① What is SDD? Explain it with own example?

A) Syntax direct derivations (SDD):

The syntax direct derivations is combination of context free grammar and Argumented attributes [semantic rules]

SDD = CFG + Argumented attributes.

* Every time we will attach attributes to a variable.
* Terminals will handle by lexical analyser
* The attribute count may be 1 or 2 or 0.
* We have two types of attributes.
  1. Synthesized attribute
  2. Inherited attribute.

Synthesized attribute: In synthesized attribute, the parent node will pick the value from children nodes.

Suppose $T \rightarrow ABC$

$T.val = A.val$

$T.val = B.val$

$T.val = C.val$

Inherited Attribute: In inherited attribute, here children node will pick values from its sibling nodes

Suppose $t \rightarrow ABC$

$B.val = A.val$

$B.val = C.val$

Example:

② Solve the given problem by using production rule in SDD

2*3+4n

$L \rightarrow En$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

T → F

F → Number

Firstly, we have to write SDD.

L.val → E.val

E.val → E.val + T.val

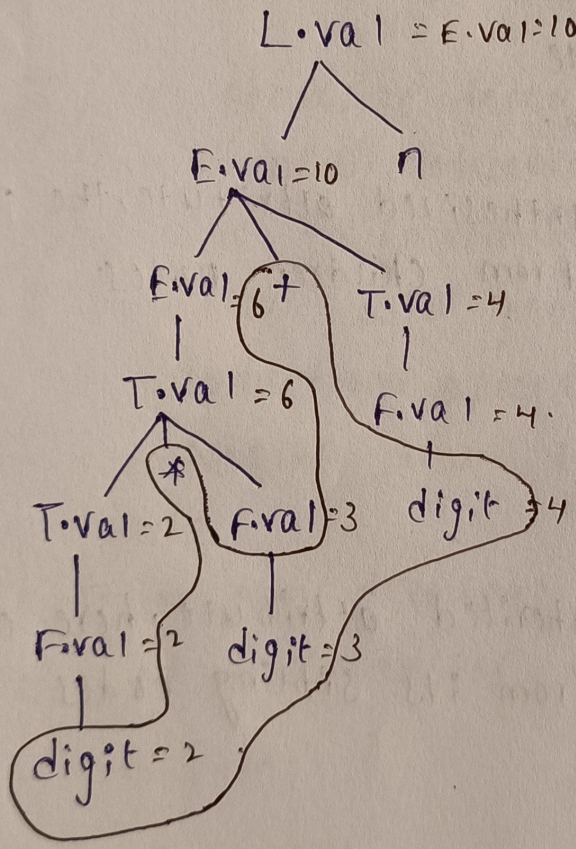
E.val → T.val

T.val → T.val * F.val

T.val → F.val

F.val → Digit (Lexinumeral)

for these we have to generate parse tree.



Types of SDD:

We have 2 types of SDD

1. S attributed SDD

2. L attributed SDD.

S attributed SDD

→ It will follow Synthesized attribute format
→ Right most children value will be picked by the parent
→ And here parsing will happen in bottom up mode to solve regular equations we use 'S' attribute format.

L attribute SDD

→ Both inherited and synthesized attributes support this one.
→ Child node will pick the values from the siblings.
→ only left side values will be picked
→ parsing will happen from left to right.
→ compared with S attribute implementation is difficult.

② write about Recursive decent parsing

→ If a derivation of a production rule, derives terminal first, you have to increment the input content
→ If the derivation will generate a non terminal write the production according to it.
→ If the derivation non terminal having 3 productions
write each non terminal functionality.
→ No need of declarations and functional assignments.

production rule:

Example: $E \rightarrow iE'$

$E' \rightarrow +iE' / \varepsilon$

$E \rightarrow iE'$

Ans: $E \rightarrow ( )$

{

if (derivation = ";") (Terminal)

{

```
input ++;
}
else
{
  E prime();
}
E prime()
{
  if (Derivation = "+")
  {
    input ++;
  }
  if (Derivation = "i")
  {
    input ++;
  }
  else
  E prime();
}
```

Example 2:

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow id$

$F \rightarrow (E)$

$E \rightarrow E+T$

```
E()
  if (Derivation = "E")
  {
    E();
  }
  else if (Derivation = "+")
    input ++;
    else
      T();
}
```

(3) Explain about LL(1) grammar

Rules:

Elimination of left factorisation.

Elimination of left Recursion.

first and follow Algorithm.

Construction of parse table.

Checking of given string whether it exists in LL(1)

Example: $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

for removal of ambiguity we need to convert the left recursion into right recursion.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Variable Action form	first	follow
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \varepsilon$	{+, id, (}	{$,), +}
$T \rightarrow FT'$	{id, (}	{+, $,)}
$T' \rightarrow *FT' \mid \varepsilon$	{*, id, (}	{*, +, $,)}
$F \rightarrow id \mid (E)$	{id, (}	{*, +, $, id, (,)}

E → T

```
E()
{
if(Derivation = T)
{
 T();
}}
```

T → T * F

```
T()
{
if (Derivation = T)
{
 T();
}
else if (Derivation = "*")
{
 input ++;
}
 else
 {
   F();
 }}
```

T → F

```
T()
if (Derivation = F)
{
 F();
}
```

F → id

```
F()
{
 if (Derivation = "id")
 {
  input +++;
}}
```

F → (E)

```
if (Derivation = "(")
{
 input +++;
}
else (Derivation = ")")
{ input +++;
}
```

→ In a parse trees table rows and variables, columns are terminals

→ To construct a parse table we need to follow 3 rules:-

(i) If variable derives a variable, pick first of it

(ii) If variable derives ε, pick follow of it

(iii) If variable derives terminal, pick the terminal only.

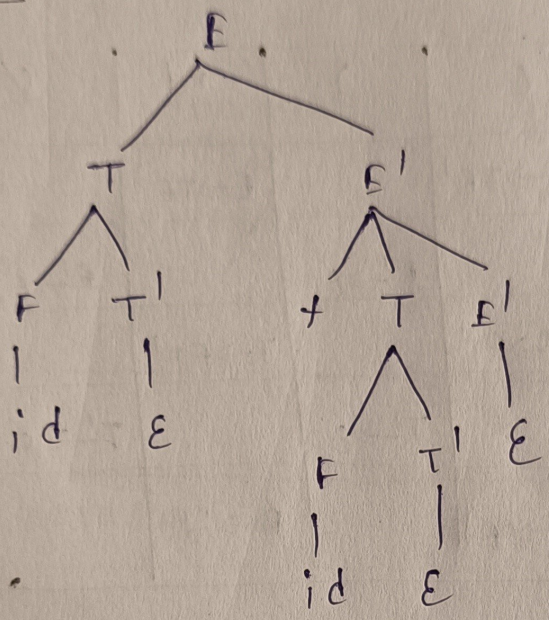
Variables	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE' / \varepsilon$			$E' \to \varepsilon$		$E' \to \varepsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to *FT' / \varepsilon$	$T' \to *FT' / \varepsilon$		$T' \to \varepsilon$		$T' \to \varepsilon$
F			$F \to (E)$		$F \to id$	

Checking of string

Stack	Input string	Action
$ E	id + id $	$E \to TE'$
$ E'T	id + id $	$T \to FT'$
$ E'T'F	id + id $	$F \to id / (E)$
$ E'T'id	id + id $	pop
$ E'T'	+ id $	$T' \to *FT' / \varepsilon$
$ E'	+ id $	$E' \to +TE' / \varepsilon$
$ E'T+	+ id $	pop
$ E'T	id $	FT'
$ E'T'F	id $	$F \to id / (E)$

$ E'T'id	id$	pop.
$ E'T'	$	$T' \rightarrow *FT'/\varepsilon$
$ E'	$	$T' \rightarrow *FT'/\varepsilon$.
$	$	$E' \rightarrow +TE'/\varepsilon$.

Parse Tree



(4) What is Three address code in details? And Optimization of basic blocks.

(A) Three Address Code :

Three Address code which is useful to convert high level language into Assemble language. in a easier way.

The three address code addresses divided into 3 types

(i) Quadraples

(ii) Triplets.

(iii) Abstract Triplets.

Divide complex equation into simple way.

$a+b=t_1$ — (1)

$c+d=t_2$ — (2)

$t_1 * t_2 = t_3$ — (3)

$s=t_3$ — (4)

Quatraplets: We have to pick 4 quatraplets.

operator	Argument t_1	Argument 2	Result
+	a	b	t_1
+	c	d	t_2
*	t_1	t_2	t_3

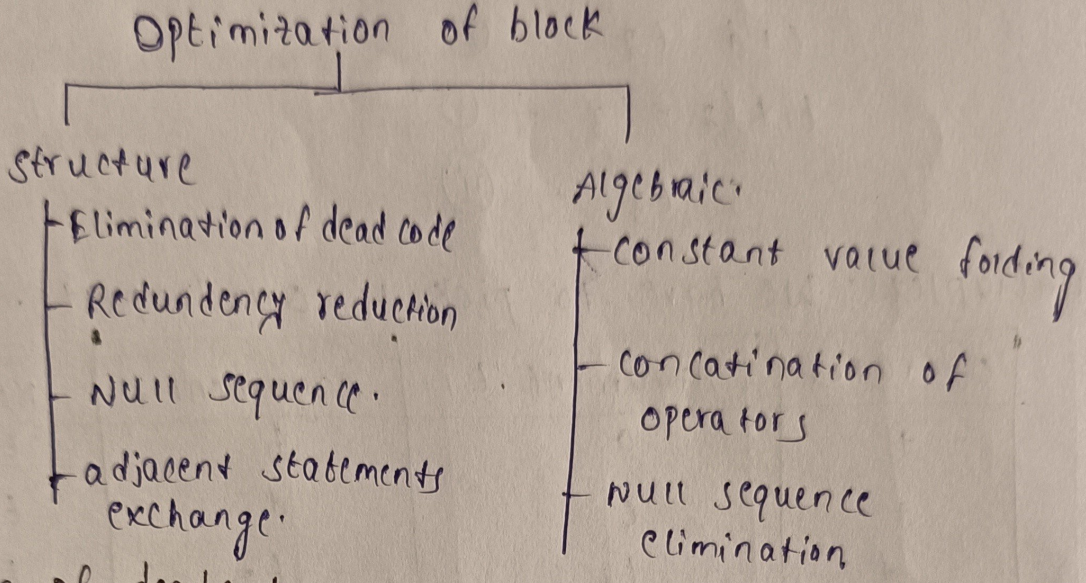
Triplet:

operator	Arg 1	Arg 2
+	a	b
+	c	d
*	t_1	t_2
=	s	t_3

Abstract triplets:

Abstract form	Argument 1	Argument 2.
(1)	a	b
(2)	c	d
(3)	(1)	(2)
(4)	(5)	(3)

Optimization of Blocks.



Elimination of deadcode;

→ Continous iterative loops elimination is nothing but dead code elimination.

Redundency reduction:

Ex: $Y = x+3$
$i = Y$
$z = i$
$w = z * 3$
$Y = x+3$
$w = Y*3$

→ By reducing the production rules based on redundency.

Null sequence:

→ By eliminating null sequence we can optimize the code.

$a+0 = a$
$a-0 = a$
$a*1 = a$
$a/1 = a$ } a.

adjacent statement exchange:

→ exchange of variables is happen to reduce the computational time

$y = x * 3$

computional time is high.

$y = x * x$ (or) $y = 3 * 3$ } → computational time is low.

Algebraic:

constant value folding:

$y = 2 * 3$

$y = 6$

Concatination of operators:

$S \Rightarrow (a+b) * (c+d)$

$a+b = t_1$

$c+d = t_2$

$t_1 * t_2 = t_3$

$S = t_3$

Null sequence elimination:

→ By eliminating null sequence we can optimize the code.

$a+0 = a$
$a-0 = a$
$a * 1 = a$
$a/1 = a$ } a

→ we can write a by replacing the 4 equations.