B. Tech.

SIXTH SEMESTER EXAMINATION, 2005-2006

COMPILER CONSTRUCTION

Time : 3 hours *Total Marks : 100*

Note : *(i) Answer All questions.*

(ii) All questions carry equal marks.

(iii) In case of numerical problems assume data wherever not provided.

(iv) Write your answers as concisely as possible, consistent with providing a complete answer to the question.

(v) Be precise in your answer.

Q. 1. Attempt any *four* parts of the following : — **(5 × 4 = 20)**

(a) Explain bootstrapping in detail.

Ans. Bootstrapping :—A compiler is a complex enough program that would like to write it in friendlier language than assembly language. In the unit programming evironment, complier are written in C. Each C compiler are written in C. Using the facilities offered by a language to compile itself is the essence of bootstrapping.

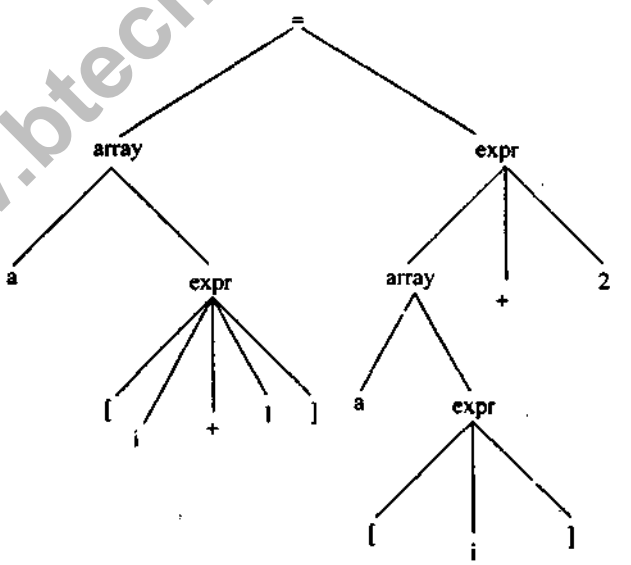
For bootstrapping purpose; a compiler is characterized by three languages. The three languages such that it compiles. All target language T that it generates code for and the implementation language I that it is written in.

Q. 1. (b) Given the Cassignement

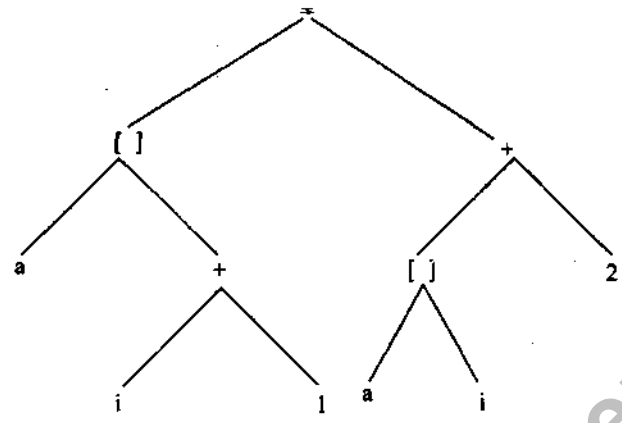
$a[i+1] = a[i] + 2$

draw a parse tree ans syntax tree for the expression.

Ans. $a[i+1] = a[i] + 2$



Parse tree

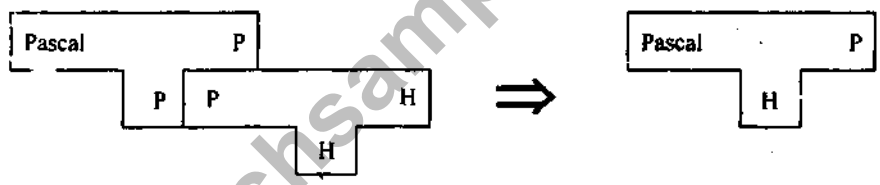


Syntax tree

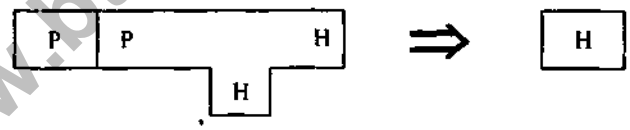
Q. 1. (c) Suppose, you have a Pascal to C translator written in C and a working C compiler. Use T diagram to describle the steps you would take to create a working Pascal Compiler.

Ans.

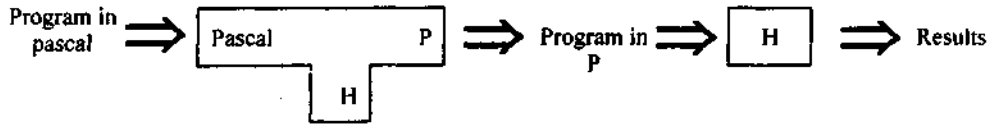
Step –1



Step –2



Step –3

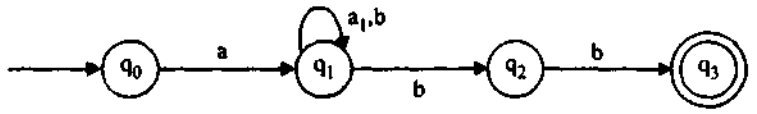


Q. 1. (d) Construct a deterministic finite automata (DFA) for the regular exression.

$a.(a+b)^*.b.b$

Ans. $a(a+b)^*bb$]

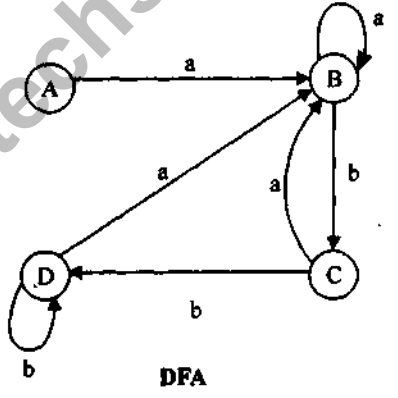
Final control NFA



	a	b
$\rightarrow q_0$	a_1	—
q_1	a_1	a_1, a_2
q_2	—	a_3
q_3	—	—

Now *DFA*

	a	b	
$\rightarrow q_0$	q_1	—	$q_0 = A$
q_1	q_1	q_1, q_2	$q_1 = B$
q_1, q_2	q_1	q_1, q_2, q_3	$q_1, q_2 = C$
q_1, q_2, q_3	q_1	q_1, q_2, q_3	$q_1, q_2, q_3 = D$



DFA

Q. 1. (e) Write English description for the language generated by the following regular expressions :

(i) $(a \mid b)^* a (a \mid b \mid c)$

(ii) $(aa \mid b)^* \ (a \mid bb)^*$

Ans. (i) A string starts with either with a's or b's or followed by single and end and with either a or b or c.

(ii) A string start with even number of a's or any number b and end with either any number of a or even number of b's.

Q. 1. (f) Write a context sensitive grammar that generates strings of the form *xcx*, where *x* is a string of a's and b's .

Ans. $X \longrightarrow aX \mid bX$

$X \longrightarrow cX$

$X \longrightarrow \varepsilon$

Q. 2. Attempt any *two* parts of the following : — **(10 × 2 = 20)**

(a) Construct an LALR (1) parsing table for the following grammar :

D → L : T

L → L, id/id

T → integer.

Ans. LALR (*i*) parsing table :—

grammar is :—

D → L : T

L → L, id

L → id

T → integer

⇒

(I) argument grammar

D' → D

D → L : T

L → L ; id

L → id

T → integer

(II) D' → D, $ then compare it with A → α.B β,a

⇒ A = D'

α = ∈

B = D ⇒ find first (βa)

β = ∈ = (∈ $) = $

a = $

∴ D' = ·D, $

D = ·L : T, $

I_0 = L → ·L, id, :

L → ·id, :

T → ·integer, ,

I_1 = goto (I_0, D)

D' = D·, $

I_2 = goto (I_0, L)

D = L· : T, $

L = L·, id, :

I_3 :– goto (I_0, id)

L → id· , :

I_4 :– goto (I_0, integer)

T → integer· , ,

I_5 :– goto (I_2, :)

D = L : ·T, $

I_6 :– goto (I_2, :)

L = L,· id, :

I_7 :– goto (I_5, T)

D = L : T·, $

I_8 :– goto (I_6, id)

L = L, id·, :

Now,

graph from abotve p-states are :

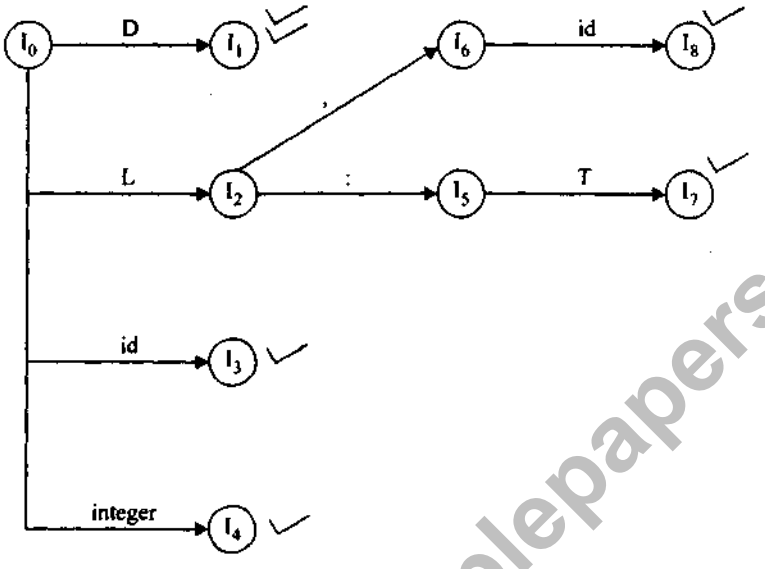


Table :

State	Action					Goto		
	:	,	id	integer	$	D	T	L
I_0			S_3	S_4		1		2
I_1					acc			
I_2	S_5	S_6						
I_3	r_3							
I_4		r_4						
I_5							7	
I_6			S_8					
I_7					r_1			
I_8	r_2							

Q. 2. (b) Cosider the following grammer :

E → EBE

E → num

E → (E)

B → +

B → -

B → *

**B → **

(i) Explain why this grammar is not suitable to form the basis for a recursive descent parser.

(ii) Use left-factoring and left-recursion removal to obtain an equivalent grammar which can be used as the basis for a recursive descent parser.

Ans. $E \rightarrow EBE$

$E \rightarrow num$

$E \rightarrow (E)$ $\Rightarrow$ The grammar becomes

$B \rightarrow +$ $\quad E \rightarrow E+E \mid E-E \mid E*E \mid E \setminus E$

$B \rightarrow -$ $\quad E \rightarrow num$

$B \rightarrow *$ $\quad E \rightarrow (E)$

$B \rightarrow \setminus$

(i) Above grammar is not suitable to form the basis for recursive descent parser because above grammar is left recursive grammar and such grammar causes a recursive-descent parser, even one with backtracking and to go an infinite loop. So, in many case we try to eliminate left recursion and left factoring from it.

(ii) Using left factoring and left recursion removal to obtain as equivalent grammar which can be used as basis for recursive descent parser is :—

Now remoing left recursion

$E \rightarrow E+T \mid T$	$\Rightarrow$	$E \rightarrow TE'$
$T \rightarrow T-F \mid F$		$E' \rightarrow +TE' \mid t$
$F \rightarrow F*G \mid G$		$T \rightarrow FT'$
$G \rightarrow G \setminus H \mid H$		$T' \rightarrow -FT' \mid E$
$H \rightarrow num \setminus (E)$		$F \rightarrow GF'$
(removed ambiguity)		$F' \rightarrow *GF'$
		$G \rightarrow HG'$
		$G' \rightarrow \setminus HG' \setminus t$
		$H \rightarrow num \setminus (E)$

Thus, the grammar which can be used as basis for recursive descent parser is :—

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow -FT' \mid t$

$F \rightarrow GF'$

$F' \rightarrow *GF' \mid t$

$G \rightarrow HG'$

$G' \rightarrow \setminus HG' \setminus t$

$H \rightarrow num \setminus (E)$

Q. 2. (c) Explain shift reduce parsing and operator precedence parsing.

Ans. In shift reduce parsing :—We use stack to hold grammar symbols and an input buffer to hold the string to be parsed. In this, we use '$' to mark the bottom of the stack and also the right end of the Input/Output. Initially stack is empty and string is on the I/P. The parser operates by shifting zero or more input symbol onto the stack and perform shift or reduce operations. The parser repeats the cycle until it has detected an error or until the stack contains the start symbol and the I/P is empty.

Example :— I/P string→ $id_1 + id_2 * id_3$

Production is :—

$E \rightarrow E + E/E*E/(E)$

$E \rightarrow id$

Now,

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

In this, four possible actions a shift-reduce parser can take. They are :—

(1) In shift action, the next I/P symbol is shifted onto the top of the stack.

(2) In reduce action, the parser knows right end of handle is at the top of stack. It must then locate left end within the stack and decide with what non-terminal to replace the handle.

(3) In an accept action, the parser announces successful completion of parsing.

(4) In an error action, parser discovers that a syntax error has occured and calls an error recovery routine.

Now,

In operator preceding parsing, we define three disjoint precedence relations, $<\cdot, =, R\cdot>$ between certain pairs of terminals and this is several disadvantages also. For each two terminals symbols a and b we say :

Relations	Meaning
$a <\cdot b$	a" yields precedence to "b
$a \doteq b$	a" has the same precedence as "b
$a \cdot> b$	a " takes precedance over "b

Example :— Suppose operator—precedence relation table is given i.e.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

String is :— id*id*id

then

String become :— $id+id*id$

$ <· id ·> + <· id ·> * <· id ·> $

i.e. <· is insterted between the left most $ and id and so as on. Now,

(1) Scan the string from left end until the first ·> is encountered.

(2) This scan backwards over any $\doteq$ l_s until a <· is encountered.

(3) This replace the terminals between <· and ·> by their corresponding non-terminals.

i.e. $ E+ <· id ·> * <· id ·> $ [we reduce id to E]

$\Rightarrow$ $ E + E* <· id ·> $

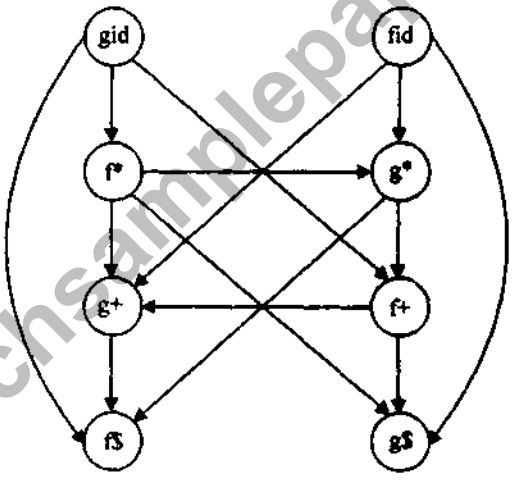
$\Rightarrow$ $ E + E* E $

Now consider string as :— $ + * $ again insert relations between them,

$ <· + <· * ·> $

$\Rightarrow$ $ E + E $

$\Rightarrow$ $ E $



Graph for above is

and related precedence function is

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

Q. 3. Attempt any *two* parts of the following : — **(10 × 2 = 20)**

(a) Consider the following code fragment generate three address code for it.

for (i = 1; i < 10; C++)

if a < b then *x = y + z*.

Ans. (1) i = 1

(2) if i < 10 goto 3

(3) if a < b goto

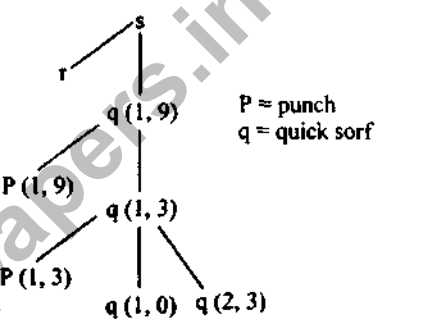
goto L next

(5) $t_1 = y + z$

(6) $x = t_1$
(7) $i = i + 1$
(8) goto z
L next :

Q. 3. (b) Activations of functions within a program are allocated a stack frame. What; is the layout of a typical stack frame on the stack ? Identify the kinds of data values which are stored there.

Ans. Stack allocation :— A stack allocation is based on the idea of a control stack; storaged is organized as a stack, and activation records are purled and popped an activation begin and respectively. Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus locals are bound to fresh storage. In each activation, because a new activation record is purled on to the stack when call is made. Further, the values of locals are deleted when activation ends; i.e, the value are lost because the storage for locals disappear when the activation is popped.



We first describe a form of stack allocation in which this sizes of all activation records are known at compile time.

Let an activation tree.

Stack allocation of activation record is as follows describe.

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)

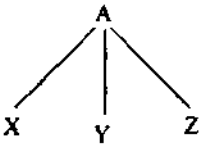
Q. 3. (c) Discuss in details about Syntax directed translation schemes, parse tree ans syntax trees.

Ans. Syntax-directed translation schemes :— There are two notation for associating semantic rules with production, syntax-directed definition and translation schemes. Syntax-directed definitions are high-level secification for translations. They hide many implementation details and free the user form having to specify explicitly. The order in which translation take place. Translation scheme indicates the order in which semantic rule are to be evaluated, so they allow now implmentation details to be shown.

Conceptually, with both syntax directed defnitions and translations schemes. We parse the input. Token stream build the parse tree and then traverse the tree a needed to evaluate the semantic rules at the parse tree no dis evalation of semantic rules may generate code, save information in a symbol table, issue error message or perform any other activities. The translation of token stream is the result obtined by evaluation the semantic rules.

Input string → Parse tree → Dependency graph → Evaluation order for semantic rules

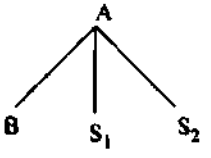
Parse tree :—A parse tree pietorially shows how the start symbol of grammar derives a string in the language. If nonterminal A has a production. A →*XYZ*; then a parse tree may have an interior node lablled A with . 3 children lablled X, Y, and Z, from left to right.



Formally a parse tree is a tree with the following properties :

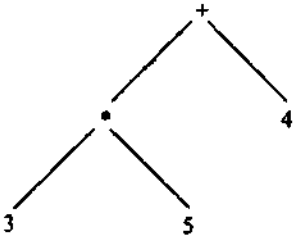
1. All root is label by the start symbol.
2. Each leaf is label by a token or by ε.
3. Each interior node is labelled by a nonterminal.
4. If A is the nonterminal labbelling save interior node and $X_1\ X_2$ ..... X_m are the labels of the children of that node from left to right, then $A \rightarrow X_1X_2.....X_m$ is a production. Here X_1, X_2 .....X_m stand for a symbol that is either a terminal or a nonterminal. As special case if $A \longrightarrow \varepsilon$ then node is tablled A may have a single labelled ε.

Syntax tree :— An (abstract) syntax tree is a condensed form of parse tree. Useful for representing languge contraction. The construt. $S \rightarrow$ if B then S_1 else S_2 might appear in a syntax tree as



In a syntax tree, operations and keywords do not appear a leaves, but rather are associated with the interior node that would be the pasent of those leaves in the parse tree. Another implification found in syntax tree is that chain of single production may be collapsed, the parese tree becomes the syntax tree.

Syntax directed tranlation is based on syntax as well as passe trees



Q. 4. Attempt any *two* parts of the following : — **(10 × 2 = 20)**

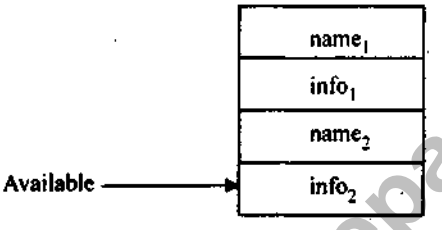
(a) What are several methods of organizing the symbol table? Explain.

Ans. Symbol Table : — A symbol table is a data sturcture used by a compiler to keep track of scope/ binding information about names. These names are used in the source program to identify the various program elements i.e. variables, constants, procedures and tables of statements.

There are several methods or approaches of organizing the symbol table. These methods are discussed below :

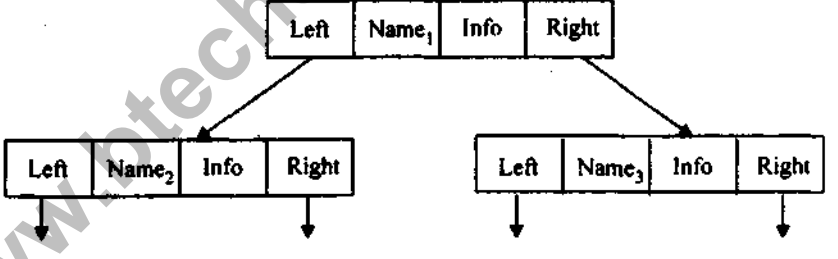
1. The linear list : — It is the easiest way to implement a symbol table. New names are added to the table in the order that they arrive. Whenever a new name is added to the table, the table is first searched linearly or sequently to check whether or not the name is already present in the table.

If the name is not present, then the second for new name is created and added to the list at a position specified by the available pointer as shown in the below.



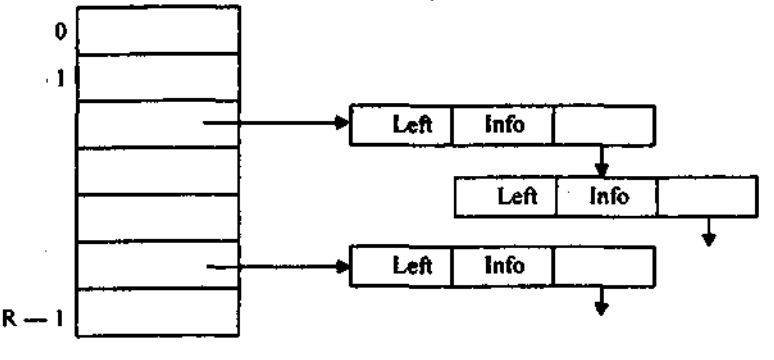
To reterieve the information about the name the table is searched sequentially, starting from the first record in the table. The average number of comparison, P, required for search are $p = (n+1)/2$ for successful search and $p = n$ for an unsuccessful search.

2. Search tree :— It is more efficient approach to symbol table organization. We add two links left and right, in each record and these links point to the record in search tree. Whenever a name is to be added, the first name is searched in the tree. If it does not exist, then a record for the new name is created and added at proper position in the search tree.



The expected time needed to enter n names and to make m queries as proportional to $(m+n)\log_2 n$.

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 with the symbol table. To enter a name into symbol table, we find out at the hast value of the name by applying a suitable hash function.



Q. 4. (b) Write the syntax directed translations to go alongwith the LR parser for the following :

L → id elist

elist → elist [E] | [E]

E → E + T | T

T → T* F | F

F → id

Ans.

SDS

(1) L ⟶ id, elist

L ⟶ id .....(i)

L ⟶ elist .....(ii)

(i) L ⟶ id {L . place : = id . place ;
L . offset : = null /* L is a simple id */
}

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

(2) elist ⟶ elist , [E]|[E]
{t : = newtemp ;
m : = Elist . ndim + 1 ;
emit (t ' : = ' $Elist_1$. place ' * ' limit ($Elist_1$. arrary, m)) ;
emit (t ' := ' t ' + ' E . place) ;
Elist. arrary : = $Elist_1$. assary;
Elist . Place : = t;
Elist. ndim : = m }

(3) E ⟶ E +T | T { E : place : = newtemp;
T : place : = newtemp;
emit (E. place ' : = E . place ' + ' T place)
emit (E . place' : =' T place)}

(4) T ⟶ T * F| F { T : Place : = newtemp;
F : Place : = newtemp;
emit (T. place ' : = , E place ' * ' T place)
emit (T . place' : =' F place)}

(5) F ⟶ id {F : Place : = id . Place;
F. offset : = null }

Q. 4. (c) There are syntactic errors in the following constructs : For each of these constructs, find out which of the input's next tokens will be detected as an error by the LR parser.

(i) While *a = b* do *x = y + z*

(ii) *a + b = c*

(iii) *a* + b + c*

Ans. There are syntactic errors in the following constructs : for each these constructs, find out which of the *I / P's* next tokens will be detected as an error by the LR Parser.

(i) While $a = b$ do $x = y + z$

(ii) $a + b = c$

(iii) $a^* + b + c$

each have syntactic errors.

(i) Semicolon missing

(ii)

(iii) $a^* + b + c$ There is semicolon missing and syntax error.

Q. 5. Attempt any *two* parts of the following : — **(10 × 2 = 20)**

(a) (i) Give the sequencea of three-address code instructions corresponding to each of the arithmetic expressions :

$2 + 3 + 4 + 5$

(ii) Describe how a for-statement can be systematically turned into a corresponding while statement. Does it make sense to use this to generate code ?

Ans. Three address code is a sequence of statments of the general form :

$x := y\ op\ z$ ...(i)

Where x, y and z are names, constants, or compiler generated temporaries; of op stands for any operator, such as fixed or floating point arithmethic operator, or a logical operator a bolleoan valued data. Thus a source language expressions like $x + y^* z$ might be translated into a sequance.

$b_1 := y * z.$

$b_2 := x + t_1$

where b_1 and b_2 are complier generated temporary names.

According to the questions ;

$x := 2 + 3 + 4 + 5$

The three address code for the above sequance is as follows ;

(1) — $t_1 := 2$

(2) — $t_2 := t_1 + 3$

(3) — $t_3 := t_2 + 4$

(4) — $t_4 := t_3 + 5$

(5) — $x := t_4$

Q. 5. (b) (i) Explain global data flow analysis.

Ans. In order to the code optimization and a good job of code generation, a compiler needs to collect imformation about the program as whole and to distribute this information to each block in the flow graph. For example, we need to know that variables are live on exist from each block could improve register usage and how could we use knowlege of global common sub expressions to eliminate redundant ampitations and we also need to know how a complier could take advantage of reading definitions", such as knowing where a variable like debug was last defind before reaching a given block, in order to perform tranformations like constant folding and dead-lock elimination. These facts are the data-flow information that an optimizing compiler, collects by a process known as data-flow analysis.

Data-flow information can be collected by a program setting up and solving systems of information or equations that relate imforamtion at various points in a program. A typical equation has the form :

out [S] = gen [S] U (in [S]-kill [S]) ..... (i)

and can be read as, " the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statements". Such statements are called data-flow equations.

Q. 5. (b) (ii) Explain DAG representation.

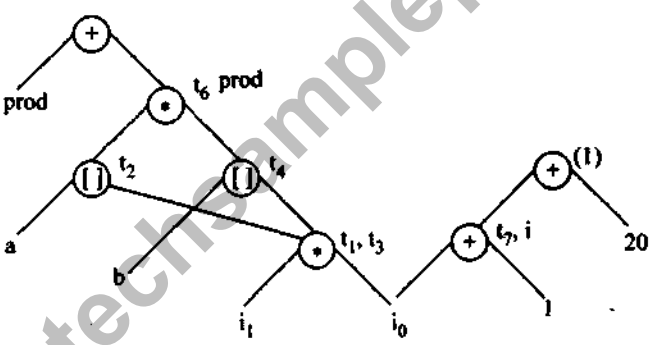
Ans. Directed acyclic graphs are useful data structures for implementing transformation on basic block. A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statments of the block. Constructing a dag from three addres statements is a good way of determining common subexressions within a block, determining which names are used inside the block but evaluated outside the block, and determining which staements of the block could have their cmputed value used outside the block.

A dag for a basic block is a directed acyclic graph with the following lobes on the nodes :

1. Leaves are labelled with identifiers either, variable names or contants from the operator applied to a name whether the l-value or r-value of a name is needed; most leaves represent r-values.

2. Interior nodes are labelled by an operator symbol.

3. Nodes are also optionally given a sequence of indtifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labelling a node are deemed to have that value.



Dag for a block

Q. 5. (c) Discuss loop optimization technique with suitable examples.

Ans. The running time of a program may be improved if we decrease the no. of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for the loop optimization :

1. Code motion :— *which move code outside the loop.*

2. Induction variable elimination :—which we apply to eliminate i.e. from the inner loops.

3. Reduction in strength :—which replace an experience operation by a cheaper one, such as multiplication by addition.

Now, Code Motion :—

An important modification that decrease the amount of code in a loop is code motion. The tranformation takes an expressions that yields the same result independent of the no. of times a loop of is executed and place the expession before the loop. For example, evaluation of limit-2 is a loop invariant computation in the following while statement.

while (i < = limit — 2) ...(i)

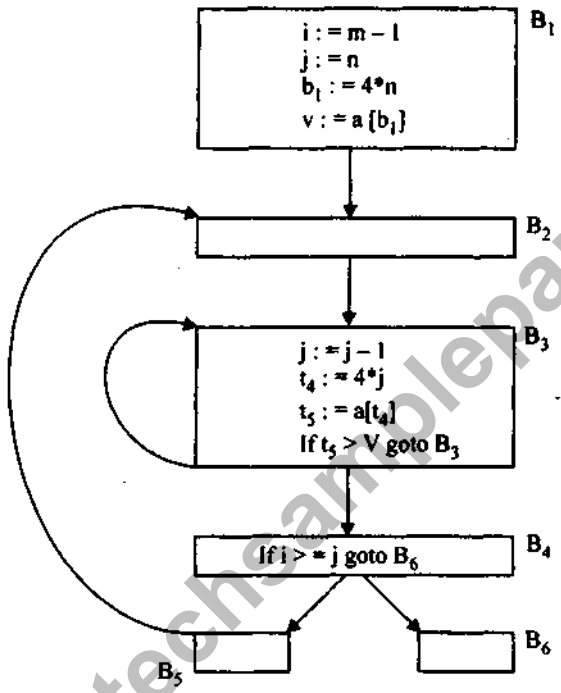
Code motion will result in the equivalent of

b = limit — 2 ...(ii)

while (i < = t) ...(iii)

Induction variables and reduction in strength.

Consider an example of a block; note that values of j and by remain in lock—step; every time the value of j decreases of 1, that of d_4 decreases by 4 because $4 * j$ is assigned to t_4. Such identifiers are called "indcution variables."



When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction variable elemination. For the inner loop around B_3, we cannot get rid of either j or t_4 completly; t_4 is used in B_3 and j in B_4. Now we can use the reduction in strength and illustrate a part of process of induction variable elemination. Evedently will be eliminated when the outer loop or $B_2 - B_5$ is considered.

As the relationship $t_4 = 4 * j$ surely holds after such an assignment to t_4 in the figure and b_4 is not changed elsewhere in the inner loop around B_3; it follows that just after the statement $j = j - 1$, the relationship $t_4 := 4 * j - 4$ must hold.

We may therefore replace the assignment $t_4 := 4 * j$ by $t_4 := t_4 - 4$. The only problem is that t_4 does not have a value when we enter block b_3 for the first time, so we place on initialization of t_4 at the end of block where j is initialized.