

Hall Ticket Number:

--	--	--	--	--	--	--	--	--	--

III/IV B.Tech (Regular) DEGREE EXAMINATION**November, 2016****Fifth Semester****Time:** Three Hours*Answer Question No.1 compulsorily.**Answer ONE question from each unit.***1.** Answer all questions**Information Technology
COMPILER DESIGN****Maximum : 60 Marks**

(1X12 = 12 Marks)

(4X12=48 Marks)

(1X12=12 Marks)

- a Differentiate phase and pass of a compiler.
- b What are the issues to be considered in top down parsing?
- c What is the role of lexical analyzer?
- d Differentiate canonical LR and LALR parsing.
- e What is handle pruning?
- f Differentiate synthesized translation and inherited translation.
- g List out different storage allocation strategies.
- h Write the contents of a symbol table.
- i What are the issues of a source language?
- j Give the three address code for the statement “ $W = -X + Y * Z$ ”
- k Write the rules to identify basic blocks.
- l What is peephole optimization?

UNIT – I

- 2.a Draw a transition diagram for recognizing tokens identifier, constant, and relational operator like $<$, $<=$, $=$, $>$, $>=$. 6M
- 2.b What is LEX? Explain in detail LEX source program. 6M

(OR)

- 3.a What is left recursion? Eliminate left recursion of the following grammar
 $S \rightarrow (L) / a, L \rightarrow L, S / S$. 6M
- 3.b Write the rules to compute first and follow set of a given grammar. Calculate first and follow set of the given grammar $S \rightarrow A, A \rightarrow aB / Ad, B \rightarrow bBC / f, C \rightarrow g$. 6M

UNIT – II

- 4.a Show the following grammar $S \rightarrow AaAb / BbBa, A \rightarrow \epsilon, B \rightarrow \epsilon$ is not SLR (1). 8M
- 4.b Explain stack implementation of shift reduce parser. 4M

(OR)

- 5.a Construct an LALR parsing table for the following grammar
 $D \rightarrow L : T, L \rightarrow L, id / id, T \rightarrow integer$. 8M
- 5.b Discuss in detail construction of syntax trees. 4M

UNIT – III

6.a Consider the following program:

```
Void foo ()
{
    float a, b, c; /* level 0 declaration */
    .....
    .....
    {
        float a, b; /* level '1x' declaration */
        .....
        .....
    }
    {
        int d, e; /* level '1y' declaration */
        {
            int f; /* level '2' declaration */
            .....
            .....
        }
    }
}
```

8M

Show the stack position after execution of level 0, level 1x, level 1y and level 2.

6.b Discuss in detail Heap allocation strategy.

4M

(OR)

7.a Describe the representation of scope information in the symbol table. Consider the following program structure and give its symbol table organization:

6M

Program main

Var x, y : integer;

Procedure P

Var X, a : boolean;

Procedure Q

Var x, y, z : real;

7.b What is the significance of symbol table at runtime and compile time. Discuss it.

6M

UNIT – IV

8.a Write an SDT scheme for Boolean expressions.

6M

8.b Write quadruples, triples for the expression $(a + b) * (c + d) - (a + b + c)$

6M

(OR)

9.a Construct DAG for the following code: $a = a + b, e = a + d + e$

6M

9.b Write an algorithm for simple code generation.

6M

November, 2016**Fifth Semester****Time:** Three Hours**Information Technology****Compiler Design****Maximum:** 60 Marks***Scheme of Evaluation & Answers*****1. Answer all questions**

(1 x 12 = 12 Marks)

a) Differentiate phase and pass of a compiler?

Ans: phase is used to classify compilers according to the construction, while pass is used to classify compilers according to how they operate.

b) What are the issues to be considered in top down parsing?

Ans: Left recursion and left factoring

c) What is the role of lexical analyzer?

Ans: The LA is the first phase of a compiler. Its main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

d) Differentiate canonical LR and LALR parsing?

Ans: In CLR, we have to find LR (1) items and we identify final items and check for conflicts.

In LALR, we have to merge equal states with look ahead different and find conflicts.

If a grammar is not CLR grammar it definitely not LALR. CLR parser is more powerful than LALR.

e) What is handle pruning?

Ans: Right-most derivation in reverse order is called handle pruning or canonical reduction sequence.

f) Differentiate synthesized translation and inherited translation?

Ans: The value of a synthesized attribute at a node is computed from the values of the attributes at the children of that node in the parse tree.

The value of a inherited attribute at a node is computed from the values of the attributes at the siblings and parent of that node.

g) List out different storage allocation strategies?

Ans: Static storage allocation

Stack storage allocation

Heap storage allocation

h) write the contents of a symbol table?

Ans: • Name: a string.

• Attribute: Reserved word, Variable name, Type name, Procedure name, Constant name.

• Data type.

• Storage allocation, size.

• Scope information: where and when it can be used.

i) What are the issues of a source language?

Ans: Procedures

Activation trees

Control stacks

The scope of a declaration and binding of names

j) Give the three address code for the statement $W = -X + Y * Z$

Ans: $T1 = -X$
 $T2 = Y * Z$
 $T3 = T1 + T2$
 $W = T3$

l) Write the rules to identify basic blocks?

Ans: i) The first statement is a leader
 ii) Any statement that is the target of a conditional or unconditional goto is a leader.
 iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

l) Define peephole optimization?

Ans: A method for trying to improve the performance of the target program by examining a short sequence of target instructions and replacing these instructions by a shorter or faster sequence, whenever possible.

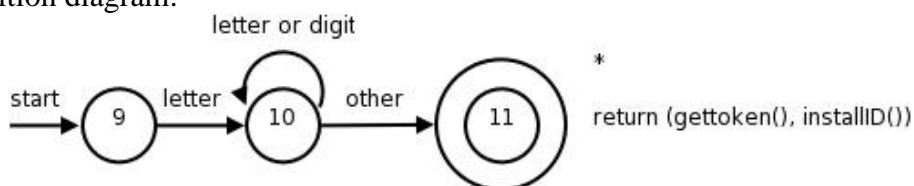
Note: Give 1 Mark for attempt

UNIT-I

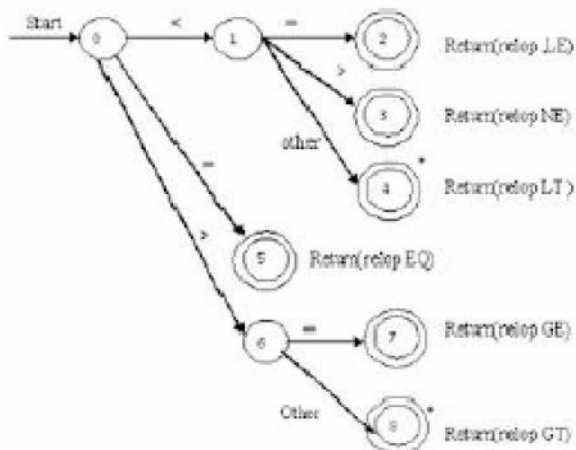
2 a) Draw a transition diagram for recognizing tokens identifier, constant, and relational operators like $<$, $<=$, $=$, $>$, $>=$? 6M

Ans:

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
 2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
 3. One state is designed the state ,or initial state ., it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used. As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.
 Identifier Transition diagram:

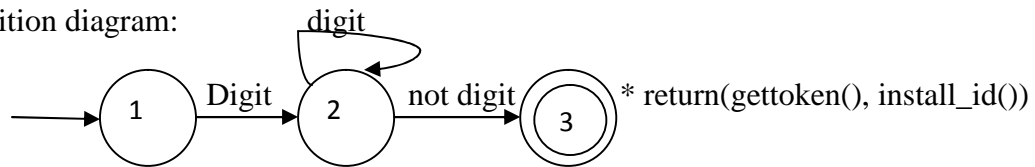


Relational operators Transition diagram:



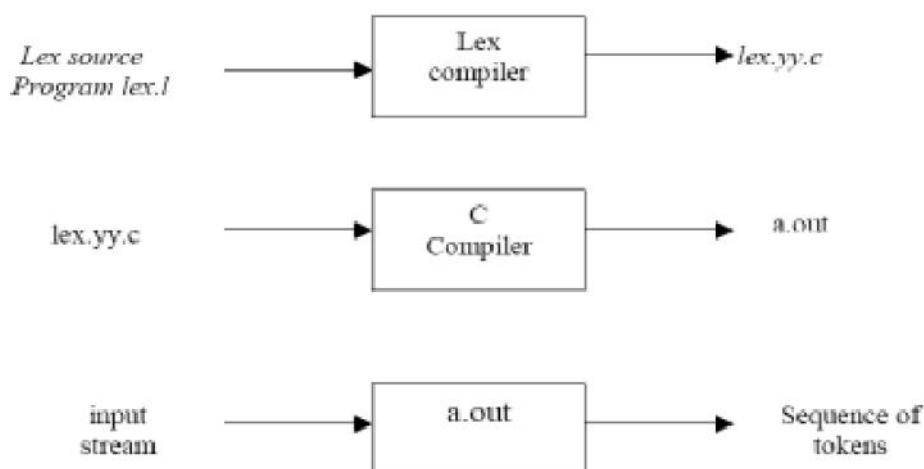
Identifier diagram- 2 Marks
Constant diagram- 1 Mark
Relational operators diagram- 3 Marks

Constant transition diagram:



2 b) What is LEX? Explain in detail LEX source program? 6M

Ans: We describe a particular tool, called Lex, that has been widely used to specify lexical analyzers for a variety of languages. We refer to the tool as the Lex compiler, and to its specification as the Lex language. First, a specification of a lexical analyzer is prepared by creating a program `lex.l` in the Lex language. Then, `lex.l` is run through the Lex compiler to produce a C program `lex.yy.c`. The program `lex.yy.c` consists of a tabular representation of a transition diagram constructed from the regular expressions of `lex.l`, together with a standard routine that uses the table to recognize lexemes. Finally, `lex.yy.c` is run through the C compiler to produce an object program `a.out`, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



LEX – 1 Mark

Explanation- 2 Marks

LEX program – 3 Marks

Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

1. The *declarations* section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. `#define PIE 3.14`), and regular definitions.

2. The *translation rules* of a Lex program are statements of the form:

p1 {action 1}

p2 {action 2}

p3 {action 3}

... ..

... ..

Pn {action n}

Where, each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions*. Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

LEX source program:

```

%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE
    IF, THEN, ELSE, ID, NUMBER, RELOP */
}%
/* regular definitions */
delim      [\t \n]
ws         [delim]+
letter     [A-Z a-z]
digit      [0-9]
Id         {letter} ({letter}|{digit})*
%%
{ws}      { /* no action and no return */ }
if        { return (IF) }
then      { return (THEN) }
{ id }    { yylval = install_id(); return ( ID ); }
.....
%%
install_id() {
.....
}
install_num() {
.....
}

```

(OR)

3 a) What is left recursion? Eliminate left recursion of the following grammar

$$S \rightarrow (L) / a, L \rightarrow L, S / S.$$
6M

Ans: A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A$ for some string . Top-down parsing methods cannot handle left-recursive grammars.

Hence, left recursion can be eliminated as follows:

If there is a production $A \rightarrow A | A' A'$ it can be replaced with a sequence of two productions

Without changing the set of strings derivable from A.

Elimination of left recursion for the given grammar is:

$$S \rightarrow (L) / a,$$

$$L \rightarrow L, S / S$$

S production has no left recursion

L production has left recursion, so eliminate it from L production.

$$L \rightarrow SL'$$

$$L' \rightarrow ,SL' / \epsilon$$

So, After elimination of left recursion the resultant grammar is

$$S \rightarrow (L) / a$$

$$L \rightarrow SL'$$

$$L' \rightarrow ,SL' / \epsilon$$

Left recursion – 2 Marks

Elimination of Left recursion-

4 Marks

3 b) Write the rules to compute first and follow set of given grammar. Calculate first and follow set of the given grammar $S \rightarrow A, A \rightarrow aB / Ad, B \rightarrow bBC / f, C \rightarrow g$. 6M

Ans: The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST
2. FOLLOW

Rules of FIRST & FOLLOW – 2 Marks
FIRST set – 2 Marks
FOLLOW set – 2 Marks

Rules for first():

1. If X is terminal, then FIRST(X) is {X}.
2. If X is a production, then add to FIRST(X).
3. If X is non-terminal and $X \rightarrow a$ is a production then add a to FIRST(X).
4. If X is non-terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1, \dots, Y_{i-1} \Rightarrow$. If is in FIRST(Y_j) for all $j=1, 2, \dots, k$, then add to FIRST(X).

Rules for follow():

1. If S is a start symbol, then FOLLOW(S) contains \$.
2. If there is a production $A \rightarrow B$, then everything in FIRST() except is placed in follow(B).
3. If there is a production $A \rightarrow B$, or a production $A \rightarrow B$ where FIRST() contains , then everything in FOLLOW(A) is in FOLLOW(B).

FIRST set of the given grammar

FIRST (S) = { a }
 FIRST (A) = { a }
 FIRST (B) = { b, f }
 FIRST (C) = { g }

FOLLOW set of the given grammar

FOLLOW (S) = { \$ }
 FOLLOW (A) = { d, \$ }
 FOLLOW (B) = { g, d, \$ }
 FOLLOW (C) = { g, d, \$ }

UNIT-II

4 a) Show the following grammar $S \rightarrow AaAb / BbBa, A \rightarrow \epsilon, B \rightarrow \epsilon$ is not SLR(1). 8M

Ans: The given grammar is

- 1) $S \rightarrow AaAb$
- 2) $S \rightarrow BbBa$
- 3) $A \rightarrow \epsilon$
- 4) $B \rightarrow \epsilon$

Augmented grammar – 1 Mark

Find LR(0) items – 3 Marks

SLR Table construction – 4 Marks

Step 1: Convert given grammar into augmented grammar

$S' \rightarrow S$
 $S \rightarrow AaAb$
 $S \rightarrow BbBa$
 $A \rightarrow \epsilon$
 $B \rightarrow \epsilon$

Step 2: Find LR (0) items

I0: $S' \rightarrow \cdot S$
 $S \rightarrow \cdot AaAb$
 $S \rightarrow \cdot BbBa$
 $A \rightarrow \cdot \epsilon$
 $B \rightarrow \cdot \epsilon$

GOTO (I0, S)
 I1: $S' \rightarrow S \cdot$

GOTO (I0, B)
 I3: $S \rightarrow B \cdot bBa$

GOTO (I2, a)
 I5: $S \rightarrow Aa \cdot Ab$
 $A \rightarrow \cdot \epsilon$

GOTO (I0, A)
 I2: $S \rightarrow A \cdot aAb$

GOTO (I0, ϵ)
 I4: $A \rightarrow \cdot \epsilon$
 $B \rightarrow \cdot \epsilon$

GOTO (I3, a)
 I6: $S \rightarrow Bb \cdot Ba$
 $B \rightarrow \cdot \epsilon$

GOTO (I5, A)
I7: $S \rightarrow AaA.b$

GOTO (I5, €)
I8: $A \rightarrow \epsilon$

GOTO (I6, B)
I9: $S \rightarrow BbB.a$

GOTO (I6, €)
I10: $B \rightarrow \epsilon$

GOTO (I7, b)
I11: $S \rightarrow AaAb.$

GOTO (I9, a)
I12: $S \rightarrow BbBa.$

Follow (S) = { \$ }
Follow (A) = { a,b }
Follow(B) = { a,b }

Step 3: Construction of an SLR parse table

State	ACTION				GOTO		
	a	b	€	\$	S	A	B
0			S4		1	2	3
1				Accept			
2	S5						
3		S6					
4	r3/r4	r3/r4					
5			S8			7	
6			S10				9
7		S11					
8	r3	r3					
9	S12						
10	r4	r4					
11				r1			
12				r1			

The table has multiply defined entries, so the given grammar is not an SLR(1).

4 b) Explain stack implementations of shift reduce parser?

4M

Ans:

Actions in shift reduce parser – 2 Marks

Stack example – 2 Marks

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
\$	id ₁ +id ₂ *id ₃ \$	shift
\$ id ₁	+id ₂ *id ₃ \$	reduce by E→id
\$ E	+id ₂ *id ₃ \$	shift
\$ E+	id ₂ *id ₃ \$	shift
\$ E+id ₂	*id ₃ \$	reduce by E→id
\$ E+E	*id ₃ \$	shift
\$ E+E*	id ₃ \$	shift
\$ E+E*id ₃	\$	reduce by E→id
\$ E+E*E	\$	reduce by E→ E *E
\$ E+E	\$	reduce by E→ E+E
\$ E	\$	accept

(OR)

5 a) Construct an LALR parsing table for the following grammar

 $D \rightarrow L:T, L \rightarrow L, id / id, T \rightarrow integer$

8M

Ans: The given grammar is

 $D \rightarrow L:T$ $L \rightarrow L, id$ $L \rightarrow id$ $T \rightarrow i$ $i \rightarrow integer$

Step-1: Convert given grammar into augmented grammar

 $D' \rightarrow D$ $D \rightarrow L:T$ $L \rightarrow L, id$ $L \rightarrow id$ $T \rightarrow i$

Step 2: Find LR (1) items

I0: $D' \rightarrow .D, \$$
 $D \rightarrow .L:T, \$$
 $L \rightarrow .L, id, :$
 $L \rightarrow .id, :$

GOTO (I0, D)
 I1: $D' \rightarrow D. , \$$

GOTO (I0, L)
 I3: $L \rightarrow id. , :$

GOTO (I0, L)
 I5: $L \rightarrow L.,id, :$

GOTO (I0, L)
 I2: $D \rightarrow L.:T, \$$
 $L \rightarrow L.,id, :$

GOTO (I2, :)
 I4: $D \rightarrow L.:T, \$$
 $T \rightarrow .i, \$$

GOTO (I4, T)
 I6: $D \rightarrow L:T., \$$

Augmented grammar – 1 Mark

Find LR(1) items – 3 Marks

LALR Table construction – 4 Marks

GOTO (I4, i)
I7: T → i, \$

GOTO (I5, id)
I2: L → L, id, :, :

Step-3: Construction of an LALR parsing table

State	ACTION					GOTO		
	:	,	i	id	\$	D	L	T
0				S3		1	2	
1					Accept			
2	S4	S5						
3	r3							
4			S7					6
5				S8				
6					r1			
7					r4			
8					r2			

5 b) Discuss in detail construction of syntax trees.

4M

Ans: The construction of a syntax tree is similar to the translation of the expression into postfix form. We construct sub trees for the sub expressions by creating a node for each operator and operand. The children of an operator node are the roots representing the sub expressions constituting the operands of that operator.

Each node in a syntax tree can be implemented as 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 for the operands. The operator is often called the label of the node.

We use the following functions to create the node of syntax trees for expressions with binary operators. Each function returns a pointer to newly created node.

mknode (op,left,right)- Creates an operator node with label op and two fields containing pointers to left and right.

mkleaf (id, entry)- Creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for the identifier.

mkleaf (num, val)- creates a number node with label num and a field containing val, the value of the number.

UNIT-III

6 a) Consider the following program

8M

```
void foo ( )
{
    float a, b, c;      /* level 0 declaration */
    .....
    .....
    {
        float a, b;    /* level '1x' declaration */
        .....
        .....
    }
    {
        int d, e;      /* level '1y' declaration */
    }
}
```

Stack position at level0 – 2 Marks
 Stack position at level1x – 2 Marks
 Stack position at level1y – 2 Marks
 Stack position at level2 – 2 Marks

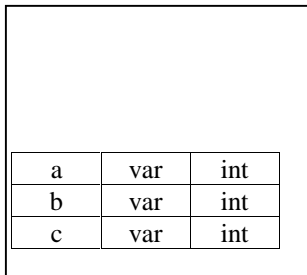
```

        {
            int f;          /* level '2' declaration */
            .....
            .....
        }
    }
}
    
```

Show the stack position after execution of level 0, level 1x, level 1y and level 2.

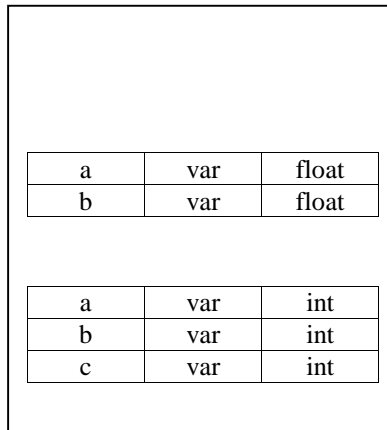
Ans:

Stack position at level 0:



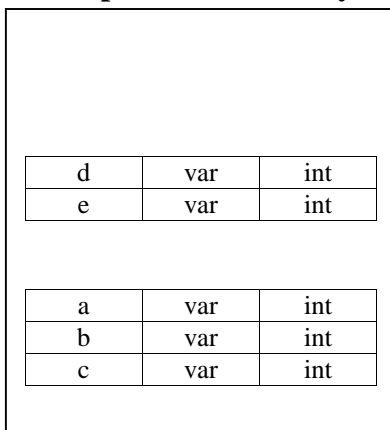
Stack

Stack position at level 1x:



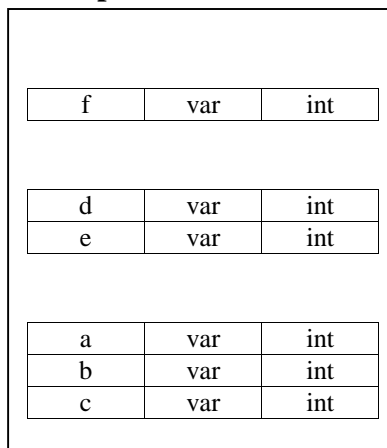
Stack

Stack position at level 1y:



Stack

Stack position at level 2:



Stack

6 b) Discuss in detail heap allocation strategy?

4M

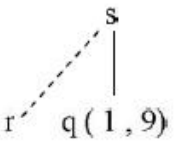
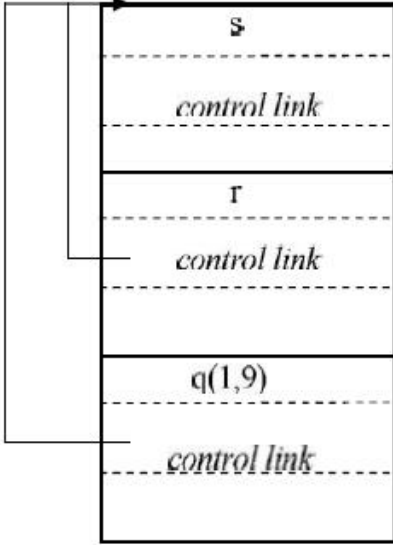
Ans:

Heap explanation – 2 Marks
Heap diagram- 2 Marks

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1, 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

(OR)

7 a) Describe the representation of scope information in the symbol table.

6M

Consider the following program structure and give its symbol table organization:

```

Program main
    var x,y : integer
Procedure P
    var x, a : Boolean
Procedure Q
    var x, y, z : real

```

Representation of scope information explanation - 3 Marks

Symbol table organization diagram – 3 Marks

Ans: Most languages have facilities for defining names with limited scopes.

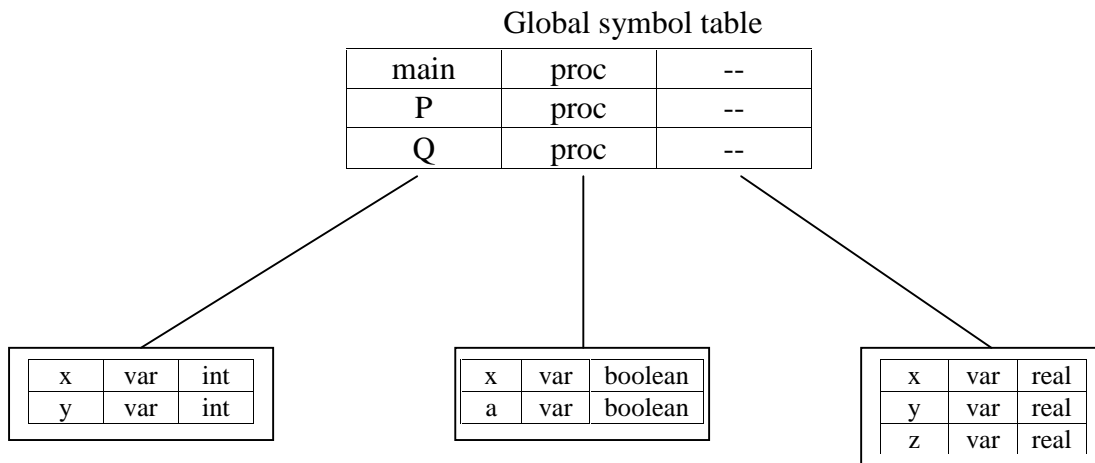
Examples:

- FORTRAN: where the scope of a name is a single subroutine.
- ALGOL: where the scope of a name is the block or procedure in which it is declared.

This situation allows the possibility that in the same program the same identifier may be declared several times as distinct names, possibly with different attributes and usually with different intended storage locations. It is thus the responsibility of the symbol table to keep different declarations of the same identifier distinct.

The usual method of making the distinction is to give a unique number to each program element that may have its own local data.

The number of the currently active subprogram is computed by semantic rules associated with productions that recognize the beginning and end of a subprogram. The subprogram number is a part of all names declared in that subprogram. The representation of the name inside the symbol table is a pair consisting of the corresponding identifier and the subprogram number.



7 b) What is the significance of symbol table at run time and compile time. Discuss it? 6M

Ans:

A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called a symbol table.

A symbol table is a compile-time data structure.

Determine whether a given name is in the symbol table

Access the information associated with a given name

Information must appear in the symbol table to denote the locations in storage belonging to data objects at run time

It's not used during run time by statically typed languages.

Any relevant information can be considered.

Significance of S.T at Compile time – 3 Marks

Run time – 3 Marks

UNIT-IV

8 a) Write an SDT scheme for Boolean expressions? 6M

Ans: An SDT scheme for Boolean expressions is as follows

SDT Scheme- 6 Marks

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code gen(E_1.false ':') E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code gen(E_1.true ':') E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true;$

$E \rightarrow id_1 \text{ relop } id_2$

$E_1.false := E.false;$
 $E.code := E_1.code$

$E \rightarrow true$

$E.code := gen('goto' E.true)$

$E \rightarrow false$

$E.code := gen('goto' E.false)$

8 b) Write quadruples, triples for the expression $(a + b) * (c + d) - (a + b + c)$

6M

Ans:

Three address code for the expression

$t1 = a + b$
 $t2 = c + d$
 $t3 = t1 * t2$
 $t4 = a + b$
 $t5 = c$
 $t6 = t4 + t5$
 $t7 = t3 - t6$

Quadruple structure – 3 Marks

Triple structure – 3 Marks

Quadruple: Quadruple is record structure with four fields, which are OP, Arg1, Arg2 and Result.

OP	Arg1	Arg2	Result
+	a	b	t1
+	c	d	t2
*	t1	t2	t3
+	a	b	t4
assign	c	-	t5
+	t4	t5	t6
-	t3	t6	t7

Triples: Three address statements can be represented by records with three fields: OP, Arg1, Arg2.

	OP	Arg1	Arg2
(1)	+	a	b
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	a	b
(5)	assign	c	-
(6)	+	(4)	(5)
(7)	-	(3)	(6)

(OR)

9 a) Construct DAG for the following code $a = a + b, e = a + d + e$

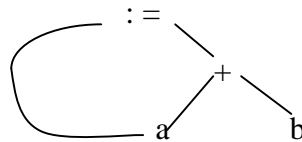
6M

Ans:

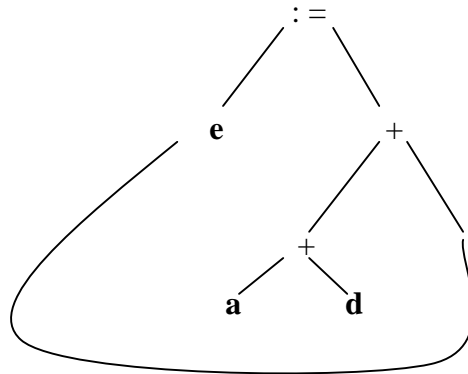
DAG for $a = a + b$ – 3 Marks

DAG for $e = a + d + e$

DAG for the expression $a = a + b$ is as follows:



DAG for the expression $e = a + d + e$ is as follows:



9 b) Write an algorithm for simple code generation?

6M

A code-generation algorithm:

Code generation algorithm- 6 Marks

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form $x := y \text{ op } z$, perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation $y \text{ op } z$ should be stored.
2. Consult the address descriptor for y to determine y' , the current location of y . Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L , generate the instruction **MOV y' , L** to place a copy of y in L .
3. Generate the instruction **OP z' , L** where z' is a current location of z . Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L . If x is in L , update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of $x := y \text{ op } z$, those registers will no longer contain y or z .

Scheme prepared by

Signature of the HOD, IT DEPT.

Paper Evaluators:

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