PAGE NO. _____

DATE _____

SPCC - DEC 2016

Q1. What is Handle Pruning?

Ans. HANDLE PRUNING is the general approach used in shift-and-reduce parsing.

A Handle is a substring that matches the body of a production. Handle reduction is a step in the reverse of rightmost derivation. A rightmost derivation in reverse can be obtained by handle pruning.

A handle of a string is a substring that matches the right hand side of a production and whose reduction to the non terminal on the left hand side of the terminal of the production represents one step along the reverse of the rightmost derivation that ultimately leads to the start symbol. If replacing a string does not ultimately lead to the start symbol it cant be a handle.

The implementation of handle pruning involves the following data-structures :- a stack - to hold the grammar symbols; an input buffer that contains the remaining input and a table to decide handles.

Handles During a Parse $id_1 * id_2$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

$E \rightarrow T$. T is not a handle in $T \overset{*}{\Rightarrow} id2$

If we replace T by E° we get $E \overset{*}{\Rightarrow} id2$ which can not be derived from E

leftmost substring that matches production body need not to be a handle.

Q.2. what is the role of finite automata in compiler theory?

Ans. An automation is a self operating machine. The word is sometimes used to describe a robot, more specifically an autonomous robot. Automation is more often used to describe non-electronic moving machines, especially those that have been made to resemble human being or animal actions.

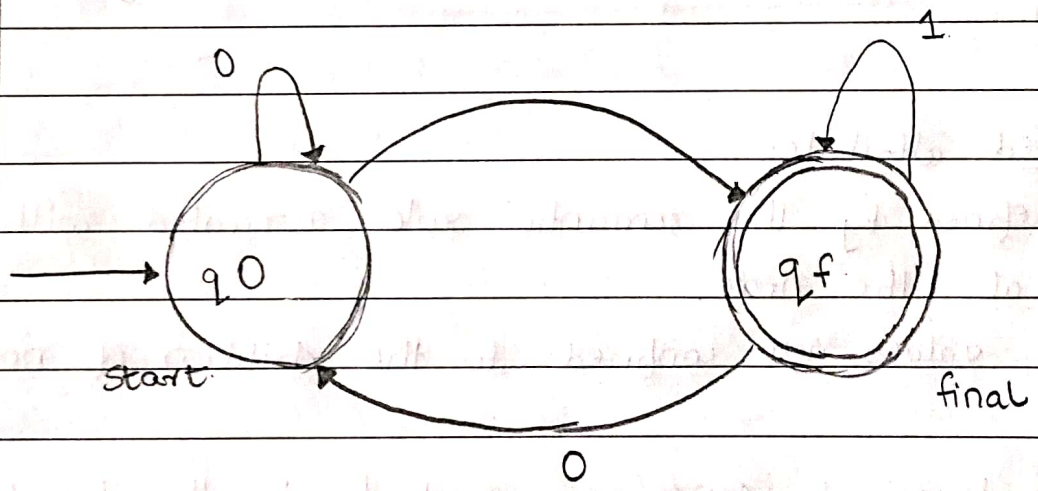
Let L(r) be a regular language recognized by some finite automata (FA).

- States :- circles represent the states of FA. The names of the state are written inside the circles.
- Start state :- The start state is the state at which the automata start. It has an arrow pointed towards it.
- Intermediate state :- The intermediate states have two arrow one arrow pointing to it and another arrow pointing out from the intermediate states.
- Final state :- The automata are expected to be in the final state when the input string is parsed successfully, and is represented by double circles. The final state may have odd number of arrows pointing towards it and even number of arrows pointing out from it. The

number of odd arrows are one greater than even, i.e. odd = even + 1.

- Transition :- when a desired symbol is found in the input, transition from one state to another state happens. The automata can either move to the next state or stay in the same state. The directed arrow displays the movement from one state to another and the arrow points towards the destination state. If automata stay on the same state, an arrow pointing from a state to itself is drawn.

Example : For instance, it is assumed that three digit binary values ending with 1 is accepted by FA. Then FA = { Q (q0, qf), Σ (0,1), q0, qf, δ }.



Q.3. what are the different type of attributes of SDD? Explain with Example.

Ans. Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions. It is a context free grammar with attributes and

rules together which are associated with grammar symbols and productions respectively.

The process of syntax directed translation is two-fold:

- construction of syntax tree and
- computing values of attributes at each node by visiting the nodes of syntax tree.

Types of attributes.

- Inherited attributes
  - It is defined by the semantic rule associated with the production at the parent of node.
  - Attributes values are confined to the parent of node, its siblings and by itself.
  - The non-terminal concerned must be in the body of the production.

- Synthesized attributes
  - It is defined by the semantic rule associated with the production at the node.
  - Attributes values are confined to the children of node & by itself.
  - The non terminal concerned must be in the head of production
  - Terminals have synthesized attributes which are the lexical values (denoted by lexval) generated by the lexical analyzer.

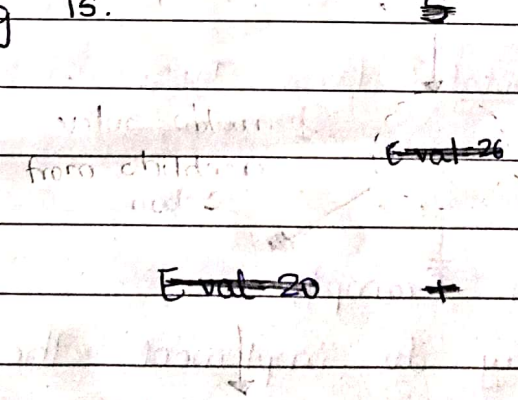
Example: Consider the following grammar

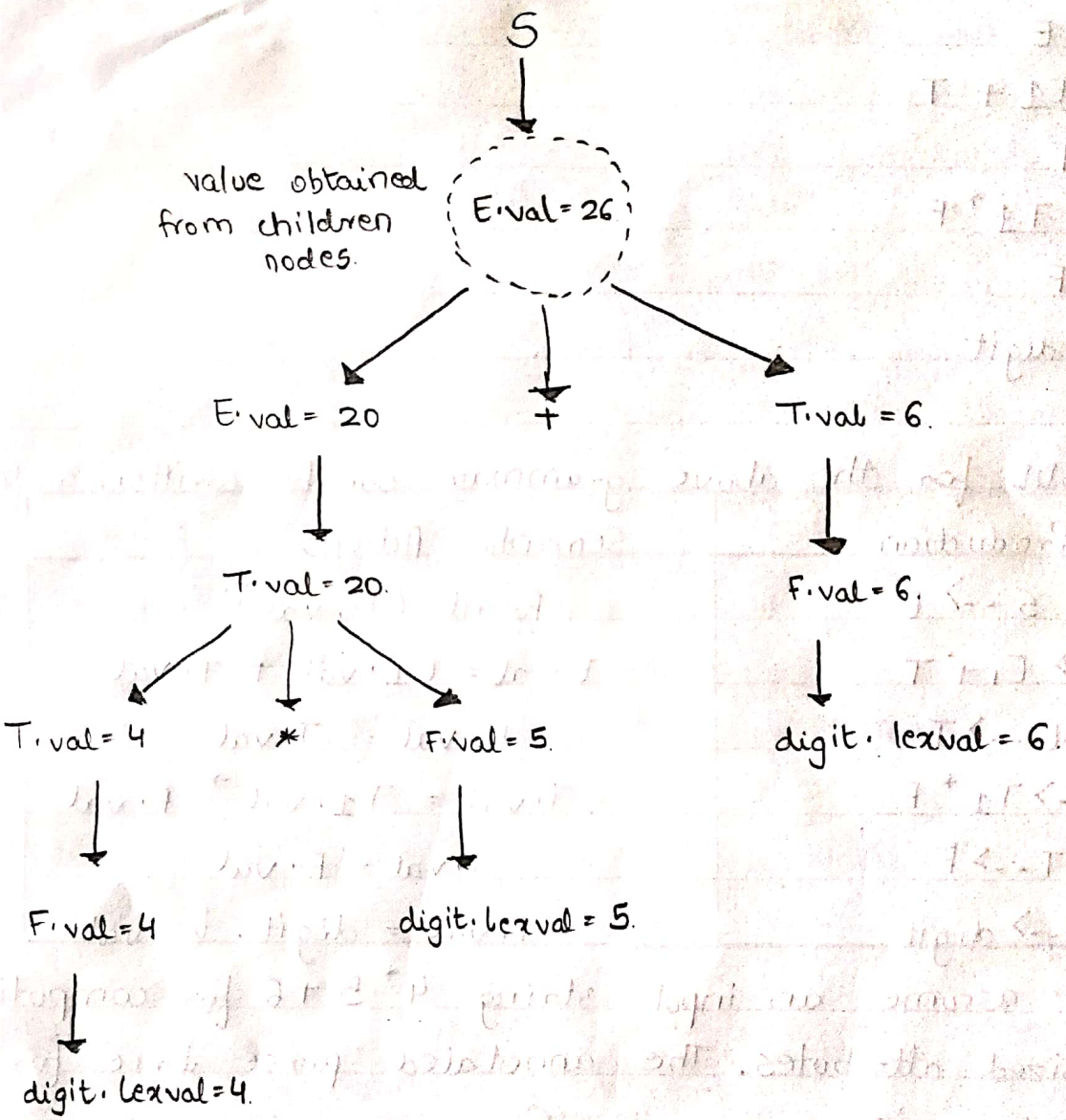
S --> E
E --> E1 + T
E --> T
T --> T1 * F
T --> F
F --> digit

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print (E.val)
E --> E1 + T	E.val = E1.val + T.val
E --> T	E.val = T.val
T --> T1 * F	T.val = T1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string 4 * 5 + 6 for computing synthesized attributes. The annotated parse tree for the input string is.





Annotated Parse Tree.

Q.4 Backpatching with Example.

Ans: The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for Boolean expressions and flow-of-control statements in a

single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. we call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions :

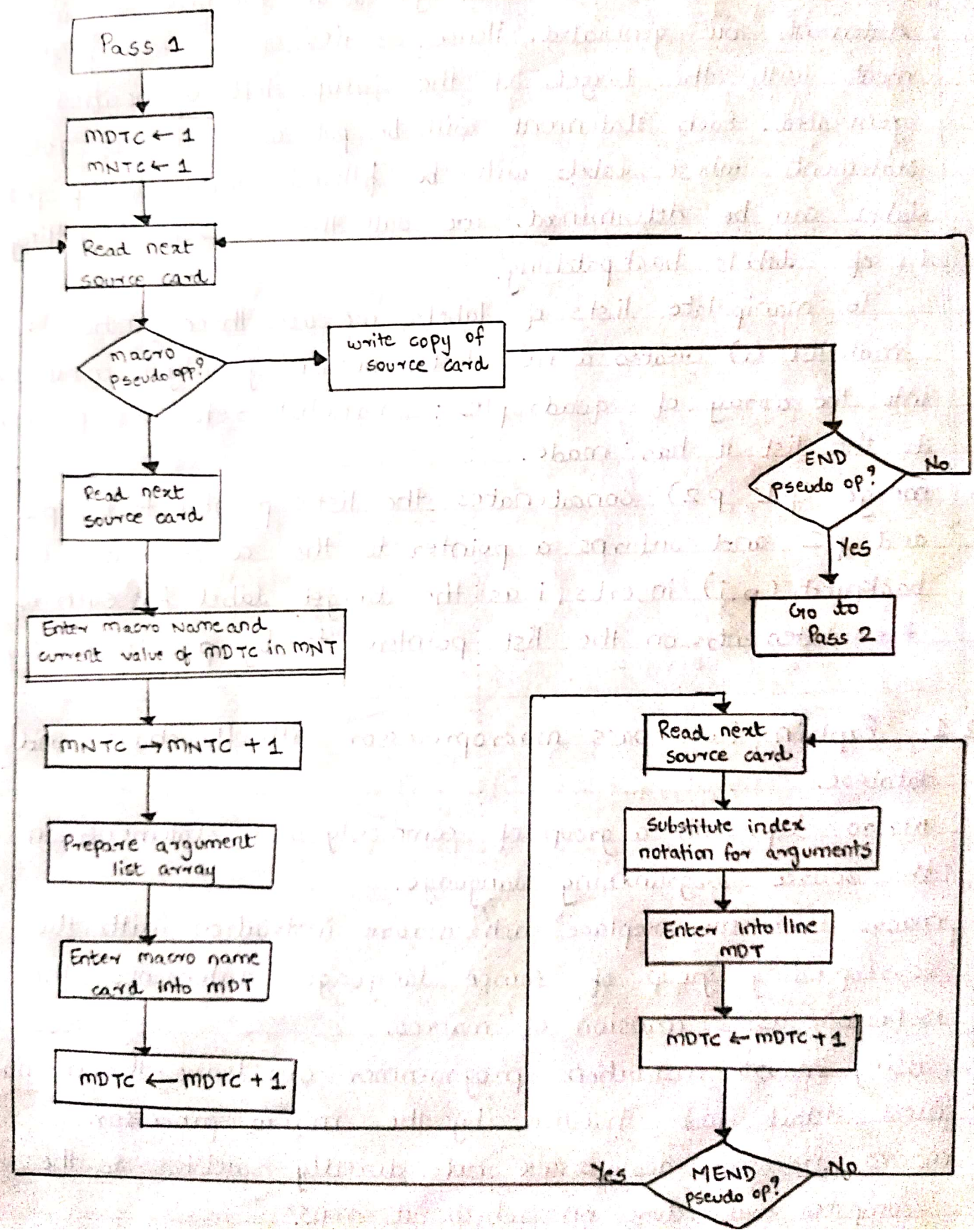
1. makelist (i) creates a new list containing only i, an index into the array of quadruples ; makelist returns a pointer to the list it has made.
2. merge (p1, p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list
3. backpatch (p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Q.2 1. Explain two pass macroprocessor with flowchart and database.

Ans. Macro represents a group of commonly used statements in the source programming language.

- Macro Processor replace each macro instruction with the corresponding group of source language statements. This is known as expansion of macros.
- Using macro instructions programmer can leave the mechanical details to be handled by the macro processor.
- Macro Processor designs are not directly related to the computer architecture on which it runs.

The flowchart for a 2 pass macro assembler is :



Specification of database formats :

Specification of database format :-

1. MDT (MACRO DEFINITION TABLE)

macro Definition Table 80 bytes per entry

Index	card		
:			
15	f LAB	INCR	f arg 1, f arg 2, f arg 3
16	#0	A	1, #1
17		A	2, #2
18		A	3, #3
19		MEND	
:		:	

- MDT is a table of text lines
- Every line of each macro definition except the macro line is stored in the MDT.
- MEND is kept to indicate the end of the dependencies
- The macro name line is retained to facilitate keyword argument replacement.

2. MNT (MACRO NAME TABLE)

MACRO NAMES TABLE (MNT):

macro Name Table

Index	8 bytes Name	4 bytes MDT index
3	"INCR"	15

- Each MNT entry consists of
- A character string (the macro name f
- A pointer (index) to the entry in MDT that corresponds to the beginning of macro definition.

3. ALA (Argument list Array)
   - ALA is used for both Pass 1 and Pass 2 but for some what reverse function.
   - During pass 1 in order to simplify later argument replacement using macro expansion, dummy arguments are replaced with positional indicators when definitions are stored.
   - symbolic dummy argument are retained on macro name to enable o enable the macroprocessor o handle argument replacement y name rathan than position.

Q2.2 Explain various loop optimization techniques with example.

Ans: The optimization performed on inner loops is called loop optimization.

Generally, inner loop is a place where program spends large amount of time. Hence, if number of instructions is less in inner loop the running time of the program decreases.

The following techniques can be performed on inner loops:

- Code motion / loop Invariant
- Induction variable
- Reduction in strength
- Loop Fusion / Loop Jamming
- loop unrolling

Code motion / loop Invariant :

The optimization performed on inner loop, in which the code moves outside the loop called as code motion

If there are a number of lines inside the loop whose result remains same even after executing the loop for several times, such an expression should be placed outside the loop, i.e., just before the loop.

Example :

```
int i, max = 10;
for (i= 10; I <= max - z; i ++)
{
Printf ("•%d", i);
}
```

In the example code, the result of an expression max - 1, remains same for 11 iterations. Hence, this code can be

optimized by removing the computing of max-1 outside the loop. i.e. by placing this expression before the loop thereby avoiding multiple computations.

The optimized code is

```
int I, max=10, x ;
x = max - 1;
for (i= 0; i<=x; i++);
 {
Printf ("%d", i);
 }
```

Induction variable :

A variable x is called an induction variable of loop L every time the variable x changes values, it is incremented or decremented by some constant.

Example 1 :

```
int i, max = 10, x;
x = max - 1;
for (i= 10; i<=x; i++)
 {
Printf ("%d", i);
 }
```

In the above code, variable i is called induction variable as values of I get incremented by 1, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Reduction in strength :

The strength of certain operators is higher than other operators

For example, strength of * is higher than +. usually, compiler takes more time for higher strength operators & execution speed is less.

Replacement of higher strength operator by lower strength operator is called a strength reduction technique

optimization can be done by applying strength reduction technique where higher strength can be replaced by lower strength operators.

Example :

```
for (i = 1; i <= 10; i++)
 {
sum = I * 7;
printf ("%d", sum);
 }
```

In the above code replacement of * by + will speed up the object code. Thus, optimization is done without changing the meaning of a code.

The optimized code is

```
temp = 7;
for(i = 1; i <= 10; i++)
 {
temp = temp + 7;
 sum = temp;
 printf ("%d", sum)
  }
```

Note : This technique is not applied to the floating point expressions because such a use may yield different results.

Loop Fusion / Loop Jamming :

This technique combines the bodies of two loops whenever the same index variable and number of iterations are shared.

Example :

```
for ( i=0; i <= 10; i++)
 {
Printf ( "Toc");
 }
For (i=0; i<=10; i++)
{
Printf ("CD");
}
```

The above code can be merged on one loop and optimized code can be rewritten as

```
for (i=0; i<=10; i++)
{
Printf ("Toc");
Printf ("CD");
}
```

Loop unrolling :

In this technique the number of jumps and tests can be optimized by writing the code to times without changing the meaning of a code

PAGE NO. _____

DATE _____

Example :

```
int i= 1;
while (i<100)
{
a[i] =b[i];
i++;
}
```

The example code can be optimized as

```
int i= 1;
while (i<100)
{
a[i]= b[i];
i++;
a[i] =b[i];
i++;
}
```

The first code loop repeats 50 times whereas second code loop repeats 25 times. Hence, optimization is done.

SPCC Paper

(Q3)(a) $S \to (S)S$

$S \to \varepsilon$

first(S) = {(, ε}

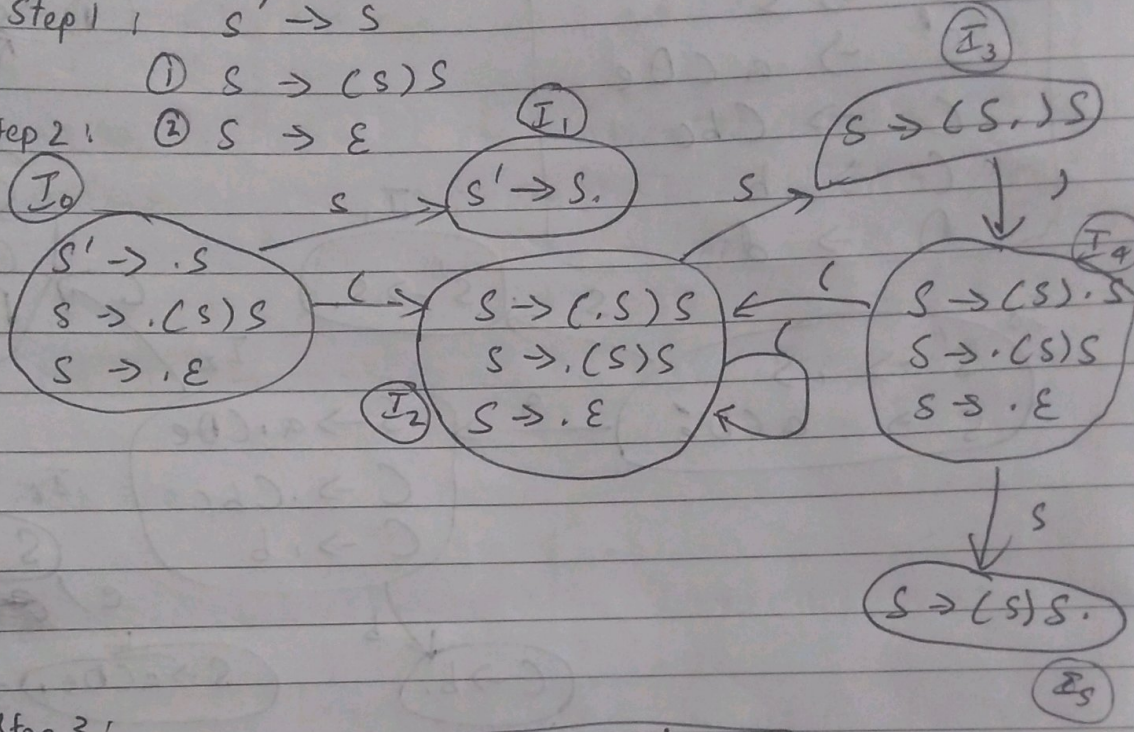
follow(S) = {), $}

i/p → ()() $

Step 1 : $S' \to S$

① $S \to (S)S$

Step 2 : ② $S \to \varepsilon$



Step 3 :

	(	)	$	S
0	S_2	r_2	r_2	1
1			accept	
2	S_2	r_2	r_2	3
3		S_4		
4	S_2	r_2	r_2	5
5		r_1	r_1	

Step 4 :

Stack	i/p	action
$0	()()$	shift
$0(2	)()$	reduce S → ε
$0(2S3	)()$	shift
$0(2S3)4	()$	shift
$0(2S3)4(2	)$	reduce S → ε
$0(2S3)4(2S3	)$	shift
$0(2S3)4(2S3)4	$	reduce S → ε
$0(2S3)4(2S3)4S5	$	reduce S → (S)S

$0(2S3)4S5	$	reduce S→(S)S
$0S1	$	accept

Q3) b)

Ans) Databases Required for Pass I

1) Input Source program
2) Location Counter (LC) to keep track of each instructions location and to assign address for each symbol defined.
3) Machine Operation Table (MOT) that indicates the symbolic mnemonic for each instruction and its length.
4) Pseudo Operation Table (POT) that indicates the pseudo-opcode and the action to be taken in pass I
5) Symbol Table - that is used to store each label and its corresponding value.
6) Literal Table - that is used to store each literal encountered and its corresponding assigned location.
7) A copy of i/p to be used later by Pass 2

Databases Required for Pass 2

1) Copy of src prg i/p to pass I
2) Location Counter (LC)
3) MOT that indicates for each instruction
   (a) Symbolic mnemonic
   (b) Length
   (c) Binary machine op-code
   (d) format (RR, RX, ---)

4) POT that indicates for each pseudo op the symbolic mnemonic and action to be taken in pass 2

5&6) Symbol Table (ST) & Literal Table (LT) prepared by pass 1

7) Base Table (BT) that indicates which registers are specified as base registers by USING pseudo opcodes and what the specified contents of the register are.

8) Instruction workspace which is used to hold each instruction as its various parts is being assembled together.

9) Print Line workspace used to produce printed listing

10) Punch card workspace used to punch the assembled instructions in the format needed by the loader.

11) A output deck of assembled instructions in the format needed by the loader.

Q6)

a) $E \to E + E$
$E \to E * E$
$E \to id$

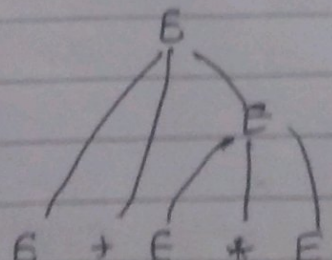
i/p : id + id * id

	id	+	*	$
id	E	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$

E : Error

$id \gtrdot \emptyset$
$\$ \lessdot \emptyset$
$+ \gtrdot +$, $+ \lessdot *$, $* \gtrdot *$

stack	input	a	b	action
$	id + id * id $	$	id	$ \lessdot$ id shift
$ id	+ id * id $	id	+	id $\gtrdot$ + reduce
$ E	+ id * id $	$	+	$ \lessdot$ + shift
$ E +	id * id $	+	id	+ \lessdot id shift
$ E + id	* id $	id	*	id $\gtrdot$ * reduce
$ E + E	* id $	+	*	+ \lessdot * shift
$ E + E *	id $	*	id	* \lessdot id shift
$ E + E * id	$	id	$	id $\gtrdot$ $ reduce
$ E + E * E	$	*	$	* $\gtrdot$ $ reduce
$ E + E	$	+	$	+ $\gtrdot$ $ reduce
$ E	$	$	$	accept



Skeletal Syntax tree.

ADITYA KHATURIA
&341

PAPER - SOLUTION

Q.4.a)
The following are commonly used intermediate code representation :

1. Postfix Notation -
The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example - The postfix representation of the expression (a-b)*(c+d)+(a-b) is : ab - cd + * ab - +.

2. Three - Address Code -
A statement involving no more than three references (two for operands and one for result) is known as three address code. Three address statement is of the form x = y op z. here x, y, z will have address (memory location). Sometimes a statement might contain less than three reference but it is still called three address statement

Example - The three address code for the expression a + b * c + d :

T1 = b * c
T2 = a + T1
T3 = T2 + d
T1, T2, T3 are temporary variables.

3. Syntax Tree -

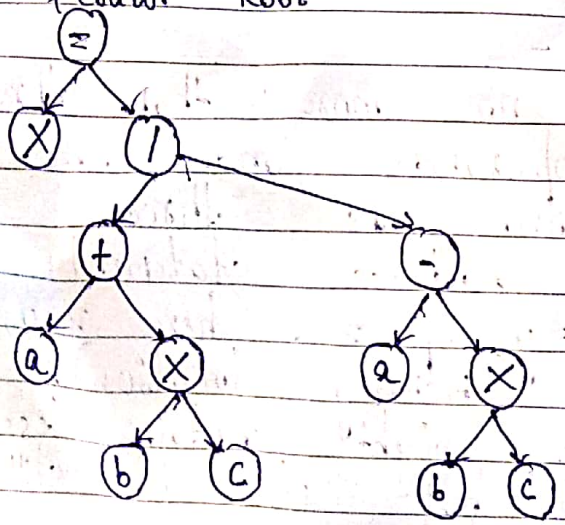
Syntax Tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example -

x = (a+b*c) / (a-b*c)

↓

Operator Root

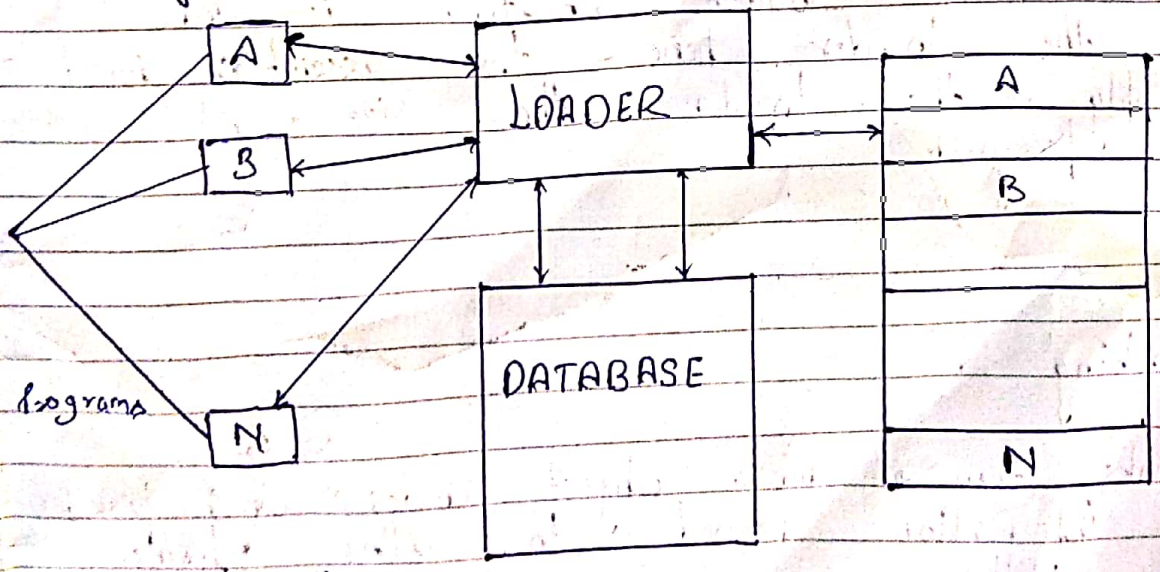


Q.4. b)

Loader

- A loader is a system program, which takes the object code of a program as input and prepares it for execution.
- Programmers usually define the Program to be loaded at some predefined location in the memory.
- But this loading address given by the programmer is not be coordinated with the OS.
- The loader does the job of coordinating with the OS to get initial loading address for the EXE file and load it into the memory

Diagram



Loader Function : The loader performs the following functions

1) Allocation
2) Linking
3) Relocation
4) Loading

1. Allocation :

- Allocates the space in the memory where the object program would be loaded for Execution
- It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.
- In absolute loader allocation is done by the programmer and hence it is the duty of the programmer to ensure that the programs do not get overlap.
- In reloadable loader allocation is done by the loader hence the assembler must supply the loader the size of the program.

2. Linking :

- It links two or more object codes and provides the information needed to allow references between them
- It resolves the symbolic references (code / data) between the object modules by assigning

all the user subroutine and library subroutine addresses. This activity is called linking.

- In absolute loader linking is done by the programmer as the programmer is aware about the runtime address of the symbols.

3. Relocation :

- It modifies the object program by changing the certain instructions so that it can be loaded at different address from location originally specified.
- There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.
- In absolute Loader relocation is done by the assembler as the assembler is aware of the starting address of the program.
- In relocatable loader, relocation is done by the loader and hence assembler must supply to the loader the location at which relocation is to be done.

4. Loading :

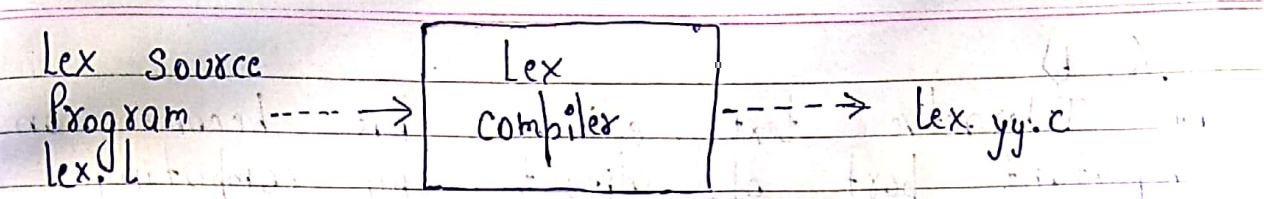
- It brings the object program into the memory for execution.
- Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution; this activity is called loading.
- In both the loaders (absolute, relocatable)

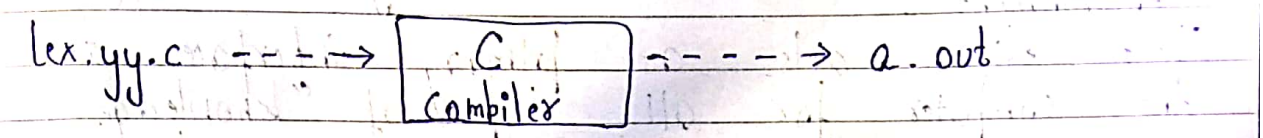
Loading is done by the loader and hence the assembler must supply to the loader the object program.

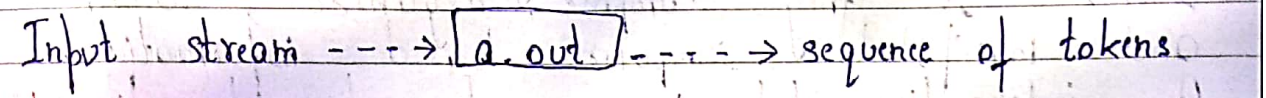
① .6. a)

LEX:

- Lex is officially known as a "Lexical Analyser".
- Its main job is to break up an input stream into more usable elements. Or in, other words, to identify the 'interesting bits' in a text file.
- For example, if you are writing a compile for the C programming language, the symbols { } ( ) ; all have significance on their own.
- The letter a usually appears as part of a keyword or variable name, and is not interesting on its own.
- Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this".
- All of these things are handled by the Lexical Analyser.
- A tool widely used to specify lexical analyzers for a variety of languages.
- We refer to the tool as Lex compiler, and to its input specification as the Lex language.



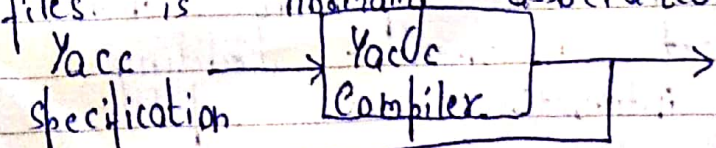


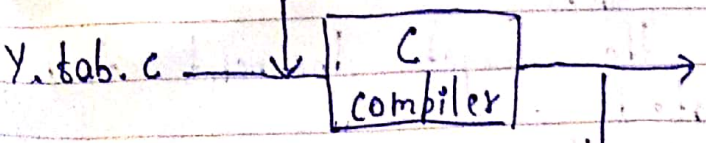


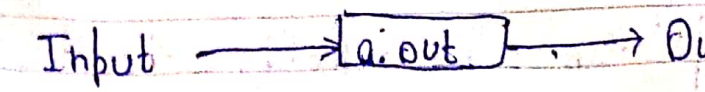
Creating a lexical analyzer with Lex

YACC :

- Yacc is officially known as a "parser".
- It's job to analyse the structure of the input stream, and operate of the "big picture".
- In the course of it's normal work, the parser also verifies the input is syntactically sound.
- Consider again the example of C-compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a (or (a = There should be exactly one } for each { in the program.
- YACC stands for "Yet another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers







Q.6. b)

An Interactive text editor has become an important part of almost any computing environment.

Text editor acts as primary interface to the computer for all type of "knowledge workers" as they compose, organize, study and manipulate computer - based information.

An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs.

Many such system are available during these days. Our discussion is broad in scope, giving the overview of interactive debugging systems - not specific to any particular existing system.

Text Editors Overview

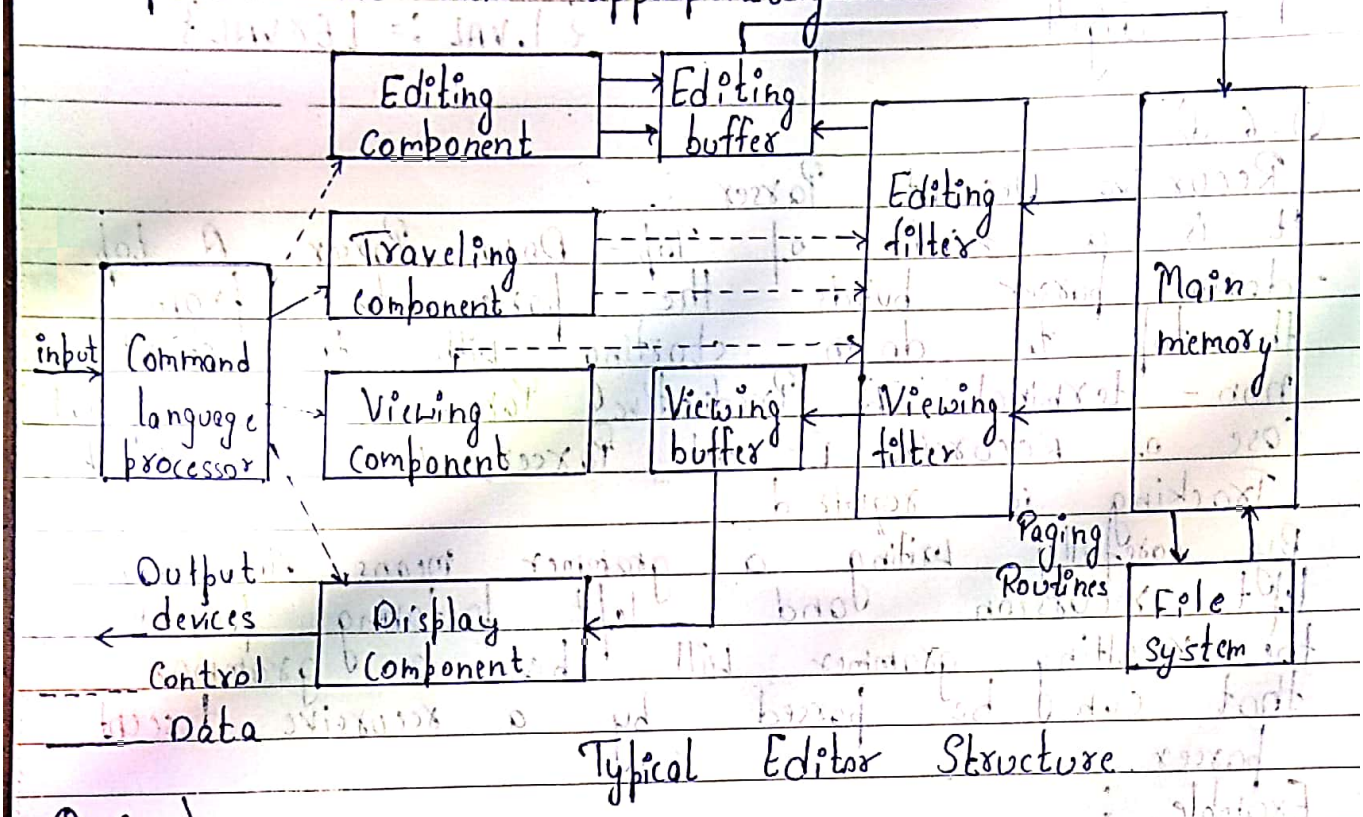
An interactive editor is a computer program that allows a user to create and revise a target document.

- Document includes objects such as computer diagrams, text, equations tables, diagrams, line art, and photographs.

Here we restrict to text editors, where character strings are the primary elements of the target text. Document editing process in an interactive user - computer dialogue has four tasks.

- Select the part of the target document to be viewed and manipulated
- Determine how to format this view on-line and how to display it
- Specify and execute operations that modify the target

document.
- Update the view appropriately



Typical Editor Structure

Q.6 c)

- The Syntax directed translation scheme is a context-free grammar.
- The Syntax directed translation scheme is used to evalute the order of semantic rules.
- In itranslation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
S → E $	{ print E.VAL }
E → E + E	{ E.VAL := E.VAL + E.VAL }
E → E * E	{ E.VAL := E.VAL * E.VAL }
E → (E)	{ E.VAL := E.VAL }

$E \rightarrow I$	$\{ E.VAL := I.VAL \}$
$I \rightarrow I$ digit	$\{ I.VAL := 10^{*} I.VAL + LEXVAL \}$
$I \rightarrow$ digit	$\{ I.VAL := LEXVAL \}$

Q.6.d)

Recursive Descent Parser :

It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal. A Predictive Parser is a special case of Recursive Descent Parser, where no Back Tracking is required.

By carefully writing a grammer means eliminating left recursion and left factoring from it, the resulting grammer will be a grammar that can be parsed by a recursive descent parser.

Example :

Before Removing left Recursion	After Removing left Recursion
~~left Recursion~~	
$E \rightarrow E + T \mid T$	$E \rightarrow TE'$
$T \rightarrow T^{*} F \mid F$	$E' \rightarrow + TE' \mid \epsilon$
$F \rightarrow (E) \mid id$	$T \rightarrow FT'$
	$T \rightarrow {}^{*} FT' \mid \epsilon$
(Here ϵ is Epsilon)	$F \rightarrow (E) \mid id$