



# **University of Asia Pacific**

Department of Computer Science and Engineering

## **Lab Manual**

Course No: CSE 430

Course Title: Compiler Design Lab

Semester: 2<sup>nd</sup> Year: 4<sup>th</sup>

### **Prepared by**

Baivab Das

Lecturer

baivab@uap-bd.edu

Date: 23 March 2023

## Contents

Instructions for the Lab .....	5
Administrative Policy of the Laboratory .....	5
Course Outline .....	6
Lab 1 & Lab 2: Creating a Lexical Analyzer.....	8
Learning Outcomes .....	8
Task: Write a program for lexical analysis i.e. takes a line from file or keyboard and specify each word or character into the following tokens. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. You can try for identifying duplicate identifiers also. ....	8
Sample Input .....	9
Sample output.....	9
Lab 3 & Lab 4: Symbol Table .....	10
Learning Outcomes .....	10
Task: Construct a simple hash-based symbol table (data-dictionary) based on chaining.....	10
Usage of symbol table by all the phases of a compiler .....	10
Symbol Table Entries .....	10
Example .....	11
Symbol Table Data Structure .....	11
Hash Table .....	11
Chaining.....	13
Sample Inputs and Outputs.....	13
Lab 5 & Lab 6: Creating a Lexical Analyzer using Lex.....	15
Learning Outcomes .....	15
Task: Create a lexical analyzer using lex on linux environment which will be able to recognize keywords, digits, identifiers and comments .....	15
File format of Lex.....	15
Function of Lex .....	16
Installation of Lex .....	16
Compiling lex programs.....	16
Pattern Matching Primitives .....	18
Pattern Matching Examples .....	18

Sample Input .....	19
Sample Output .....	19
Lab 7 & Lab 8: Elimination of Left Recursion in a grammar .....	20
Learning Outcomes .....	20
Task: Create a program that will be able to recognize left recursion in a given grammar and generate the modified grammar with left recursion eliminated.....	20
Sample Input(s) and Output(s).....	20
Lab 9 & Lab 10: First and Follow Function .....	22
Learning Outcomes .....	22
Task: Create a data structure to store a Context Free Grammar and write functions to find FIRST and FOLLOW of it .....	22
FIRST( $\alpha$ ) .....	22
Rule 1.....	22
Example 1: .....	22
Rule 2.....	22
Example 2: .....	22
Rule 3.....	22
Example 3 .....	23
Example 4 .....	23
Example 5 .....	23
Example 6 .....	23
FOLLOW (A) .....	23
Rule 1.....	24
Rule 2.....	24
Rule 3:.....	24
Example 1 .....	24
Example 2 .....	24
Example 3 .....	25
Sample Inputs and outputs .....	25
Lab 11 & Lab 12: Three Address Code Generation .....	27
Learning Outcomes .....	27

Task: Write a program to generate three address code from a given expression .....	27
General representation of TAC .....	27
Example of TAC .....	27
Implementation Details.....	28
Precedence .....	28
Example: .....	28
Left associativity .....	28
Example .....	29
Things to Remember .....	30
Input .....	31
Output .....	31
Lab 13 & Lab 14: Project Submission .....	32
Project Showcase .....	32
Guidelines.....	32

## Instructions for the Lab

- Each person may only use one computer at a time.
- Handle instruments with care. Do not remove any peripherals without approval of the lab instructor or lab staff.

## Administrative Policy of the Laboratory

- 1) Class assessment tasks must be performed by students individually, without help of others.
- 2) Viva for each program will be taken and considered as a performance.
- 3) Plagiarism is strictly forbidden.

## Course Outline

<b>Program:</b>	BSc in Computer Science and Engineering
<b>Course Title:</b>	Compiler Design Lab
<b>Course Code:</b>	CSE 430
<b>Semester:</b>	Fall-2022
<b>Level:</b>	4 <sup>th</sup> Year 2 <sup>nd</sup> Semester
<b>Credit Hour:</b>	1.5
<b>Name &amp; Designation of Teacher:</b>	<b>Baivab Das</b> , Lecturer
<b>Office/Room:</b>	7th Floor
<b>Class Hours:</b>	<b>Section A1:</b> Monday (8:00AM – 10:50AM) <b>Section A2:</b> Monday (2.00PM – 4.50 PM) <b>Section B1:</b> Sunday (2.00PM – 4.50 PM) <b>Section B2:</b> Wednesday (2.00PM – 4.50 PM)
<b>Consultation Hours:</b>	<b>Section A:</b> Monday (11:00 PM – 12:20 PM) <b>Section B:</b> Sunday (12:00 PM – 1:20 PM)
<b>E-mail:</b>	baivab@uap-bd.edu
<b>Mobile</b>	+8801963325240
<b>Pre-requisite (if any):</b>	None
<b>Rationale:</b>	Required course in the CSE program. The technology to build compilers which translate high-level programming languages has made the proliferation of computer use possible. The knowledge and skills in compiler construction are essential for computing professionals.
<b>Course Synopsis:</b>	This course will cover the main aspects of the Compiler Designing. Student will learn how to use scanner and parser generator tools (e.g., Flex, Yacc, etc). Students will then learn the designing and implementation of lexical analyzer, symbol tables, parser, intermediate code generator and code generator.

<b>Course Objectives:</b>	1. Teach various aspects of Compiler Design. Describe programming approaches that avoid common coding errors. 2. Teach various aspects of Compiler Design. 3. Demonstrate how to use various compiler-construction tools.					
<b>Course Outcomes (COs):</b>	<b>Upon successful completion of the course, students should be able to:</b> CO1. <b>Solve</b> problem using Structured Programming approaches. CO2. <b>Develop</b> solutions for real life problems using Structured Programming approach.					
<b>CO-PO Mapping:</b>		<b>CO No.</b>	<b>Corresponding POs (Appendix-1)</b>	<b>Bloom’s taxonomy domain/level (Appendix-2)</b>		
		CO1	b	1/Apply		
		CO2	c	1/Apply		
		CO3	c	2/Manipulation		
<b>Assessment Methods:</b>		<b>Assessment Type</b>	<b>% weight</b>	<b>CO1</b>	<b>CO2</b>	
		Classwork Performance, Assignment	60%	40%	20%	
		Viva	10%	10%		
		Project	30%	20%	10%	
		<b>Total</b>	<b>100%</b>	<b>70</b>	<b>30</b>	
<b>Grading System:</b>	As per the approved grading scale of University of Asia Pacific (Appendix-3).					
<b>Textbook:</b>	Compilers - Principles, Techniques, and Tools, Aho, Sethi, Ullman					
<b>Recommended References:</b>	Flex & Bison, John R. Levine A Compact Guide to Lex & YACC, Thomas Niemann					

## Lab 1 & Lab 2: Creating a Lexical Analyzer

### Learning Outcomes

- Create a basic lexical analyzer.
- Understand the concept of a scanner.
- Understand how tokenization is done.

Task: Write a program for lexical analysis i.e. takes a line from file or keyboard and specify each word or character into the following tokens. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. You can try for identifying duplicate identifiers also.

Lexical Analysis or Scanning is the first phase of a compiler. It is the process of breaking down a sequence of characters (such as source code) into a series of tokens or lexemes, which are meaningful units of a programming language. This process is also known as tokenization or scanning.

During lexical analysis, the source code is scanned from left to right, and each character is classified into different categories such as letters, digits, and symbols. The resulting tokens are then passed on to the next phase of the compilation process, which is syntax analysis.

The lexemes are tokenized in this manner:

- Any word either combination of characters and digits or combination of characters: Identifier.
- Any number : Constant
- Single character tokens:
  - Parenthesis : ( ), { }, [ ]
  - Punctuation sign(s): ;(semicolon), : (colon), , (comma)
  - Arithmetic Operator(s): + , - , \* , /
- Logical Operator(s): > , >= , < , <= , == , !=
- Keyword(s): There are total 32 keywords in C. They are:

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while



### Sample Input

(The input can be a console Input or a file Input)

```
void main()  
{  
int a, b, c;  
int a = b*c + 10;  
}
```

### Sample output

Keyword (2): void, int

Identifier (4): main, a, b, c

Arithmetic Operator (3): =, \*, +

Constant (1): 10

Punctuation (2): ;

Parenthesis (2): {, }

## Lab 3 & Lab 4: Symbol Table

### Learning Outcomes

- Learn the function of a symbol table.
- Understand the attributes of symbol table.
- Create a symbol table by using appropriate data structures.

Task: Construct a simple hash-based symbol table (data-dictionary) based on chaining.

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces etc. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate target code.

### Usage of symbol table by all the phases of a compiler

- i. Lexical analysis: Creates new entries for each new identifiers.
- ii. Syntax Analysis: Adds information regarding attributes like type, scope, dimension, line of reference and line of use.
- iii. Semantic Analysis: adds information regarding attributes like type, scope, dimension, line of reference and line of usage.
- iv. Intermediate Code Generation: information in symbol table helps to add temporary variable's information.
- v. Code Optimization: information in symbol table used in machine-dependent optimization by considering address and variable information.
- vi. Target Code Generation: generates the code by using the address information of identifiers.

### Symbol Table Entries

each entry in the symbol table is associated with “attributes” that support the compiler in different phases the attributes are:

- Name
- Size
- Dimension (used if it is an array)
- Type
- Line of declaration (where the variable is declared to generate errors)
- Line of usage (link list to keep track of multiple usage of a variable)
- Address

## Example

Name	Type	Size	Dimension	Line of code	Line of usage	Address
John	Char	4	1	5	12	0x6dfed4
Age	Int	2	0	3	5	0x7ffdd8747

## Symbol Table Data Structure

To store data in a symbol table we can use various types of data structures such as:

Data Structure	Complexity
Linear List (Linked List)	$O(n)$
Binary Search Tree	$O(\log n)$
Hash Table	$O(1)$

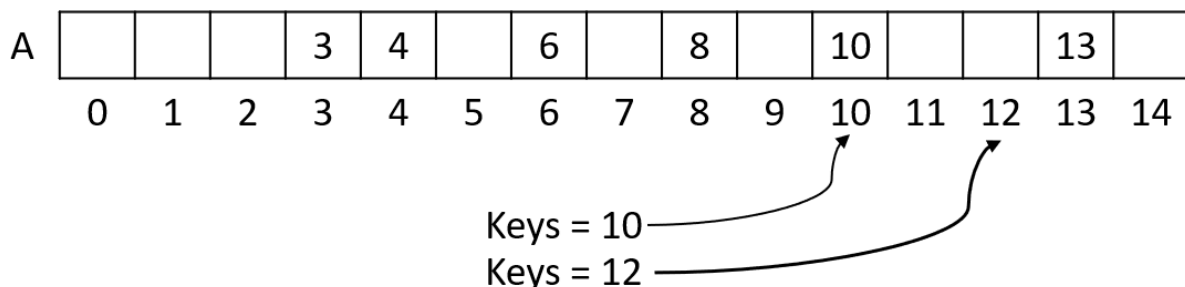
## Hash Table

We will use hash table as a data structure for our symbol table. A hash table allows very fast retrieval of data. It is widely used in database indexing, caching and error checking. At the high level, a hash table is a key value lookup.

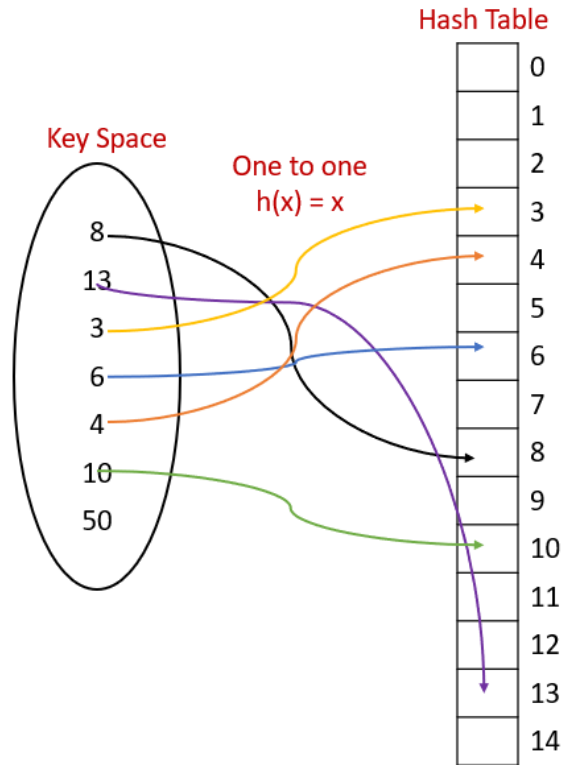
In linear search, the elements are arranged in an array randomly. And for searching we start from left to right. This is the reason of the time complexity  $O(n)$ . In binary search the elements are sorted and kept in an array. Binary search improves the search method and makes it faster to  $O(\log n)$ . But, sorting the elements will take time. This is the reason hashing technique is used as it takes constant time  $O(1)$ .

In hashing technique if list of elements are given and array space is given where we have to store the list of elements then we store the element upon its own index. We take the value of an element itself as the index and we store it at same index.

Keys: 8, 3, 13, 6, 4, 10



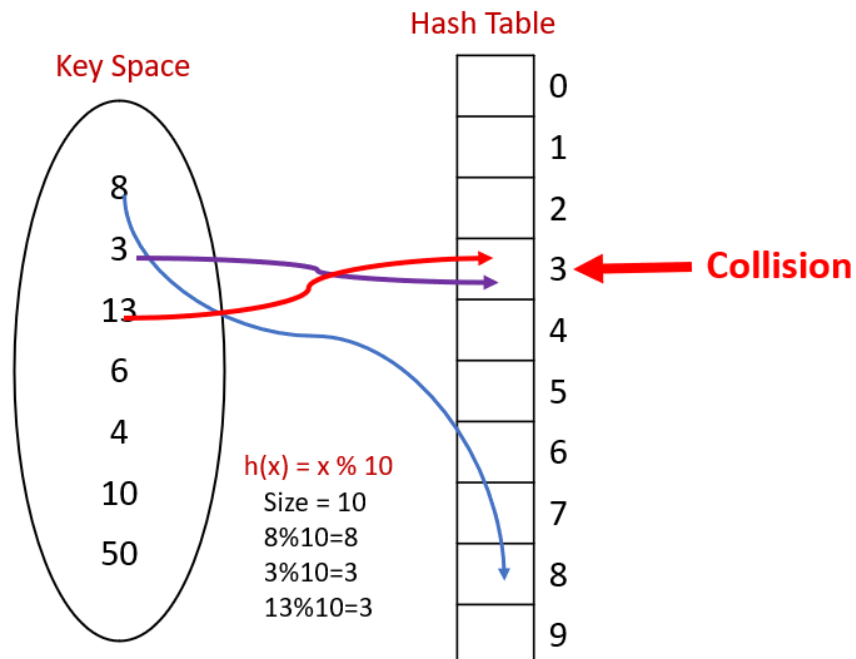
Here, keys 8, 3, 13, 6, 4, 10 are stored on the index 8, 3, 13, 6, 4, 10 itself. But, problem will occur if the key is 50 then we will have to store it on index 50 and a lot of space will be wasted. To improve this technique, we adapt a mathematical model for hashing based on functions.



The above diagram shows the use of the hash function  $h(x) = x$  and here we have the same space problem. So, we modify the function as:

$$h(x) = x \% 10 \quad \text{where, 10 is the size of hash table and } x \text{ is the key}$$

As the modified hash function is used, we are able to solve the space problem at some extend.



We take the key 8 and store it on  $h(8)=8\%10=8$  index. Similarly, we store 3 as  $3\%10=3$  index. But, while we try to store key 13,  $13\%10=3$  which is overlapping with already existing key 3. This scenario is called 'collision'.

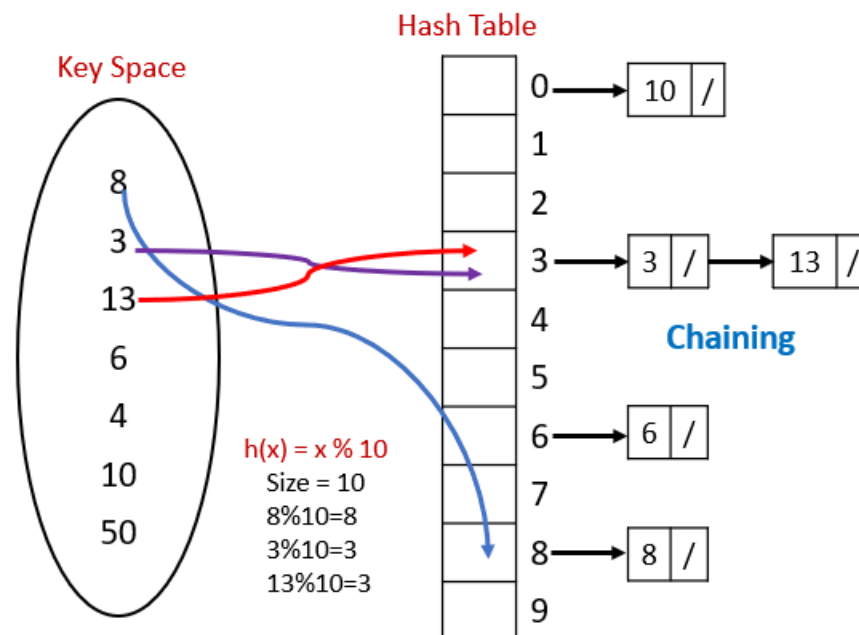
## Chaining

Chaining is an open hashing collision resolution technique which uses the same modified hash function as shown above. But, here the element will not be stored directly at the index. A chain of linked list is formed to store the elements. So, we can use a distinct hash table size as per as our choice and we can store as much as elements in the hash table we want.

For searching a key in the chaining based hash table, we use the same hash function

$$h(x) = x \% 10 \quad \text{where, 10 is the size of hash table and } x \text{ is the key}$$

So, if we need to search for the element 13,  $13\%10=3$ , we search the chain at index 3 for the key 13. When an element is not found in the chain, it is considered as a unsuccessful search.



## Sample Inputs and Outputs

The input to your program will be a sequence of six tuples . They are:

- Type
- Size
- Dimension
- Line of Code
- Address

where each element in each tuple is a string

An example of input sequence is given below:

x, ID, 2, 1, 5, 0x6dfed4

The symbol table should be able to store multiple rows and the symbol table should have the following functionalities:

1. **Insert** a new symbol/name along with its type into the symbol table
2. **Search** a symbol/name along with its type from the symbol table
3. **Delete** a symbol/name along with its type into the symbol table
4. **Show** the contents of the symbol table in the console
5. **Update** an already existing entry in the symbol table
6. **getHashKey()** function is used to show the hash value

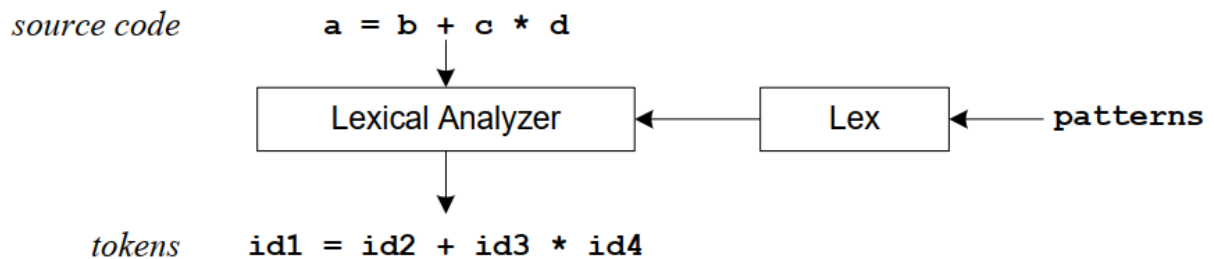
## Lab 5 & Lab 6: Creating a Lexical Analyzer using Lex

### Learning Outcomes

- Learn the basics of lex tool.
- Learn the basic use of virtual machines.
- Learn how to install lex and yacc on linux environment.
- Understand the syntax of the lexical analyzer tool (lex) and use of regular expressions .
- Create a lexical analyzer using lex on linux environment for recognizing tokens.

Task: Create a lexical analyzer using lex on linux environment which will be able to recognize keywords, digits, identifiers and comments

Lex is a tool or software which automatically generates a lexical analyzer (finite Automata). It takes as its input a LEX source program and produces lexical Analyzer as its output. Lexical Analyzer will convert the input string entered by the user into tokens as its output. We define regular expressions by using lex. The following is the process where the source code is converted into tokens by using lex tool:



### File format of Lex

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

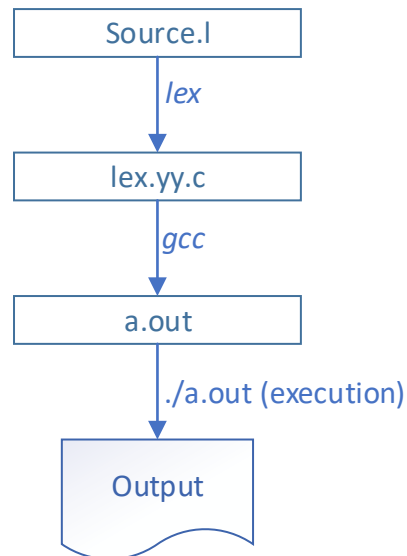
**Definitions** include declarations of constant, variable and regular definitions.

**Rules** define the statement of form `p1 {action1} p2 {action2}...pn {action}`.

Where **pi** describes the regular expression and **action** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

## Function of Lex

The process of compiling a lex program is show below:



## Installation of Lex

Assuming the student has access to an updated Linux environment (preferably Debian based systes), follow the process shown below to install flex:

Open terminal and type “sudo apt-get install flex”. A prompt will occur asking permission for downloading the flex package. Press ‘Y’ and the installation will be completed automatically.

```
baivab@baivab: ~  
baivab@baivab:~$ sudo apt-get install flex  
[sudo] password for baivab:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  libfl-dev libfl2 m4  
Suggested packages:  
  bison flex-doc m4-doc  
The following NEW packages will be installed:  
  flex libfl-dev libfl2 m4  
0 upgraded, 4 newly installed, 0 to remove and 361 not upgraded.  
Need to get 534 kB of archives.  
After this operation, 1,486 kB of additional disk space will be used.  
Do you want to continue? [Y/n]
```

## Compiling lex programs

Follow the process to compile a lex program to recognize patterns:

1. The regular expression of an identifier is:

*letter(letter|digit)\**



We will use this regular expression in our lex program.

2. Create a file with '.l' extension such as: test.l and write the following code:

```
%{
#include<stdio.h>
%}

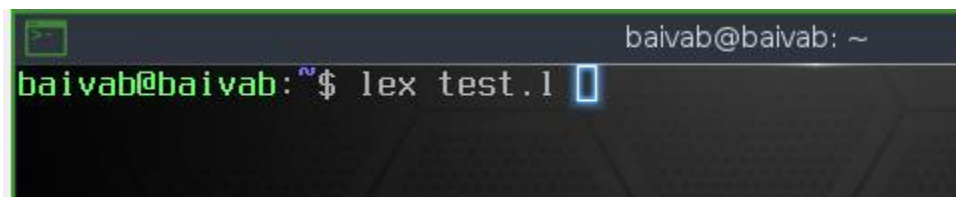
%%

[a-zA-Z][a-zA-Z0-9]* {printf ("Identifier \n");}

%%
int main()
{
yylex();
}
```

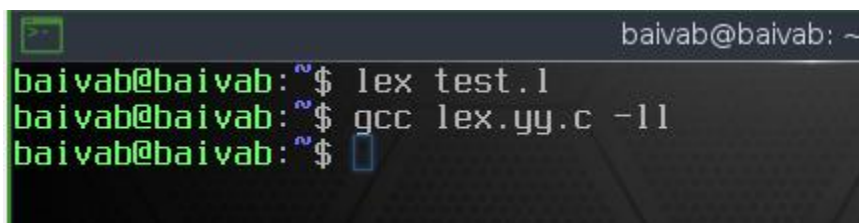
Here, the regular expression `[a-zA-Z][a-zA-Z0-9]*` is used for recognizing identifiers and the function `yylex()` returns a value indicating the type of token that has been obtained.

3. This test.l file will be now converted to a C program file by using the lex tool. Open terminal and change directory to the same directory where the test.l file is saved.
4. Type `lex test.l` and a new file name `lex.yy.c` will be created.



```
baivab@baivab: ~
baivab@baivab:~$ lex test.l
```

5. We can observe that this is a C file. So, we will use the gcc compiler to compile this c program. Type the command `gcc lex.yy.c -ll` to compile. A new file 'a.out' will be created which is the executable machine code file.



```
baivab@baivab: ~
baivab@baivab:~$ lex test.l
baivab@baivab:~$ gcc lex.yy.c -ll
baivab@baivab:~$
```

6. Now we execute the 'a.out' executable file to run the program by typing `./a.out`

```
baivab@baivab: ~  
baivab@baivab:~$ ./a.out  
hi  
Identifier  
  
a  
Identifier  
  
a1  
Identifier  
  
a121  
Identifier
```

7. We can see by giving input which matches with the regular expression of identifier it is recognizing the identifiers.

### Pattern Matching Primitives

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
\$	end of line
a b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

### Pattern Matching Examples

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbcb abcbcbcb ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\ -z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[ \t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b

[a b]	one of: a,  , b
a b	one of: a, b

### Sample Input

```
void main(){ int a, b2, c;    //hello    a = b2 * c + 10; }
```

### Sample Output

void	keyword
main	identifier
int	keyword
a	identifier
b2	identifier
c	identifier
//hello	comment
10	digit

## Lab 7 & Lab 8: Elimination of Left Recursion in a grammar

### Learning Outcomes

- Learn the concept of left recursion in context free grammars.
- Understand if a given grammar is left recursive or not.
- Create a program to eliminate left recursion.

Task: Create a program that will be able to recognize left recursion in a given grammar and generate the modified grammar with left recursion eliminated

Whenever there is a production in the form of:

$$A \rightarrow Aa$$

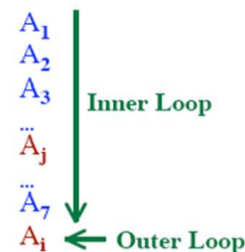
It is considered as left recursion.

There are two types of left recursion:

- Immediate left recursion
- Non-immediate left recursion

The algorithm for eliminating left recursion is given below:

```
Assume the nonterminals are ordered  $A_1, A_2, A_3, \dots$ 
(In the example: S, A, B)
for each nonterminal  $A_i$  (for  $i = 1$  to  $N$ ) do
  for each nonterminal  $A_j$  (for  $j = 1$  to  $i-1$ ) do
    Let  $A_j \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_N$  be all the rules for  $A_j$ 
    if there is a rule of the form
       $A_i \rightarrow A_j \alpha$ 
    then replace it by
       $A_i \rightarrow \beta_1 \alpha \mid \beta_2 \alpha \mid \beta_3 \alpha \mid \dots \mid \beta_N \alpha$ 
    endIf
  endFor
  Eliminate immediate left recursion
  among the  $A_i$  rules
endFor
```



Sample Input(s) and Output(s)

**Sample Input 1:**

$E \rightarrow E+T \mid T$

**Sample Output 1:**

After elimination of left recursion the grammar is:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
**Sample Input 2:**
$$T \rightarrow T * F \mid F$$
**Sample Output 2:**

After elimination of left recursion the grammar is:

$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$

## Lab 9 & Lab 10: First and Follow Function

### Learning Outcomes

- Learn the concept predictive parsers in context-free grammar.
- Understand the FIRST and FOLLOW function of a grammar.
- Create a program to generate FIRST and FOLLOW function of a grammar.

Task: Create a data structure to store a Context Free Grammar and write functions to find FIRST and FOLLOW of it

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, **FIRST** and **FOLLOW**, allow us to fill in the entries of a predictive parsing table for G, whenever possible. First and Follow sets are needed so that the parser can properly apply the needed production rule at the correct position. Pre-requisite for generating first and follow functions:

- i. Elimination of left recursion from the grammar
- ii. Elimination of left factoring from the grammar

### FIRST( $\alpha$ )

First( $\alpha$ ) is a set of terminal symbols that begin in strings derived from  $\alpha$ .

#### Rule 1

If FIRST ( $\alpha$ ) is the set of terminals that begin the strings derived from  $\alpha$ .

#### Example 1

Consider the production rule-

$A \rightarrow abc \mid def \mid ghi$

Then, we have-

$$\text{First}(A) = \{a, d, g\}$$

#### Rule 2

If  $\alpha \rightarrow \epsilon$ , then  $\epsilon$ , is also in FIRST ( $\alpha$ ).

#### Example 2

For a production rule  $X \rightarrow \epsilon$ ,

$$\text{First}(X) = \{ \epsilon \}$$

#### Rule 3

If FIRST ( $\alpha$ ) is the set of non-terminals:

If  $\alpha \rightarrow Y_1 Y_2 Y_3 \dots Y_k$  is a rule then

If  $a$  is in First ( $Y_1$ ) then

Add a to First ( $\alpha$ )

If  $\epsilon$  is in First ( $Y_1$ ) and a is in First ( $Y_2$ ) then

Add a to First ( $\alpha$ )

If  $\epsilon$  is in First ( $Y_1$ ) and  $\epsilon$  is in First ( $Y_2$ ) and a is in First ( $Y_3$ ) then

Add a to First ( $\alpha$ )

.....

If  $\epsilon$  is in First ( $Y_i$ ) for all  $Y_i$  then

Add  $\epsilon$  to First ( $\alpha$ )

### Example 3

First ( $Y_1$ )  $\rightarrow$  {a, b,  $\epsilon$ }

First ( $Y_2$ )  $\rightarrow$  {c, d,  $\epsilon$ }

First ( $Y_3$ )  $\rightarrow$  {e, f,  $\epsilon$ }

Then First ( $X$ )  $\rightarrow$  {a, b, c, d, e, f,  $\epsilon$ }

### Example 4

First ( $Y_1$ )  $\rightarrow$  {a, b,  $\epsilon$ }

First ( $Y_2$ )  $\rightarrow$  {c, d,  $\epsilon$ }

First ( $Y_3$ )  $\rightarrow$  {e, f}

Then First ( $X$ )  $\rightarrow$  {a, b, c, d, e, f}

### Example 5

First ( $Y_1$ )  $\rightarrow$  {a, b,  $\epsilon$ }

First ( $Y_2$ )  $\rightarrow$  {c, d}

First ( $Y_3$ )  $\rightarrow$  {e, f,  $\epsilon$ }

Then First ( $X$ )  $\rightarrow$  {a, b, c, d}

### Example 6

First ( $Y_1$ )  $\rightarrow$  {a, b}

First ( $Y_2$ )  $\rightarrow$  {c, d,  $\epsilon$ }

First ( $Y_3$ )  $\rightarrow$  {e, f,  $\epsilon$ }

Then First ( $X$ )  $\rightarrow$  {a, b}

## FOLLOW (A)

FOLLOW (A), for nonterminal A, to be the set of terminals that can appear immediately to the right of A in some sentential form. That is, the set of terminals such that there exists a derivation of the form  $S \rightarrow aA\alpha\beta$  for some  $\alpha$  and  $\beta$ .

To compute FOLLOW (A) for all nonterminal A, apply the following rules until nothing can be added to any FOLLOW set:

#### Rule 1

Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right end marker.

#### Rule 2

If there is a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST ( $\beta$ ), except for  $\epsilon$ , is placed in FOLLOW (B).

#### Rule 3

If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where FIRST ( $\beta$ ) contains  $\epsilon$  (i.e.,  $\beta \rightarrow \epsilon$ ),

Then everything in FOLLOW (A) is in FOLLOW (B).

#### Example 1

$S \rightarrow aSe \mid B$	$\text{FIRST}(S) = \{a, b, c, d, \epsilon\}$
$B \rightarrow bBCf \mid C$	$\text{FIRST}(B) = \{b, c, d, \epsilon\}$
$C \rightarrow cCg \mid d \mid \epsilon$	$\text{FIRST}(C) = \{c, d, \epsilon\}$

#### According to Rule 1:

$\text{FOLLOW}(S) = \{\$\}$

#### According to Rule 2:

$\text{FOLLOW}(C) = \{f, g\}$   
 $\text{FOLLOW}(B) = \{c, d, f\}$   
 $\text{FOLLOW}(S) = \{\$, e\}$

#### According to Rule 3:

$\text{FOLLOW}(C) = \{f, g\} \cup \text{FOLLOW}(B) = \{c, d, e, f, g, \$\}$   
 $\text{FOLLOW}(B) = \{c, d, f\} \cup \text{FOLLOW}(S) = \{c, d, e, f, \$\}$   
 $\text{FOLLOW}(S) = \{\$, e\}$

#### Example 2

Consider the expression grammar stated below:

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \epsilon$

$T \rightarrow F T'$



$T' \rightarrow * F T' \mid \epsilon$

$F \rightarrow (E) \mid id$

Then:

$FIRST(E) = FIRST(T) = FIRST(F) = \{ (, id \}$

$FIRST(E') = \{ +, \epsilon \}$

$FIRST(T') = \{ *, \epsilon \}$

$FOLLOW(E) = FOLLOW(E') = \{ ), \$ \}$

$FOLLOW(T) = FOLLOW(T') = \{ +, ), \$ \}$

$FOLLOW(F) = \{ +, *, ), \$ \}$

### Example 3

Consider the expression grammar stated below

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \epsilon$

$D \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

The first and follow functions are as follows-

$First(S) = \{a\}$

$First(B) = \{c\}$

$First(C) = \{b, \epsilon\}$

$First(D) = \{First(E) - \epsilon\} \cup First(F) = \{g, f, \epsilon\}$

$First(E) = \{g, \epsilon\}$

$First(F) = \{f, \epsilon\}$

$Follow(S) = \{\$ \}$

$Follow(B) = \{First(D) - \epsilon\} \cup First(h) = \{g, f, h\}$

$Follow(C) = Follow(B) = \{g, f, h\}$

$Follow(D) = First(h) = \{h\}$

$Follow(E) = \{First(F) - \epsilon\} \cup Follow(D) = \{f, h\}$

$Follow(F) = Follow(D) = \{h\}$

### Sample Inputs and outputs

#### Sample Input 1

You can take input from a **text file/console**. Instead of **epsilon ( $\epsilon$ )** use **hash (#)** symbol.

```
E -> TR
R -> +T R | #
T -> F Y
Y -> *F Y | #
F -> (E) | i
```

### Sample Output 1

```
First (E) = {(, i,}
First(R) = {+, #,}
First (T) = {(, i,}
First(Y) = {*, #,}
First (F) = {(, i,}
```

---

```
Follow (E) = {$, ),}
Follow(R) = {$, ),}
Follow (T) = {+, $, ),}
Follow(Y) = {+, $, ),}
Follow (F) = {*, +, $, ),}
```

### Sample Input 2

```
S -> aSe | B
B -> bBCf | C
C -> cCg | d | #
```

### Sample Output 2

```
FIRST(S) = {a, b, c, d, #,}
FIRST (B) = {b, c, d, #,}
FIRST(C) = {c, d, #,}
```

---

```
FOLLOW (S) = {$, e}
FOLLOW (B) = {c, d, e, f, $}
FOLLOW (C) = {c, d, e, f, g, $,}
```

## Lab 11 & Lab 12: Three Address Code Generation

### Learning Outcomes

- Learn the concept intermediate code generation.
- Understand the use of Three Address Code.
- Create a program to generate TAC.

**Task:** Write a program to generate three address code from a given expression

**Three-address code** (often abbreviated to TAC or 3AC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Three address code is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in a temporary variable generated by the compiler. The compiler decides the order of operation given by three address code.

### General representation of TAC

$$a = b \text{ op } c$$

Where a, b, or c represents operands like names, constants, or compiler-generated temporaries and op represents the operator.

### Example of TAC

$$x = (-b + \sqrt{b^2 - 4*a*c}) / (2*a)$$

TAC of the above expression will be:

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

## Implementation Details

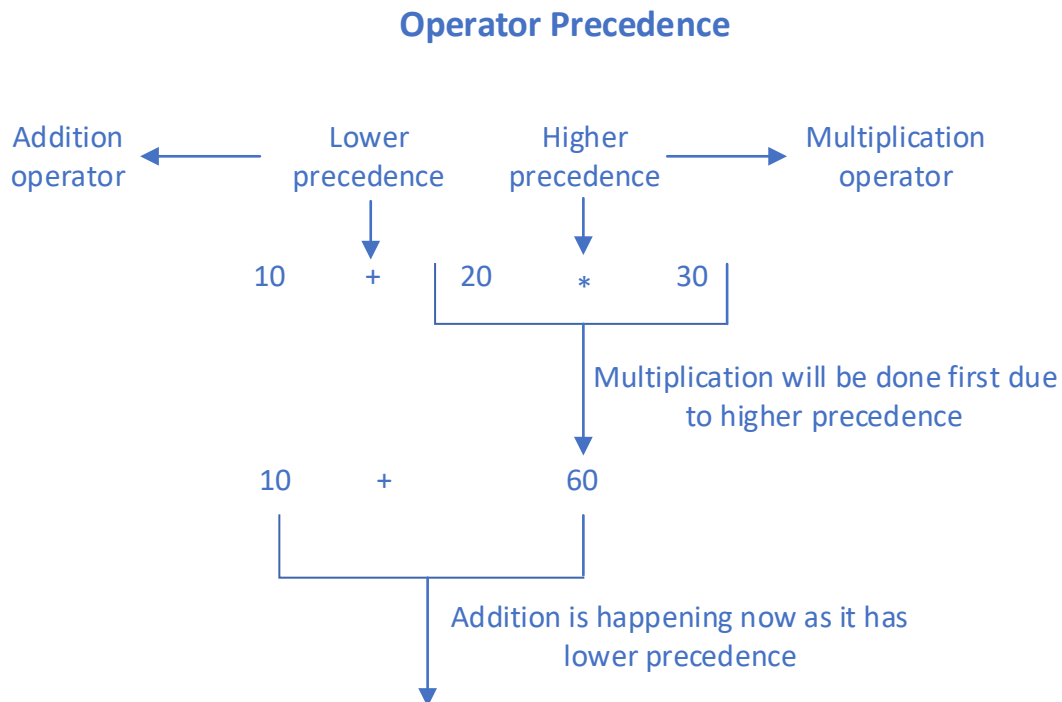
### Precedence

While implementing the TAC, the first thing to keep in mind is precedence. **Precedence/ order of operations** is a collection of rules that reflect conventions about which procedures to perform first in order to evaluate a given mathematical expression.

The precedence of an operator specifies how "tightly" it binds two expressions together. For example, in the expression  $1 + 5 * 3$ , the answer is 16 and not 18 because the multiplication (" $*$ ") operator has higher precedence than the addition (" $+$ ") operator. Parentheses may be used to force precedence, if necessary. For instance:  $(1 + 5) * 3$  evaluates to 18.

Example:

**Solve:  $10 + 20 * 30$**

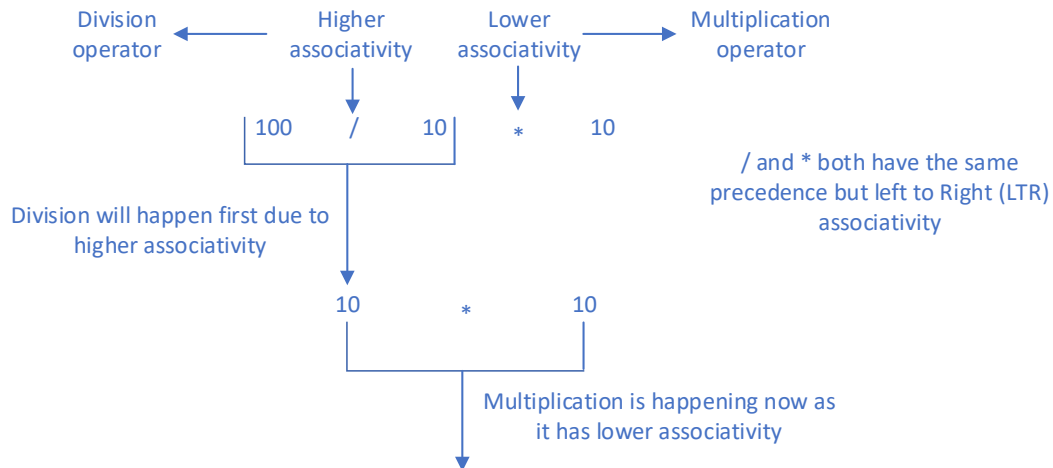


### Left associativity

Operators Associativity is used when two operators of the same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.

For example: ' $*$ ' and ' $/$ ' have the same precedence and their associativity is Left to Right, so the expression " $100 / 10 * 10$ " is treated as " $(100 / 10) * 10$ ".

## Operator Associativity

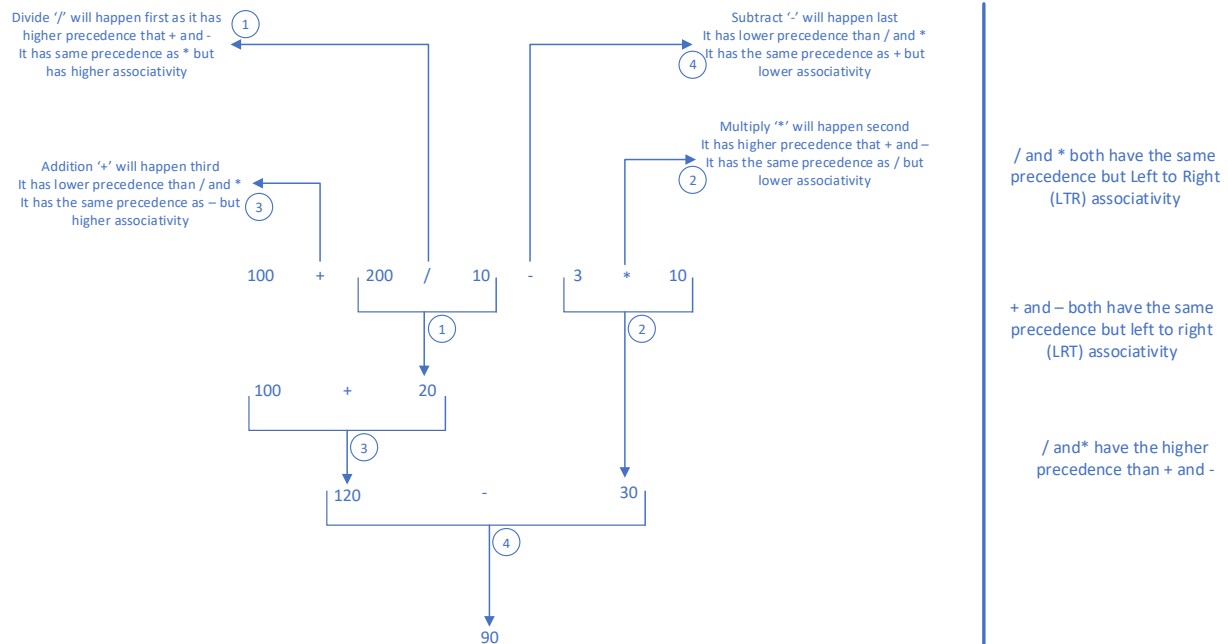


Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets.

## Example

$$100 + 200 / 10 - 3 * 10$$

### Operator Precedence and Associativity



## Things to Remember

- 1) Associativity is only used when there are two or more operators of the same precedence.
- 2) All operators with the same precedence have the same associativity.
- 3) Precedence and associativity of postfix ++ and prefix ++ are different.

Precedence	Operator	Description	Associativity
<b>1</b>	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal	
<b>2</b>	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement	
<b>3</b>	* / %	Multiplication, division, and remainder	Left-to-right
<b>4</b>	+ -	Addition and subtraction	
<b>5</b>	<< >>	Bitwise left shift and right shift	
<b>6</b>	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
<b>7</b>	== !=	For relational = and ≠ respectively	
<b>8</b>	&	Bitwise AND	
<b>9</b>	^	Bitwise XOR (exclusive or)	
<b>10</b>		Bitwise OR (inclusive or)	
<b>11</b>	&&	Logical AND	
<b>12</b>		Logical OR	

<b>13</b>	<b>?:</b>	Ternary conditional	Right-to-left
<b>14</b>	<b>=</b>	Simple assignment	

It is good to know precedence and associativity rules, but the best thing is to use brackets, especially for less commonly used operators (operators other than +, -, \*, .. etc). Brackets increase the readability of the code as the reader doesn't have to see the table to find out the order.

For simplicity in this experiment, use only first brackets and arithmetic operators (^, \*, /, +, -, %, =) and some unary operators.

### Input

You should take inputs from a file/console.

-(a x b) + (c + d) - (a + b + c + d)

### Output

- (1) T1 = a x b
- (2) T2 = uminus T1
- (3) T3 = c + d
- (4) T4 = T2 +T3
- (5) T5 = a + b
- (6) T6 = T3 + T5
- (7) T7 = T4

## Lab 13 & Lab 14: Project Submission

### Project Showcase

#### Guidelines

Below are some guidelines for the project:

- Project can be an individual project or a group project.
- Include all the source code and other supporting documents required for the project.
- For lexical analyzer and parser use lex and yacc respectively. But, other programming languages can also be used to generate same output as lex and yacc if needed.
- The project should be fully functional before submission.
- Project demo must be shown to the assigned faculty member.
- Proper documentation of design and implementation of the project is mandatory.
- Plagiarism is completely prohibited.



## **Appendix-1:**

### **Washington Accord Program Outcomes (PO) for engineering programs:**

- (a) Apply knowledge of mathematics, natural science, engineering fundamentals and an engineering specialization as specified in K1 to K4 respectively to the solution of complex engineering problems.
- (b) Identify, formulate, research literature and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences. (K1 to K4)
- (c) Design solutions for complex engineering problems and design systems, components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal, and environmental considerations. (K5)
- (d) Conduct investigations of complex problems using research-based knowledge (K8) and research methods including design of experiments, analysis and interpretation of data, and synthesis of information to provide valid conclusions.
- (e) Create, select and apply appropriate techniques, resources and modern engineering and IT tools, including prediction and modeling, to complex engineering activities with an understanding of their limitations.
- (f) Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to professional engineering practice and solutions to complex engineering problems. (K7)
- (g) Understand and evaluate the sustainability and impact of professional engineering work in the solution of complex engineering problems in societal and environmental contexts. (K7)
- (h) Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice. (K7)
- (i) Function effectively as an individual, and as a member or leader in diverse teams and in multi-disciplinary settings.
- (j) Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- (k) Demonstrate knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

(I) Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

In addition to incorporating the above-listed POs (graduate attributes), the educational institution may include additional outcomes in its learning programs. An engineering program that aims to attain the abovementioned POs must ensure that its curriculum encompasses all the attributes of the Knowledge Profile (K1 – K8) as presented in Table 4.1 and as included in the PO statements. The ranges of Complex Problem Solving (P1 – P7) and Complex Engineering Activities (A1 – A5) are given in Tables 4.2 and 4.3, respectively.

### **Knowledge Profile**

K1     A systematic, theory-based understanding of the natural sciences applicable to the discipline

K2     Conceptually based mathematics, numerical analysis, statistics and formal aspects of computer and information science to support analysis and modelling applicable to the discipline

K3     A systematic, theory-based formulation of engineering fundamentals required in the engineering discipline

K4     Engineering specialist knowledge that provides theoretical frameworks and bodies of knowledge for the accepted practice areas in the engineering discipline; much is at the forefront of the discipline

K5     Knowledge that supports engineering design in a practice area

K6     Knowledge of engineering practice (technology) in the practice areas in the engineering discipline

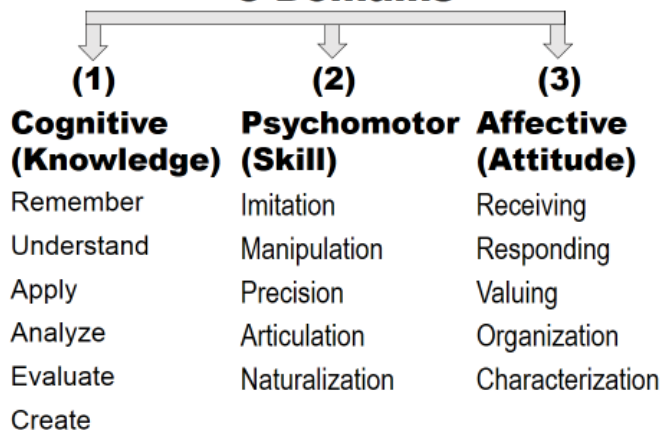
K7     Comprehension of the role of engineering in society and of the identified issues in engineering practice in the discipline: ethics and the engineer's professional responsibility to public safety; the impacts of engineering activity in economic, social, cultural, environmental and sustainability terms

K8     Engagement with selected knowledge in the research literature of the discipline

## Appendix-2

### **Bloom's Taxonomy (Taxonomy of Learning)**

#### **3 Domains**



## Appendix-3:

### **UAP Grading Policy:**

<b>Numeric Grade</b>	<b>Letter Grade</b>	<b>Grade Point</b>
80% and above	A+	4.00
75% to less than 80%	A	3.75
70% to less than 75%	A-	3.50
65% to less than 70%	B+	3.25
60% to less than 65%	B	3.00
55% to less than 60%	B-	2.75
50% to less than 55%	C+	2.50
45% to less than 50%	C	2.25
40% to less than 45%	D	2.00
Less than 40%	F	0.00