**14IT502**

**Hall Ticket Number:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

**III/IV B.Tech (Regular / Supplementary) DEGREE EXAMINATION**

**November, 2019** **Information Technology**

**Fifth Semester** **Compiler Design**

**Time:** Three Hours **Maximum:** 60 Marks

*Answer Question No.1 compulsorily.* (1X12 = 12 Marks)

*Answer ONE question from each unit.* (4X12=48 Marks)

1 Answer all questions (1X12=12Marks)

a) Define assembler.
b) Define symbol table.
c) Define lexeme.
d) List the possible actions can make in shift-reduce parsing.
e) Define reduce/reduce conflict.
f) Define syntax-directed definition.
g) Define control stack.
h) Define activation record.
i) Define call by reference.
j) Define register allocation.
k) Define basic block
l) Define flow graph.

**UNIT I**

2 a) Illustrate phases of compiler with an assignment statement a=(b+c)*(b+c)*2 8M

b) Find whether the following grammar is LL(1) or not 4M

S-> abSa|aaAb
A->baAb|b

**(OR)**

3 a) Find the predictive parser for the following grammar and parse the sentence id+id*id 8M

E→ E+T|T
T→ T*F|F
F→ (E)| id

b) Differentiate the lexical analysis with parsing. 4M

**UNIT II**

4 a) Construct the SLR parsing table for the grammar 8M

S->(L)|a
L->L,S|S

b) Construct Syntax tree for b + 5 – a 4M

**(OR)**

5 a) Construct the LALR parsing table for the grammar 8M

S->L=R|R
L->*R|id
R->L

b) Illustrate the construction of input/output translator with Yacc. 4M

**UNIT III**

6 a) Illustrate stack and heap storage allocation strategies for strings and records. 8M

b) Demonstrate the representing of scope information. 4M

**(OR)**

7 a) Explain register assignment and allocation with an example. 4M

b) Demonstrate different data structures to symbol tables 8M

## UNIT IV

| | | | |
|---|---|---|---|
| 8 | a) | Describe simple target machine model. | 6M |
| | b) | Demonstrate determining the liveness and next-use information for each statement in a basic block. | 6M |

**(OR)**

| | | | |
|---|---|---|---|
| 9 | a) | Illustrate back patching. | 4M |
| | b) | Illustrate the issues in design of code generator | 8M |

**III/IV B.Tech (Regular / Supplementary) DEGREE EXAMINATION**

**November, 2019** **Information Technology**

**Fifth Semester** **Compiler Design**

**Time:** Three Hours **Maximum:** 60 Marks

## *Scheme of Evaluation & Answers*

**1.** ***Answer all questions*** (1 x 12 = 12 Marks)

**a)** **Define assembler.**

**Ans** An assembler is a program that converts assembly language into machine code.

**b)** **Define symbol table.**

**Ans** **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. **Symbol table** is used by both the analysis and the synthesis parts of a compiler.

**c)** **Define lexeme.**

**Ans** A **lexeme** is a sequence of characters in the source program that matches the pattern for a **token** and is identified by the lexical analyzer as an instance of that **token**

**d)** **List the possible actions can make in shift-reduce parsing.**

**Ans** 1.Shift

2.Reduce

3.Accept

4.Error

**e)** **Define reduce/reduce conflict.**

**Ans** A **reduce/reduce conflict** occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.

**f)** **Define syntax-directed definition.**

**Ans** A **syntax directed definition** is a context-free grammar in which. each grammar symbol X is associated with two finite sets of values: the synthesized attributes of X and the inherited attributes of X, each production A is associated with a finite set of expressions of the form.

**g)** **Define control stack.**

**Ans** **Control stack** or runtime **stack** is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed.

**h)** **Define activation record.**

**Ans** Information needed by a single execution of a procedure is managed using contiguous block of storage called an activation record.

**i) Define call by reference.**

**Ans** The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the **call**.

**j) Define register allocation.**

**Ans** Register allocation refers to the practice of assigning variables to registers as well as handling transfer of data into and out of registers.

**k) Define basic block.**

**Ans** A **basic block** is a sequence of consecutive statements in which **flow** of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

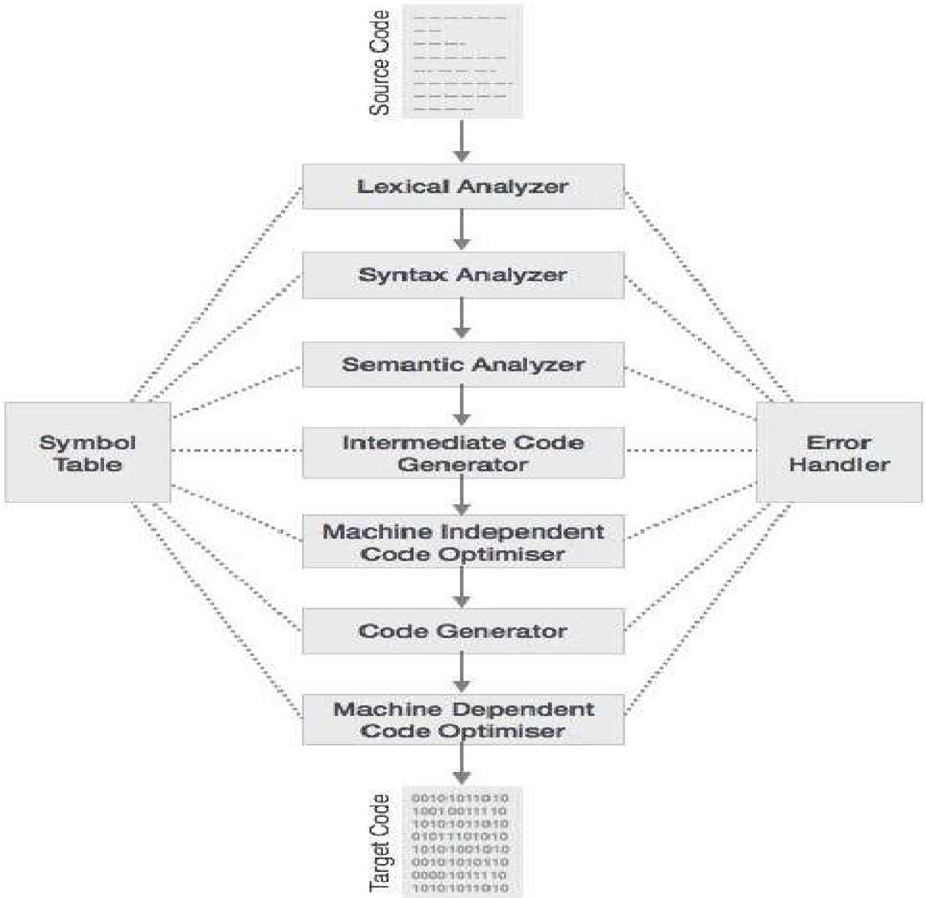
**l) Define flow graph.**

**Ans** **Flow graph** is a directed **graph**. It contains the **flow** of control information for the set of basic block. A control **flow graph** is used to depict that how the program control is being parsed among the blocks.

## UNIT-I

**2 a) Illustrate phases of compiler with an assignment statement a=(b+c)*(b+c)*2 8M**

**Ans:**




**→2 Marks**

**Lexical Analysis:**

LA or Scanner reads the source program one character at a time, separates the source program into a sequence of atomic units called **tokens.** The usual tokens are keywords such as WHILE, FOR, DO or IF, identifiers such as X or NUM, operator symbols such as <,<=,+,>,>= and punctuation symbols such as parentheses or commas. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase.

**Syntax Analysis:**

The second phase is called Syntax analysis or parser. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

**Semantic Analysis:**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

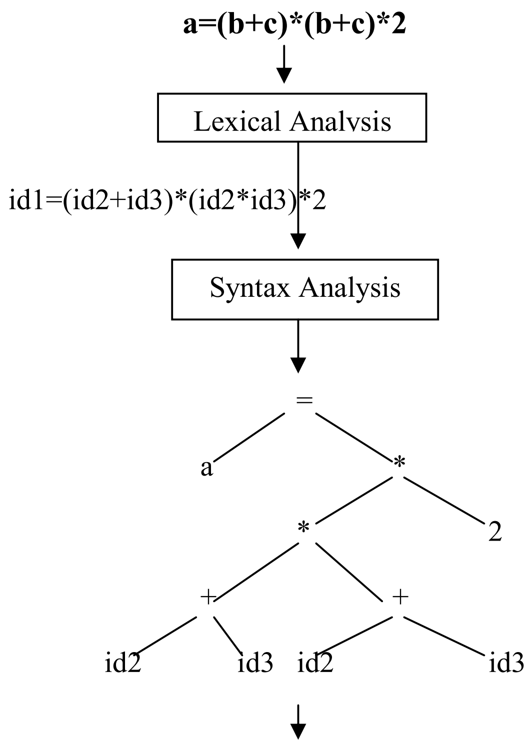
**Intermediate Code Generations:**

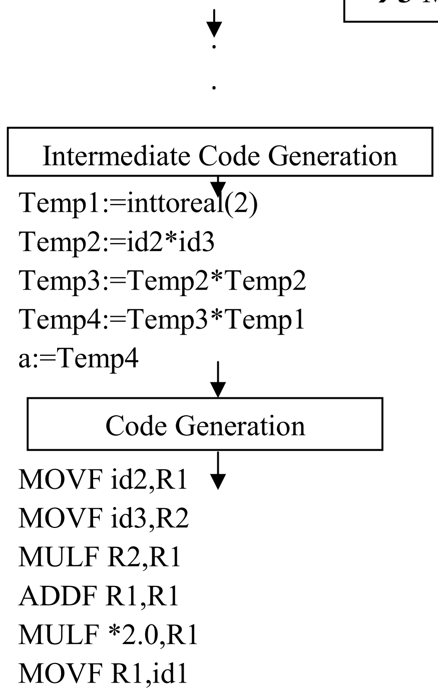
After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code. This phase bridges the analysis and synthesis phases of translation.

**Code Generation:**

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

**→3 Marks**




**→3 Marks**

**2 b) Find whether the following grammar is LL(1) or not** **4M**

**S-> abSa|aaAb**

**A->baAb|b**

**Ans:** The given grammar contains left factoring, so we have to eliminate left factoring, then grammar will be $S \rightarrow aS^1$ $S^1 \rightarrow bSa \mid aAb$ $A \rightarrow bA^1$ $A^1 \rightarrow aAb \mid €$

Find the FIRST & FOLLOW of variable **→2 Marks**

FIRST(S) = { a } FIRST($S^1$) = { b,a } FIRST(A) = { b } FIRST($A^1$) = { a, € }

FOLLOW(S) = { a,$ } FOLLOW($S^1$) = { a,$ } FOLLOW(A) = { b } FOLLOW($A^1$) ={b }

Parse table:

| | a | b | $ |
|---|---|---|---|
| S | $S \rightarrow aS^1$ | | |
| $S^1$ | $S^1 \rightarrow aAb$ | $S^1 \rightarrow bSa$ | |
| A | | $A \rightarrow bA^1$ | |
| $A^1$ | $A^1 \rightarrow aAb$ | $A^1 \rightarrow €$ | |

**→2 Marks**

The parse table contains uniquely defined entries, Hence the given grammar is LL(1) grammar.

**(OR)**

**3 a) Find the predictive parser for the following grammar and parse the sentence id+id*id** **8M**

**E→ E+T|T**

**T→ T*F|F**

**F→ (E)| id**

**Ans:** First eliminate the left recursion for E as

E → TE’

E’ → +TE’ | ε

Then eliminate for T as

T → FT’

T’→ *FT’ | ε

Thus the obtained grammar after eliminating left recursion is

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id **→2 Marks**

FIRST and FOLLOW set of variables

**First( ) :**

FIRST(E) = { ( , id}

FIRST(E’) ={+ , ε }

FIRST(T) = { ( , id}

FIRST(T’) = {*, ε }

FIRST(F) = { ( , id }

**Follow( ):**

FOLLOW(E) = { $, ) }

FOLLOW(E') = { $, ) }

FOLLOW(T) = { +, $, ) }

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

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

**→2 Marks**

**Predictive parsing table :**

| NON-TERMINAL | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | E' → +TE' | | | E' → ε | E'→ ε |
| T | T → FT' | | | T → FT' | | |
| T' | | T'→ ε | T'→ *FT' | | T' → ε | T' → ε |
| F | F → id | | | F → (E) | | |

Parser moves for the input string id+id*id$

**→2 Marks**

**Stack implementation:**

| stack | Input | Output |
|---|---|---|
| $E | id+id*id $ | |
| $E'T | id+id*id $ | E → TE' |
| $E'T'F | id+id*id $ | T → FT' |
| $E'T'id | id+id*id $ | F → id |
| $E'T' | +id*id $ | |
| $E' | +id*id $ | T' → ε |
| $E'T+ | +id*id $ | E' → +TE' |
| $E'T | id*id $ | |
| $E'T'F | id*id $ | T → FT' |
| $E'T'id | id*id $ | F → id |
| $E'T' | *id $ | |
| $E'T'F* | *id $ | T' → *FT' |
| $E'T'F | id $ | |
| $E'T'id | id $ | F → id |
| $E'T' | $ | |
| $E' | $ | T' → ε |
| $ | $ | E' → ε |

**→2 Marks**

**3 b) Differentiate the lexical analysis with parsing.** **4M**

**Ans:** The main difference between lexical analysis and syntax analysis is that lexical analysis reads the source code one character at a time and converts it into meaningful lexemes (tokens) whereas syntax analysis takes those tokens and produces a parse tree as an output.

| Lexical Analysis | Paising |
|---|---|
| First phase of the compilation process. | Second phase of the compilation process. |
| Process of converting a sequence of characters into a sequence of tokes. | Process of analyzing a string of symbols conforming to the rules of a formal grammar. |
| Lexing and tokenization are other names for lexical analysis. | Syntactic analysis and parsing are the other names for syntax analysis. |
| Reads the source program one character at a time and converts it into meaningful lexemes (tokens). | Takes tokens as input and generates a parse tree as output. |

**→4 Marks**

## UNIT-II

**4 a) Construct the SLR parsing table for the grammar** **8M**

**S→ (L) | a**

**L→ L , S | S**

**Ans:** The given grammar G is
1. S→ (L)
2. S→ a
3. L→ L , S
4. L→ S

Augmented grammar $G^1$ for the given grammar G is:

$S^1$→ S
S→ (L)
S→ a
L→ L , S
L→ S

**→1 Mark**

LR(0) items for the grammar are:

I0:
$S^1$→ . S
S→ . (L)
S→ . a

Goto( I0,S)
I1: $S^1$→ S.

Goto( I0, ( )
I2:S→ (.L)
L→ .L , S
L→ .S
S→ .(L)
S→ . a

Goto( I0,a)
I3: S→ a.

Goto( I2, L )
I4: S→ (L.)
L→ L. , S

Goto( I2, S )
I5: L→ S.

Goto( I2, ( ) = I2

Goto( I2, a ) = I3

Goto( I4, ) )
I6: S→ (L).

Goto( I4, **,**)
I7: L→ L , S.

Goto( I7,S)
I8: L→ L , S.

Find FOLLOW of variables:
FOLLOW(S) = { ), **,** , $ }
FOLLOW(L) = { ), **,** }

**→4 Marks**

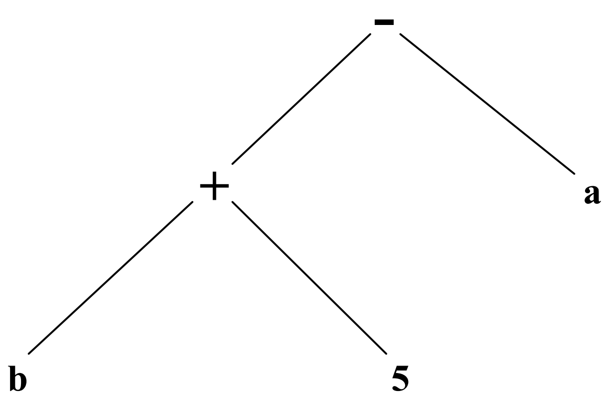
Construction of SLR parse table:

| State | ACTION | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | ( | ) | , | a | $ | S | L |
| 0 | S2 | | | S3 | | 1 | |
| 1 | | | | | Accept | | |
| 2 | S2 | | | S3 | | 5 | 4 |
| 3 | | r2 | r2 | | r2 | | |
| 4 | | S6 | S7 | | | | |
| 5 | | r4 | r4 | | | | |
| 6 | | r1 | r1 | | r1 | | |
| 7 | | | | | | 8 | |
| 8 | | r3 | r3 | | | | |

**→3 Marks**

**4 b) Construct Syntax tree for b + 5 - a** **4M**

**Ans:** Syntax Tree or Abstract Syntax Tree is a condensed form of parse tree. In the **syntax tree**, interior nodes are operators and leaves are operands.




**→2 Marks**

Each node in a syntax tree for an (arithmetic) expression is a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes of the operands. The operator is often called the label of the node.

p1:= mkleaf( id, entryb);

p2:=mkleaf(num,5);

p3:=mknode('+',p1,p2);

p4:=mkleaf(id,entrya);

p5:=mknode('-',p3,p4);

**→2 Marks**

**(OR)**

**5 a) Construct LALR parsing table for the grammar S → L=R S → R L →*R L →id R → L 8M**

**Ans:** The given grammar is

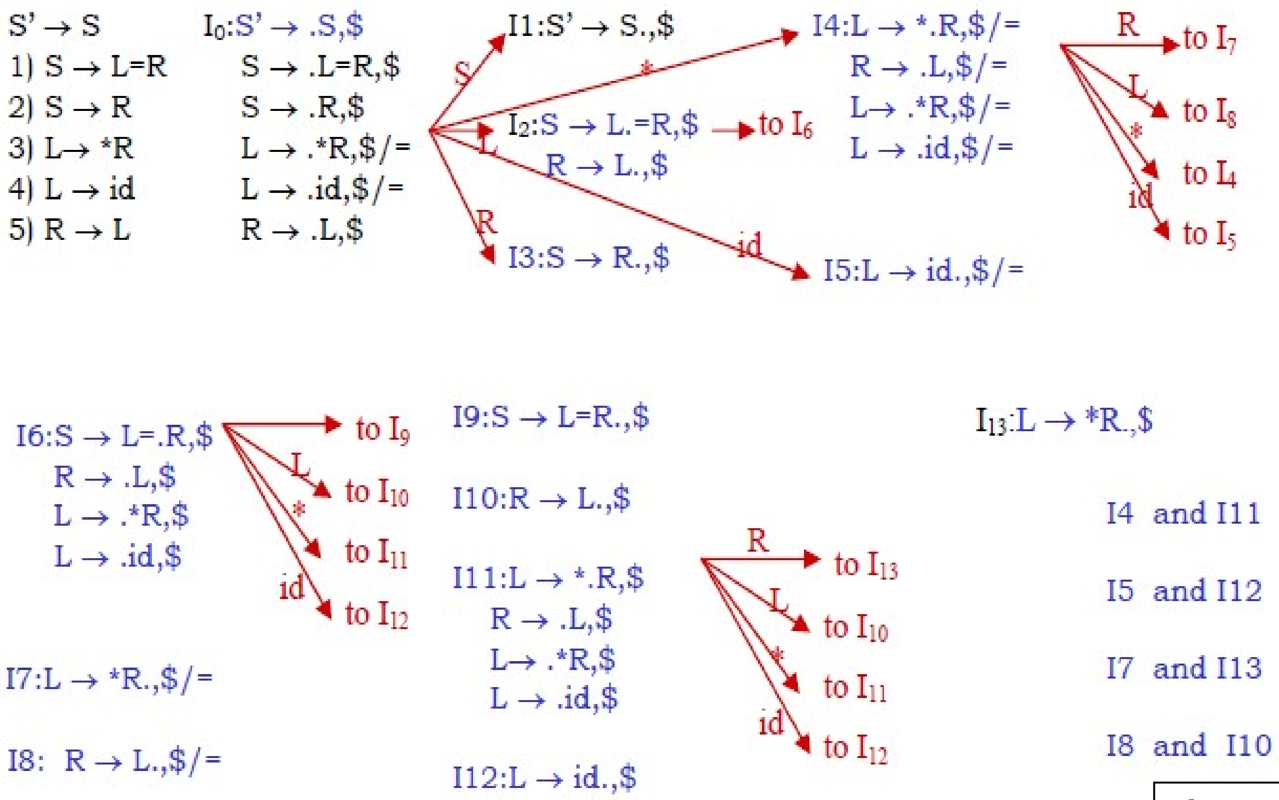
1) S→ L = R 2) S→ R 3)L→*R 4)L→id 5) R→L

Step-1: Convert given grammar into augmented grammar

$S^1 \rightarrow S \quad S \rightarrow L = R \quad S \rightarrow R \quad L \rightarrow *R \quad L \rightarrow id \quad R \rightarrow L$

**→1 Mark**

Step-2: Find LR(1) items




**→4 Marks**

Step 3: Construction of parse table:

| State | ACTION | | | | GOTO | | |
|---|---|---|---|---|---|---|---|
| | = | * | id | $ | S | L | R |
| 0 | | S411 | S512 | | 1 | 2 | 3 |
| 1 | | | | ACCEPT | | | |
| 2 | S6 | | | r5 | | | |
| 3 | | | | r2 | | | |
| 411 | | S411 | S512 | | | 810 | 713 |
| 512 | r4 | | | r4 | | | |
| 6 | | S11 | S12 | | | 810 | 9 |
| 713 | r3 | | | r3 | | | |
| 810 | r5 | | | r5 | | | |
| 9 | | | | r1 | | | |

**→3 Marks**

All blanks are error.

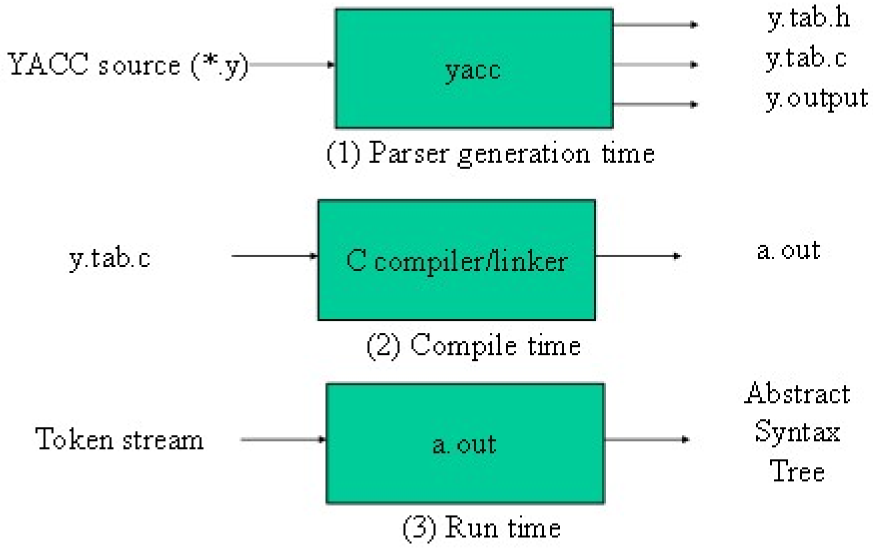
**5 b) Illustrate the construction of input/output translator with YACC.** **4M**

**Ans:** Yacc: Yet Another Compiler-Compiler

YACC is a tool which will produce a parser for a given grammar.

**→1 Mark**

YACC (Yet Another Compiler Compiler) is a program, designed to compile a LALR (1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar.




**→2 Marks**

**YACC file format:**

%{

*C declarations*

%}

*yacc declarations*

%%

*Grammar rules*

%%

*Additional C code*

– Comments enclosed in /* ... */ may appear in any of the sections.

**→1 Mark**

**UNIT-III**

**6 a) Illustrate stack and heap storage allocation strategies for strings and records. 8M**

**Ans: Stack Allocation**

- Stack allocation manages the runtime storage as a stack, i.e., control stack
- Activation records are pushed and popped as activation begins and end respectively
- Locals are always bound to fresh storage in each activation, because a new activation is onto a stack when a call is made
- Values of locals are deleted as activation ends
- The data structure can be created dynamically for stack allocation

**Limitations**

- Values of locals cannot be retained once activation ends
- The memory addressing can be done using pointers and indexed registers
- This type of allocation is slower than static allocation

**→4 Marks**

**Heap allocation**

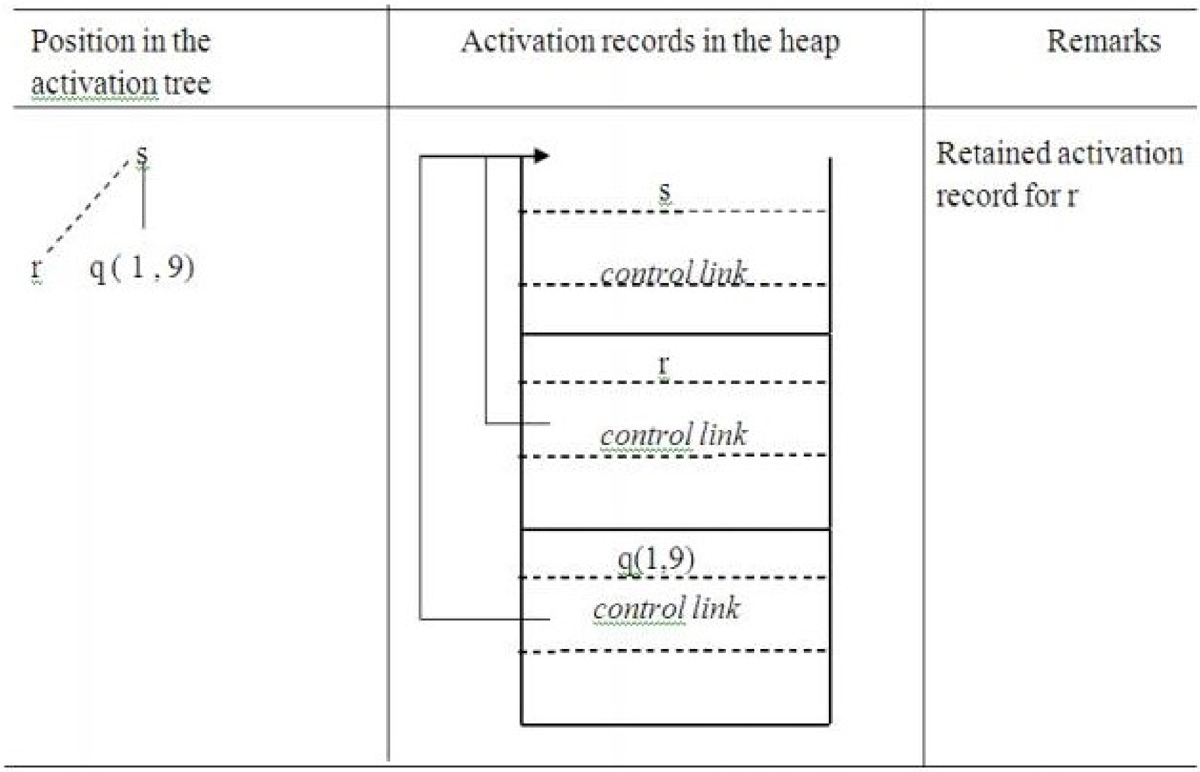
- Storage can be allocated and de allocated in any order
- If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation

- It is used for retaining of local variables
- The heap allocation allocates the continuous block of memory when required for storage of activation records. This allocated memory can be deallocated when activation ends
- Free space can be further reused by heap manager
- It supports for recursion and data structures can be created at runtime

**Limitation**

- Heap manages overhead.

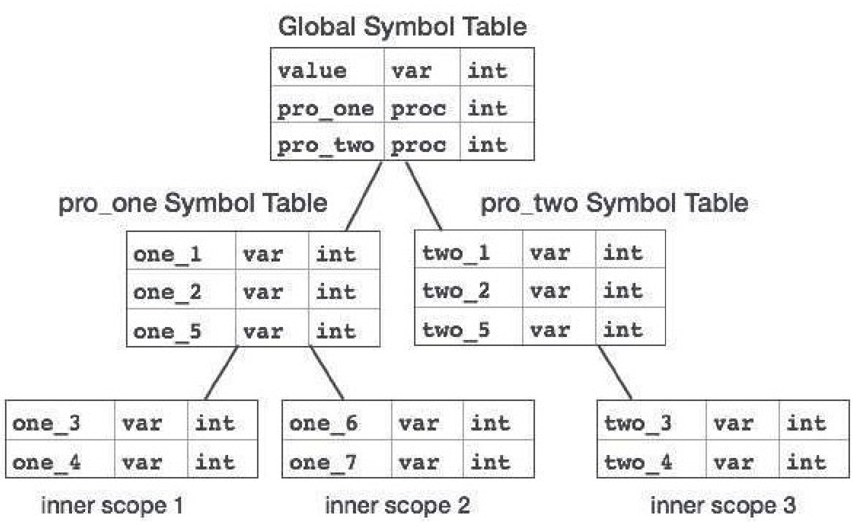
**→4 Marks**




**6 b) How to represent scope information in symbol table? 4M**

**Ans:** A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:




**→2 Marks**

The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- First a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- Either the name is found or global symbol table has been searched for the name.

**→2 Marks**

**(OR)**

**7 a) Explain register assignment and allocation with an example. 4M**

**Ans: Register Assignment-**The specific register that a variable will reside in is picked.

Certain machine requires even-odd *register pairs* for some operands and results.
For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair
y – divisor
even register holds the remainder
odd register holds the quotient

**→2 Marks**

**Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers. A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or **registers**. Code generator decides what values to keep in the **registers**.

**→2 Marks**

**7 b) Demonstrate different data structures to symbol tables. 8M**

**Ans:** There are a number of data structures that can be used to implement a symbol table:

• An Ordered List Symbols are stored in a linked list, sorted by the symbol's name. This is simple, but may be a bit too slow if many identifiers appear in a scope.

• A Binary Search Tree Lookup is much faster than in linked lists, but rebalancing may be needed. (Entering identifiers in sorted order turns a search tree into a linked list.)

• Hash Tables the most popular choice.

**→2 Marks**

**The Linear List**

- A linear list of records is the easiest way to implement a symbol table.
- The new names are added to the table in the order mat they arrive.
- Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether or not the name is already present in the table.

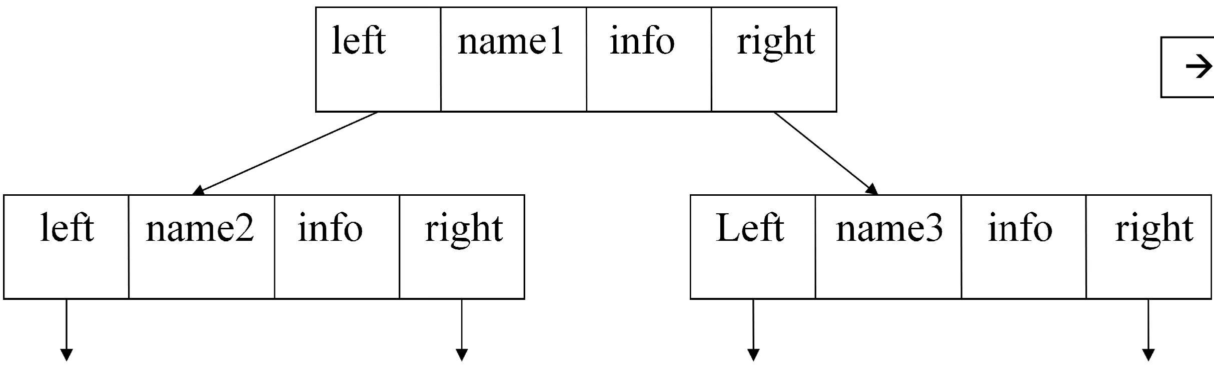
- . If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer
- To retrieve the information about the name, the table is searched sequentially, starting from the first record in the table.
- The average number of comparisons, *p,* required for search are $p$ = (n + l)/2 for successful search and $p = n$ for an unsuccessful search, where $n$ is the number of records in symbol table.
- The advantage of this organization is that it takes less space, and additions to the table are simple.
- This method's disadvantage is that it has a higher accessing time.

**→2 Marks**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | info | Name | info | Name | info | Name | info | Name | info |

**Search Trees**

- A search tree is a more efficient approach to symbol table organization.
- add two links, left and right, in each record, and these links point to the record in the search tree.
- Whenever a name is to be added, first the name is searched in the tree.
- If it does not exist, then a record for the new name is created and added at the proper position in the search tree.
- This organization has the property of alphabetical accessibility; that is, all the names accessible from name;. by following a left link, precede name, in alphabetical order.
- Similarly, all the name accessible from name, by following right link follow name, in alphabetical order
- The expected time needed to enter $n$ names and to make $m$ queries is proportional to *(m + n)* Iog2n; so for greater numbers of records (higher n) this method has advantages over linear list organization.



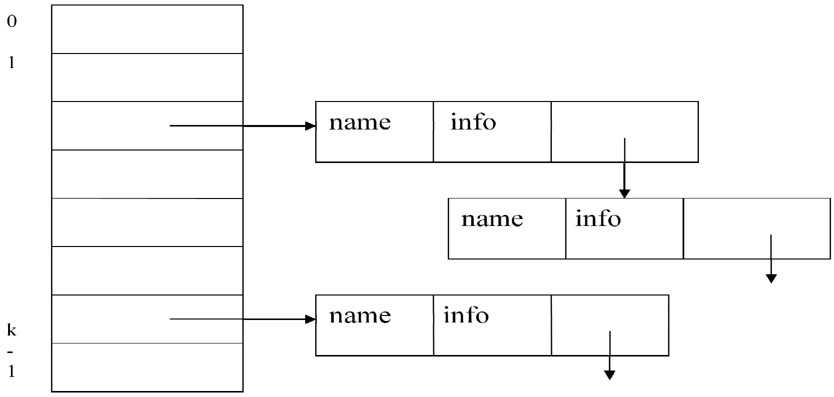

**→2 Marks**

**Hash Tables**

- A hash table is-a table of pointers numbered from zero to *k*~\ that point to the symbol table and a record within the symbol table.
- To enter a name into symbol table, we find out the hash value of the name by applying a suitable hash function.

- The hash function maps the name into an integer between ze>o and *k-lt* and using this value as an index in the hash table, we search the list of the symbol table records that are built on that hash index.

If the name is not present in that list, we create a record for name and insert it at the head of the list.




**→2 Marks**

Hash Table

## UNIT-IV

**8 a) Describe simple target machine model.** **6M**

**Ans:**

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n*general-purpose registers, $R_0, R_1, \ldots, R_{n-1}$.
- It has two-address instructions of the form:

  ***op source, destination*** *where, op is an op-code, and source and destination are data fields.*

  It has the following op-codes:

  MOV (move *source* to *destination*)

**→3 Marks**

The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

**Address modes with their assembly-language forms**

| MODE | FORM | *ADDRESS* | ADDED COST |
|---|---|---|---|
| *absolute* | M | M | 1 |
| *register* | R | R | 0 |
| *indexed* | *c*(R) | *c*+*contents*(R) | 1 |
| *indirect register* | *R | *contents* (R) | 0 |
| *indirect indexed* | **c*(R) | *contents*(*c*+ *contents*(R)) | 1 |
| *literal* | #*c* | *c* | 1 |

**→3 Marks**

**8 b) Demonstrate determining the liveness and next-use information for each statement in a basic block. 6M**

**Ans:**

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

**Input:** Basic block B of three-address statements

**Output:** At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

**Method:** We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

**→4 Marks**

**Symbol Table:**

| Names | Liveliness | Next-use |
|---|---|---|
| x | not live | no next-use |
| y | Live | i |
| z | Live | i |

**→2 Marks**

**(OR)**

**9 a) Illustrate back patching. 4M**

**Ans:** The problem in generating three address codes in a single pass is that we may not know the labels that control must go to at the time jump statements are generated. So to get around this problem a series of branching statements with the targets of the jumps temporarily left unspecified is generated.

**Back Patching** is putting the address instead of labels when the proper label is determined.

Back patching Algorithms perform **three types of operations**

1) **Makelist (i)** – creates a new list containing only i, an index into the array of quadruples and returns a pointer to the list it has made.

2) **Merge (i, j)** – concatenates the lists pointed to by i and j, and returns a pointer to the concatenated list.

3**) Backpatch (p, i)** – inserts i as the target label for each of the statements on the list pointed to by p.

**→4 Marks**

**9 b) Illustrate the issues in designing of code generator? 8M**

**Ans:** The following issue arises during the code generation phase:

1. **Input to code generator:**

   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's etc. Assume that they are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. **Target Program:**

   Target program is the output of the code generator. The output may be absolute machine language, reloadable machine language, assembly language.

   - Absolute machine language as an output has advantages that it can be placed in a fixed memory location and can be immediately executed.
   - Reloadable machine language as an output allows subprograms and subroutines to be compiled separately. Reloadable object modules can be linked together and loaded by linking loader.

   **→3 Marks**

   - Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.

3. **Memory Management –**

   Mapping the names in the source program to addresses of data objects is done by the front end and the code generator. A name in the three address statement refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

**4. Instruction selection –**

Selecting best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into latter code sequence as shown below:

```
P:=Q+R
S:=P+T
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

**→3 Marks**

**5. Register allocation issues –**

Use of registers makes the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers is subdivided into two sub problems:

1. During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.
2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

**→1 Mark**

**6. Evaluation Order:**

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results.

**→1 Mark**

**Scheme prepared by**

**Mr. G.Prasad, Asst.Prof.,**

**Department of I.T.**

**Signature of the HOD, IT DEPT.**

**Paper Evaluators:**

| S.No | Name of the College | Name of the Examiner | Signature |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |