CONTENTS

Chapters and Topics **page**

CHAPTER 17 ***(DAY 3 Afer noon)***
RECURSION
RECURSIVE DEFINITION 216
RECURSION vs. ITERATION 217
FACTORIAL 218
OTHER RECURSIVE FUNCTIONS (Reading) 219
FIBONACCI 219
BINARY SEARCH 221
TOWERS OF HANOI 224
RECURSIVE APPROACH 225
REMOVING RECURSION (Reading) 225

CHAPTER – 17

RECURSION

OTHER RECURSIVE FUNCTIONS

recursive function for multiplying two numbers:

```
int mult (int a, int b)
{
        return (b == 1  ?  a  :  mult (a, b – 1) + a);
}       /* end of mult  */
```

recursive function for exponent:

```
long power (int m, int n)
{
        return (n == 1  ?  (long) m :  (long) m * power (m, n – 1) );
}       /* end of power */
```

REMOVING RECURSION

Every recursive subroutine can be transformed into a non-recursive one by a series of simple steps. The necessary steps are described in many good text books on Data Structures, e.g., [2]. The transformation assumes that our programming language supports gotos. In case it doesn't (as it is usually the case today), we can transform it to some pseudo-language and then simply replace the unconditional jumps with iterative constructs, e.g., while loops. The following rules assume that the labels 1, 2,..., k are not used in the recursive subroutine. Moreover, k is one more than the number recursive calls in a given subroutine.

1. At the beginning of the subroutine, code is inserted which declares stacks associated with each formal parameter, each local variable, and the return address for each recursive call. Initially all stacks are empty.

2. The label 1 is attached to the first executable program statement.
3. If the subroutine is a function, i.e., returns some value, then we must replace all return statements with assignment statements, i.e., we introduce a fresh variable, say z, which has the same type as that of the function, and replace each return e statement with a z=e statement.

   Now, each recursive call is replaced by a set of instructions which do the following:

4. Store the values of all parameters and local variables in their respective stacks. The stack pointer is the same for all stacks.
5. Create the i-the label, **i**, and store **i** in the address stack. The value **i** of this label will be used as the return address. This label is placed in the subroutine as described in rule 8.
6. Evaluate the arguments of this call (they may be expressions) and assign these values to the appropriate formal parameters.
7. Insert an unconditional branch to the beginning of the subroutine.
8. If this is a procedure, add the label created above to the statement immediately following the unconditional branch. If this is a function then follow the unconditional branch by code to use the value of the variable z in the same way a return statement was handled earlier. The first statement of this code is given the label that was created above.

   These steps are sufficient to remove all recursive calls from a subroutine. Finally, we need to append code just after the last executable statement to do the following:

9. If the recursion stacks are empty, then return the value of z, i.e., return z, in case this is a function, or else simply return.
10. If the stacks are not empty, then restore the value of all parameters and of all local variables. These are at the top of the each

stack. Use the return label from the corresponding stack and execute a branch to this label. This can be done using a switch statement.