Scheme of Solution

SRN										

PES University, Bangalore

UE16CS351

(Established under Karnataka Act No. 16 of 2013)

END SEMESTER ASSESSMENT (ESA) - B.TECH VI SEMESTER – May 2019

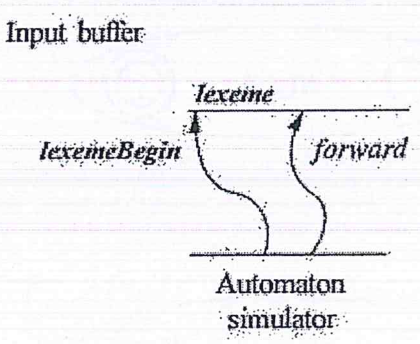
UE16CS351 - COMPILER DESIGN

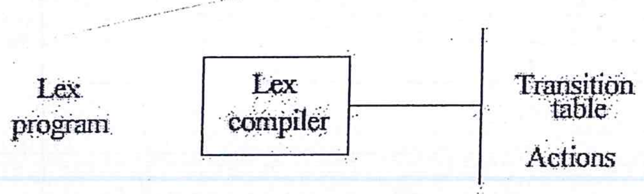
Time: 3 Hrs | Answer All Questions | Max Marks: 100

1	a	Explain the working(design) of a lexer with a neat diagram. Clearly specify how a lexeme is identified.	5

Solution:

The program that serves as the lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.





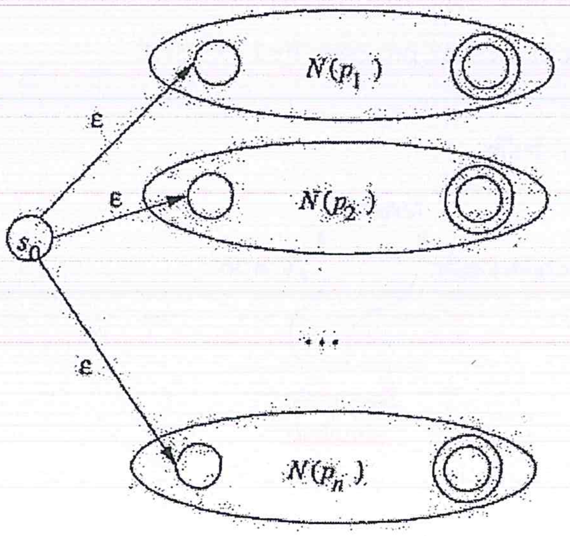
These components are:

A transition table for the automaton.

SRN

Those functions that are passed directly through Lex to the output. The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

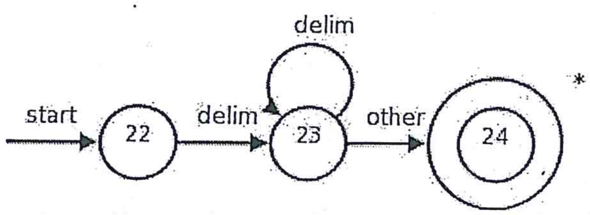
To construct the automaton, we begin by taking each regular-expression pattern in the Lex program and converting it to an NFA. We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with e-transitions to each of the start states of the NFA's *N{* for pattern *pi.*



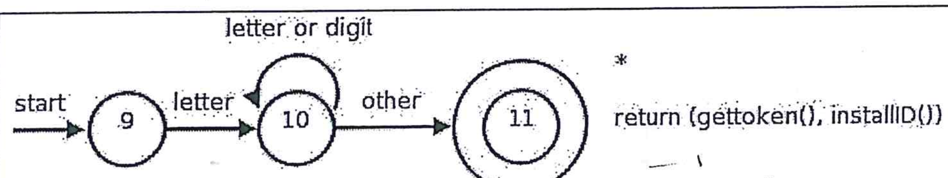
b Construct a Transition diagram for : 5

(2.5+2.5)

a) Whitespaces
b) identifiers



SRN



2 0.5

c Mention two major disadvantages of Recursive descent parser with backtracking. **5**

Solution:

a) Parser enters into infinite loop in case the grammar is left recursive
Example : A -> Aa | b

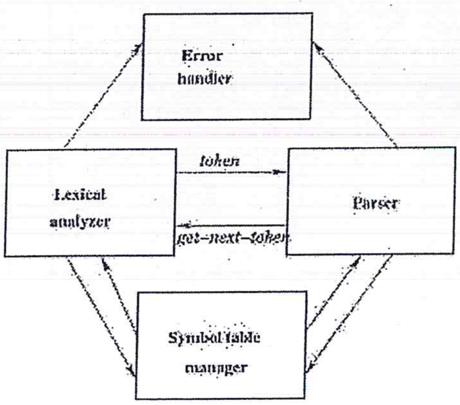
b) The alternatives of a Non Terminal are tried in the way they are listed due to which few strings which belong to the L(G) are rejected by the Parser.
Example : S -> cAd
A -> a | ab
cabd belongs to L(G) but since A -> a is listed first, A() returns true to S() on seeing an a and hence b does not match with d due which parser declares that string is not accepted

d Explain with a diagram the interaction between Scanner and a Parser. **5**

Provide an example for: (3 + 2)

I. A Lexical Error
II. A Parser Error 2 1 1

Example :



A lexical Error : Anything for which a pattern is not defined. A garbage character.
A Parser Error : missing paranthesis or a semicolon

2 a Consider the following grammar: **10 (5 + 5)**

A → B b | a
B → c A | ϵ

(a) Construct a LL(1) parser for this grammar.
(b) Show the steps taken by your LL(1) parser to parse the input "cbb".

Solution:

SRN										

(a) Calculate FIRST and FOLLOW for A, B:

Nonterminals	FIRST	FOLLOW
A	{ a ; c, b }	{ b, $ }
B	{ c, ϵ }	{ b }

(b) Construct a LL(1) parser for this grammar.

	a	b	c	$
A	A → a	A → Bb	A → Bb	
B		B → ϵ	B → cA	

(c) Show the steps taken by your LL(1) parser to parse the input "cbb".

Stack	Input
$ A	c b b $
$ bB	c b b $
$ bAc	c b b $
$ bA	b b $
$ bbB	b b $
$ bb	b b $
$ b	b $
$	$

b Shift-reduce parsing : 5

Given the following ACTION/GOTO table, show the parse if the current stack contents are 0 b 4 B 1 (i.e., current state is 1) and the remaining input is "d$".

State	ACTION			GOTO	
	d	e	$	S	B
0	Shift 2	Reduce S → Bb	Reduce S → bB	2	1
1	Shift 3	Reduce B → d	Accept	3	4
2	Shift 1	Shift 4	Accept	2	4
3	Reduce S → Bd	Reduce B → d	Reduce S → bBd	1	2
4	Reduce B → e	Shift 3	Reduce S → bSe	4	3

Solution:

SRN

Stack	Input	Action
0 b 4 B 1	d $	Shift 3
0 b 4 B 1 d 3	$	Reduce S → bBd
0 S	$	Goto 2
0 S 2	$	Accept

c Consider the following LR(1) items in a single state of a shift/reduce parser. List any conflicts that exist and describe for what lookaheads they occur. 5

[A → a • bAb, d]
[B → a • bc, c]
[C → da • da, a]
[D → aa • d, c]
[E → ca • c, a]
[F → aAa • , a]
[G → ba • , a]
[H → Ha • , c]

Solution:
Shift/reduce conflict: [E -> ca •.c, a] and [H -> Ha • , c] for lookahead c (3)
Reduce/reduce conflict: [F -> aAa •, a] and [G -> ba • , a] for lookahead a (2)

3 a Write appropriate rules to construct a Syntax tree for the following set of statements: 5

P -> S
S -> if (C) S | id = num
C -> id

Solution:
P -> S { P.node = S.node;}
S -> if(C) S { S.node = new Node ("if", C.node, S1.node)}
S -> id = num {S.node = new Node("=" , new Leaf(id, id.lexval), new Leaf(num, num.lexval)}
C -> id {C.node = new Leaf(id, id.lexval)}

b Write appropriate rules to generate 3-address code for the following statements: 5

S -> id = E
E -> E + T | T
T -> id

Solution:
S -> id = E {S.code = E.code || gen(id.lexval "=" E.addr)}

E -> E + T {E.addr = new Temp(); E.code = E1.code || T.code || gen(E.addr "=" E1.addr "+" T.addr);}

E -> T {E.addr = T.addr; E.code = T.code;}
T -> id {T.addr = id.lexval; T.code = " ";}

SRN

c

Append appropriate rules to the given CFG to count the number of occurrences of the substring ab in a string made up of a's and b's.

10 (5 + 5)

S -> Sa | Sb | a | b

For example the string aabbaba contains two occurrence of ab, whereas the string baaaba contains only one occurrence of ab.

Justify the working of your rules by carrying out the SDD over the input string : aabbaba

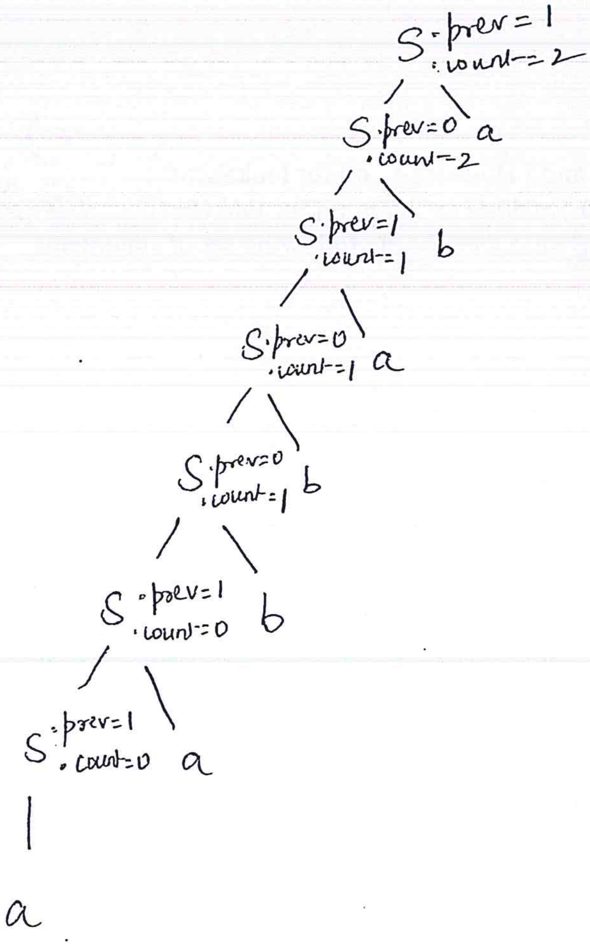
Solution:

S -> S1 a {S.prev = 1; S.count = S1. count;}

S -> S1 b {if (S1.prev == 1){S.count = S1.count + 1; S.prev = 0;} else {S.prev = 0; S.count = S1.count }

S -> a {S.prev = 1; S.count = 0;}

S -> b {S.prev = 0; S.count = 0;}



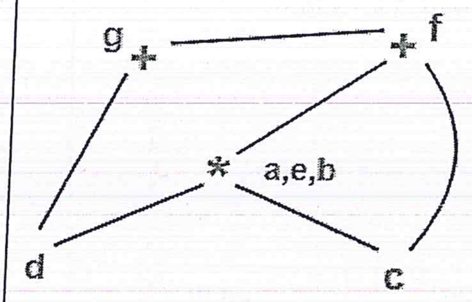
SRN										

<table>
<tr><td>4</td><td>a</td><td>Locally optimize the following block by constructing a DAG for it:

a = b * c
d = b
e = d x c
b = e
f = b + c
g = f + d

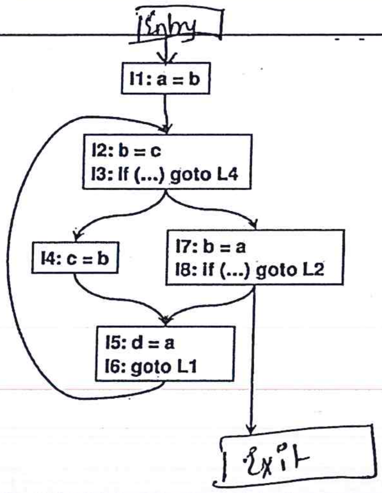
Solution-

Directed Acyclic Graph for the given block is-



Directed Acyclic Graph</td><td>5</td></tr>
<tr><td></td><td>b</td><td>Construct a Control flow graph for the given intermediate code :
(Note : Also introduce Entry and Exit nodes)
a := b
L1: b := c
if (...) goto L4
c := b
L2: d := a
goto L1
L4: b := a
if (...) goto L2
Solution:</td><td>5</td></tr>
</table>

SRN										

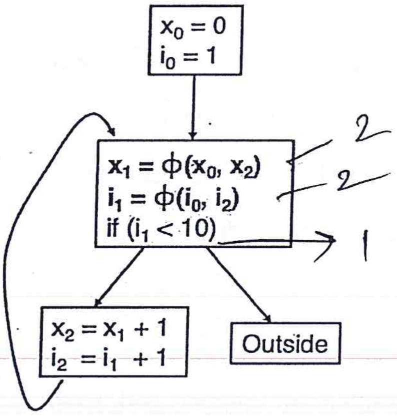


c	Provide quadruple representation for the following intermediate code: a) a = 5 b) b = a * 5 c) Label L1 d) x[t3] = 3 e) goto L2	5

op	arg1	arg2	result
=	5		a
*	a	5	b
Label			L_1
[ ] =	x	t_3	3
goto			L_2

d	Convert the following code to SSA form:
	x = 0
	i = 1
	while (i<10)
	{
	x = x+i
	i = i+1
	}
	Solution:
		5

SRN										



5	a	Construct target code for the following procedures assuming stack allocation: main() x = y + 2 call foo foo() z = x * x return Assume the Stack area starts at location 800. The code area for main and foo starts at address 100 and 400 respectively. The Activation record size of main() and foo() is 60 bytes and 30 bytes respectively. Solution:	05

```
// code for main()
100: MOV SP, #800      — 1
112: SUB SP, SP, #60
128: LD R1, y
140: ADD R1, R1, #2
156: ST x, R1
168: SUB SP, SP, #30   — 0.5
184: ST 0(SP), #204    — 1
196: BR 400            — 1
204: ADD SP, SP, #30   — 0.5
220: ADD SP, SP, #60

// code for foo()
400: LD R1, x
412: MUL R1, R1, R1
428: ST z, R1
440: BR 0(SP)          — 1
```

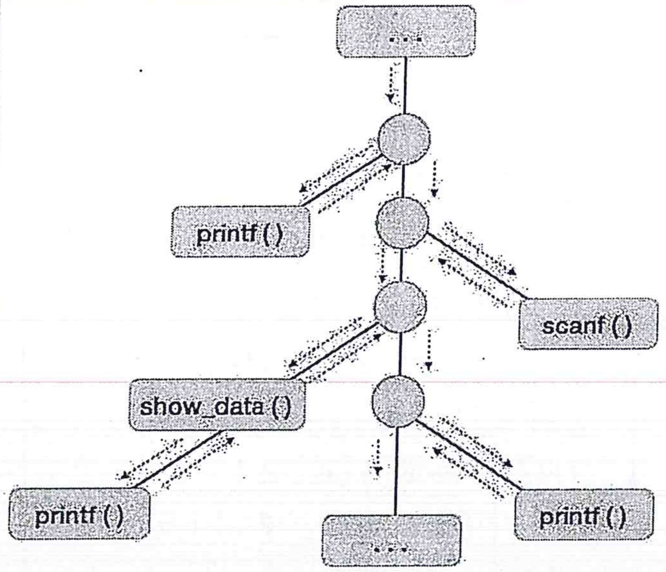
SRN										

	b	What is an activation tree? Explain with an example. Solution: A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure. We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.	5 (2+3)

```
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue…");
. . .
int show_data(char *user)
  {
  printf("Your name is %s", username);
  return 0;
  }
. . .
```

Below is the activation tree of the code given.

SRN										



c Briefly explain the Activation record structure. 5

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

(3+2)

SRN										

	d	Perform live-variable analysis:	5