Ajay Kumar Garg Engineering College, Ghaziabad

Department of CSE

UT Solution- Odd Semester (2021-22)

Course	:	B.Tech.
Subject Code	:	KCS-502
Subject Name	:	Compiler Design
Semester	:	Vth
Written By	:	Mr. Jay Kant Pratap Singh ( )
Reviewed By	:	Mr. Pradeep Gupta ( )

HOD signature : Dr. Sunita Yadav Sunita 18/01/2022

PAPER ID-411134

Subject Code: KCS502

Roll No: | | | | | | | | | | | | | |

B.TECH.
(SEM V) THEORY EXAMINATION 2021-22
COMPILER DESIGN

Time: 3 Hours *Total Marks: 100*

Note: 1. Attempt all Sections. If require any missing data; then choose suitably.

SECTION A

1. Attempt *all* questions in brief. **2 x 10 = 20**

a. What is the difference between parse tree and abstract syntax tree?
b. Explain the problems associated with top-down Parser.
c. What are the various errors that may appear in compilation process?
d. What are the two types of attributes that are associated with a grammar symbol?
e. Define the terms Language Translator and compiler.
f. What is hashing? Explain.
g. What is do you mean by left factoring the grammars? Explain.
h. Define left recursion. Is the following grammar left recursive?

$E \rightarrow E+E \mid E*E \mid a \mid b$

i. What is an ambiguous grammar? Give example.
j. List down the conflicts during shift-reduce parsing.

SECTION B

2. Attempt any *three* of the following: **10 x 3 = 30**

a. Construct the LALR parsing table for the given grammar

$S \rightarrow BB$
$B \rightarrow aB / b$

b. What is an activation record? Explain how it is related with runtime storage organization?
c. Write the quadruple, triple, indirect triple for the following expression

$(x + y)*(y + z) + (x + y + z)$

d. Discuss the following terms:
   i. Basic block
   ii. Next use information
   iii. Flow graph
e. Construct predictive parse table for the following grammar.

$E \rightarrow E + T/T$
$T \rightarrow T *F/F$
$F \rightarrow F /a/b$

SECTION C

3. Attempt any *one* part of the following: **10 x 1 = 10**

a. Construct the SLR parse table for the following Grammar

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow id$

b. Differentiate between stack allocation and heap allocation.

4. Attempt any *one* part of the following: 10 x 1 = 10

a. Write syntax directed definition for a given assignment statement:

S → id=E
E → E+E
E →E*E
E→-E
E → (E)
E→id

b. What are the advantages of DAG? Explain the peephole optimization.

5. Attempt any *one* part of the following: 10 x 1 = 10

a. What do you understand by lexical phase error and syntactic error? Also suggest methods for recovery of errors.

b. Discuss how induction variables can be detected and eliminated from the given intermediate code

B2: i:= i+1
t1:=4*j
t2:=a[t1]
if t2<10 goto B2

6. Attempt any *one* part of the following: 10 x 1 = 10

a. Test whether the grammar is LL(1) or not, and construct parsing table for it.

S→1AB / ε
A→1AC/0C
B→0S
C→1

b. Distinguish between static scope and dynamic scope. Briefly explain access to non local names in static scope.

7. Attempt any *one* part of the following: 10 x 1 = 10

a. What are the various issues in design of code generator & code loop optimization?

b. Generate the three address code for the following code fragment.

```
while(a>b)
{
if(c<d)
x=y+z;
else
x=y-z;
}
```

Section A

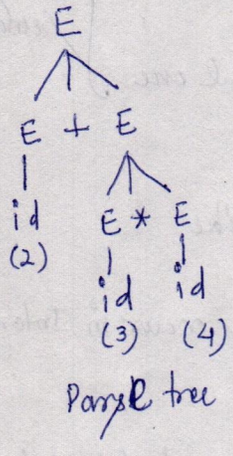
Q1 Attempt all questions in brief

a) what is the difference between parse tree and abstract syntax tree?

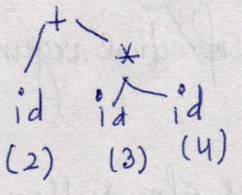
Soln:-

- Parse tree is a graphical representation of replacement process in a derivation. Each interior node represent a non terminal and each leaf node represents a terminal or operator.
- Syntax tree is condensed form of parse tree in which operators appear as interior node in the tree and not as leaves. Attribute attached to every node of syntax tree

✓ In syntax tree nonterminals are not shown.



Parse tree



abstract syntax tree

(b) Explain the problems associated with top down parser

Soln:- problems associated with the top down parsers are

a) Backtracking
b) left recursion
c) Left factoring
d) Ambiguity

(c) What are various errors that may appear in compilation process?

Soln:- In compilation process, every phase of compiler may suffer from error but

the major contribution is of 3 types of error:

a) Lexical error, (b) Syntatic error (c) Semantic errors

a) Lexical errors are :-

i) exceeding length of identifier
ii) the appearance of illegal characters
iii) unmatched string

} occurs during lexical analysis

b) syntatic errors are:

i) error in structure
ii) Missing operator
iii) Misspelled keywords
iv) unbalanced parenthesis

} occurs during syntax analysis

c) Semantic errors are :-

i) incompatible type operands
ii) undeclared variables
iii) Not matching of actual arguments with formal one

} occurs during Semantic analysis

Besides of these other phases also suffered from errors like

→ Error due to incompatibility of operands type for operator may occur in Intermediate code generation phase

→ Error occurs during control flow analysis due to some unreachable code may occur in code optimization phase

→ incompatibility with computer Architecture type error may occur in code generation phase.

d) What are the two types of attributes that are associated with grammar symbol.

Soln:- there are two types of attributes attached to the grammar symbols are

✓ synthesis attribute
✓ inherited attribute

- **synthesis attribute**

the value of variable of LHS of the production depends on the value of variables on the RHS of the production rule.

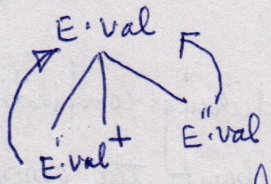
OR

the value of synthesized attribute, will be computed from attributes of its children node in parse tree.

for example

$$E \to E+E \quad (E.val = E'.val + E''.val)$$

Here E on LHS of production rule can be computed by sum of value of E' and E'' which is right hand side of production rule ie LHS variable depend on RHS variable.



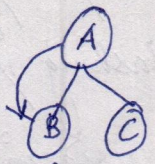
In parse tree given above, the value of E ie root node can be computed by the help of its children nodes E' and E''. Hence value is a synthesized attribute.

- **Inherited attribute** :- the value of nonterminal at RHS depends on nonterminals on LHS or nonterminals on RHS of production Rule

OR

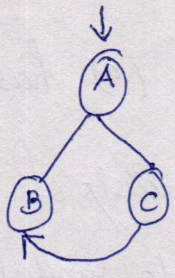
the value of inherited attribute will be computed from the value of sibling and parent nodes of a parse tree.

for ex :- $A \to BC \ \{ B.val = A.val * 5 \}$ ⟶



B.val = A.val * 5

$A \to BC \ \{ B.val = 2 * C.val \}$



B.val = 2*C.val

Q(e) Define the terms language translator and compiler.

Soln :-

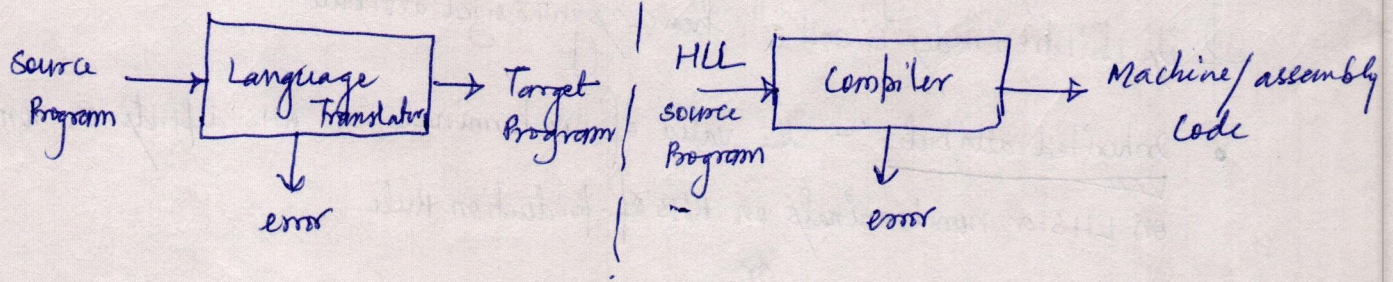
Language translator (also called language processor) is a program that translates an input program written in a programming language (called source program) into functionally equivalent program in another language (called

target program). Apart from translation, translator should have error detection capability. Any violation of source language specification would be detected and reported to the programmer. various language translators are preprocessor, compiler, Assembler etc

compiler is a program that converts source program written in high level language (HLL) into functionally equivalent machine or low level language program. (also called target program)

target program will called by the user to process the input to produce the output.

It is also noticed here, during compilation or translation process, compiler also informs the programmer about the errors present in the source program.



(f) What is hashing? explain.

Soln:- Hashing is an important technique used to search table. this method is used in symbol table organization and implemented by using two tables

(a) hash table (b) symbol table.

hash table consists of k entries from 0 to k-1. these entries are basically pointer to symbol table pointing to names of symbol table.

- to determine whether the name is in symbol table, we use a hash function h such that h(name) will result integer between 0 to k. we can search any name by position = h(name)

using this position we can obtain the exact location of name in Symbol table.

- ✓ the hash function should result in uniform distribution of names in symbol table.
- ✓ Hash function should be such that there will be minimum number of collision in hash table. collision is a situation where hash function returns in same location for string names.
- ✓ various collision resolution strategies are open hashing, chaining rehashing.

the advantage of hashing is quick search is possible and disadvantage is that hashing is complicated to implement and some extra space is required.

- ✓ type of hash function used in hashing
  1. Mid square method
  2. Division method
  3. folding method

(g) what do you mean by left factoring the grammars? Explain

Sol^{n}:- when productions contain common prefixes, it is difficult to choose a valid syntax rule while applying the production rule. Hence such situation a left factoring is used.

For example:- consider the grammar with two productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

the common prefixes α in RHS of productions in the grammar make it difficult to choose between $\alpha\beta_1$ and $\alpha\beta_2$. For expanding A. Hence grammar is left factored and production can be rewritten as

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

General form $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \text{---} | \alpha\beta_n | \gamma$

Left factored grammar

$A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 | \text{---} | \beta_n$

Note :- common prefixes at LHS also called left factor creates a problem that when we chek the string the left factor is same but right factor might not be same that we backtrack again and again.

let $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \text{---} | \alpha\beta_{100}$

so we backtrack 99 times to derive $\alpha\beta_{100}$ (say). So to over come this limitation we remove the left factor.

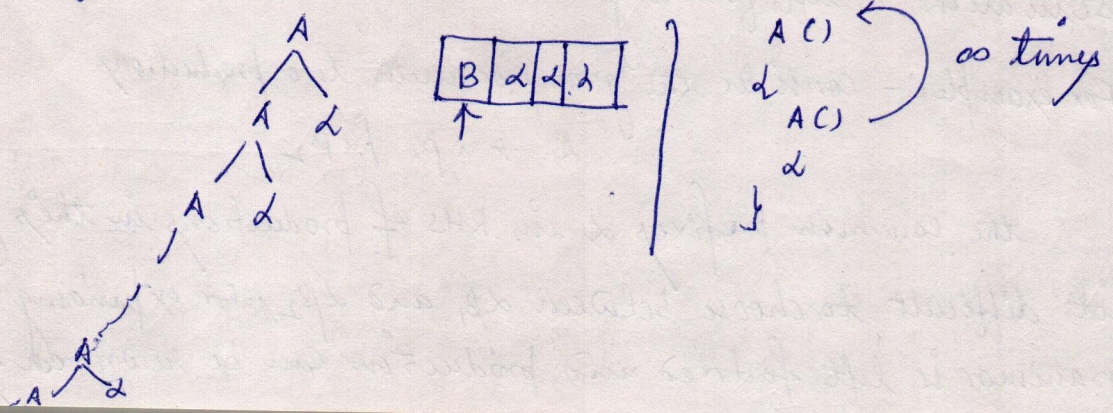
b. Define left recursion. Is the following grammar is left recursive?

$E \rightarrow E+E | E*E | a | b$

Sol^n :- A left recursive grammar is a grammar where productions of form $A \rightarrow A\alpha | \beta$.

- If left recursion present in the grammar, then topdown parser enter into infinite loop.

for ex :- $A \rightarrow A\alpha | \beta$ is grammar, in this grammar we want to generate a string $\beta\alpha\alpha\alpha$

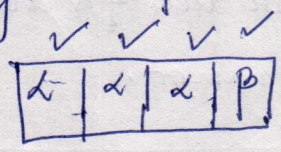


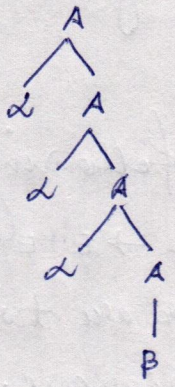
the expansion of A causes further expansion of A only due to generation of A, Aα, Aαα, Aααα ——, the input pointer will not advanced.

In order to remove left recursion we convert the left recursive grammar into right recursive grammar because right recursive grammar never enters into infinite loop during derivation of string

for ex $A \to A\alpha \mid \beta$

string → ααα β



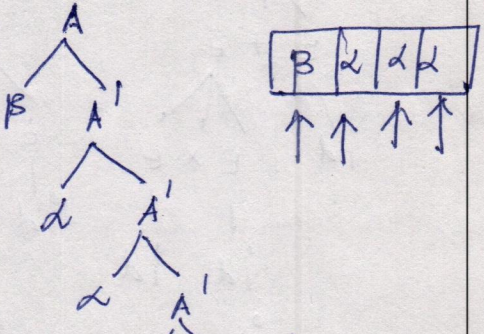


Removal of left recursion :- Left recursive grammar to right recursive grammar

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$



General form

$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$

Non left factored grammar

$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \cdots \alpha_n A' \mid \epsilon$

$E \to E+E \mid E*E \mid a \mid b$

$A \to A\alpha_1 \quad A\alpha_2 \quad \beta_1 \quad \beta_2$

so grammar is left recursive grammar because left hand side non terminal present extremely left on right hand side so production is of form $A \to A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$.

After removing left factor grammar will be

$$E \to aE' \mid bE'$$
$$E' \to +EE' \mid *EE' \mid \epsilon$$

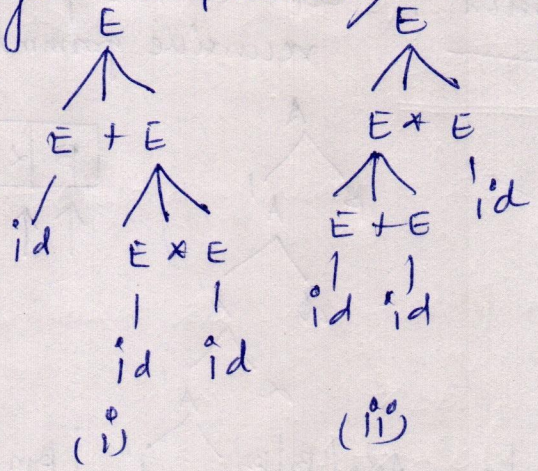
(i) What is an ambiguous grammar? Give example.

A grammar G is said to be ambiguous grammar if it generates more than one parse tree for some string belonging to the language generated by the grammar.

for example :- consider the following grammar

$$E \to E+E \mid E*E \mid (E) \mid id$$

for string id+id*id, there are two parse tree given below hence above given language Grammar is ambiguous.



Q) List down the conflict during shift reduce parsing?

there are two conflicts in shift-reduce parsing

(a) shift - reduce conflicts

(b) reduce - reduce conflicts

Shift reduce conflicts

Let us consider the grammar which have two production $A \to \beta$ and $B \to \beta a \gamma$. Let β be at top of stack and next input symbol is a. At this stage parser (S-R) has two options

(a) Reduce the handle using rule $A \to \beta$. In this case, the parser misses the other handle $B \to \beta a$ when the parsing progresses.

(b) ignore β and shift a to make the reduction using $B \to \beta a$ possible. In this case, the parser misses the handle β.

this situation is called shift reduce conflict as parser is not able to decide whether to shift a or to to reduce using $A \to \beta$ given the entire stack contents and next symbol.

2) Reduce - Reduce conflict

consider the grammar containing two production rules $A \to \alpha$ and $B \to \alpha$. If α is on the top of stack, the parser has two reduction possibilities and it is not able to decide which reduction to make. this situation is known as a reduce - reduce conflict

Section B

Q 2 :- Attempt any three of the following

(a) construct the LALR parsing table for the following grammar

$$S \to BB$$
$$B \to aB \mid b$$

Sol^{n} :-

Augment the grammar by $S' \to S$

Now closure $(S' \to \cdot S, \$)$

$I_0 = \{ S' \to \cdot S, \$,\ S \to \cdot BB, \$,\ B \to \cdot aB, a/b,\ B \to \cdot b, a/b \}$

$I_1 = \text{Goto}(I_0, S) = \{ S' \to S \cdot, \$ \}$

$I_2 = \text{Goto}(I_0, B) = \{ S \to B \cdot B, \$,\ B \to \cdot aB, \$,\ B \to \cdot b, \$ \}$

$I_3 = Goto(I_0, a) = \{B \to a \cdot B,\ a/b,\ B \to \cdot aB,\ a/b,\ B \to \cdot b,\ a/b\}$

$I_4 = Goto(I_0, b) = \{B \to b \cdot,\ a/b\}$

$I_5 = Goto(I_2, B) = \{S \to BB \cdot,\ \$\}$

$I_6 = Goto(I_2, a) = \{B \to a \cdot B,\ \$,\ B \to \cdot aB,\ \$,\ B \to \cdot b,\ \$\}$

$I_7 = Goto(I_2, b) = \{B \to b \cdot,\ \$\}$

$I_8 = Goto(I_3, B) = \{B \to aB \cdot,\ a/b\}$

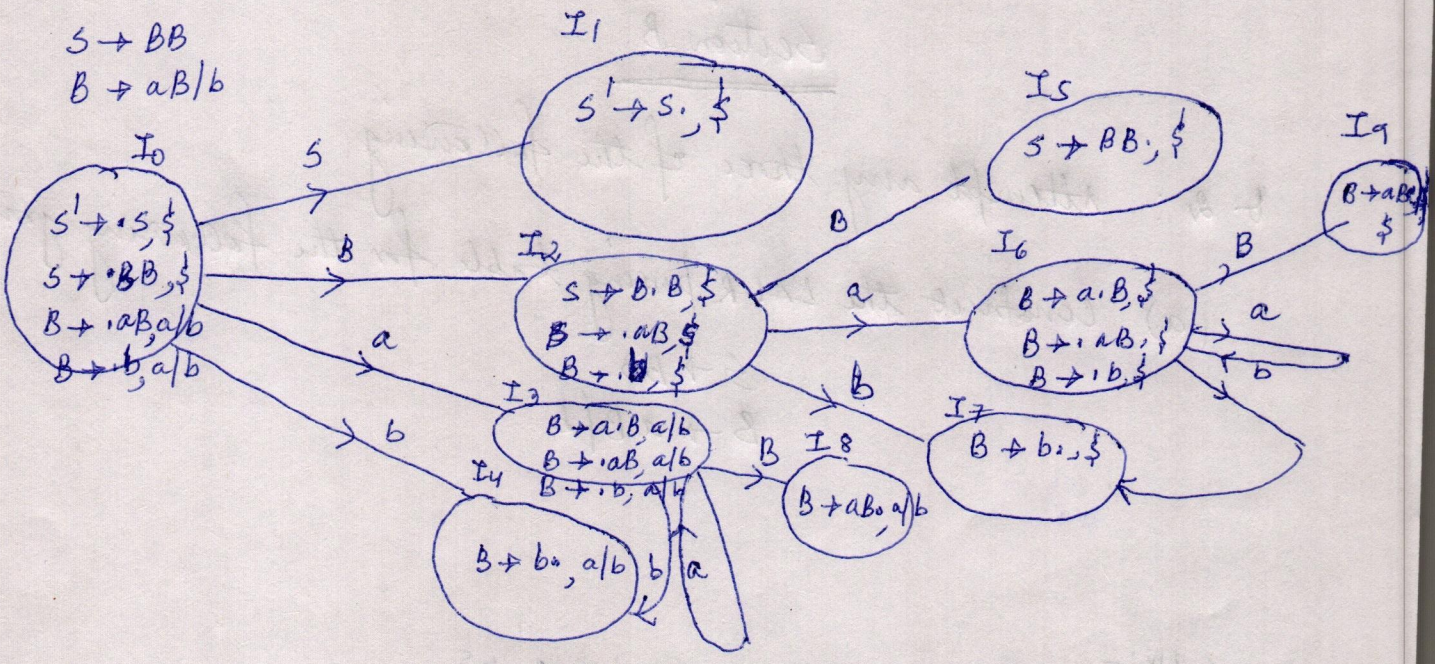
✓ $I_3 = Goto(I_3, a) = \{B \to a \cdot B,\ a/b,\ B \to \cdot aB\ a/b,\ B \to \cdot b,\ a/b\}$

✓ $I_4 = Goto(I_3, b) = \{B \to b \cdot,\ a/b\}$

$I_9 = (I_6, B) = \{B \to aB \cdot,\ \$\}$

$I_6 = (I_6, a) = \{B \to a \cdot B,\ \$,\ B \to \cdot aB/\$,\ B \to \cdot b,\ \$\}$

$I_7 = (I_6, b) = \{B \to b \cdot,\ \$\}$



Now construct the CLR parsing table

Set of LR(1) items in canonical collection represent the rows of table for example if $C = \{I_0, I_1, \ldots I_n\}$ then rows are 0 to n where ith row represent of LR(1) item sets.

	Action			Goto	
	a	b	$	S	B
0	s_3	s_4		1	2
1			accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Construction of LALR(1) parsing table

compare core part of all LR(1) items computed previously, of two or more sets containg similar first part then replace all of them with their union.

eg $I_i^\circ = A \to \alpha \cdot B\beta, a$

$I_j^\circ = A \to \alpha \cdot B\beta, b$

union

I_{ij} $A \to \alpha \cdot B\beta, a/b$

hence first the set LR(1) items can be constructed as follows with merging states

$I_0 = \{S' \to \cdot S, \$,\ S \to \cdot BB, \$,\ B \to \cdot aB, a/b,\ B \to \cdot b, a/b\}$

$I_2 = \{S \to B \cdot B, \$,\ B \to \cdot aB, \$,\ B \to \cdot b, \$\}$

$I_{36} = \{B \to a \cdot B, a/b/\$,\ B \to \cdot aB, a/b/\$,\ B \to \cdot b, a/b/\$\}$

$I_{47} = \{B \to b \cdot, a/b/\$\}$

$I_5 = \{S \to BB \cdot, \$\}$

$I_{89} = \{B \to aB \cdot, a/b/\$\}$

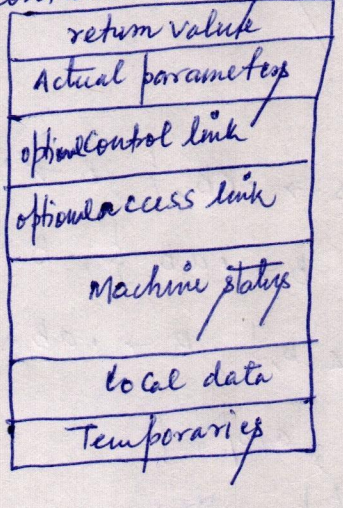
Hence parsing table

	Action			Goto	
	a	b	$	S	B
0	s_{36}	s_{47}		1	2
1			Accept		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

b) What is an activation record? Explain how it is related with runtime storage organization

Soln:- information needed by a single execution of a procedure is managed using a contiguous block of storage called activation record.

- An activation record is allocated when a procedure entered and it is dealloated when that procedure exited.
- the size of each field can be determined at compile time (although actual location of the activation record is determined at run time. A typical activation record contains various kind of data is pictorially represent as shown below



- Return value :- the return value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for return value.
- actual parameters :- the field for actual parameters is used by the calling procedure to supply parameters to the called procedure.
- optional control link points to the activation record of the caller.
- the optional access link is used to refer to non local data held in other activation records.
- saved machine status :- the field for saved machine status holds information about the state of machine before the procedure is called.
- local data field holds data that local to an execution of procedure.
- temporaries field is used to store temporaries that are generated by compiler during evaluation of long arithmatic expression.

Runtime storage organization

When procedures or functions are used by any programming language, the compiler manage part of the runtime as a stack. Each time the procedure is called, the local variables, return address and the parameters are pushed onto stack as activation record play an important role for run time storage management.

Let us see how it is managed :-

- each execution of procedure is called as activation of that procedure.
- An execution of a procedure starts at the beginning of the procedure body
- when the procedure completed, it returns the control to the point immediately after the place where the procedure is called.
- each execution of a procedure is called as its activation

- life time of an activation of a procedure is the sequence of steps b/w the first and last steps in the execution of that procedure (including the other procedure called by that procedure)
- if a and b are procedure activations, then their lifetimes are either non overlapping or are nested.
- if the procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.

Activation tree

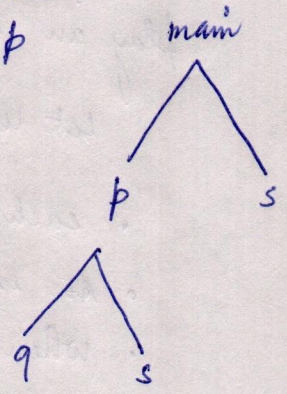
We can use a tree (called activation tree) to show the way control enters and leaves activations

- In activation trees
  - each node represents an activation of procedure
  - the root node represents the activation of the main program
  - the node a is parent of the node b iff the control flows from a to b
  - the node a is left to the node b iff the life time of a occurs before the life time of b.

for ex:-

```
program main
  procedure s
    begin ... end
  procedure p;
    procedure q
      begin ... end;
    begin q; s; end
  begin p; s; end
```

```
enter main
enter p
enter q
exit q
enter s
exit s
exit p
enter s
exit s
exit main
```



Control stack :- the flow of the control in a program corresponds to DFS of the activation tree that

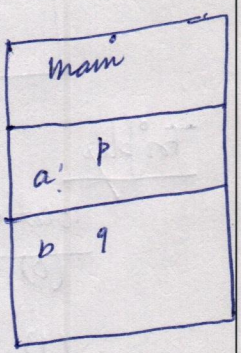
- start at root
- visit a node before its children and
- and recursively visit children at each node an a left to right order

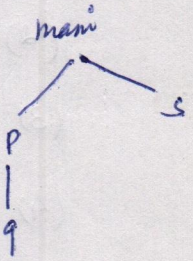
- A stack (called control stack) can be used to keep track of live procedures activations
  - An activation record is pushed onto the control stack as the activation starts.
  - the activation record is popped when that activation ends
  - when node n is at the top of stack, the stack contains the
- along the path from n to the root.

hence Activation record with control stack manage the runtime storage.

for ex :

```
Program main
Procedure p
  var a : real
  Procedure q
    var b : integer
    begin - - end
  Procedure s;
    var c : integer
    begin - - end
begin p; s; end;
```





Q(c) Write the quadruple, triple, and indirect triples for the following expression:

$$(x+y) * (y+z) + (x+y+z)$$

Soln :- we first construct a three address code for given expression

$(x+y) * (y+3) + (x+y+3)$

$t_1 = x+y$

$t_2 = y+3$

$t_3 = t_1 * t_2$

$t_4 = t_1 + 3$

$t_5 = t_3 + t_4$

Quardruple representation

location	operator	operand 1	operand 2	result
(0)	+	x	y	t_1
(1)	+	y	3	t_2
(2)	*	t_1	t_2	t_3
(3)	+	t_1	3	t_4
(4)	+	t_3	t_4	t_5

Triples

location	operator	operator 1	operator 2
(0)	+	x	y
(1)	+	y	3
(2)	*	(0)	(1)
(3)	+	(0)	3
(4)	+	(2)	(3)

Indirect triples

location	statement
0	11
1	12
2	13
3	14
4	15

location	operator	operand 1	operand 2
11	+	x	y
12	+	y	3
13	*	(11)	(12)
14	+	(11)	3
15	+	(13)	(14)

Q(a) – Discuss the following terms
(i) Basic block
ii) Next-use information
iii) flow graph

Soln :–

(i) Basic block

A basic block is a sequence of code where the control enters at top and exist at the bottom. There are no jumps into the middle of the block and control will leave the block without halting and without branching except at the last instruction in the block.

Now question is how to obtain basic blocks :

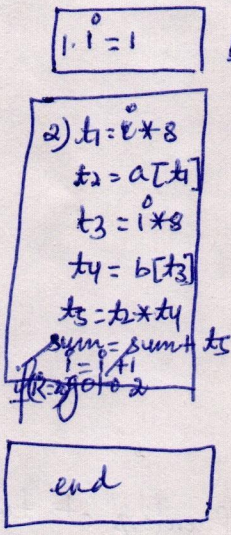
Input : A sequence of 3-Address instructions
Output : A list of basic blocks for a given sequence where each instruction is assigned to exactly one block.

Method :- Identify the leaders and obtain basic blocks as shown below :
1) Determine the set of leaders. the rules for finding out leaders are shown below :
a) the first instruction in the 3-address statements is a leader
b) Any instruction that is the target of a conditional or un conditional goto is a leader
c) Any instruction that immediately follows a conditional goto is a leader
2) All statements starting from leader up to next leader (but not including the next leader) or the end of a program represent a basic block.

for ex consider the 3 address code

1. $i = 1$ ——— L (because of first statement is leader)
2. $t_1 = i * 8$ ——— L (target of conditional jump present in line 9)
3. $t_2 = a[t_1]$
4. $t_3 = i * 8$
5. $t_4 = b[t_3]$
6. $t_5 = t_2 * t_4$
7. $sum = sum + t_5$
8. $i = i + 1$
9. if ($i <= 20$) goto 2 ——— L (it follows conditional goto)

So



B_2 Basic block

Next-to-use information

A variable that is assigned a value in a block and if that variable is used in subsequent statements in the same block and its value has never changed then it is said to have next-use information.

Observe the following point

- ✓ To generate good code, it is necessary to know when the value of a variable will be used next in block.
- ✓ If the value of a variable present in a register is never used subsequently then that register can be assigned to another variable
- ✓ So, it is necessary to know the next-use information of a variable.

Now how can be find out next to use information.

Algorithm

Input :- A basic block B of 3-address code. initially all non temporary variables are live in symbol table.

Output :- At each statement i: $x = y + z$ in block. we attach liveness to i and next to use information of x, y, z.

Procedure :- Start scaning from Last statement of the block B backwards to the start

1. Attach the information currently found in the symbol table ~~to start of block~~ regarding the next to use and liveness of x, y, z to statement i.
2. In symbol table, the x is set to "dead" (notlive) and "no next-use"

step3:- In the symbol table set y and z to live and next-use of y and z to statement i

Note:- the symbol + can be replaced by any operator.

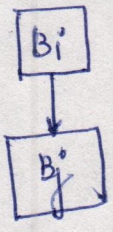
flow graph

A flow graph is a directed graph that shows how the control flows from one basic block to other basic block. A node in the flow graph is a basic block that represents various computations and the edge from one basic block to other basic block represent the flow of control.

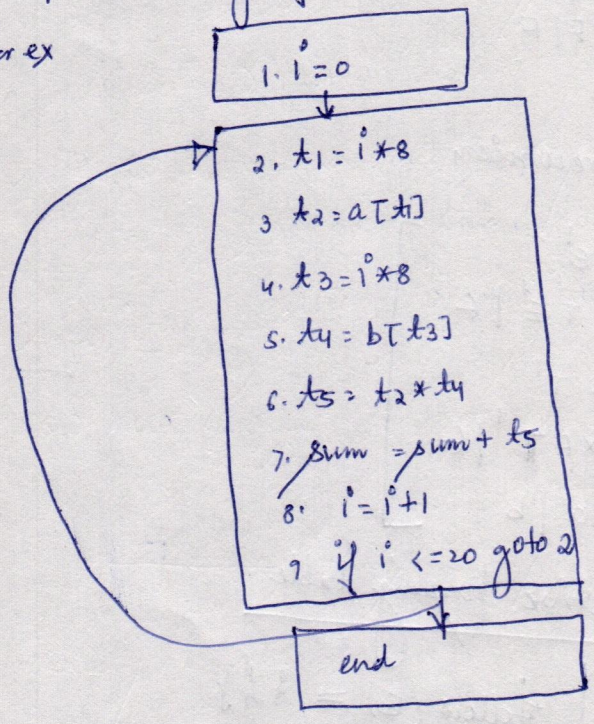
how to obtain the flow graph from the basic blocks??

the flow graph can easily obtained using basic blocks as shown below:

During execution, after executing block B_i if block B_j is executed, then place an edge from block B_i to block B_j



for ex



flow graph

Q:- e) construct predictive parse table for the following grammar.

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow F/a/b$

Soln:-

After removing left recursion

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE' \mid \epsilon \\
&T \to FT' \\
&T' \to *FT' \mid \epsilon \\
&F \to aF' \mid bF' \\
&F' \to F' \mid \epsilon
\end{aligned}
$$

we get solution as above given: here $F' \to F'$ is unit production so remove unit production

Another approach is to firstly remove unit production from the given grammar ~~and~~ then resultant grammar is

$$
\begin{aligned}
&E \to E+T \mid T \\
&T \to T*F \mid F \\
&F \to a \mid b
\end{aligned}
$$

Now remove the left recursion

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE' \mid \epsilon \\
&T \to FT' \\
&T' \to *FT' \mid \epsilon \\
&F \to a \mid b
\end{aligned}
$$

Now we construct predictive parsing table

first(E) = {a, b}
first(T) = {a, b}
First(F) = {a, b}
first(E') = {+, ε}
first(T') = {*, ε}

follow(E) = {$}
follow(T) = {+, $}
follow(F) = {*, +, $}
follow(E') = {$}
follow(T') = {+, $}

~~follow~~

$V = \{E, E', T, T', F\}$

$\Sigma = \{+, *, a, b, \$\}$

V/Σ	+	*	a	b	$
E	.		$E \to TE'$	$E \to TE'$	
E'	$E' \to +TE'$				$E' \to \epsilon$
T			$T \to FT'$	$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$			$T' \to \epsilon$
F			$F \to a$	$F \to b$	

Section C

Q-3

a) construct the SLR parsing table for the following grammar

$E \to E+E$

$E \to E*E$

$E \to id$

Sol^n :- Augment the grammar

$E' \to E$

$E \to E+E$

$E \to E*E$

$E \to id$

Now find DFA for LR(0) item sets

$I_0 = \text{closure}(E' \to \cdot E) = \{E' \to \cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

$I_1 = \text{goto}(I_0, E) = \{E' \to E\cdot,\ E \to E\cdot+E,\ E \to E\cdot*E\}$

$I_2 = \text{goto}(I_0, id) = \{E \to id\cdot\}$

$I_3 = \text{goto}(I_1, +) = \{E \to E+\cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

$I_4 = \text{goto}(I_1, *) = \{E \to E*\cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

I_5 Goto$(I_3, E) = \{E \to E+E\cdot,\ E \to E\cdot+E,\ E \to E\cdot*E\}$

I_2 = Goto$(I_3, id) = \{E \to id\cdot\}$

I_6 = Goto$(I_4, E) = \{E \to E*E\cdot,\ E \to E\cdot+E,\ E \to E\cdot*E\}$

I_2 = Goto$(I_4, id) = \{E \to id\cdot\}$

I_3 = Goto$(I_5, +) = \{E \to E+\cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

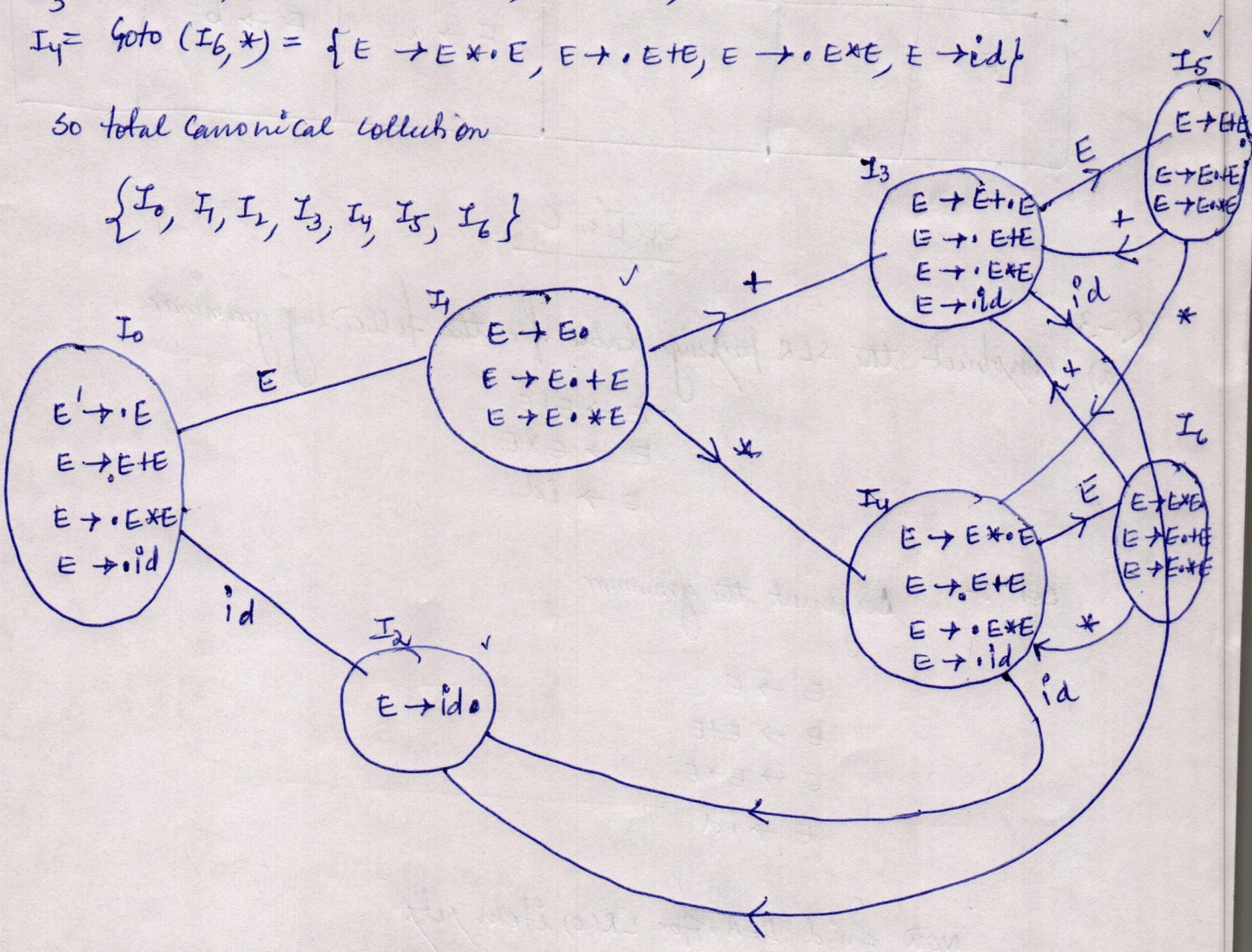
I_4 = Goto$(I_5, *) = \{E \to E*\cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

I_3 = Goto$(I_6, +) = \{E \to E+\cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

I_4 = Goto$(I_6, *) = \{E \to E*\cdot E,\ E \to \cdot E+E,\ E \to \cdot E*E,\ E \to \cdot id\}$

So total canonical collection

$\{I_0, I_1, I_2, I_3, I_4, I_5, I_6\}$



Now parsing table

	Action				goto
	+	*	id	$	E
0			s_2		1
1	s_3	s_4		accept	
2 ✓	r_3	r_3		r_3	
3			s_2		5
4			s_2		6
5 ✓	s_3/r_1	s_4/r_1		r_1	
6	s_3/r_2	s_4/r_2		r_2	

$E \to E+E$ (1)
$E \to E*E$ (2)
$E \to id$ (3)

$E \to id. \to r_3$

follow(E) = {*, +, $}

$E \to E+E. \to r_1$

$E \to E*E. \to r_2$

. So SLR parsing table for given grammar is as above

Q3(b) Differentiate between stack allocation and heap allocation.

Both the dynamic storage allocation techniques.

i) stack allocation :-

Stack allocation is one of the dynamic storage allocation where the space is reserved on the stack as and when required ie when program is executing this allocation technique is used when the storage requirements are not known at compile time. Stack based allocation is normally used in C/C++, Pascal programming language for local variables in the procedures and procedure call information. the stack storage helps in execution of recursive procedures. Now observe the following points wrt stack storage.

- All the compilers that use procedures or functions manage at least part of their run time memory stack.
- each time a procedure is called, space for its local variables is pushed onto the stack and when the procedure is terminated

that space is popped of the stack.

- All the above activities are done with the help of activation trees, activation records and with the help of calling sequence.

ii) Heap allocation

Heap allocation is one of the dynamic storage allocation where the space is reserved on the heap as and when required during run time ie when the program is being executed. this is the most flexible allocation scheme since storage can be allocated and deallocated dynamically at arbitrary times during program execution. This will be more expensive than their stack based allocation, Now observe the following points wrt heap allocation

- To support heap management "Garbage collection" enables the runtime system to detect useless data elements and reuse their storage even if the programmer does not return the space explicitly
- Automatic garbage collection is essential feature of many modern languages.

the major diff is as illustrate below:

Stack allocation	Heap allocation
1) the amount of memory allocated during runtime is fixed at compile time.	1) the amount of memory to be allocated during runtime is decided and allocated during runtime
2) the size of memory to be allocated is fixed during compile time and cannot be altered during execution time	2) As and when memory is required, memory can be allocated. If not required memory can be deallocated. the size of memory required may vary during execution.

3. used only when the data size is fixed and known in advance before processing	3. used only for unpredictable memory requirement.
4. execution is faster, since the necessary instructions to allocate memory and deallocate memory are generated during compile time itself.	4. execution is slower since memory has to be allocated and deallocated during run time and consumes time
5. Memory is allocated either in stack area (for local variables) or data area (for global and static variables)	5. Memory is allocated only in heap area.
6. recursion can be implemented by help of stack allocation	6. Dynamic ~~allo~~ data structures are implemented using heap allocation.

Q4:- (a) write syntax directed ~~trans~~ definition for a given assignment statement

$S \rightarrow id = E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow id$

Sol^n:- translation of E can have two attribute

1) E.code: is a sequence of three address code that represent/evaluate E.

2) E.place: it will tell about name that will hold the value of expression.

Production	Semantic action
(1) $S \rightarrow id = E$	$S.code = E.code \,\|\|\, gen(id.place = E.place)$
2) $E \rightarrow E + E$	$E.place = newtemp()$ $E.code = E_1.code \,\|\|\, E_2.code \,\|\|\, gen(E.place = E_1.place + E_2.place)$
3) $E \rightarrow E * E$	$E.place = newtemp()$ $E.code = E_1.code \,\|\|\, E_2.code \,\|\|\, gen(E.place = E_1.place * E_2.place)$
4) $E \rightarrow -E$	$E.place = newtemp()$ $E.code = E_1.code \,\|\|\, gen(E.place = -E_1.place)$
5) $E \rightarrow (E_1)$	$E.place = E_1.place$ $E.code = E_1.code$
6) $E \rightarrow id$	$E.place = id.place$ $E.code = ''$

the synthesized attribute S.code represent the three address code for assignment E has two attributes, namely E.place and E.code and S and id have only one attribute each S.code and id.place respectively. Note that attribute E.place designates the name that consists the value of E. while E.code represent the sequence of three address statements that evaluates E. the function newtemp() returns a new tempo -rary storage, while function gen() is used to generate the the required three address code. the operator || denotes the concatenation that join three address code.

Q4 (b) What are the advantages of DAG? explain the peephole optimization. (14)

the advantages of DAG

> Determing common subexpression
> Determining which identifier are used inside the block and computed outside the block.
> Determining which statement of the block could have their computed value used outside the block.
> Dead code elimination

Peephole optimization

this technique is applied to improve the performance of program by examining a short sequence of instructions in a window (peephole) & replace the instructions by a faster or short sequence of instructions

peephole optimization technique involve the following steps:

i. Redundant instruction elimination
ii. Removal of unreachable code
iii. flow of control optimization
iv. Algebraic simplification
v. Machine idioms

1. Redundent instructions elimination

consider the following instructions

MOV R0, a
MOV a, R0 } MOV R0, a

we can eliminate 2'nd instruction since 'a' is already in R0

2. Removal of unreachable code :- Remove statements which are unreachable (never executed)

```
    MOV #100, R0
    BRA L1
    MUL R0, R1   } unreachable
    ADD #200, R1 }
L1: MOV R0, R2
```

3) flow of control optimization — unnecessary jumps can be eliminated

```
        goto L1
        - - -
L1 :    goto L2
        . . .
L2 :    goto L3
        . . .
L3 :    MOV a, R0
```

multiple jumps can make the code inefficient. Above code can be replaced by

```
        goto L3
        - - -
L1 :    goto L3
        . . .
L2 :    goto L3
        . . .
L3 :    MOV a, R0
```

4) Algebraic simplification

eg $x = x + 0$ or $x = x * 1$] compile time removal
$x = x / 1$ $x = x - 0$]

- above statements can be eliminated because of executing those statements the result 'x' would not change.

- strength reduction

$a = x \wedge 2$ replaced by $a = x * x$

$b = y / 8$ " " $b = y >> 3$ ↑ right shift

5) use of machine idioms — (make use of architectural techniques)

* is a process of using powerful features of CPU instructions

for ex:- Auto INC / DEC features can be used in m/c / dic vari-ables

$a = a + 1 \Rightarrow INC(a)$

$a = a - 1 \Rightarrow DEC(a)$

(5)(a) What do you understand by lexical phase error and syntatic error? Also suggest metho (15)
-ds for recovery of errors.

Sol^{n}:-

1) Lexical phase errors

these errors are detected during the lexical analysis phase. Typical lexical errors are:

- Exceeding length of identifier or numeric constants
- the appearance of illegal characters
- unmatched string

ex 1:- printf("Geeks"); $
this is an lexical error since an illegal character $ appear at the end of statement.

2. this is comment */
lexical error because begining of comment is not present.

Error Recovery for lexical errors

Panic mode recovery

- In this method successive characters from input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as ; or }
- the advantage is that it is easy to implement and guarantees not to go into an infinite loop.
- the disadvantage is that a considerable amount of input is skipped without checking it for additional errors.

2) Syntatic phase errors:-
these errors are detected during the syntax analysis phase, the typical syntax errors are:

- error in structure
- Missing operator
- Miss-spelled keywords
- unbalanad parenthesis

for eg switche(ch)
{
:
}

the keyword is incorrectly written as a switche. Hence syntax error "unidentified keyword/identifier"

Error Recovery for syntatic phase recovery

1. Panic mode recovery

In this method, successive characters from input are removed one at a time until a designated set of synchronizing token is found. Panic mode correction often skip a considerable amount of input data without checking for additional errors, and it give the guarantee not to go into infinite loop.

for ex: a=b+c
d=e+f; } By using panic mode it skip a=b+c without checking the error in the code.

2. Phrase-level recovery:- When the parser ditects an error the parser may perform local correction on the remaining input. It may replace a prefix of the remaining input by some string that allows parser to continue.

- A typical local correction is done by phrase-level recovery parser by replacing comma by semicolon, delete an extraneous semicolon or inserting a missing semicolon etc.

for ex:- consider the code; while(x>0) y=a+b; In this code local correction is done by phrase-level recovery by adding 'do' and parsing is continue.

3. Error production :- If error production is used in parser, we can generate appropriate error message and parsing is continue. It takes the advantage of common errors that can be encountered then these errors can be incorporated by augmenting the grammar with error productions. for ex let us consider the grammar; E → +E/*E| *A|A, A → E

When error production *A encountered, it send an error message to the user asking the use '*' as unary or not.

4. Global correction:- the parser examines the whole program and tries to find out the closest match for it which is error free.

- the closet match program has less number of insertions, deletions and change of tokens to recover from erroneous input.
- Due to high time and space complexity, this method is not implemented practically.

Q5 (b) Discuss how induction variables can be detected and eliminate from the given intermediate code.

B_2: i := i+1
t1 := 4*j
t2 := a[t1]
if t2 < 10 goto B_2

Sol^{n}:- this is used in loop optimization. Induction variables are the variables that gets incremented or decremented by fixed amount in every iteration of loop.

for example

```
for(i=0; i<10; i++)
{ j = 17*i          i and j are induction variables
}
```

A technique to recognize the existance of induction variables and replace them with simpler computation is also required and called as strength reduction.

for example :- in above loop * has higher strength so convert it into + is called strength reduction

```
temp = 0
for(i=0; i<10; i++)
{ j = temp
  temp = temp+17;
}
```

In order to eliminate induction variables following steps need to perform:

1) find all induction variable by scanning the statements of the loops
2) find any additional induction variables and for such additional induction variable A, find the family of some basic induction variable B to which A belongs. A may be of following form

A = B*C
A = C*B
A = B/C
A = B±C

} where C is loop constant and B is basic induction variable and A is family of B

in the question

B2: $i = i + 1$

$t_1 = 4 * i$

$t_2 = a[t_1]$

if $t_2 < 10$ goto B_2

$t_1 = 4 * i$

t_1 is additional induction variables induced from i so i is basic induction variable and t_1 is additional induction variable belong to the class of i. So we have to remove i, t_1

So $i = 0, 1, 2, 3, 4 \ldots\ldots$

$t_1 = 4, 8, 12, 16, 20 \ldots\ldots$

As we know t_1 is being used in further computations so we can easily remove i.

strength reduction

we have to replace $4 * i$ by $t_1 + 4$ addition so it will take less time for execution: then we eliminate i.

B_2: $t_1 = t_1 + 4$

$t_2 = a[t_1]$

if $t_2 < 10$ goto B_2

Q6(a) Test whether the grammar is LL(1) or not and construct the parsing table for it

$S \rightarrow 1AB \mid \epsilon$

$A \rightarrow 1AC \mid 0C$

$B \rightarrow 0S$

$C \rightarrow 1$

Sol^{n}:-

to see that LL(1) grammar we have to check following conditions

1) grammar must be unambiguous, nonleft-recursive and left-factored can be LL(1) grammar

2) A single production grammar is LL(1)

3) if production are from (sufficient condition)

$$A \to \alpha_1 | \alpha_2$$

then $first(\alpha_1) \cap first(\alpha_2) = \phi$ then grammar is LL(1)

4) if any one of first ie $first(\alpha_1)$ or $first(\alpha_2)$ contain ϵ then

(i) if $first(\alpha_2)$ contain ϵ then $first(\alpha_1) \cap follow(A) = \phi$

(ii) if $first(\alpha_1)$ contain ϵ then $follow(A) \cap first(\alpha_2) = \phi$

Now check grammar

$S \to 1AB | \epsilon$ $\quad first(1AB) = \{1\}$ $\quad first(\epsilon) = \{\epsilon\}$ so find out $follow(S) = \{\ \}$

$first(1AB) \cap follow(S) = \{1\} \cap \{\ \} = \phi$

so LL(1)

Now

$A \to 1AC | 0C$ $\quad first(1AC) = \{1\}$ so $first(1AC) \cap first(0C) = \phi$

$first(0C) = \{0\}$

hence LL(1)

$B \to 0S$ is single production so LL(1)

$C \to 1$ is single production hence LL(1)

so all productions are in LL(1) hence grammar is LL(1).

Now construct the predictive parsing table

V/Σ	1	0	$
S	S→1AB		S→ϵ
A	A→1AC	A→0C	
B		B→0S	
C	C→1		

there is no conflict in table hence it is also depicted that grammar is LL(1)

Q6 b) Distinguish between static scope and dynamic scope. Briefly explain access to non-local names in static scope.

Soln:- the scope of a variable is the region of the program in which variable is declared and used. one reason of scoping is to keep variables in the different parts of program distinct from one another.

on the basis of scoping variables are divided into two types

1) Local variables :- A variable is local in program unit or block if it is declared in same program or block.

2) Non local variables :- A variable is nonlocal if it is visible with in program unit or block but not declared there.

so A scope of a variable is the range of statements in which the variable is visible. A variable is visible in a statement if it is referenced in that statement.

Scope of a variable is generally divided into two types : (1) static scope (2) dynamic scope

- static scope (lexical scope) refers to scope of a variable is defined at compile time itself, i.e. when the code is compiled a variable to be bounded to some program unit or block.
- Dynamic scope refers to scope of a variable is defined at runtime than at compile time. Perl, Javascript use dynamic scoping.

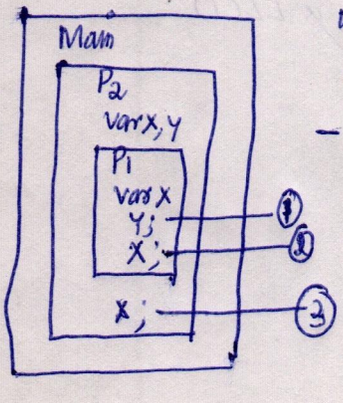
In static scope, the scope of a variable can be determined statically - by looking at the program prior to execution.

Searching process of nonlocal variables in static scope

Search declarations, first locally, then in increasingly larger enclosing unit, until one is found for a given name. If variable declaration is not found after searching largest enclosing unit then "undeclared variable error"

- enclosing static unit or scopes are called static ancestors, the nearest static ancestor is called static Parent.

← main is the static parent of P_1 and P_2. P_1 is static parent of P_1 main, P_2 are static ancestor of P_1.



If a variable 'Y' is ~~declared~~ used in P_1 but not declared there then search for static parent of P_1 i.e P_2 and see whether Y is declared there or not if found then used that defintion of Y in P_2 otherwise search for main in this manner continue till last

Q7(a) What are the various issues in design of code generator & code loop optimization?

Soln:- Code generation phase converts source program into machine code. A code generator is expected to generate correct code. the following issues arises during the code generation phase

i) 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 intermediate representation. this intermediate code can be represented using: i) 3-address code such as quadruples triples, indirect triples ii) linear representation like postfix notation iii) Graphical representation such as syntax trees or DAGs

ii) Target Program :- the architecture of target machine has a significant impact on the design of code generator. the most common architectures includes CISC & RISC. Another issue to be considered is whether to produce absolute machine language, or relocatable machine code, or assembly code. If absolute machine code is produced, it can directly placed into memory at fixed locations and can be executed immediately, if relocatable m/c code is produced then all the subprograms have separately compiled and then they are linked together and loaded for execution by linker & loader. if assembly code is produced a assembler is needed to convert assembly code to machine code.

iii) Memory Management Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in three address statements refers to the symbol table entry for name. then from the symbol table entry a relative address can be determined for name.

iv) Instruction selection :- using the intermediate representation of code, a series of machine instructions have to be generated that can be executed by the target program, of the instruction is represented using high level language, then it has to be translated into series of machine instructions. But, if the instructions are represented using low level, then for every instruction in the intermediate representation, we may have to write an equivalent machine instruction. the various point to be considered are on instruction selection, 1) the nature of instruction set of the target machine has a strong effect on the difficulties 2) instruction speeds 3) the quality of code generated may vary.

v) Register allocation In code generation, the key problem is deciding what values to hold in what registers. the values that may not hold in registers must resides in memory the instructions involving register operands are shorter and faster than those involving operands in memory the use of register divided into two subproblems
- Register allocation: we select set of variables that will reside in register at each point in program.
- Register assignment: here, we select the specific register to hold the value of variable

vi) Evaluation order the order in which computations are performed can affect the efficiency of the target code generated. some assumption orders require fewer register to hold intermediate results than others but picking best order difficult.

Q7b Generate the three address code for following code fragment

```
while (a>b)
{ if(c<d)
  x = y+z;
  else
  x = y-z;
}
```

Soln :-

1. if (a>b) then goto 3
2. goto 10
3. if (c<d) goto 7
4. $t_1 = y-z$
5. $x = t_1$
6. goto 1
7. $t_2 = y+z$
8. $x = t_2$
9. goto 1
10. end

———* ———* ———