Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

LABORATORY MANUAL

Course No.: CSE4130
Course Title: Formal Languages and Compilers Lab

For the students of 4th Year, 1st semester of
B.Sc. in Computer Science and Engineering program

TABLE OF CONTENTS

COURSE OBJECTIVES

- Comprehend the fundamental concepts of scanning and filtering a source program using techniques like lexical analysis, symbol table construction and detecting syntax errors while parsing.
- Apply Context Free Grammars for parsing to detect syntactic and semantic errors in expressions and statements.
- Analyze complex problems like LL(1) and LR(1) parsing algorithms, intermediate code generation and machine code generation.

PREFFERED TOOL(S)

- Code Blocks

TEXT/REFERENCE BOOK(S)

- Aho A. V., Lam M. S., Sethi R., Ullman J. D., Compilers: Principles, Techniques and Tools, Pearson Education, 2nd Edition, 2007.
- Hopcroft J. E., Motwani R., Ullman J. D., Introduction to Automata Theory, Languages, and Computation, Prentice Hall, 3rd Edition, 2007.

ADMINISTRATIVE POLICY OF THE LABORATORY

- Students must perform class assignments individually without the help of others.
- Viva for lab exercises and assignments will be arranged as an important component of the assessment procedure.
- Plagiarism is strictly prohibited and will be dealt with strictly.

Session 1: Scanning and Filtering a Source Program

I. OBJECTIVES

To develop a program which can filter comments and white space characters from a source program.

II. DEMONSTRATION OF USEFUL RESOURCES

Extracting the sequence of occurrences of a specified character from a source program.

Sample Input: datafile1.c

datafile1.c

```
#include <stdio.h>
int main(void)
{
   FILE *p1,*p2; char c;

   p1 = fopen("datafile1.c", "r");
   p2 = fopen("parentheses.txt","w");

   if(!p1) printf("\nFile can't be opened!");
   else {
          while((c = fgetc(p1)) != EOF) {
                 if ((c == '(') || (c == ')'))
                 fputc(c, p2);  } }
   fclose(p1);
   fclose(p2);

   p2 = fopen("parentheses.txt","r");
   while((c=fgetc(p2))!=EOF)
                 printf("%c",c);
   fclose(p2);

   return 0;
}
```

Output of the program: ()()()()(())((()))()()()((()))()()

III. LAB EXERCISE

1. Write a program to print the header files used in a source program.

Sample Input: *input.c*

```
#include <stdio.h>
int main()
{
  // printf() displays the string inside quotation
printf("Hello, World!");
return 0;
}
```

Sample Output: *stdio.h*

2. Write a program to add line numbers to a source program.

Sample Input: *input.c*

Sample Output:

```
1: #include <stdio.h>
2: int main()
3: {
4:  // printf() displays the string inside quotation
5: printf("Hello, World!");
6:    return 0;
7: }
```

IV. ASSIGNMENT #1:

A C source program with single and multiple line comments is given. As the first step toward compilation, you need to remove the comments and white space (extra spaces, tabs and newline characters). Develop a program that takes as input file the given source program and produces a filtered file as stated above. The program must also display both the files.

Sample Input: *input1.c*

```
#include<stdio.h>

int    main(void)
{

// Single    Line Comment

printf   ("Hello");
/* Multi
     Line
                Comment
*/
printf("World");
return 0;
}
```

Sample Output: *output.txt*

```
#include<stdio.h>int main(void){printf (“Hello”);printf(“World”);return 0;}
```

Session 2: Lexical Analysis

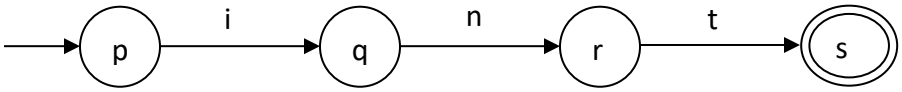
I. OBJECTIVES:

To write a program that reads any simple program as source and separates out the valid tokens from the source program.

II. DEMONSTRATION OF USEFUL RESOURCES:

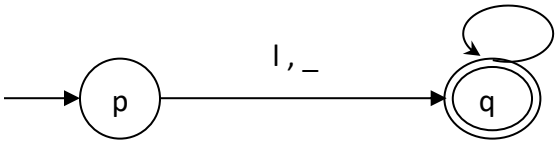
DFAs for recognition of tokens: Recognition of keywords using DFA is easier than recognition of identifiers and numbers.

1. **Keyword:** The keywords are predefined, reserved words used in programming that have special meanings to the compiler. To recognize a keyword 'int' we can just use the following DFA:



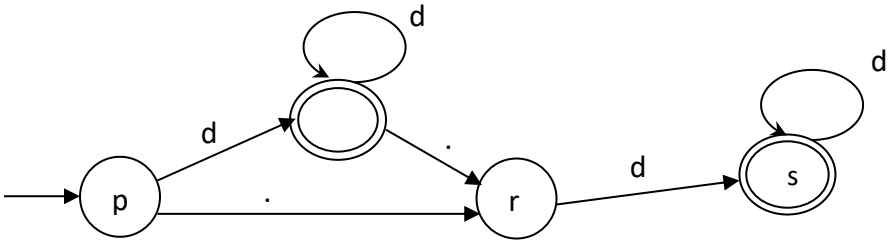
Regular expression: int

2. **Identifier:** An identifier in C is a word which starts with a letter or underscore. The 1st character can be followed repeatedly by letters, underscore or digits. No other character is allowed in the identifier. To recognize a valid C identifier the DFA might look like the one that follows:



where l stands for a|b|c|...|z|A|B|C|...|Z and d stands for 0|1|2|...|9.

3. **Numeric Constant:** A DFA for simple floating-point numbers or fixed-point numbers might take the following form:



Mark that the regular expression is dd*|d*.dd*, where d stands for 0|1|2|…|9, and mark also that the DFA has two final states.

4. Sample Implementation of DFA for Numeric Constants

```
int num_rec(char *lex)
{
  int i, l, s;
  i=0;
  if(isdigit(lex[i])) {
     s=1; i++;
  }
  else
     if(lex[i]=='.') {
       s=2; i++;
     }
  else s=0;

  l=strlen(lex);
```

```
if(s==1)
for(;i<l;i++){
  if(isdigit(lex[i]))
    s=1;
  else
    if(lex[i]=='.') {
       s=2; i++; break;
    }
    else {
       s=0; break;
    }
 }
```

```
if(s==2)
  if(isdigit(lex[i])) {
     s=3; i++;
  }
  else
     s=0;
if(s==3)
  for(;i<l;i++) {
     if(isdigit(lex[i]))
        s=3;
     else {
       s=0; break;
     }
  }

 if(s==3) s=1;
 return s;
}
```

III. LAB EXERCISE:

1. Write a program to recognize whether the entered string is a keyword or not.
   Write a program to detect whether the entered string is an identifier or not based on the finite automata described above.

Some Useful Library Functions:

Following are the functions defined in the header ctype.h:

S.N.	Function & Description
1	intisalnum(int c) This function check whether the passed character is alphanumeric.
2	intisalpha(int c) This function check whether the passed character is alphabetic.
3	intisdigit(int c) This function check whether the passed character is decimal digit.
4	intislower(int c) This function check whether the passed character is lowercase letter.
5	intisspace(int c) This function check whether the passed character is white-space.
6	intisupper(int c) This function check whether the passed character is uppercase letter.

The library also contains two conversion functions that also accept and return an "int".

S.N.	Function & Description
1	inttolower(int c) This function convert uppercase letter to lowercase.
2	inttoupper(int c) This function convert lowercase letter to uppercase.

IV. ASSIGNMENT #2:

Suppose, we have a C source program scanned and filtered as it was done in Session 1. We now take that modified file as input, and separate the lexemes first. We further recognize and mark the lexemes as different types of tokens like keywords, identifiers, operators, separators, parenthesis, numbers, etc.

Sample Input:

```
char c; int x1, x_2; float y1, y2; x1=5; x_2= 10; y1=2.5+x1*45; y2=100.o5-x_2/3; if(y1<=y2) c='y'; else c='n';
```

Step 1: Lexemes are separated. Mark that two-character relational operators are also distinguished beside separators, one-character operators, parenthesis, number constants and alphanumeric strings with or without underscore.

```
char c ; int x1 , x_2 ; float y1 , y2 ; x1 = 5 ; x_2 = 10 ; y1 = 2.5 + x1 * 45 ; y2 = 100.o5 - x_2 / 3 ; if ( y1 <= y2 ) c = ' y ' ; else c = ' n ' ;
```

Step 2: Lexemes are categorized under the categories kw for keyword, id for identifier, etc. Some may be labeled unkn (unknown).

```
[kw char] [id c] [sep ;] [kw int] [id x1] [sep ,] [id x_2] [sep ;] [kw float] [id y1] [sep ,] [id y2] [sep ;] [id x1] [op =] [num 5] [sep ;] [id x_2] [op =] [num 10] [sep ;] [id y1] [op =] [num 2.5] [op +] [id x1] [op *] [num 45] [sep ;] [id y2] [op =] [unkn 100.o5] [op -] [id x_2] [op /] [num 3] [sep ;] [kw if] [par (] [id y1] [op <=] [id y2] [par )] [id c] [op =] [sep '] [id y] [sep '] [sep ;] [kw else] [id c] [op =] [sep '] [id n] [sep '] [sep ;]
```

*Note that we need to generate an error message for [unkn 100.o5].

Session 3: Symbol Table Construction and Management

I. OBJECTIVES:

The main purpose of this session is to introduce the symbol table, the table in which all the identifiers are stored along with information about them. When a variable is declared, the compiler enters it as a new entry in the symbol table. When a variable is referred to in an expression, the compiler looks up in the symbol table to retrieve necessary information about it, such as its data type, value, etc., and the compiler performs other actions on the table like delete, update and so on.

II. DEMONSTRATION OF USEFUL RESOURCES:

Related sample programs will be demonstrated.

III. LAB EXERCISE:

Sept 1 and Step 2 of the Assignment #3 described below.

IV. ASSIGNMENT #3:

Suppose, a given C source program has been scanned, filtered and then lexically analyzed as it was done in Session 1 & 2. We have all the lexemes marked as different types of tokens like keywords, identifiers, operators, separators, parentheses, numbers, etc. Now we generate a Symbol Table describing the features of the identifiers. Then, we generate a modified token stream in accordance with the Symbol Table for processing by the next phase, that is, Syntax Analysis.

Sample source program:

Sample source program:

```
// A program fragment
float x1 = 3.125;
/* Definition of the
function f1 */
double  f1(int x)
{
  double z;
  z = 0.01;
  return z;
}
//* Beginning of 'main'
int main(void)
{
 int n1;  double z;
 n1=25;  z=f1(n1);
```

Sample input based on the program fragment:

[kw float] [id x1] [op =] [num 3.125] [sep ;] [kw double] [id f1] [par (] [kw int] [id x] [par)] [brc {] [kw double] [id z] [sep ;] [id z] [op =] [num 0.01] [sep ;] [kw return] [id z] [sep ;] [brc }] [kw int] [id main] [par (] [kw void] [par)] [brc {] [kw int] [id n1] [sep ;] [kw double] [id z] [sep ;] [id n1] [op =] [num 25] [sep ;] [id z] [op =] [id f1] [par (] [id n1] [par)] [sep ;]

Sample input based on the program fragment:

[kw float] [id x1] [op =] [num 3.125] [sep ;] [kw double] [id f1] [par (] [kw int] [id x] [par)] [brc {] [kw double] [id z] [sep ;] [id z] [op =] [num 0.01] [op +] [id x] [op *] [num 5.5] [sep ;] [kw return] [id z] [sep ;] [brc }] [kwint] [id main] [par (] [kw void] [par)] [brc {] [kw int] [id n1] [sep ;] [kw double] [id z] [sep ;] [id n1] [op =] [num 25] [sep ;] [id z] [op =] [id f1] [par (] [id n1] [par)] [sep ;]

Step 1: After complete recognition of all the lexemes only identifiers are kept in pairs for formation of Symbol Tables. The token stream should look like the one as follows:

[float] [id x1] [=] [3.125] [;] [double] [id f1] [(] [int] [id x] [)] [{] [double] [id z] [;] [id z] [=] [0.01] [;] [return] [id z] [;] **[}]** [int] [id main] [(] [void] [)] [{] [int] [id n1] [;] [double] [id z] [;] [id n1] [=] [25] [;] [id z] [=] [id f1] [(] [id n1] [)] [;]

Step 2: Symbol Table generation:

Sample source program:

```
// A program fragment
float x1 = 3.125;
/* Definition of the
function f1 */
double  f1(int x)
{
  double z;
  z = 0.01;
  return z;
}
//* Beginning of 'main'
int main(void)
{
 int n1;  double z;
 n1=25;  z=f1(n1);
```

Symbol Table:

Sl. No.	*Name*	*Id Type*	*Data Type*	*Scope*	*Value*
1	x1	var	float	global	3.125
2	f1	func	double	global	
3	x	var	int	f1	
4	z	var	double	f1	0.01
5	main	func	int	global	
6	n1	var	int	main	25
7	z	var	double	main	

Step 3: Your program should implement the following functions on symbol table.

1. *insert()*
2. *update()*
3. *delete()*
4. *search()*
5. *display()*

Step 4: Modified token stream for Syntax Analysis:

Sample source program:

```
// A program fragment
float x1 = 3.125;
/* Definition of the
function f1 */
double  f1(int x)
{
  double z;
  z = 0.01;
  return z;
}
//* Beginning of 'main'
int main(void)
{
int n1;  double z;
 n1=25;  z=f1(n1);
```

[float] [id 1] [=] [3.125] [;] [double] [id 2] [(] [int] [id 3] [)] [{] [double] [id 4] [;] [id 4] [=] [0.01] [;] [return] [id 4] [;] [}] [int] [id 5] [(] [void] [)] [{] [int] [id 6] [;] [double] [id 7] [;] [id 6] [=] [25] [;] [id 7] [=] [id 2] [(] [id 6] [)] [;]

Session 4: Detecting Simple Syntax Errors

I. OBJECTIVE:

Syntax errors are very common in source programs. The main purpose of this session is to write programs to detect and report simple syntax errors.

II. DEMONSTRATION OF USEFUL RESOURCES:

Sample programs will be demonstrated related to syntax error detection.

III. LAB EXERCISE:

Write programs to detect the following syntax errors.

1. Duplicate Identifier Declarations.
2. Unbalanced curly braces Detection.

IV. ASSIGNMENT #4:

Suppose, a given C source program has been scanned, filtered, lexically analyzed and tokenized as that were done in earlier sessions. In addition, line numbers have been assigned to the source code lines for generating proper error messages. As the first step to Syntax Analysis, we now perform detection of simple syntax errors like duplication of tokens except parentheses or braces, unbalanced braces or parentheses problem, unmatched 'else' problem, etc. Duplicate identifier declarations must also be detected with the help of the Symbol Table.

Sample Input: Sample code segment with numerous syntax errors.

```
/* A program fragment*/

float     x1 = 3.125;;;
/* Definition of function f1 */
double  f1(float a, int int x)
{if(x<x1)
double z;;
else z =          0.01;}}
else   return z;
}
/* Beginning of 'main'     */
int main(void)
{{{{
int n1;  double z;
n1=25;  z=f1(n1);}
```

Intermediate Output: Recognized tokens in the lines of code.

```
1
2
3 kw float id x1 = 3.125 ; ; ;
4
5 double id f1 ( float id a , int int id x )
6 { if ( id x < id x1 )
7 double id z ; ;
8 else id z = 0.01 ; } }
9 else return id z ;
10 }
11
12 int id main ( void )
13 { { { {
14 int id n1 ; double id z ;
15 id n1 = 25 ; id z = id f1 ( id n1 ) ; }
16
```

Sample Output: Types of detected errors

Duplicate token at line 3, Misplaced '}' at line 8, Unmatched 'else' at line 9, etc.

Guidelines:

1. Unbalanced braces or parentheses problem in an arithmetic or relational expression can be detected during tokenization in a simple way by counting the openings and closings. Stack can be used here as well.
2. Unmatched 'else' problem in its simplest form may also be detected by counting 'if's and 'else's: For every 'else' there must be an 'if' that occurs earlier.
3. Undeclared identifiers and duplicate identifier declarations in the same scope are detected during Symbol Table construction in a relatively easier way.
4. Duplicate ';' in 'for' construct of C demands additional checking. *for(;;){}*

Session 5: Use of CFGs for Parsing

I. OBJECTIVE:

We can think of using CFGs to parse various language constructs in the token streams freed from simple syntactic and semantic errors, as it is easier to describe the constructs with CFGs. But CFGs are hard to apply practically. In this session, we implement a simple recursive descent parser to parse a number of types of statements after exercising with simpler CFGs. We note that a recursive descent parser can be constructed from a CFG with reduced left recursion and ambiguity.

II. DEMONSTRATION OF USEFUL RESOURCES:

1. Observe the C code segments that implement the non-terminals of the following CFG.

$S \rightarrow b \mid AB$
$A \rightarrow a \mid aA$
$B \rightarrow b$

Language generated: {b, ab, aab, aaab, ...}

```
void S() {
   if (str[i] == 'b'){
      i++;
      f=1;
      return;
   }
   else {
      A();
      if (f) {
         B();
         return; }
   }}
```

```
void A() {
   if (str[i] == 'a') {
      i++;
      f=1;
   }
   else {
      f=0;
      return;
   }
   if (i<l-1)
      A();
}
```

```
void B() {
   if (str[i] == 'b') {
         i++;
         f=1;
         return;
   }
   else {
      f=0;
     return;}
}
```

** Find if there is any logical error in the sample code shown above.

2. A CFG to describe the syntax of simple arithmetic expressions may look like the one that follows:

<Exp>→<Term> + <Term> | <Term> - <Term> | <Term>

<Term>→<Factor> * <Factor> | <Factor> / <Factor> | <Factor>

<Factor>→(<Exp>) | ID | NUM

ID → a|b|c|d|e

NUM→ 0|1|2|…|9

Non-terminal symbols:

<Exp>, <Term>, <Factor>

Terminal symbols:

+, -, *, /, (,), a, b, c, d,e, 0, 1, 2, 3, ..., 9

Start symbol:

<Exp>

III. LAB EXERCISE:

1. Implement the following CFG in the way shown above.

$A \rightarrow aXd$
$X \rightarrow bbX$
$X \rightarrow bcX$
$X \rightarrow \varepsilon$

2. Implement the CFG shown above for generating simple arithmetic expressions.

IV. ASSIGNMENT #5:

Implement the following grammar in C.

<stat>→<asgn_stat> | <dscn_stat> | <loop_stat>

<asgn_stat>→id = <expn>

<expn>→<**smpl_expn>** <extn>

<extn>→<relop> <**smpl_expn>** | ε

<dcsn_stat>→ if (<expn>) <stat> <extn1>

<extn1>→ else <stat> | ε

<loop_stat>→while (<expn>) <stat> | for (<asgn_stat> ; <expn> ; <asgn_stat>) <stat>

<relop>→ == | != | <= | >= | > | <

Note: <**smpl_expn**> can be implemented using the materials demonstrated in this session.

Session 6: Predictive Parsing

I. OBJECTIVES:

Manual implementation of LL(1) and LR(1) parsing algorithms.

II. DEMONSTRATION OF USEFUL RESOURCES:

1. Computation of the FIRST and FOLLOW functions as described below.

❖ To Compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

a. If X is a terminal, then FIRST(X) is {X}.

b. If X is a non-terminal and $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place b in FIRST(X) if for some i, b is in FIRST(Y_i), and ε is in all of FIRST(Y_1), …, FIRST(Y_{i-1});
that is, $Y_1, \ldots, Y_{i-1}$ derives ε.

c. If ε is in FIRST(Y_j) for all j = 1, 2, …, k then add ε to FIRST(X).

Sample input and corresponding output:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

FIRST(E) = {(, id}
FIRST(T) = {(, id}
FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FIRST(F) = {(, id}

❖ To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

i. Place $ in FOLLOW(S), where S is the start symbol and $ is the right end-marker of an input.

ii. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B).

iii. If there is a production $A \rightarrow \alpha B$, then everything in FOLLOW(A) is in FOLLOW(B).

iv. If there is a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Sample input and corresponding output:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

FOLLOW(E) = {$,)}
FOLLOW(T) = {+, $,)}
FOLLOW(E') = {), $}
FOLLOW(T') = {+,),$}
FOLLOW(F) = {*, +, $,)}

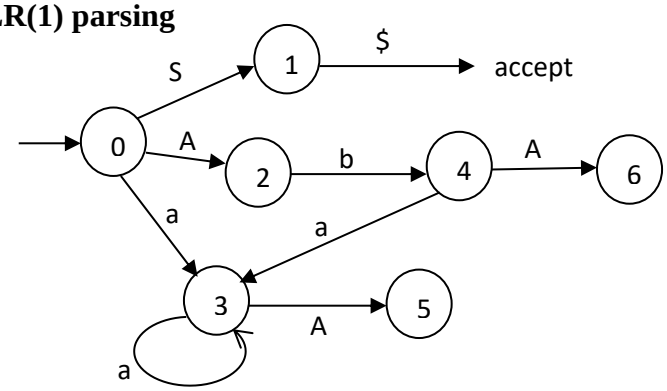
Table for LL(1) non-recursive predictive parsing with the given grammar:

Non-terminal	Input symbols					
	id	+	*	(	)	$
E	E → TE′			E → TE′		
E′		E′→+TE′			E′→ε	E′→ε
T	T → FT′			T → FT′		
T′		T′→ε	T′→ *F T′		T′→ε	T′→ε
F	F →id			F → (E)		

III. An example of construction of tools for LR(1) parsing

1. S →AbA
2. A →aA
3. A → a

S' → S
S →AbA
A→aA
A → a



State	ACTION			GOTO	
	a	b	$	S	A
0	s3			1	2
1			acc		
2		s4			
3	s3	r3	r3		5
4	s3				6
5		r2	r2		
6			r1		

FOLLOW(S) = {$}

FOLLOW(A) = {b. $}

FIRST(S) = FIRST(A) ={a}

If A →α• is in I_i, then set ACTION(i, a) to "Reduce by A →α" for all a in FOLLOW(A).

I_3 contains A→a•; I_5contains A →aA•; I_6contains S →AbA•.

IV. LAB EXERCISE:

Perform the tasks1, 2, and 3 of the Assignment #6 which is described below.

V. ASSIGNMENT #6:

Suppose, you are given the following grammar and the input string ***abcd***.

$$S \rightarrow aXd$$
$$X \rightarrow YZ$$
$$Y \rightarrow b$$
$$Y \rightarrow \varepsilon$$
$$Z \rightarrow cX$$
$$Z \rightarrow \varepsilon$$

- You are required to perform the following tasks manually:
  1. Find the FIRST and FOLLOW sets of each of the non-terminals.
  2. Construct the predictive parsing table for LL(1) method.
  3. Demonstrate the moves of the LL(1) parser on the given input.
  4. Construct the LR(0) automaton for the grammar.
  5. Construct the parsing table for LR(1) parsing with the grammar.
  6. Demonstrate the moves of the LR(1) parser on the given input.

Session 7: Intermediate Code Generation and Machine Code Generation

I. OBJECTIVES:

Writing a program to implement the intermediate code generation and the machine code generation

II. DEMONSTRATION OF USEFUL RESOURCES:

Sample programs will be demonstrated related to this session.

III. LAB EXERCISE:

Write a program which can generate 3 address code for a given expression.

Sample Input: *w = a-b*c/d+e-f*

Sample Output:

The 3-Address Code:	Expression
Z = c/d	w=a-b*Z+e-f
Y = b*Z	w=a-Y+e-f
X = Y+e	w=a-X-f
W = a-X	w=W-f
V = W-f	w=V
w = V	

IV. ASSIGNMENT #7:

Write a program to generate machine code form a 3-Address code stored in a file.

Sample Input:

```
X=a-b
Y=a+c
Z=d+b
C=a-d
```

Sample Output:

```
-----------------------------------------------------------
      Statement          Target Code
-----------------------------------------------------------
       X=a-b              MOV R0,a
                          SUB R0,b
       Y=a+c              MOV R1,a
                          ADD R1,c
      Z=d+b               MOV R2,d
                           ADD R2,b
       C=a-d              MOV R3,a
                          SUB R3,d
```

MID TERM EXAMINATION

There will be a written mid-semester examination on the materials covered in the sessions conducted during the first half of the semester.

TERM FINAL EXAMINATION

There will be a written end-of-semester examination on the materials covered in the sessions conducted during the second half of the semester.

END