B.Tech.

SIXTH SEMESTER EXAMINATION, 2004-2005

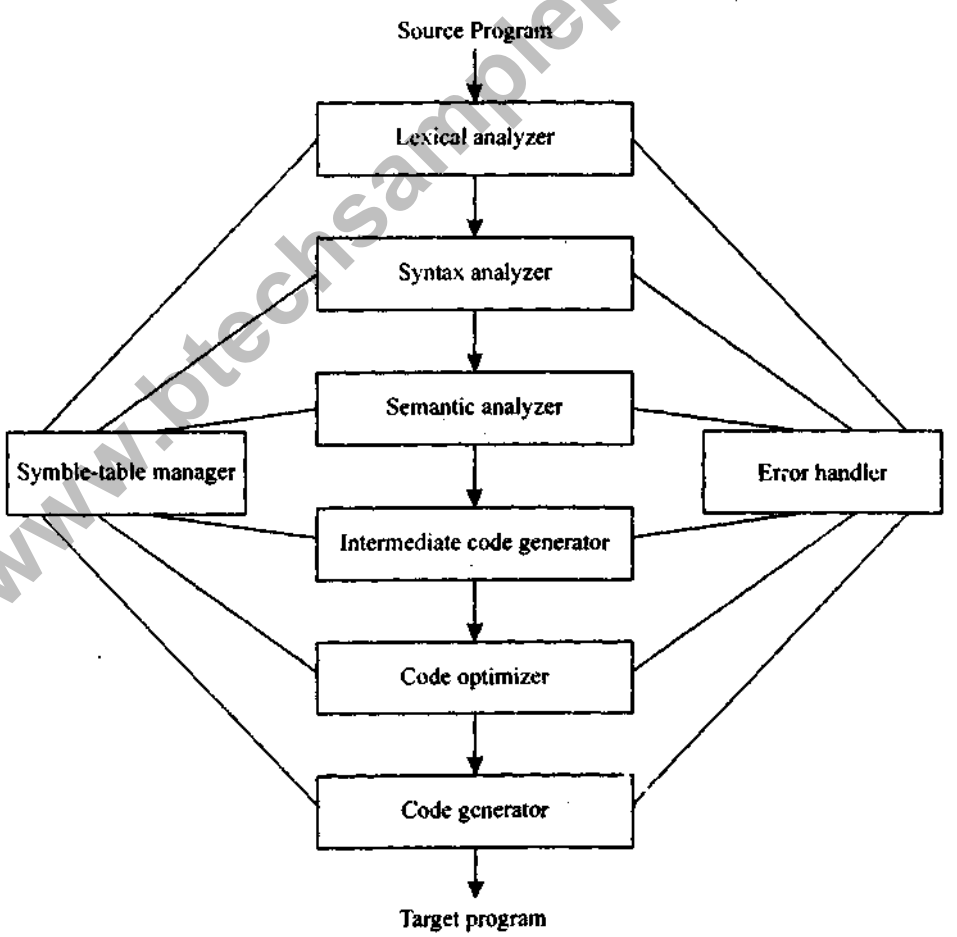
COMPILER CONSTRUCTION

Note : *(i) Attempt ALL questions.*
(ii) All questions carry equal marks.
(iii) In case of numerical problems assume data wherever not provided.

Q. 1. Answer any two parts of the following :

Q. 1. (a) What is a translator ? Discuss the role of various phases of the compiler in the translation of source program to object code.

Ans. A compiler is a program that fromulate a high-level language program into a functionally equivalent low-level langurage program. So compiler is basically a translator whose source language is a high-level language and the target language is a low-level language i.e. a a complier is used to implement a high-level language on a computer.



A compiler operates in phases, each of which transforms the source program from one representation to another.

1. Liner Aanalysis in which the stream of character making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

2. Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical pharases that are used by the complier to synthesize output.

3. The semantic analysis phase cheks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

4. The intermediate representation can have a variety of forms. Three-address code in an example of intermediate code, which is like the assembly language for a machine in which every memory location can act like a register. Three address code consist of a sequence of instruction each of which has at most three operands.

5. The code optimization phase attempts to improve the intermediate code, no faster-running machine code will result. Some optimizations are trivial.

6. The final phase of the complier is the generation of target code, consisting normally of relocatable machine code or assembly code.

Q 1. (b) Explain cross compiler. Suppose you have a working C compiler on machine A. Discuss the steps you would take to create a working compiler for another language C' on machine B.

Ans. Cross-compiler : — A compiler that run on one computer but produces object code for a different type of computer. Cross compilers are used to generate software that can run on computer with a new architecture or on special purpose device that cannot host their own compilers.

Suppose we have a new language L, that we want to make available on machine A and machine B. As a first step, we can write a small compiler : ${}^{S}C_{A}^{A}$, which will translate an S subset of L to the object code for machine A, written in a language available on A.

We then write a compiler ${}^{S}C_{S}^{A}$, which is compiled in language L and generates object code written in an S subset of L for machine A. But this will not be able to execute unless and untill it is translated by ${}^{S}C_{A}^{A}$; therefore, ${}^{S}C_{S}^{A}$ is an input to ${}^{S}C_{A}^{A}$, producing a compler for L that will run on machine A and self generated code for machine A : ${}^{S}C_{A}^{A}$

$$ {}^{S}C_{S}^{A} \longrightarrow {}^{S}C_{A}^{A} \longrightarrow {}^{L}C_{A}^{A} $$

Now if we wnat to produce another compiler to run on and produce code for machine B, the compiler can be written itself in L and made avilable on machine B by using the following steps :

$$ {}^{L}C_{L}^{B} \longrightarrow {}^{L}C_{A}^{A} \longrightarrow {}^{L}C_{A}^{B} $$

$$ {}^{L}C_{L}^{B} \longrightarrow {}^{L}C_{A}^{B} \longrightarrow {}^{L}C_{B}^{B} $$

Q 1. (c) Give the algorithm subset construction and computation of ε-closure. Using these alogorithms find the DFA for regular expression :

(a | b)* a (a | b | ∈)

Ans. Subset construction Algorithm:—

initially, ε- closure (So) is only state in Dtate and it is unmarked.

while there is an unmarked state T in Dtate do begin

mark T;

for each input symbol a do begin

U : ε- clousure (move (T_1, a));

if u is not in Dstates then

add U as an unmarked states to Dstates;

Dtran [T_1, a] = U

end

end.

Computation of ε - closure :–

Push all states in T onto stack;

initialize ε-closure (T) to T;

while stack is not empty do begin

pop t, the top element, off of stack;

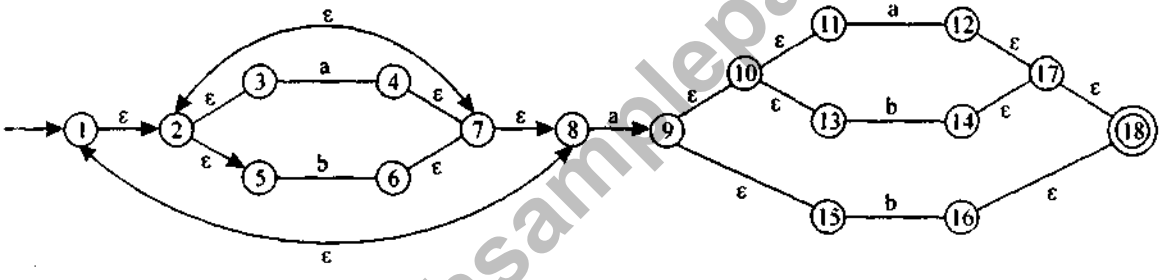
for each state u with an edge from t to u labeled ε do

if u is not in ε-closure (T) do begain

add u to ε- closure (T);

push u onto stack

end;



end;

Regular Expression (a | b)* a (a | b | ε)

ε-closure (q_1) = { 1, 2, 3, 5, 8 } = A

ε- closure (move (A, a)) = ε – closure (4, 9)

= {4, 7, 8, 2, 3, 5, 9, 10, 11, 13, 15, 16, 18}

= {2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 15, 16, 18} = B

ε- closure (move (A, b)) = ε closure (6)

= {6, 7, 8, 2, 3, 5 }

= {2, 3, 5, 6, 7. 8 } = C

ε- closure (move (B, a)) = ε- closure (4, 9, 12)

= {2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18} = D

ε- closure (move (B, b)) = ε- closure (6, 14)

= {2, 3, 5, 6, 7, 8, 14, 17. 18 } = E

ε- closure (move (C, a)) = ε- closure (4, 9) = B

ε- closure (move (C, b)) = ε- closure (6) = C

ε- closure (move (D, a)) = ε- closure (4, 9, 12,) = D

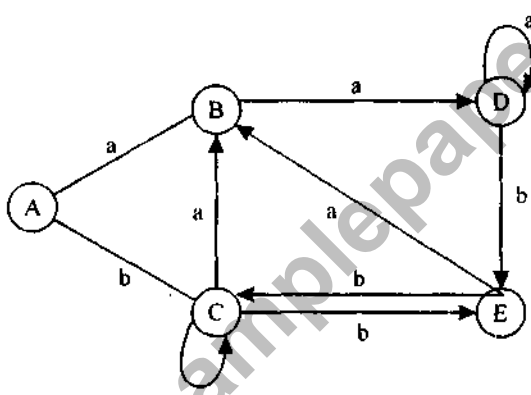
ε- closure (move (D, b)) = ε- closure (6, 14) = E

ε- closure (move (E, a)) = ε- closure (4, 9) = B

ε- closure (move (E, b)) = ε- closure (6) = C

Transition table

State / Input	a	b
A	B	C
B	D	E
C	B	C
D	D	E
E	B	C



DFA for regular Expression (a | b)* a (a | b | ε)

2. Answer any two parts of the following :

Q. 2. (a) Discuss the operator-precedence parsing alogrithm. Consider the following operator grammar and precedence functions; explain the parsing of input string id + id* id.

Grammar : E → E + E | E*E | (E) | id

Precedence functions :

	+	*	id	$
f	**4**	**2**	**4**	**0**
g	**3**	**1**	**5**	**0**

Ans— Operator- precedence parsing Algorithm :—

Input :— An input string W and a table of precedence relations.

Output :— If w is will formed, a skeletal. parse tree, with a placeholder non-termial E labeling all interior nodes; otherwise, an error indication.

Method :— Initially, the stack contains $ and the input kniffer the string w$. To parse, We execute the following program : —

(1) set i_p to point the first symbol of w$

(2) repeat forever
(3) if $ is on top of stack and i_p points to $ than return
else begin
(4) let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by i_p.
(5) if a ⋖ b or a = b then begin
(6) push b into the stack;
(7) advance i_p to the next input symbol end;
(8) else if a ⋗ b then
(9) repeat
(10) pop the stack
(11) untill the top stack terminal is related by ⋖ to the terminal most recently popped.
(12) else error ( )
end.
Now the operator precedence relation for given precedence function table

	+	*	id	$
+	⋗	⋗	⋖	⋗
*	⋖	⋗	⋖	⋗
id	⋗	⋗	⋖	⋗
$	⋖	⋖	⋖	

Grammer

E → E + E
E → E * E
E → (E)
E → id

input string : id + id + id
$ ⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $

after reducing above expression

E + E * E
E + E
E

Q. 2. (b) Consider the following grammar.

G : E →E +T | T
T → T * F | F
F → (E) | id

(i) Remove the left recursion.
(ii) Compute the FIRST and FOLLOW sets of non-terminals of the resulting grammar.
(iii) Show the resulting grammar is LL (1).
(iv) Constuct LL (1) parsing table for the resulting grammar.

Ans—

G : E → E +T | T
T→ T * F | F
F → (E) | id

(i) Eliminating the immediate left recursion.

$E \rightarrow TE'$

$E' \rightarrow TE' / \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' / \varepsilon$

$F \rightarrow (E) / id$

(ii) First (E) = First (T) = First (F) = { c, id}

First (E') = {+, ε}

First (T') = { * , ε }

Follow (E) = Follow (E') = {), $ }

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

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

(iii) First (+ T E') . ∩ First (ε) = {+} ∩ {ε} = ϕ

First (T+ E') ∩ Follow (E') = {+} {), $, } = ϕ

similarly

First (* FT') ∩ First (ε) = {*} ∩ {ε} = ϕ

First (* FT') ∩ Follow (T') > {*} ∩ {+,), $} = ϕ

Similarly

First (CE)) ∩ First (id) = {(} ∩ {id} = ϕ

Hence the grammer is LL (l) .

(iv)

Non-terminal	Input symbol					
	id	+	*	(	)	$
E	E → T E'			E →TE'		
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'			T → FT'		
T'		T' → ε	T' → *F T'		T' → ε	T' → ε
F	F → id			F → (E)		

Q. 2. (c) Discuss algorithms for computation of the sets of LR (1) items. Also show that the following grammar is LR (1) but not LALR (1).

G : S → Aa | bAc | Bc | bBa

A → d

B → d

Ans. Construction of the sets of LR(1) items : —

Algorithm :— Input :— Angumented grammer

Output : — Canonical collection of sets of LR (l) items.

1. $C_{old} = \phi$

2. add closure ($\{S_1 \rightarrow . S, \$\}$) to c

3. while $C_{old} \neq C_{new}$ do

temp = $C_{new} - C_{old}$

$C_{old} - C_{new}$

for every I in temp do

for every X in V U T (i.e. for every symbol X) do

if goto (I, X) is not empty and not in (new, then add goto (I, X) to C_{new}

}

4. $C = C_{new}$

Grammer G :—

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

I_0 : $S' \longrightarrow . S, \$$

$S \longrightarrow . Aa, \$$

$S \longrightarrow . bAc, \$$

$S \longrightarrow . Bc, \$$

$S \longrightarrow . bBa, \$$

$A \longrightarrow . d, a$

$B \longrightarrow . d, c$

$I_1 : S' \longrightarrow S . , \$$

I_2 : Goto (state 0, d) = $A \longrightarrow d ., a$

$B \longrightarrow d ., c$

I_3 : Goto (state 0, b) = $S \longrightarrow b . Ac, \$$

$S \longrightarrow b . Ba, \$$

$A \longrightarrow d ., c$

$A \longrightarrow d ., a$

I_4 : Goto (state 2, d) = $A \longrightarrow d . , c$

$B \longrightarrow d ., a$

LALR (1) Parser

Mergue lookahead for state 2 and state 4 in (R(1) parser to create new state

state = merge (state1, state 3) = $A \longrightarrow d . , a$

$A \longrightarrow d . , c$

$B \longrightarrow d ., c$

$B \longrightarrow d ., a$

Reduce/Reduce conflict forl lookahead "a" and "c" i.e. cann't decide whether to reduce d to A or B

this is not LALR (1)

Q. 3. Answer any two parts of the following :

Q. 3. (a) Consider the grammar of Q. No. 2b. Associate semantic rules with the productions for construction of syntax tree for an expression. Using the translation scheme construct the syntax tree for the expression

a + b * c

Ans. Grammer : G

$E \longrightarrow E + T / T$

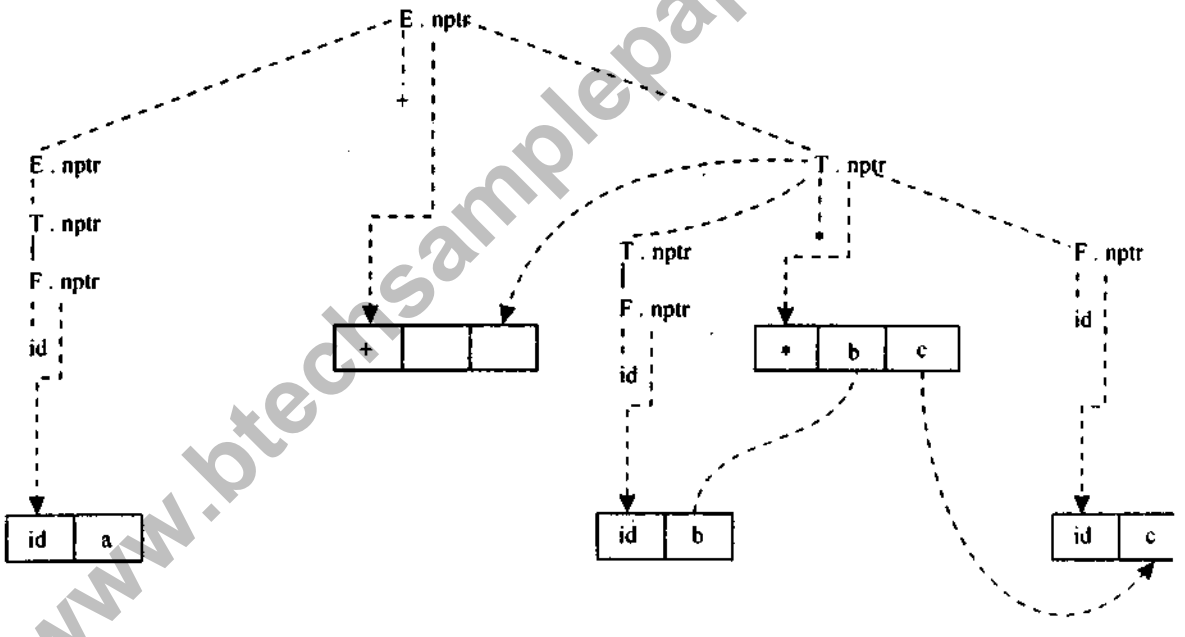
$T \longrightarrow T * F / F$

$F \longrightarrow (E) / id$

Syntax— Directed for synatx tree

Production	Semantic rule
$E \longrightarrow E_1 + T$	E . nptr = mk node ('+', E_1 . nptr, T. nptr)
$T \longrightarrow T_1 * F$	T. nptr = mk node ('*', T_1 . nptr, F. nptr)
$E \longrightarrow T$	E . nptr = T. nptr
$T \longrightarrow F$	T. nptr = F. nptr
$F \longrightarrow (E)$	F. nptr = E . nptr
$F \longrightarrow id$	F. nptr = mkleaf (id, id.entry)

syntax tree for a + b * c



Q. 3. (b) Consider the following grammer for array references. Give syntax directed translation scheme to genetate three address codes for addressing array elements. Translate the statement X = A [i, j]. Where lower bounds of i and c are 1 and 1, upper bounds are 10 and 20 respectively.

Grammar :

$S \longrightarrow L := E$

$E \longrightarrow E + E \mid (E) \mid L$

$L \longrightarrow Elist] \mid id$

$Elist \longrightarrow Elist, E \mid id [E$

Ans. Grammer G :

$S \longrightarrow L := E$

E ⟶ E + E

E ⟶ (E)

E ⟶ L

L ⟶ Elist]

L ⟶ id

Elist ⟶ Elist , E

Elist ⟶ id [E

Synatx directed translation scheme of above grammer is as :

S ⟶ L : = E { if L. offset = null then
emit (L . place ' : = ' E . place)
else
emit (L . place '[' L . offset ']' ' : = ' E . place) }

$E \longrightarrow E_1 + E_2$ { E . place : = newtemp;
emit (E . place ' : = E_1 . place ' + ' E_2 place)}

$E \longrightarrow (E_1)$ { E . place = E_1 . place }

E ⟶ L { if L . offset = null then
E . place : = L . place
else begin
E . place : = newtemp;
emit (E . place ' : = ' L . place ' [' L . offset '] ') end }

L ⟶ Elist] { L . place := newtemp ;
L . offset : = newtemp ;
emit (L . place ' : = ' c (Elist . array)) ;
emit (L . offset ' : = ' Elist . place ' * ' width (Elist . array)) }

L ⟶ id {L . place : = id . place ;
L . offset : = null }

$Elist \longrightarrow Elist_1$, E { (t : = newtemp ;
m : = $Elist_1$. ndim + 1 ;
emit (t ' : = ' $Elist_1$. place ' * ' limit ($Elist_1$. arrary, m)) ;
emit (t ' := ' t ' + ' E . place) ;
Elist. arrary : = $Elist_1$. assary;
Elist . ndim : = m }

Elist ⟶ id [E { Elist . assary : = id. place ;
Elist . place : = E. place;
Elist. ndim : = 1 }

Three address code for
X = A [i, j]

lower bound are 1,1

upper bound are 10, 20

$t_1 = i * 20$

$t_1 = t_1 + j$

$t_2 = c$ 1* constat = $base_A$ - 84 *1

$t_3 = 4 * t_1$

$t_4 = t_2 [t_3]$.

[x = t_4]

Q. 3. (c) Translate the following program segment into three-address statements :

```
switch (a+ b)
{
   case 2 : {x = y ; break ; }
   case 5 : {Switch x
      {
        Case 0 : { a = b + 1; break ;}
        Case 1 : { a = b + 3; break ;}
        default : { a = 2;}
   }
     break;
   case 9 : {x = y – 1; break ;}
   default : {a = 2;}
}
```

Ans. The three address code is

(1) $t_1 = a + b$
(2) goto (23)
(3) x = y
(4) goto NEXT
(5) goto (14)
(6) $t_3 = (b + 1)$
(7) $a = t_3$
(8) goto NEXT
(9) $t_4 = b + 3$
(10) $a = t_4$
(11) goto NEXT
(12) a = 2
(13) goto NEXT
(14) if x = 0 goto (6)
(15) if x = 1 goto (9)
(16) goto (12)
(17) goto NEXT
(18) $t_5 = j - 1$
(19) $x = t_5$
(20) goto NEXT
(21) a = 2
(22) goto NEXT
(23) if $t_1 = 2$ goto (3)
(24) if $t_1 = 5$ goto (5)
(25) if $t_1 = 9$ goto (18)
(26) goto (21)

Q. 4. Answer any two parts of the following :

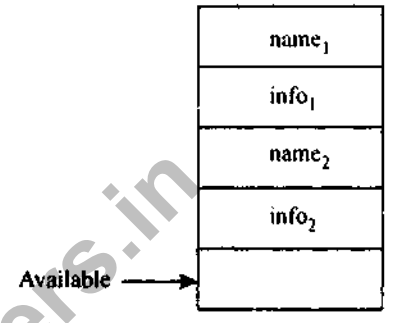
(a) What is symbol table ? Discuss the various approaches used for organization of symbol table.

Ans. Symbol Table : — A symbol table is a data sturcture used by a compiler to keep track of scope/binding information about name. This information is used in the source program to identify the various program elements like variables, constants, procedures, and tabels of statements. The symbol table is reached every time a name is encountered in the source text. When a new name or new information about an enisting name is discovered, the

content of symbol table changes. Therefore a symbol table must have an efficent mechianism for accessing the information build in the table well as for adding new entries to the symbol table.

Various data structure to symbol table organiszation : —

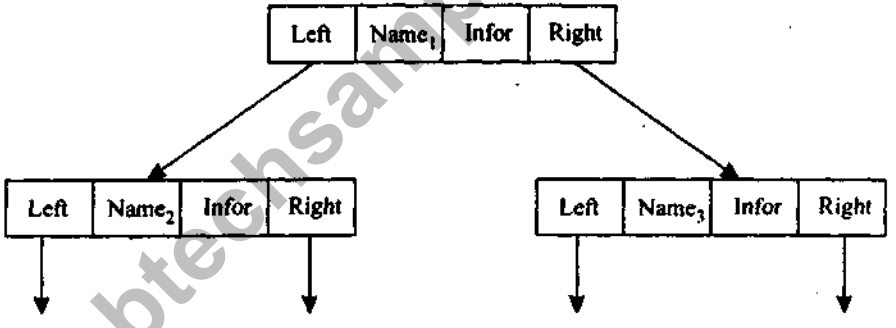
1. The liner list : — A linear list of record is the easiest way to implement a symbol table. The new name are added to the table in the order that they arrive. Whenever a new name is added to the table, the table is first searched lineartly to cheek whether or not the name is already present in the table. If the name is not present, then the record for new name is errated and added to the list at a position specified by the available pointer.



The average number of comparison, P, required for search are p = (n+1) / 2 for successful search and p = n for an unsuccessful search, wheren is a records in symbol table. The advantage of this organization is that it take less space and addiotion to table are simple. Disadvantage is that it has a higher accessing time.

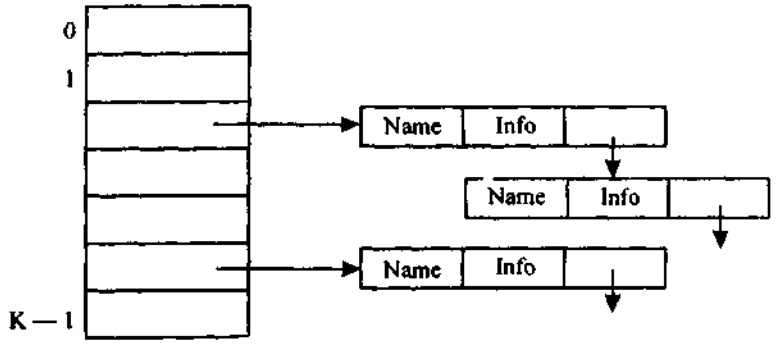
2. Search tree :— The search tree is a more efficient approach to symbol table organization. We add two links left and right in each record and these link point to the record in search tree.

Whenever a name is to be added, the first name is searched in tree. If it does not exist, then a record for the new name is created and added at proper position in the search tree. This organisation has the property of accessibility, i.e. all the names accesible from name, will, by following a left link, precede name, in alphabetical



order. Similarly for right link. The expected time needed to enter n names and to make m qureries is proportional to $(m+n)\log_2 n$; so for greator number of records (higher n) this method has advantage over linear list organization.

3. Hash table : — A hash table is a table of k pointers numbered from zero to K-1 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 zero and K-1 and using this value as an index in the hash table, we search the list of the symbol table records that is built on that index. If name is not is not present in that list, we create a record for name and insert it at the head of list. When retrieving the information associated with the name, the hash



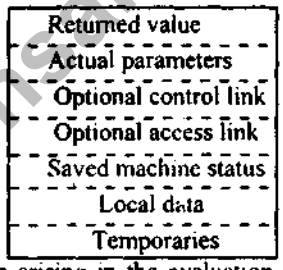
value of the name is first obtained and then the list that was built on this hash value is searched for information about the name.

Q. 4. (b) Explain activation record and display structure. Show the activation records and display structure just after the procedure called at lines marked x and y have started their execution. Be sure to indicate which of the two procedure named A you are refering to :

```
Program Test;
    Procedure A;
      Procedure B;
        Procedure A;
          ------
            end A;
begin
    Y : A;
  end B ;
 begin
    B;
 end A;
begin
    X : A;
 end Test;
```

Ans. Activation record : — Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame, consisting of collection of fields. Not all languages, nor all compilers use all of these fields; of ten register can take place of one or more of them.

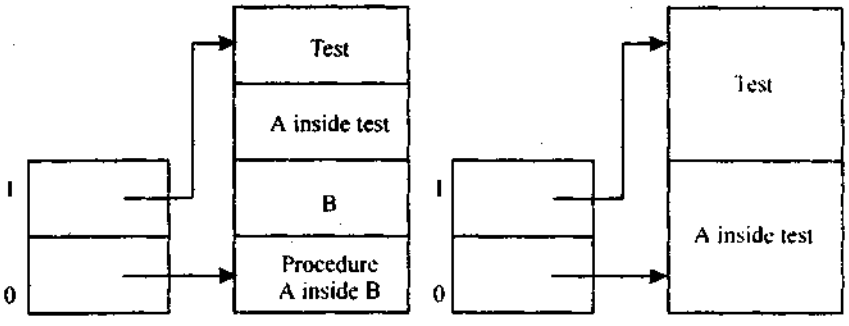
The purpose of the fileds of an activation record is as follows, starting from the field for temporaries.



1. Temporary value, such as those arising in the evaluation of expressions, are stored in the field for temporaries.

2. The field for local data holds data that is local to an execution of a procedure.

3. The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the program counter and machine registers that have to restored when control returns from the procedures.



4. The optional acess link is used to refer to non-local data held in other activation records.

5. The optional control link points to the activation record of the caller

6. The field for actual parameters, is used by the calling procedures to supply parameter to the called procedures.

7. The field for returned value is used by the called procedure to return a value to the calling procedure.

The sizes of each of there fields can be determined at the ti· e a procedure is called.

Q. 4. (c) Discuss the folling storage-allocation strategies :

(i) Stack allocation

(ii) Heap allocation

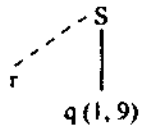
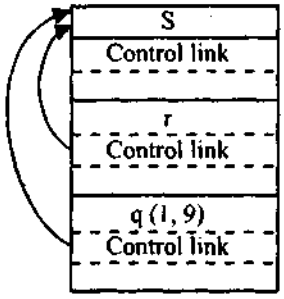
Ans. (i) Stack allocation : — Stack allocation is based on the idea of a control stack, storage is organized as a stack and activation records are push and poped as activation begin and end respectively. Storege of local in each call of a procedure is constained in the activation record for that call. Thus locals are bound to fresh stroage in each activation, because a new activation record is pushed on the stack when a call is made. The values of locals are deleted when activation ends; i.e. the valeus are lost because the stroage for locals disappear when the activation record is popped.

Ex : Position in activation tree	Activation Records on the stack	Remark
S	S a : array	frame for s
S r	S a : array r i : integer	r is activated
S r q (1, 9)	S a : array q (1, 9) i : integer	Frame for r has been popped and q (1, 9) pushed
S r q (1, 9) P (1, 9) q (1, 3) P (1, 3) q (1, 10)	S a : array q (1, 9) i : integer q (1, 3) i : integer	Control has just returned to q (1,3)

(ii) Heap allocation : — The stack allocation strategy cann't be used if either of the following is possible:

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

Heap allocation parcels out pieces of contiquous storage, as needed for activation record or other objects. Pieces may be deallocated in any order no over time the heap will consist of alternate areas that are free and in use.

Ex : Position in activation tree	Activation records on the stack	Remamk
		Retained activation record for r

Q. 5. Answer any two parts of the following :

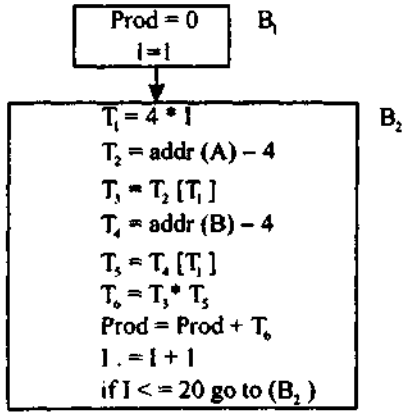
(a) Consider the following sequence of three address code :

(1) PROD : = 0
(2) I : = 1
(3) T_1 := 4* I
(4) T_2 : = addr (A) – 4
(5) T_3 : = T_2 [T_1]
(6) T_4 : = addr (B) – 4
(7) T_5 : = T_4 [T_1]
(8) T_6 : = T_3* T_5
(9) PROD : = PROD + T_6
(10) I : = I + 1
(11) if I < = 20 go to (3)

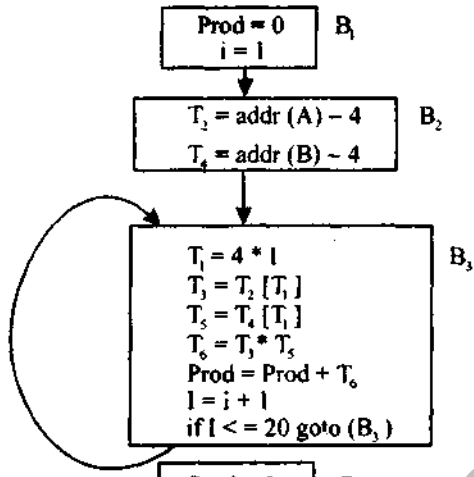
(i) Find the basic blocks and construct a flow graph.
(ii) Eliminate common subexpressions.
(iii) Move the loop-invarient computation out of the loop.
(iv) Find the induction variables and eliminate them where possible.

Ans.

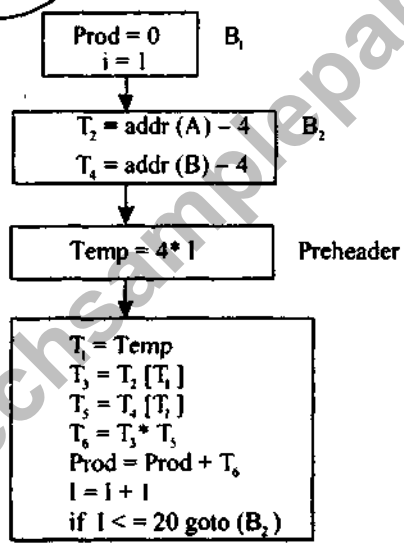
(i)



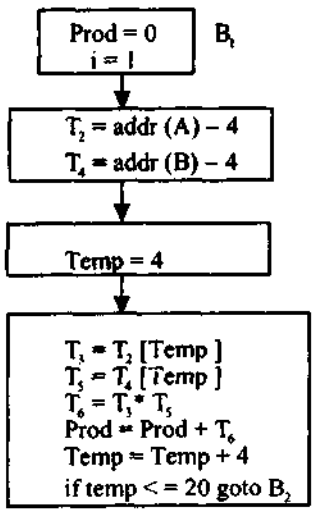
(ii)



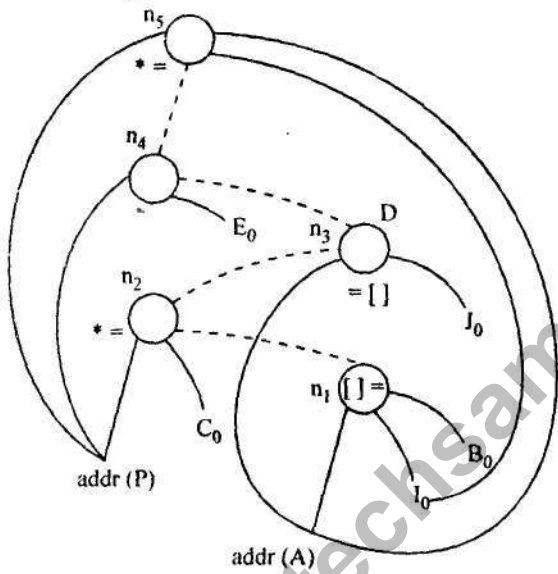
(iii)



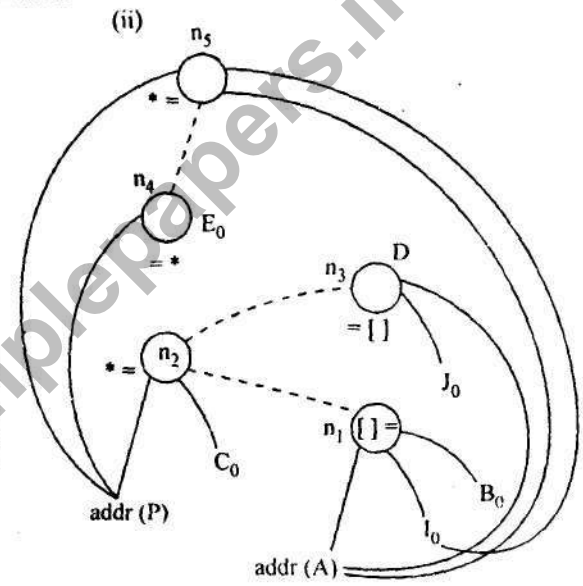
(iv)



Q. 5. (b) Construct DAG for the following code sequence :
A [I] : = B
***P : = C**
D : = A [J]
E : = * P
***P := A : [I]**
Assume that :
(i) P can point anywhere
(ii) P Points to only B or D
Do not forget to show the implied order constraints.
Ans. (i)



(ii)



Q. 5. (c) Semantic Errors
Context free Grammer

(i) Semantic Errors : —Semantic errors can be detected both at compile and at run time. The most common semantic error that can be detected at compile time are errors of declaration and scope.

Typical excamples are undeclared or multiply- decleared identifiers.

Type incompatibilities between operators and operands and between formal and actual parameter are another common source of semantic errrors, that can be detected in many languages at compile time. The amount of type checking that can be done depends on the language at hand.

(ii) Context free Grammer : —CFG notation specifies a context-free language that consist of terminals, non-terminal, a start symbol and productions.

Terminal are nothing more than takens of the language, used to form the language constructs. Non-Terminals are variables that denotes a net of string.

Therefore context-free grammer is a four-tuple denoted as :—

$G = (V, T, P, S)$

Where

V = is finite set of symbol called as non-terminal of variables.

T = is a set of symbol that are called as terminal

P = is a set of production.

S is a member of V, called as start symbol.