Code No.:

QUESTION BANK.

UNIT-3

1). Explain the translation to produce three address code for boolean expressions.

2) obtain a Quadruple, Triple, Syntax tree and indirect triple for the following expression.

$(a-b) * (c-d) - (a+b)$

3) Explain:
   i) Syntactic errors
   ii) Semantic errors.

4) Describe the different data structures used for maintaining symbol table.

5) Explain error recovery strategies

6) Explain backpatching.

7) What is an Intermediate code? why is it necessary in the design of the compiler?

8) Explain with the help of an example the most closely nested rule for accessing non local names.

9) Translate the following code statements into:
   i) Quadruples
   ii) Triples
   iii) Indirect triple.

```
while (a<c and b>d)
  { c=c+1,
    d=d+1,
  }.
```

10) write a short note on the errors that can occur at different phases of compiler and how can they be resolved.

11) write the translation scheme for flow of control statements.

12) what is a symbol table? Explain the contents of symbol table and data structures used to create symbol table.

13) Explain with the help of a diagram, the structure of activation record.

14) Explain static scope and dynamic scope.

15) Write the translation scheme for Assignment statements.

16) Provide an algorithm to partition 3-address code into basic block. Indicate the basic blocks and draw flow graph for the following source code:

```
i = 0;
val = 0;
while (i++<10)
    {
        i = 2*i;
        val += i;
    }.
```

17) Define the following:

a) Annotated parse tree.

b) Dependency graph.

18) Differentiate between static and Heap allocation strategies.

19) What do you mean by Backpatching and how it is been carried out?

Code No.:

QUESTION BANK

UNIT - 4

1) Explain the following:
   i) Register descriptor
   ii) Address descriptor.
2) Describe the different issues in the design of code generator.
3) With the help of an example, explain the code generation algorithm.
4) Construct DAG for the following:
   $(a*b) + (c-d) * (a*b) + b$.
5) Consider

```
int a[10]
Sum = 0;
for (i=0; i<=10; i++)
  sum = sum + a[i];
```

Construct basic blocks and flow graphs.

6) Explain the following techniques of code optimization with examples.
   i) Induction variable.
   ii) Dead code elimination.
7) Write a note on next use information.
8) Explain the rules used for the construction of the basic blocks.
9) Explain the significance of the construction of the DAG w.r.t the following example.
   B[i] = C[i] * D[i] / E
10) Consider the following statements given below.
    x = a-b
    y = a-d
    z = x*y.
    i) Write the assembly code for the above statements
    ii) Write the contents of address and register descriptor for the above statement.

iii) Calculate the total cost of executing the above statement.

11) Explain the loop unrolling technique with an example.

12) Explain the characteristic of peephole optimization.

13) obtain the assembly code generated after compiling the following statements. Show the content of address and register descriptor.

```
T1 = a + b
T2 = t1 + c.
```

14) Construct DAG for the following code:

```
t1 = a + b
x = t1
t2 = a - b
y = t2
z = x + y.
```

15). obtain assembly language code generated after compiling the following statements. Show the contents of address descriptor and register descriptor.

```
t = a - b
v = t + u.
```

16). Write a short note on Peephole optimization.

17) Explain the following code optimization techniques:

a) Copy Propagation.
b) Common subexpression elimination
c) Dead code elimination.

18) Describe the local transformations that can be applied to basic blocks.

19) Compute the cost of the following.

```
MOV *R1, *R0
ADD *R2, *R0.
```

20). Consider the following statements:

```
for(int i=0, i<10, i++)
 { a = a+b[i]; }
```

i) Construct the basic block for the code.

ii) Optimise the code using different optimisation techniques.

21) Construct DAG for the following code.

```
x = a+b
y = a-b
w = a+c
z = x*y.
```

What are the applications of DAG?

22) Write the 3-address code for the fragment given below. Convert it to basic blocks and draw flow graph. Further optimize the code.

```
count = 0,
result = 0,
while (count++<20)
   {
      inc = 2*count;
      result += inc;
   }.
```