BYJU'S EXAM PREP

Computer Science & IT

Compiler Design

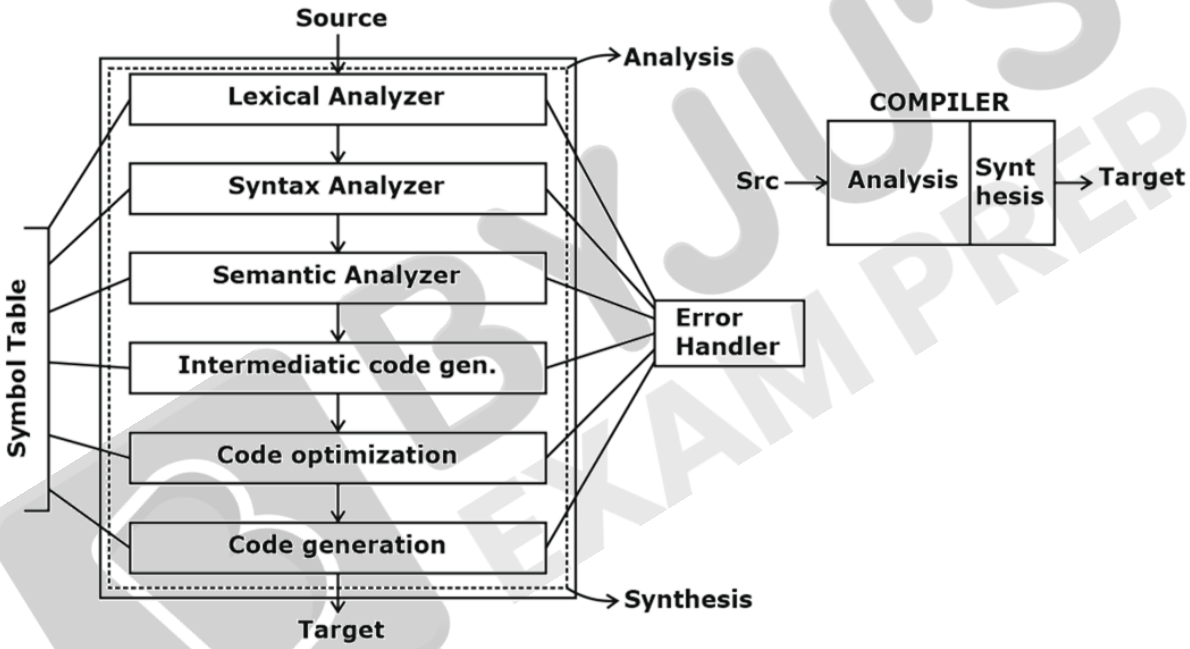
SHORT NOTES

Short Notes — COMPILER DESIGN

- **Definition:** It converts high level language to low level language.

- **High Level Language** can perform more than one operation in single statement.

- **Analysis and synthetic model of compiler:**



- There are 6 phases of the compiler.
  1. **Lexical Analyzer :**
     Program of DFA, it checks for spelling mistakes of the program.
  2. **Syntax Analyzer :**
     It checks grammatical errors of the program. (Parser)
     * Parser is a DPDA.
  3. **Semantic Analyzer :**
     Checks for meaning of the program.
     [Eg. Type miss match, stack overflow]
     * w/o Error handler, compiler can still work.
  4. **Intermediate Code generator :**
     This phase is used to make the work of next 2 phases much easier.
     Enforces reusability and portability.

5. **Code Optimization :**
    1. Loop invariant construction
    2. Common sub expression elimination
    3. Strength Reduction
    4. Function in Lining
    5. Dead code elimination
6. **Symbol Table :**
    1. Data about data (meta data)
    2. Date structure used by compiler and shared by all the phases.

    * w/o symbol table compiler cannot work.

- **CD – Grammar**

  ⇒ In compiler we only use: Type – 2 (CFG) and Type – 3 (RG) Grammers.

  ⇒ Compiler = Program of Grammar

  ⇒ Compiler = Membership Algorithm

  * Every programming Language is CSG. (CSL)

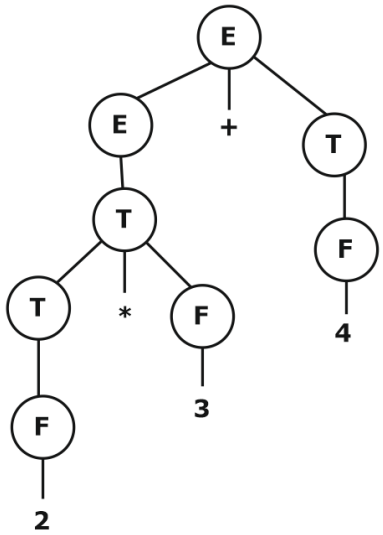
- **Parse Tree and Syntax Tree :**

  G: $E \rightarrow E + T/T$ $E \rightarrow T + T \rightarrow T + T \rightarrow T * F + T \rightarrow F * F + T$

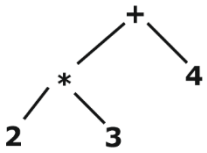
  $T \rightarrow T * F \mid F$ $\rightarrow 2 * F + T \rightarrow 2 * 3 + T \rightarrow 2 * 3 + F$

  $\boxed{E \rightarrow 2 * 3 + 4}$

- **Parse Tree :** **Syntax Tree :**



→ Procedural
→ Int Node = Variable
→ Whole derivation is represented.



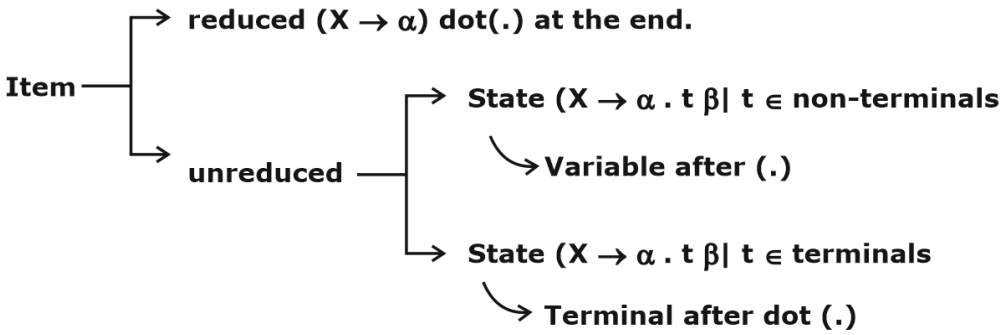
→ non procedural
→ Int. node = operation
→ Discuses the concept only.

* **To check the priority / Associativity :**

Randomly derive till you have enough operations, then check which one done first.

* If priority of 2 operators is same and both are Left and Right associative → Ambiguous Grammar. [USELESS]

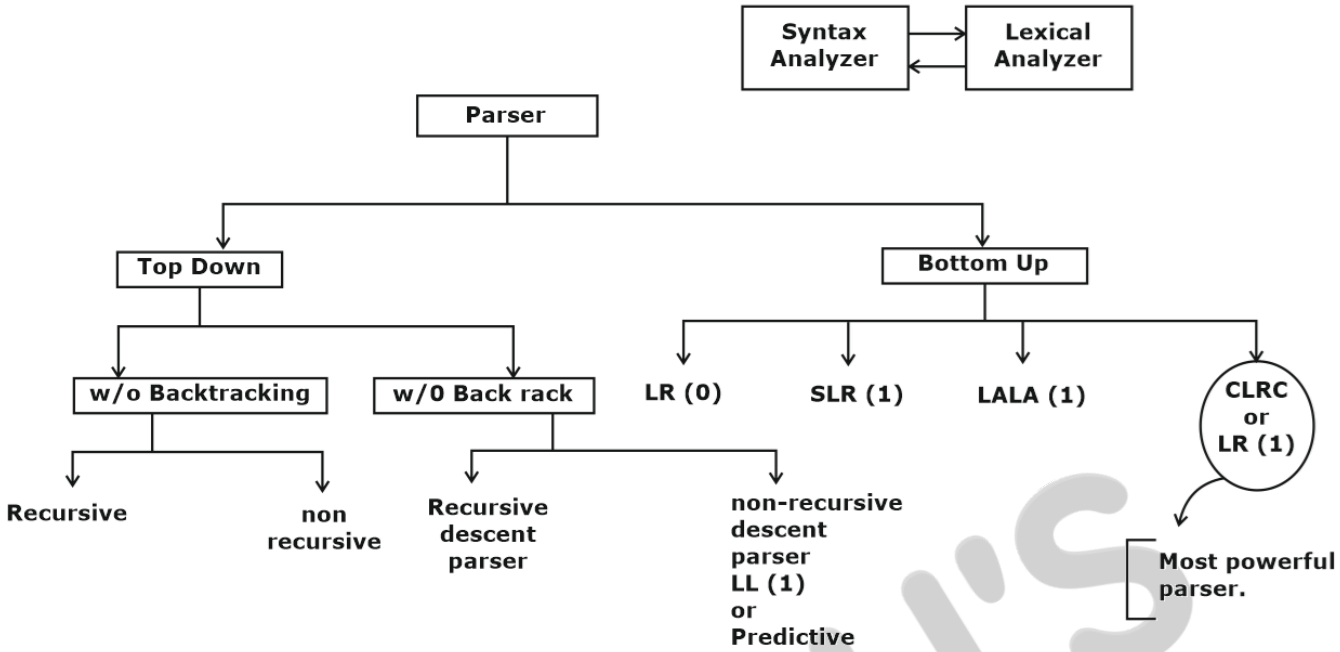
- **Type of item in Bottom up : [CD – Parser]**



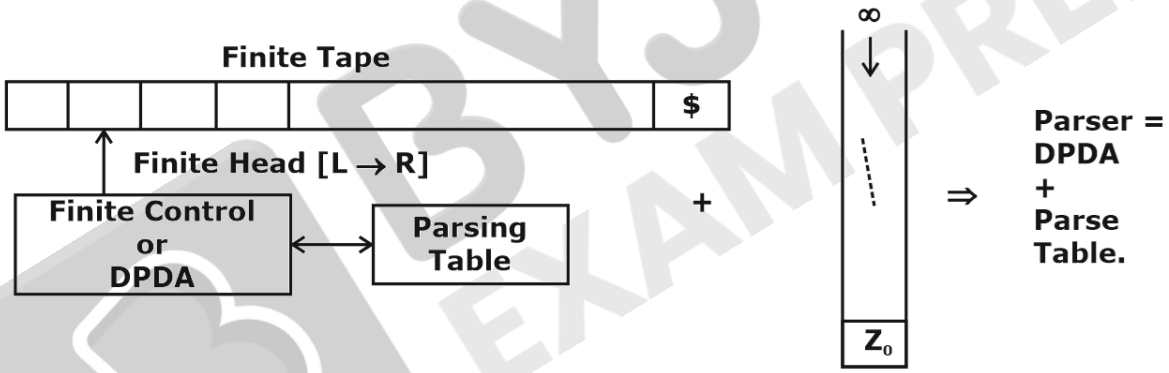
- **CD – Syntax Analysis Parsing**

Grammatical errors are checked with the help of parsers.

* Parsers are basically DPDA's.



- **Mathematical Model of Parser:**



* Parsers generate parse tree, for a given string by the given grammar.

- **Top down parser (LL (1)):**

  * It uses LMD and is equivalent to DFS in Graph.

- **Algorithm to construct parsing table :**

  1. Remove Left Recursion if any.
  2. Remove Left Factoring if any. [Remove common prefix.]
  3. Find 1st and follow set.
  4. Construct the Table.

  * If we increase the look ahead symbol:

  → strength of parser ↑

  → complexity of parser ↑

  * [**Due to common prefix:** Back track]

  [Due to left Recursion : ∞ RCC.]

Removal of common Prefix : (Left factor).

1. $S \to \underline{a} \mid \underline{a}b \mid \underline{a}A$

$\Rightarrow S \to aY$

$Y \to \in \mid b \mid A.$

2. $A \to \underline{ab}A \mid \underline{a}A \mid b.$

$\Rightarrow A \to ax \mid b \mid A \to Ax \mid b$

$X \to bA \mid A \mid x \to \underline{b}A \mid ax \mid \underline{b}$

* Indirect common perfix

$\Rightarrow A \to ax \mid b$

$x \to ax \mid bY$

$Y \to A \mid \in$

- **First and Follow:**
  - **First set** → The extreme Left terminal from which the string of that variable starts.

    * It never contains variables, but may contain 'ϵ'.

    * We can always find the first of any variable.
  - **Follow set** → Follow set contains terminals and $.

    It can never contain variable and "ϵ".

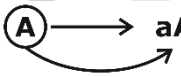
    How to find follow set?

    1. Include $ in follow of start variable.
    2. If production is of type →

    $A \to \alpha B \beta$ [α, β → strings of grammar symbol.]

    $\boxed{\text{follow}(B) = \text{first}(\beta)}$

    If, $\beta \xrightarrow{*} \in$, ie $A \to \alpha\beta$, then $\boxed{\text{follow}(B) = \text{follow}(A)}$

    * **Production Like :**  gives No follow set.
  - **Examples of first and follow set :**

    1.

    $S \to AB|CD$

    $A \to aA|a$

    $B \to bB|b$

    $C \to cC|c$

    $D \to dD|d$

	First	Follow
S	a,c	$
A	A	b
B	B	$
C	C	d
D	D	$

- **Entry into Table : Top down :**

  1. No of rows = No of unique variables in Grammar.
  2. No. of columns = [Terminals + $]
  3. For a variable (ROW) fill the column (terminal) if it is there in its first's w/o the production reqd.
  4. If ϵ is in first put V → ϵ under $ and its follows.

  * If any cell has multiple times, then it not possible to have LL(1) parser. Since that will be ambiguous.

  * [In top down we do : derivation]
  [In Bottom up we do : Reduction]

2. Construct LL(1) Parsing Table for the given grammar:

$$E \to E + T \mid T \; ; \; T \to T * F \mid F \; ; \; F \to (E) \mid id \; ; \; \}G_0$$

- **Removing Left Recursion :**

$$\left.\begin{array}{l} E \to TE' \\ E' \to +TE' \mid \in \\ T \to FT' \\ T' \to *FT' \mid \in \\ F \to (E) \mid id \end{array}\right\} G_1$$

	First	Follow
E	c, id	$,)
E′	+, ϵ	$,)
T	c, id	+, $,)
T′	*, ϵ	+, $,)
F	c, id	*, +, $

* left factoring not required.

- **Construction of Table : [LL (1)]**

	+	*	(	)	id	$
E	error	error	E → TE'	error	E → TE'	error
E'	E'→ + TE'	error	error	E' → ϵ	Error	E' → ϵ
T	error	error	T → FT'	error	T→ FT'	error
T'	T' → ϵ	T' → * FT'	error	T' → ϵ	error	T' → ϵ
F	error	error	F → (ϵ)	error	F → id	error

* Since for G_1, Table constructed w/o no multiple entries, hence successfully completed.

Hence G_1 is LL(1).

Q. Construct LL(1) Parsing Table for the following grammar :

$S \rightarrow L = R \mid R$; $L \rightarrow * R \mid id$; $R \rightarrow L$ } G_0

- **Left Factoring :**

$$\left.\begin{array}{l} S \rightarrow L = R \mid L \\ L \rightarrow * R \mid id \\ R \rightarrow L \end{array}\right| \Rightarrow \left.\begin{array}{l} S \rightarrow L\,X \\ X \Rightarrow = R \mid \epsilon \\ L \rightarrow *R \mid id \\ R \rightarrow L \end{array}\right\} G_1$$

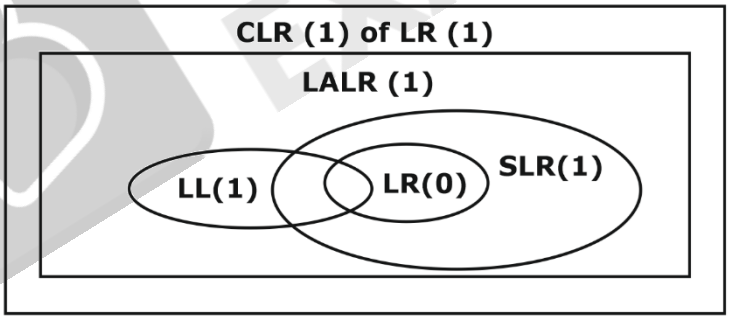
	First	Follow
S	*, id	$
X	=, ϵ	$
L	*, id	$, =
R	*, id	$, =

- **Construction of Table :**

	*	=	id	$
S	S → LX	error	S → LX	error
L	L → *R	error	L → id	error
R	R → L	error	R → L	error
X	Error	X → =R	error	X → ϵ

* G_1 is a LL(1) Grammar.

- **Hierarchy of Parsers :** [for ϵ-free Grammar]



* For ϵ-producing grammars, every LL(1) may not be LALR(1).

  - **NOTE:-**

    We can't construct any parser for ambiguous grammar.

    Except : operator precedence, parser possible for some ambiguous grammar.

    * There are some unambiguous grammar, for which there are no parsers.

  - **Example:**

    1. G : S → a S a | b s b | a | b

       L(G) = w(a + b) wR

       (odd palindrome)

- **Unambiguous but no parser.**

  ⇒ Every RG is not LL(1) as it may be ambiguous, or recursive or having common prefix.

  ⇒ Parsers exist only for the grammar if the Lang. is DCFL.

  * There are some grammar whose Lang is DCFL but no parser is possible for it.

- **Operation Precedence Grammar :**

  Format :

  1. No 2 or more variable should come side by side.
  2. No ϵ production.

  ○ **Example:**

1. $E \to E = T \mid T$
   $T \to T * F \mid F$
   $F \to (E) \mid id$
   O.G.

2. $E \to E + E$
   $E \to E \times E$
   $E \to a \mid b$
   O.G.

3. $S \to a\ S\ a \mid b\ S\ b \mid a \mid b.$
   O.G.

4. $S \to A\ B$
   $A \to a\ A \mid \epsilon$
   $B \to b\ B \mid \epsilon$ **{Not O.G.}**

- **Checking LL(1) w/o table :**

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3,$ then $\to$ $A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \in$

$$\text{first}(\alpha_1) \cap \text{first}(\alpha_2) = \phi$$
$$\text{first}(\alpha_1) \cap \text{first}(\alpha_3) = \phi$$
$$\text{first}(\alpha_2) \cap \text{first}(\alpha_3) = \phi$$

$$\text{first}(\alpha_1) \cap \text{first}(\alpha_2) = \phi$$
$$\text{first}(\alpha_1) \cap \text{first}(\alpha_3) = \phi$$
$$\text{first}(\alpha_2) \cap \text{first}(\alpha_3) = \phi$$
$$\text{follow}(A) \cap \text{first}(\alpha_1) = \phi$$
$$\text{follow}(A) \cap \text{first}(\alpha_2) = \phi$$
$$\text{follow}(A) \cap \text{first}(\alpha_1) = \phi$$

- **BOTTOM – UP Parsers:**

→ It uses RMD in reverse and has no problem w/o :

(a) Left Recursion
(b) Common Prefix.

→ No Parser possible for ambiguous grammar.

→ There are some unambiguous grammar for which, there are no Parser.

→ The Language of the grammar must be DCFL.

$$LR(1) = LR(0) + 1 \text{ look a head.}$$

- **Basic Algorithm for Construction :**

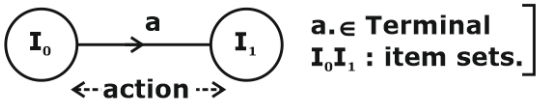
  → Augment the grammar and expand it and give numbers to it.

  → Construct LR(0) or LR(1) items.

  → From these items fill the entries in the Table accordingly.

1. Shift Entries :

Transitions on terminals

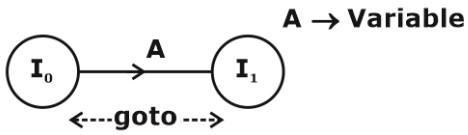


entry :

	a
I_0	S_1

2. State entry :

Transition on non-terminal (variables)



entry :

	A
I_0	1

*** Shift entries are same for all Bottom-up Parser.**

*** Same for all Bottom up Parser.**

(2) Reduce Entry :

Done for each separate production in the item set of type :

$i > X \to \alpha.$

where $\begin{bmatrix} i \to \text{Prod. No} \\ X \to \text{Producing var.} \\ \alpha \to \text{Grammar String.} \end{bmatrix}$

In :

(a) LR(0) Parser: Put R_i in <u>every cell</u> Of the set in action Table **(ALL)**

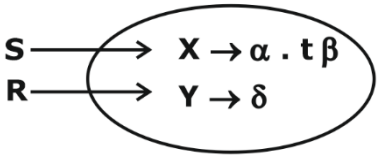
(b) SLR(1) Parser : Put R_i only in the <u>follow(x)</u> from the Grammar. **(Follow (x))**

(c) LALR(1) and CLR(1) : Put R_i only in the <u>look-ahead</u> of the production **(Lookaheads)**

- **Conflicts :**

LR(0) Parser :

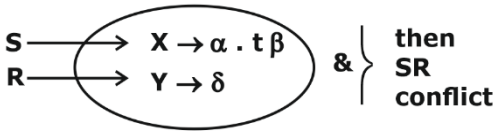
SR : Shift Reduce Conflict



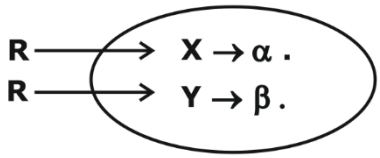
Reduce conflict, then RR. **RR**

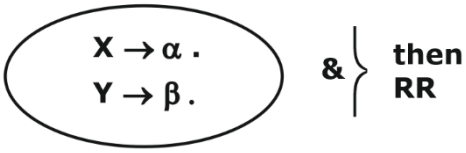
SLR (1) Parser

SR



RR : Reduce



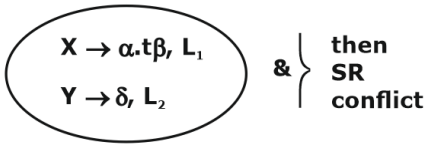


Follow (x) ∩ follow (y) ≠ Φ

- **LALR(1) and CLR(1) :**

  Same as SLR(1), but instead --------- use the provided lookahead.

SR



$t \in L_2$

RR

$L_1 \cap L_2 \neq \varphi$

- **Inadequate Static :** A static having ANY conflict is called a conflicting static or inadequate static.

  NOTE The static $S' \to S.$ or $S' \to S.,\$$ is excepted static, and this is not a reduction.

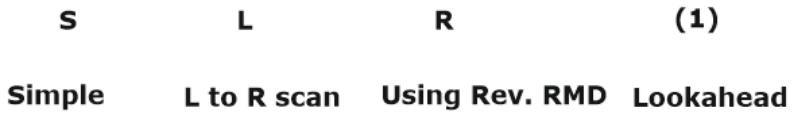
  * The only difference b/w CLR(1) and LALR(1) is that, the states with the similar items, but different lookaheads are merged together to reduce space.

$$\#\text{state in } LR(0) = \#\text{Static in } SLR(1) = \#\text{Static in } LALR(1) = \leq \#\text{Static in } CLR(1)$$

- **Important Points :**

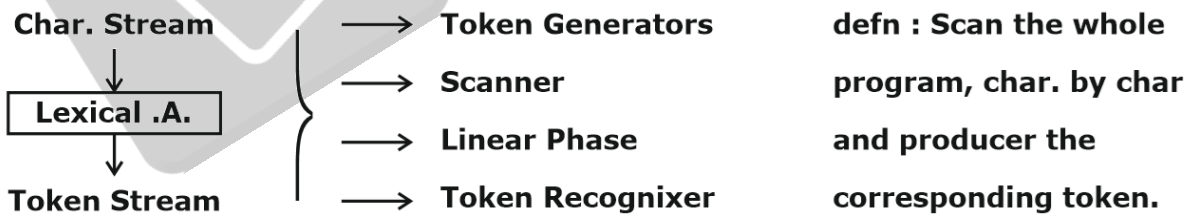
  1. If CLR (1) doesn't have any conflict, then conflict may or may not arise after merging in LALR(1)
  2. If LALR (1) has SR-conflict, then we can conclude that CLR(1) also has SR-conflicts.
  3. LALR (1) has SR-conflict if and only if CLR (1) also has SR.

  * We can construct Parser for every unambiguous regular grammar [CLR (1) Parser].



- **Very Important Point :**
  LALR (1) Parser can parse non LALR (1) grammar which only has SR-conflict by favouring shift over reduce.
  Eg.
  E → E + E | E * E | id | 2 + 3 * 5 ⇒ E + E. * 5

- **CD – Lexical Analysis**



  * Also produce/reports the text Lexical Errors (if any)

- **Functions of Lexical Analyzer:**

  i) Scans all the characters of the program.

  ii) Token Recognizer.

  iii) Ignores the comment & spaces

  iv) Maximal Matching Rule [Longest prefix match].

  - **NOTE:**

    The Lexical analyser uses, the Regular Expression.

    - Prioritization of Rules.

    - Longest Prefix match

    Lexeme → Smallest unit of program or Logic

    Token → Internal representation of Lexeme.

    Lexmes
    LA
    Tokens

    Types of Token:

    1. Identifier
    2. Keywords
    3. Operators
    4. Literals/constants
    5. Special symbol

    Token Separation:

    1. Spaces
    2. Punctuation

    Implementation:

    → LEX tool ⇒ Lex. yy. C

    * All identifier will have entry in symbol Table, LA, gives entries into the symbol Table.

    [Regular Expression → DPA → Lexical Analyzer]

- **Find no. of Tokens :**

  1. void main ( )
  { printf (“gate”);
  [10 Tokens]

  2. int x, * P ;
  X = 10;
  P = & x;
  x + +;
  [18 tokens]

  3. int x;
  x = y ;
  x = = y;
  [11 Tokens]

  4. int 1 x 2, 3 ;
  [Lexical Error]

5. Char ch = 'A' ;

[5 Token]

6. char ch = 'A ;

Lexical Error

7. char * P = "gate";

[6 Tokens]

8. char * P = "gate";

[Error]

9. int x = 10 ;

/* comment

x = x + 1;

Error

10. int x = 10;

Comment * /

x = x + 1;

[14 Tokens]

- **CD - Syntax Directed Translation**

  - **Example :**

    $S \rightarrow S_1 S_2$ [S. count = S_1 count + S_2 count]

    $S \rightarrow (S_1)$ [S. count = S_1 count + 1]

    $S \rightarrow \epsilon$ [S. count = 0]

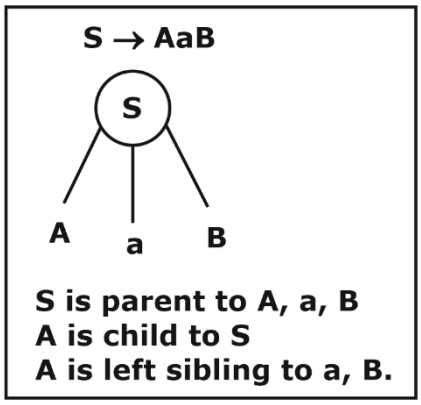
    * Count is an attribute for non-terminal.

- **Application of SDT :**

  1. Used to perform **Semantic Analysis**
  2. Produce Parse Tree
  3. Produce intermediate Rep.
  4. Evaluate an expression
  5. Convert infix to prefix or postfix.

- **Attributes :**

  1. Inherited Attribute
  2. Synthesized Attribute



- **Inherited Attribute : (RHS)**

{A. x = f(B.x | S.x)}

The computation at any node (non-terminal) depends on **parent or sibling(s)**.

* In Above example x is inherited attribute.

- **Synthesized Attribute : (LMS)**

{S.x = f(A.x | B.x}

x is synthesized attribute.

The computation of any node (non-terminal) depends on children.

- **Identifying Attribute Type :**

* Always check every Translation.

1.

$D \rightarrow T : L$; {L. Type = T. Type} inherited.

$L \rightarrow L_1$, id ; { L1 . Type = L. Type} inherited

$L \rightarrow id$

$T \rightarrow$ integer {T. type = int} synthesized.

Type in neither inherited nor syntesized.

2.

$E \rightarrow E_1 + T$ {E.val = E_1.val + T.val} synth.

$E \rightarrow T$ {E. val = T. val} synth.

$T \rightarrow T_1 - F$ {T.val = T_1 . val - F. val} synth.

$F \rightarrow$ id {F. val = id. val} synth.

Val is synthesized attribute.

3.

$S \rightarrow AB$ {Aa = B.x ; S.y = A. x} x is inherited | y is synth

$A \rightarrow a$ {A. y = a} y is synth

$B \rightarrow b$ {B.y = a.y} y is synth

x ⇒ Inherited

y ⇒ Synthesized

4.

$D \rightarrow T\ L$ {L. in = T. type} inherited(in)

$T \rightarrow$ int {T. type] int} synth (type).

$L \rightarrow$ id {Add Type (id. entry, L. in)}

in ⇒ Inherited

type ⇒ Synthesized

- **Syntax Directed Definitions (SDDs) : (Attribute Grammar)**

1. L - Attributed Grammar :

→ **Attribute is synthesized or restricted inherited.**

→ **Parent**

→ **Left Sibling**

Only

→ **Translation can be appended any where is RHS of production.**

2. S - attributed Grammar :

→ **Attribute is synthesized only**

→ **The translation is placed only at the end of production.**

$S \rightarrow AV \{A.x = S.x + 2\}$

or, $S \rightarrow AB \{ B.x = f(A.x \mid S.x) \}$

or, $S \rightarrow AB \{ S.x = f(A.x \mid B.x) \}$

Eg. $S \rightarrow AB \{ S.x = f(Ax \mid Bx) \}$

→ **Evaluation :** Rev. RMD (Bottom up Parsing)

L – Attri.

S – Attri.

- **Identify SDD :**

$E \rightarrow E_1 + E_2$ {E. type = if (E_1. Type = = int & & E. type = = int) then int} synth.

$E \rightarrow id$ {E. type = lookup (id .entry)} else type error. Synth.

∴ type is synthesized, hence S-Attribute and also L – attributed Grammar.

* Every S-attributed Grammar is also L-attributed Grammar.

* For L-attributed Evaluation, use the In-order of annotated Pares Tree.

* For S-attributed, Reverse of RMD is used.

→ Find RMD order

→ Consider in Reverse

- CD- Intermediate Representation

Representations

Linear Form

→ **Postfix code**

→ **3-Attr. code (3AC)**

→ **static - single assignment (SSA code).**

Non-Linear Form

→ **Syntax Tree**

→ **Directed A cyclic graph DAG**

→ **Control Flow Graph**

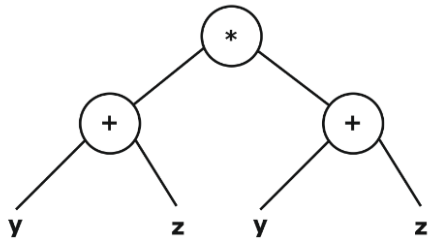
- **Example expression :** (y + z) * (y + z)

Postfix → $\boxed{yz + yz + *}$

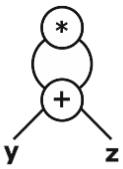
SSA → $\boxed{\begin{matrix} f_1 = y + z \\ f_2 = f_1 \times f_1 \end{matrix}}$ ⇒ f_1 and $f_2$cannot be reassigned

3AC → $\boxed{\begin{matrix} t_1 = y + z \\ t_2 = t_1 * t_1 \end{matrix}}$

Syntax Tree →



DAG →



Reverse the atready existing common sub expression.

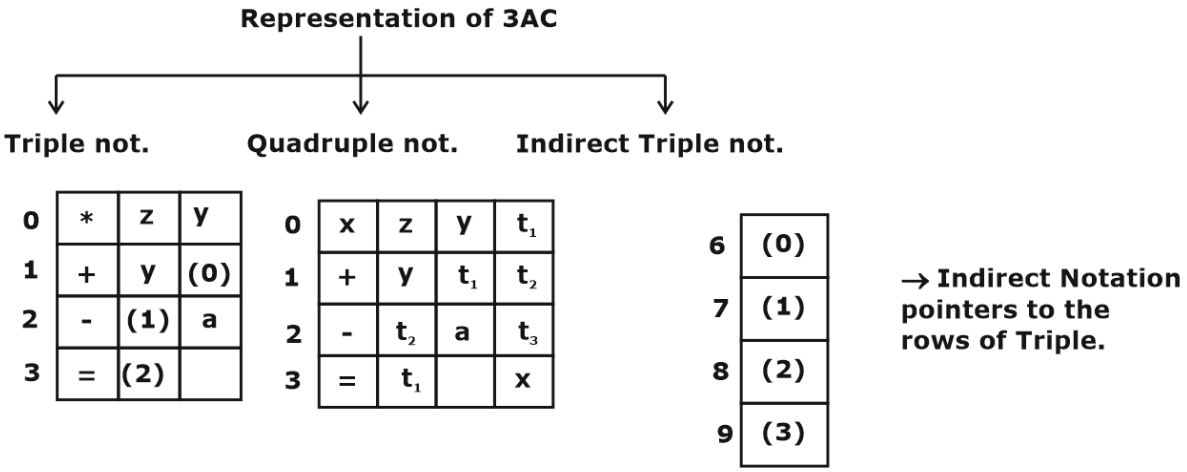
- **3-Address Code :** Code in which, at most 3 addresses. [including LHS]

address → name, variable, constant

$x = y + z * y - a$

1, 2, 3

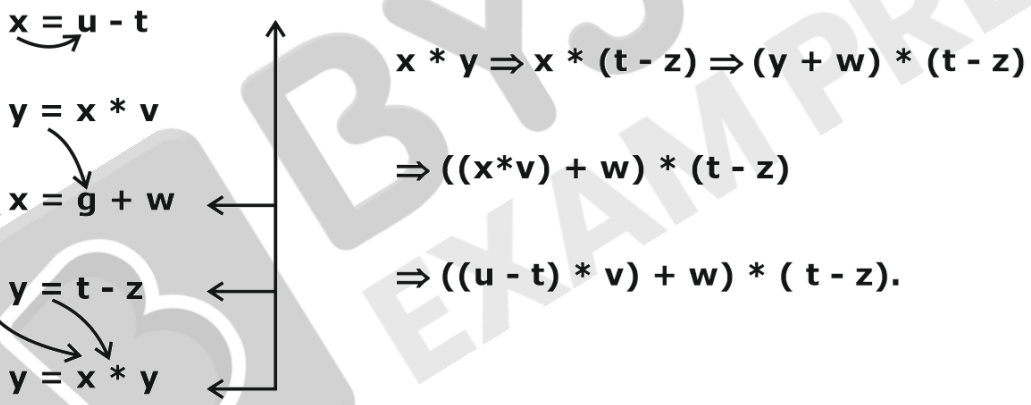
⇒ $\begin{bmatrix} t_1 = z * y \\ t_2 = y + t_1 \\ t_3 = t_2 - a \\ n = t_3 \end{bmatrix}$ Equivalent 3-adder code



Triple Notation	**Quadruple**
→ Space efficient	→ space ineft.
→ time in inefficient	→ time eff.

* 3AC done using operator precedence.

- Find min no of variable reqd. in equivalent 3AC :



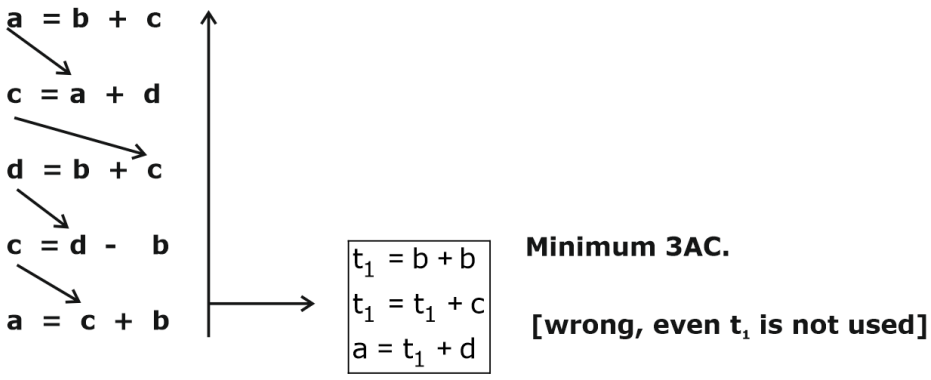
$$\begin{bmatrix} u = u - t & v = t - z \\ v = u * u & z = w * v \\ w = v + w & \end{bmatrix} \Rightarrow 5 \text{ variables only.}$$

$a \Rightarrow \underline{e} + b \Rightarrow \underline{d} - b + b \Rightarrow b + \underline{c} - b + b$

$\Rightarrow b + \underline{a} + d - b + b$

$\Rightarrow b + b + c + d - b + b$

$\Rightarrow b + b + c + d$



∴ Minimum :

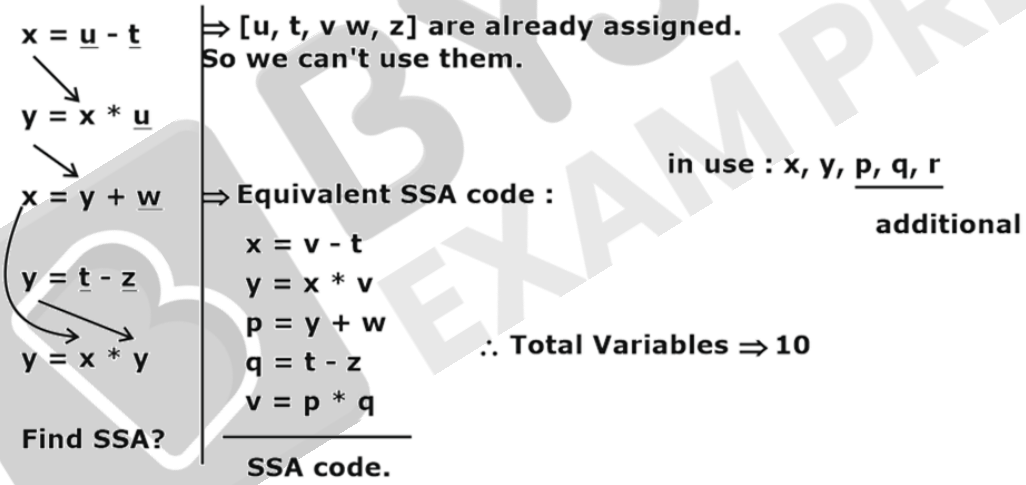
$\left.\begin{matrix} b = b + b \\ c = b + c \\ a = c + d \end{matrix}\right\} \Rightarrow$ only 3 variable [Most optimal]

- **Static Single Assignment code (SSA code) :**

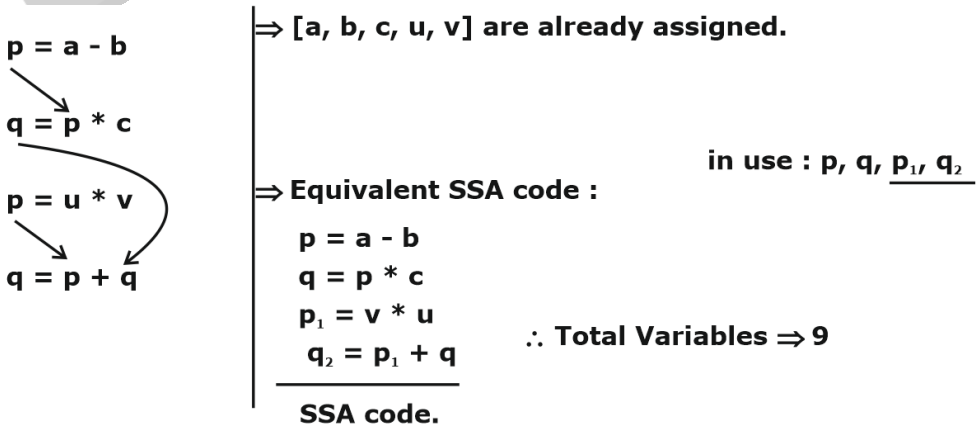
  Every variable (addr) in the code has single assignment.

  [Single meaning] + 3 AC.

  (1)



  (2)



- **Control flow Graphs :**

  Basic Blocks : Seq. of 3-addr code, which control entire from 1st stmt and exists from last.

  * Basic blocks can never contain jump statement in b/w.

  Find Leaders to identify basic blocks.

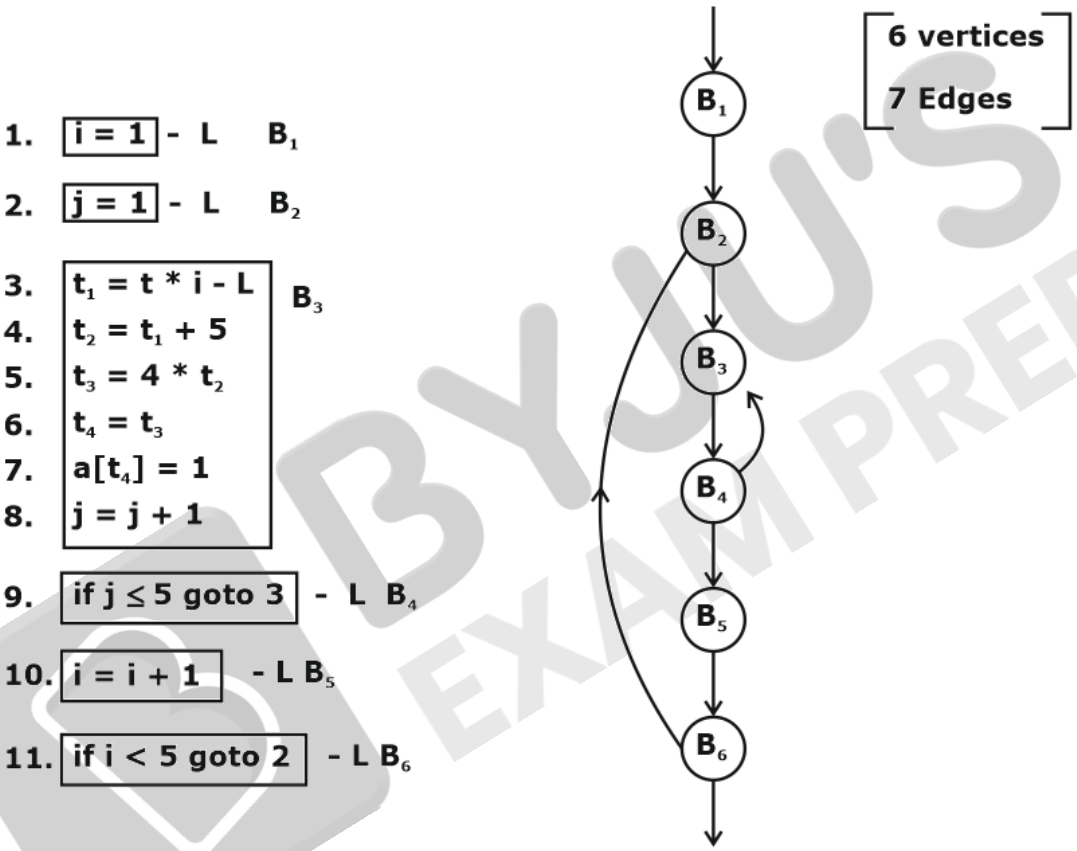
  → 1st 3 AC is leader

  → Target of Jumps are Leader

  → Statement Just below Jump are Leaders

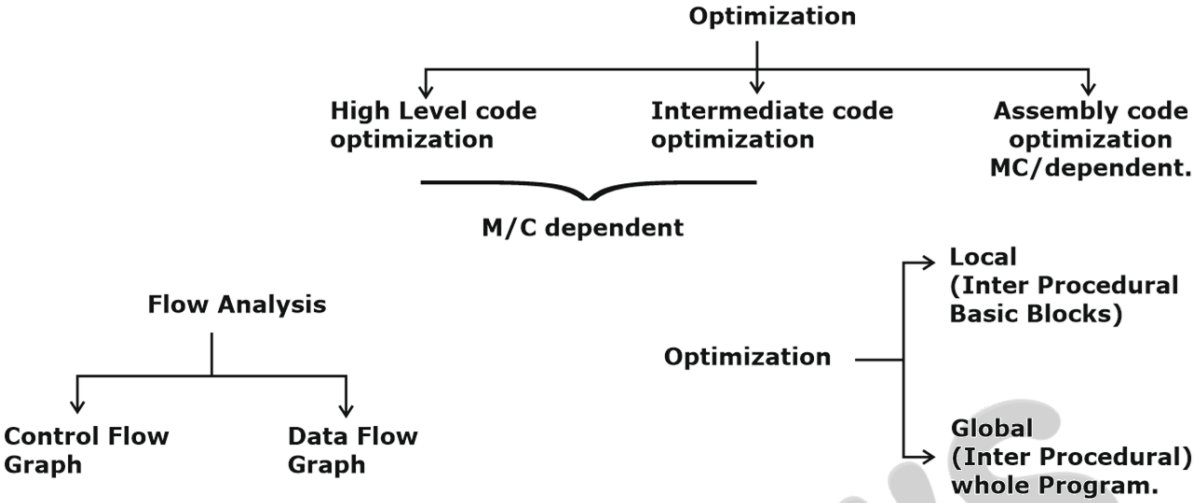
  → Jump is itself a Leader.

  - **Example :**



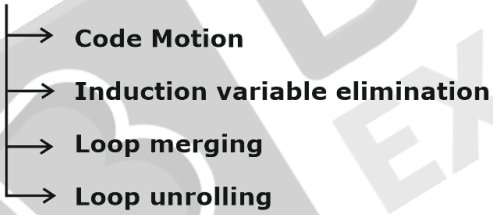
- **CD – Code Optimization**

  → Saves space/time. (Basic Objective)

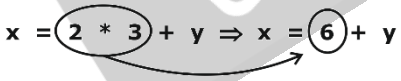


- **Optimization Methods :**

  1. Constant folding
  2. Copy propagation
  3. Strength reduction
  4. Dead code elimination
  5. Common sub expr elimination
  6. Loop optimization



1. Constant Folding :



Folding

x = 2 + y * 3] Can't fold the constants.

2. Copy Propagation :

i) Variable propagation :

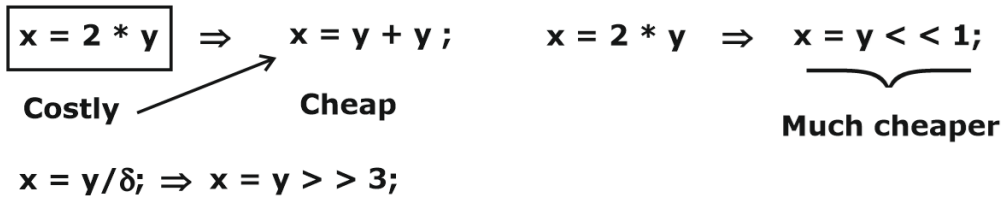
x = y ;
z = x + 2; ⇒ **z = y + 2;**

ii) Constant propagation :

x = 3
z = x + a; ⇒ **z = 3 + a;**

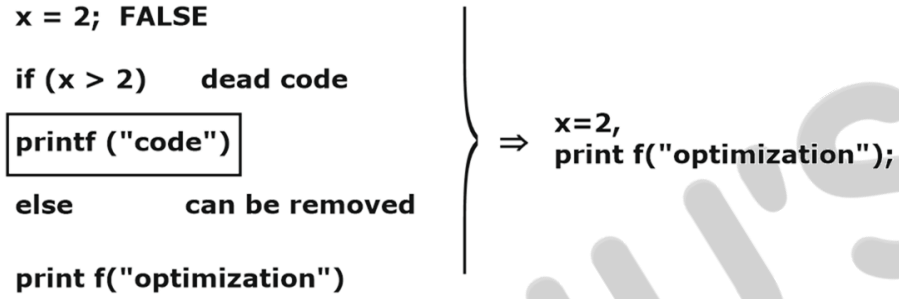
3. Strength Reduction :

Replace expensive statement/instruction with cheaper one.



$x = y/\delta; \Rightarrow x = y >> 3;$

4. Dead code Elimination :



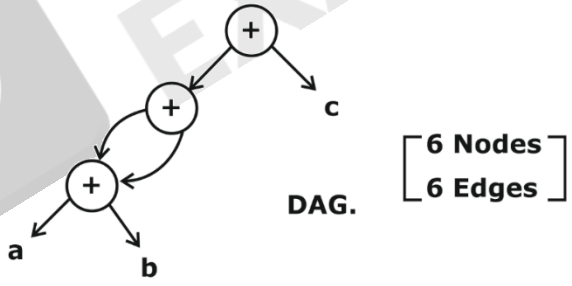
*** Hence, its a code, that never execute, during execution. We can always delete such code.**

5. Common Subexpression elimination

DAT is used to eliminate common sub expression.

Eg. $x = (\underline{a + b}) + (\underline{a + b}) + c ; \Rightarrow \quad t_1 = a + b$

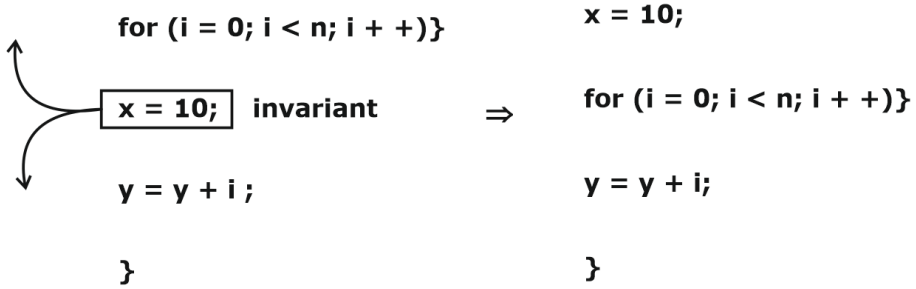
$x = t_1 + t_1 + c$



6. Loop Optimization :

(i) Code Motion – Freq Reduction :

Move the loop invariant code outside of Loop.



(ii) Induction Variable elimination :

```
i1 = 0 ;                          for (i = 0; i < n; i ++){
i2 = 0 ;                    ⇒         A [i] = B [i];
for (i = 0; i < n; i + + ) {      }
    A[i1 ++] = B [i2 ++];         Only 1 induction variable : i
}
```

i,i1, i2 : 3 induction variables

(iii) Loop Merging/combining : (Loop Jamming)

```
3n + 2                    for (i = 0; i < n; i ++)          for (i = 0; i < n; i ++){
                              A [i] = i + 1;                    A [i] = i + 1
3n + 2                    for (j = 0; j < n; j ++)      ⇒       B [i] = i - 1
------                        B [j] = j + 1;                }
6n + 4                                                                          4n + 2
```

Reduced

(iv) Loop cooling:

(1)

```
for (i = 0; i < 3; i ++)          ⇒      Print f("CD");
    print f ("CD");                      Print f("CD");
3 × 3 + 2 = 11 Statements         →      3 statements
```

(2)

```
for (i = 0; i < 2n; i ++) {       ⇒      for (i = 0; i < n; i ++) {
    print f("CD");                       printf ("CD");
{                                        P
(2 × 3n + 2) = 6n + 2                    (4n + 2)
```

****