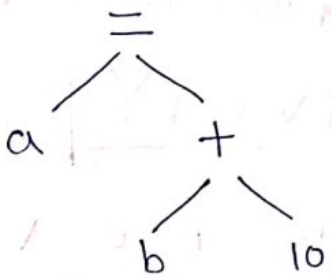
UNIT II : Syntax Analysis (Part-I)

* **Parser** : A Parsing or syntax analysis is a process which takes the input string 'w' & Produces either a Parse tree (syntactic structure) or generates the syntactic errors.

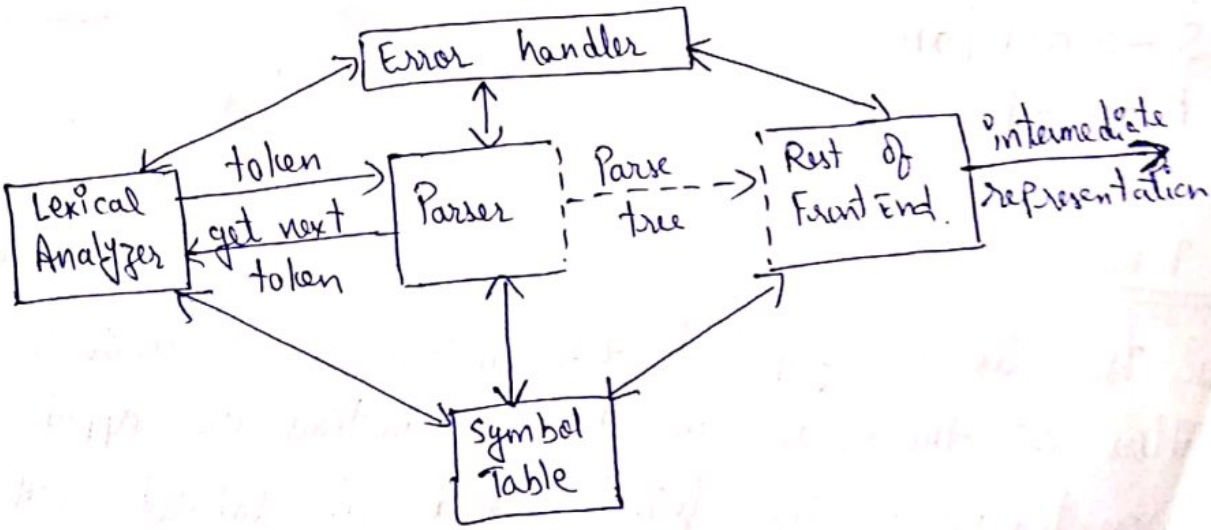
→ The syntax analyzer takes the tokens as input & generates a tree like structure called '**Parse tree**'.

Eg : $a = b + 10$



Parse tree for $a = b + 10$.

(*)



Position of Parser in compiler model

→ Specification of input can be done by using "Context-Free Grammar" (CFG).

Scanned by CamScanner

* Context-Free Grammar:

The CFG, 'G' is a collection of the following:

$$G_1 = (V, T, S, P)$$

- 'V' is a set of non-terminals (All upper-case letters) (Syntactic variables that denote sets of strings)
- 'T' " " " " terminals (basic symbols from which strings are formed)
- 'S' is a start symbol
- 'P' is a set of Production rules.

→ The Production rules are must be in the format:

$$\boxed{\text{Non-terminal} \longrightarrow (V \cup T)^*}$$

(or) $\boxed{A \longrightarrow \alpha}$ where $A \in V$

$\alpha \in (V \cup T \cup \epsilon)^*$.

eg: $S \rightarrow aSb \mid aA$

$A \rightarrow a \mid \epsilon$.

* Parse trees:

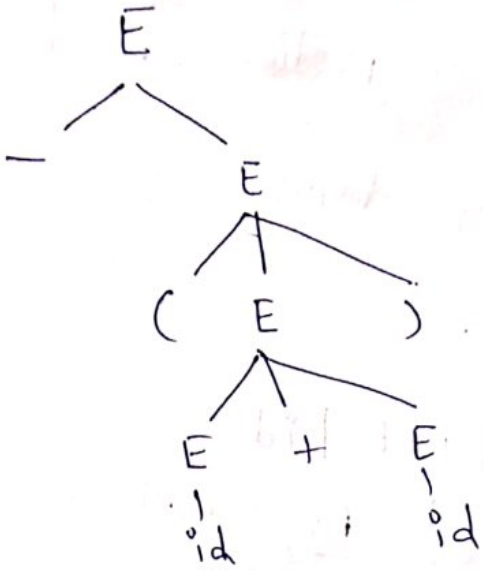
A Parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals. The interior node is labeled with the non-terminal.

→ Each interior node of a Parse tree represents the application of a Production.

Scanned by CamScanner

Eg: $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$.

generate Parse tree for $-(id+id)$

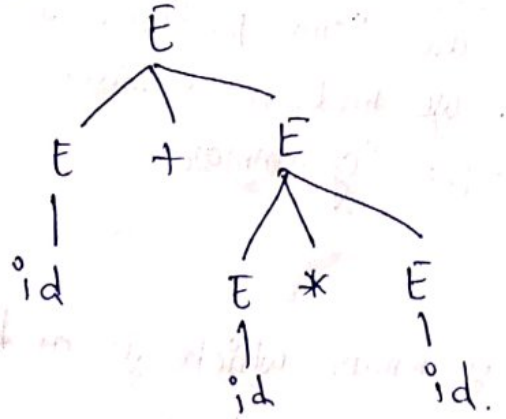


Parse tree

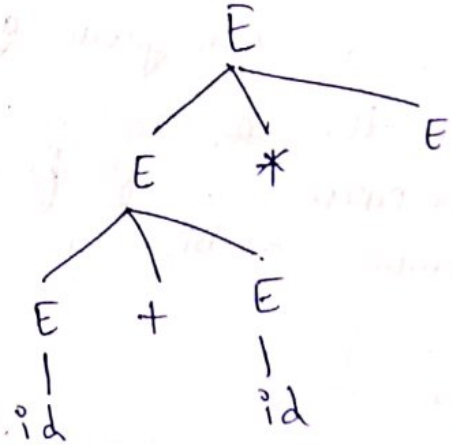
* **Ambiguous Grammar:** A grammar is said to be ambiguous if it generates more than one parse trees for same sentence of language $L(G)$.

Eg: $E \rightarrow E+E \mid E*E \mid id$.

id+id*id:



Parse tree 1



Parse tree 2

Since we have more than one Parse tree for id+id*id, the given grammar is Ambiguous Grammar.

Scanned by CamScanner

* **Derivation** : Derivation from 'S' means generation of string 'w' from 'S'. For constructing derivation two things are important.
  - choice of Non-terminal from several others.
  - choice of rule from Production rules for corresponding non-terminal.

→ we have two types of derivations:
  a) Left most derivation
  b) Right most derivation.

* **Example** : $E \rightarrow E+E \mid E*E \mid id$

Leftmost derivation

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

Right most derivation

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

* **Eliminating ambiguity** :

Sometimes an Ambiguous grammar can be rewritten to eliminate the ambiguity. There are some problems like left recursion, left factoring. we need to eliminate these to remove ambiguity in the grammar.

(1) **Left Recursion** :

The left recursive grammar is a grammar which is as below :

$A \overset{+}{\Longrightarrow} A\alpha$ where A is a non-terminal
α denotes some i/p string.

(A ... same) $\overset{+}{\Longrightarrow}$ means deriving the i/p in one or more steps.

Scanned by CamScanner

→ Because of left recursion, the top down parser can enter into infinite loop. So we need to eliminate it.

* Removing left recursion

> $A \longrightarrow A\alpha \mid \beta$ is rewritten as :
>
> $A \longrightarrow \beta A'$
> $A' \longrightarrow \alpha A' \mid \epsilon$

Example :

$E \longrightarrow E+T \mid T$
$T \longrightarrow T*F \mid F$
$F \longrightarrow id \mid (E)$

we have left recursion in the following Production rules.

$E \longrightarrow E+T \mid T$ (same A; α = $+T$; β = T)

$T \longrightarrow T*F \mid F$ (same A; α = $*F$; β = F)

removing left recursion we get :

$E \longrightarrow TE'$
$E' \longrightarrow +TE' \mid \epsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' \mid \epsilon$
$F \longrightarrow id \mid (E)$

$\frac{E}{A} \longrightarrow \frac{E}{A}\frac{+T}{\alpha} \mid \frac{T}{\beta}$

$\frac{T}{A} \longrightarrow \frac{T}{A}\frac{*F}{\alpha} \mid \frac{F}{\beta}$

Scanned by CamScanner

② Left factoring :

Consider a grammar :

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Here it is not possible for us to take a decision whether to chose first rule or second. The above grammar can be left factored as : rewritten as :

$$A \longrightarrow \alpha A'$$
$$A' \longrightarrow \beta_1 \mid \beta_2$$

* Example : Grammar : $S \longrightarrow iEtS \mid iEtSeS \mid a$
$E \longrightarrow b$

The above grammar after left factoring becomes as :

$S \longrightarrow iEtSS' \mid a$
$S' \longrightarrow eS \mid \epsilon$
$E \longrightarrow b.$

$S \longrightarrow \underset{\alpha}{iEtS} \mid \underset{\alpha}{iEtSeS} \mid \underset{\beta_2}{a}$

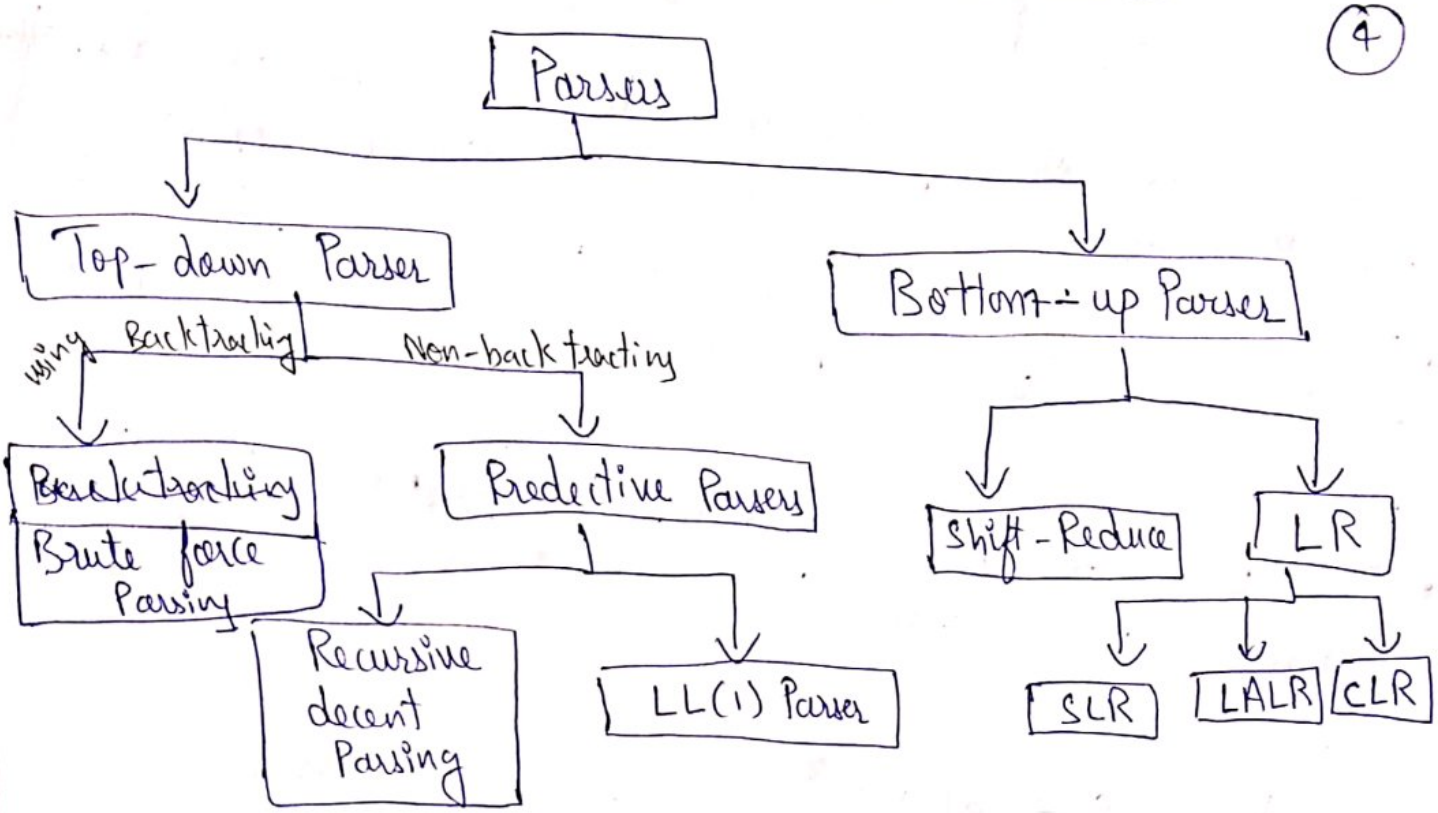
→ By left factoring we may be able to rewrite the Production in which the decision can be deferred until enough of the input is seen to make the right choice.

* Parsing Techniques :

There are two Parsing techniques :

1. Top - Down Parsing (we will Parse from Root to leaf node)
2. Bottom - Up Parsing. (" " " leaf to root node)

Scanned by CamScanner



Types of Parsing Techniques

* (1) Top Down Parsing :-

The top-down construction of a Parse tree is done by starting with the root, labeled with the starting non-terminal, & repeatedly Performing the following two steps

- At node n, labeled with non-terminal A, select one of the Productions for 'A' & construct children at 'n' for the Symbols on the right side of the Production.
- Find the next node at which a sub-tree is to constructed.

→ Top-down Parsing can be viewed as an attempt to find a left-most derivation for an input string. A general form of top-down Parsing is called "Recursive descent Parsing".

Scanned by CamScanner

(a) Brute-force Parsing :

- Given a particular non-terminal that is to be expanded, the first Production for this non-terminal can be applied.
- Then, within this newly expanded string, the left most non-terminal is selected for expansion & its first Production is applied.
- This Process of applying Productions is repeated for all subsequent non-terminals that are selected untill the Process cannot be continued.

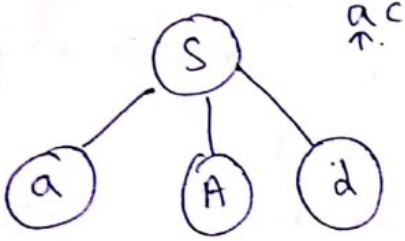
* Example : S → aAd | aB , w = accd.

A → b | c

B → ccd | ddc

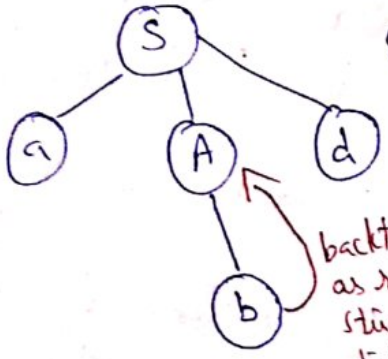
step 1 : S — accd.

step 2 :



accd

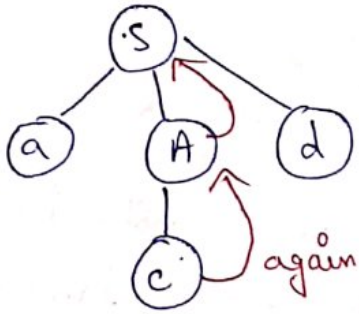
step 3 :



accd

Generated : ab.

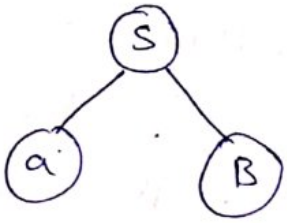
step 4 :



acd we got

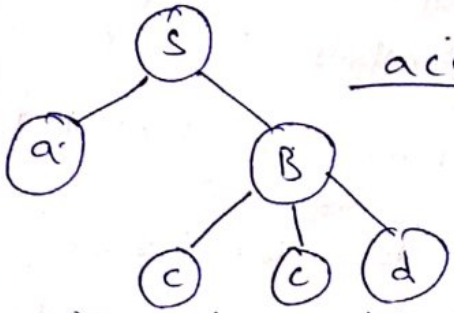
accd

step 5 :



accd

step 6 :



accd

here the given string match with generated string is done so it is ~~success~~ successful Passing.

Scanned by CamScanner

(b) Recursive Descent Parsing:

A Recursive-descent parsing program consists of a set of Procedures, one for each non-terminal. Execution begins with the Procedure for the start symbol, which halts & announces success if its Procedure body scans the entire input string.

→ CFG is used to build the recursive routines. The RHS of the Production rule is directly converted to a Program. For each non-terminal a separate Procedure is written & body of the Procedure (code) is RHS of the corresponding non-terminal.

* Basic steps for construction of Recursive descent Parser:

The RHS of the rule is directly converted into program code symbol by symbol

(1) If the input symbol is non-terminal then a call to the Procedure corresponding to the non-terminal is made.

(2) If the input symbol is terminal then it is matched with the lookahead input. The lookahead pointer has to be advanced on matching of the input symbol.

(3) If the Production rule has many alternates then all these alternates has to combined into a single body of Procedure

(4) The Parser should be activated by a Procedure corresponding to the start symbol.

Scanned by CamScanner

```
Void A() {
    Choose an A-production, A → X1 X2 .... Xk ;
    for (i = 1 to k) {
        if (Xi is a nonterminal)
            call Procedure Xi() ;
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol ;
        else /* an error has occured */ ;
    }
}
```

A typical procedure for a non-terminal in a top-down Parser

(c) Predictive LL(1) Parsing :

Predictive LL(1) Parsing is of non-recursive type. In this type of Parsing a table is built.

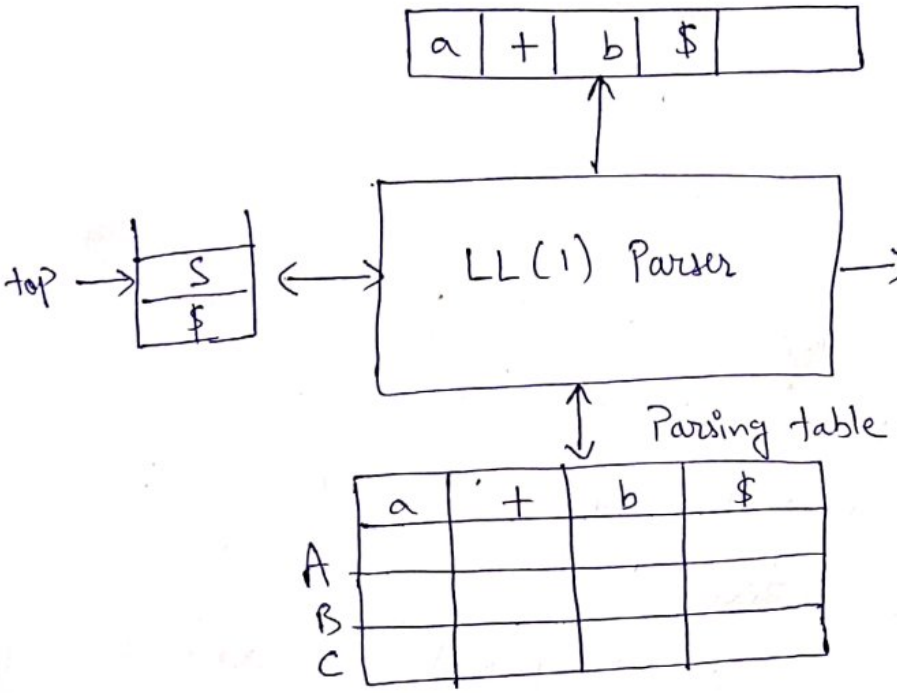
LL (1) → lookahead (uses only one i/p symbol to Predict Parsing Process)

- i/p is scanned from left to right.
- leftmost derivation

→ The data structures used by LL(1) are a) input buffer, b) stack, c) Parsing table. The LL(1) parser uses input buffer to store the input tokens. The stack is used to hold the left sentential form. The symbols in RHS of rule are pushed into the stack in reverse order i.e from right to left.

Scanned by CamScanner

Use of stack makes this parsing non-recursive. The table is basically a two dimensional array. The table can be represented as M[A, a] where 'A' is a non-terminal & 'a' is current input symbol..



Model for LL(1) Parser

→ The Parsing Program reads top of the stack & a current input symbol. with the help of these two symbols the Parsing action is determined. The Parser consults the table M[A, a] each time while taking the parsing actions. Hence this type of Parsing method is called "table drive Parsing".

* Construction of Predictive LL(1) Parsing:

~~Step 1:~~ ~~Comp~~

→ In LL(1) Parsing, Pre-Processing steps are as follows:

Scanned by CamScanner

1. Elimination of left Recursion.
2. left-factoring the grammar.
3. Computation of FIRST & FOLLOW.
4. Constructing Predictive Parsing Table using FIRST & FOLLOW.
5. Parse the input string with the help of Predictive Parsing table.

* FIRST( ) :

FIRST(α) is a set of terminal symbols that are first symbols appearing at RHS in derivation of 'α'.

→ Following are the rules used to compute FIRST functions:

- If 'X' is a terminal then First(X) is just 'X'.
- If there is a Production $X \rightarrow \epsilon$(epsilon) then add 'ϵ' to first(X).
- If there is a Production $X \rightarrow aB$ then add 'a' to first(X) where 'a' is a terminal & 'B' is non-terminal.
- If there is a Production $X \rightarrow Y_1Y_2 \cdots Y_k$ then first($Y_1Y_2 \cdots Y_k$) to first(X).
- First($Y_1Y_2 \cdots Y_k$) is either:
  - i) First(Y_1) (if First(Y_1) doesn't contain 'ϵ')
  - ii) OR (if First(Y_1) does contain 'ϵ') then First($Y_1Y_2 \cdots Y_k$) is everything in First(Y_1) <except for 'ϵ'> as well as everything in First($Y_2 \cdots Y_k$).

Scanned by CamScanner

iii) If First(Y_1) First(Y_2) ... First(Y_k) all contain 'ϵ' then add 'ϵ' to First($Y_1 Y_2 \cdots Y_k$) as well.

* Example :

(1). $E \rightarrow E+T \mid T$ (Left Recursion exists.)
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

sol : we need to remove left recursion from the grammar.
After removal of left recursion the grammar is :

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id.$

(*) First(E) = $E \rightarrow T$ now go to 'T' RHS production & look.
$T \rightarrow FT'$ (again non-terminal)
so go to 'F' RHS side production
$F \rightarrow (E) \mid id$ (id → terminal.)
∴ First(E) = { (, id }

- First(E) = First(T) = First(F) = { (, id }
- First(E') = { +, ϵ }
- First(T') = { *, ϵ }

(2). $S \rightarrow aB \mid aC \mid Sd \mid Se$
$B \rightarrow bBc \mid f$
$C \rightarrow g.$

sol : First(S) = { a }
First(B) = { b, f }
First(C) = { g }

Scanned by CamScanner

* FOLLOW ( ) :

Let $S \rightarrow \alpha A a \beta$ where α & β may be terminals or non-terminals. FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of 'A'.

→ Rules for computing FOLLOW are :

- For the start symbol 'S' place "$" in FOLLOW (S).
- If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST (β) without 'ϵ' is to be placed in FOLLOW(B).
- If there is a Production $A \rightarrow \alpha B \beta$ @L $A \rightarrow \alpha B$ and FIRST (β) = {ϵ} then FOLLOW (A) = FOLLOW (B) i.e everything in FOLLOW (A) is in FOLLOW (B).

* Examples :

(1)
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$.

Sol : • FOLLOW (E) = { $,) }

check all RHS side Productions where 'E' is there, we have
$F \rightarrow (E) \mid id$
↳ first(')') = ')'.
∴ FOLLOW(E) = { $,) }
since 'E' is start symbol place '$' in its follow.

- FOLLOW(E') = { $,) }
- FOLLOW (T) = { +, $,) } ⇒ $E \rightarrow TE'$
  first(E') = {+, ε} → to follow (E) = FOLLOW (T)
  FOLLOW(T) = {+, $,) }
- FOLLOW (T') = { +, $,) }
- FOLLOW (F) = { *, +, $,) }

Scanned by CamScanner

(2). $S \rightarrow aB \mid aC \mid Sd \mid Se$

$B \rightarrow bBc \mid f$

$C \rightarrow g$.

Sol: FOLLOW(S) = { $, d, e }

FOLLOW(B) = { c }

FOLLOW(C) = { d, e, $ }

* Construction of Predictive Parsing table:

The construction of Predictive Parsing table is an important activity in Predictive Parsing method. This algorithm requires FIRST & FOLLOW functions.

Input: Context-free grammar 'G'.

Output: Predictive Parsing table 'M'.

Algorithm: For the rule $A \rightarrow \alpha$ of grammar 'G':

- For each 'a' in FIRST(α) create entry M[A, a] = $A \rightarrow \alpha$ where 'a' is a terminal symbol.
- For 'ε' in FIRST(α) create entry M[A, b] = $A \rightarrow \alpha$ where 'b' is the symbols from FOLLOW(A).
- All the remaining entries in the table 'M' are marked as syntax error.

→ This algorithm can be applied to any grammar 'G' to produce a Parsing table 'M'. If the grammar 'G' is left recursive or ambiguous then 'M' will have atleast one multiple defined entry.

Scanned by CamScanner

* LL(1) grammars :- A grammar whose Parsing table has no multiple-defined entries is said to be LL(1). The LL(1) grammars are:

- Scanned from left-to-right.
- are Parsed by a leftmost derivation
- have one symbol look ahead.

* Problems :-

(1) Verify whether the following grammar is LL(1) or not.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

sol :- we need to remove left recursion. After removal the grammar is:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

- First(E) = First(T) = First(F) = {(, id}
- First(E') = {+, ε}
- First(T') = {*, ε}

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

Scanned by CamScanner

* Predictive Parsing Table :

Non-terminals → Input | terminals → id, +, *, (,), $ | First Productions → (| Follow sets →), $

Input	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$.
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Since there are no multiple entries in the above table, the given grammar is LL(1) grammar.

Input string : id + id * id $ is parsed using above table as below :

Stack	Input	Action
$ E' T → store in reverse order	id + id * id $	$E \to TE'$
$ E' T' F	id + id * id $	$E \to FT'$
$ E' T' id	id + id * id $	$F \to id$
$ E' T'	+ id * id $	
$ E'	+ id * id $	$T' \to \epsilon$
$ E' T +	+ id * id $	$E' \to +TE'$
$ E' T	id * id $	stack top & current pointer of input points to same character. Hence Pop it.

Scanned by CamScanner

stack	Input	Action
$ E' T' F	id * id $	T → F T'
$ E' T' id	id * id $	F → id
$ E' T'	* id $	
$ E' T' F *	* id $	T → * F T'
$ E' T' F	id $	
$ E' T' id	id $	F → id
$ E' T'	$	
$ E'	$	T → ϵ
$	$	E' → ϵ

∴ Thus input string gets parsed.

(2) check whether the following grammar is LL(1) grammar or not.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Sol:- After left factoring, the above grammar becomes as:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

First & Follow for the above grammar:-

Scanned by CamScanner

- First (S) = { i, a }
- First (S') = { e, ϵ }
- First (E) = { b }
- Follow (S) = { $, e }
- Follow (S') = { $, e }
- Follow (E) = { t }

→ Predictive Parsing table can be constructed as:

	a	b	e	i	t	$
S	S → a			S → i E t S S'		
S'			S' → ϵ S' → e S	multiple entries.		S' → ϵ
E		E → b				

Since we got multiple entries in M[S', e], this grammar is not LL(1) grammar.

* Error Recovery in Predictive Parsing:

e/n error is detected during Predictive Parsing when the terminal on top of stack does not match the next input symbol or when non-terminal 'A' is on top of the stack, 'a' is the next input symbol, & the parsing table entry M[A, a] is empty.

Scanned by CamScanner

There are two types of error recovery techniques :-

(a) Panic-mode error recovery :- It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

(b) Phase level Recovery :- It is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert or delete symbols on the input & issue appropriate error messages. They may also pop from the stack.

* Bottom-Up Parsing :-

A Bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) & working up towards the root (the top).

→ Input string is taken first & we try to reduce this string with the help of grammar & try to obtain the start symbol. The process of Parsing halts successfully as soon as we reach to start symbol.

Scanned by CamScanner

* Reduction :- Parser tries to identify RHS of production rule & replace it by corresponding LHS. This activity is called "Reduction".

→ The Primary task in bottom-up Parsing is to find the productions that can be used for reduction. The bottom up Parse tree construction process indicates that the tracing of derivations are to be done in reverse order.

* Handle Pruning :-

Handle is a string of substring that matches the right side of Production & we can reduce such string by a non-terminal on left hand side Production.

(OR)

Handle of right sentential form γ is a production $A \rightarrow \beta$ & a position of γ where the string β may be found & replaced by 'A' to produce the previous right sentential form in rightmost derivation of γ.

*Example :- $E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow id$

Consider string id. + id. * id.

$E \xrightarrow{rm} E+\underline{E}$
$E \xrightarrow{rm} E+\underline{E*E}$
$E \xrightarrow{rm} E+E*\underline{id}$

Scanned by CamScanner

$E \xrightarrow{rm} E + \underline{id} * id$

$E \xrightarrow{rm} \underline{id} + id * id$

→ A right most derivation in reverse order can be obtained by "Handle Pruning".

Right Sentential Form	Handle ← Handle	Reduced Production.
$\underline{id} + id * id$	id	$E \rightarrow id$
$E + \underline{id} * id$	id	$E \rightarrow id$
$E + E * \underline{id}$	id	$E \rightarrow id$
$E + \underline{E * E}$	$E * E$	$E \rightarrow E * E$
$\underline{E + E}$	$E + E$	$E \rightarrow E + E$
E		

* Shift - Reduce Parsing :

Shift-reduce Parsing is a form of bottom-up Parsing in which a stack holds grammar symbols & an input buffer holds the rest of the string to be parsed.

→ The initial configuration of shift reduce Parser is as below:

STACK	INPUT
$	ω $

where stack is empty & 'ω' is a string.

Scanned by CamScanner

The shift-Reduce Parser can Perform the following operations:

(1) Shift : Moving of symbols from input buffer onto the stack.

(2) Reduce : If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means RHS of rule is popped off & LHS is pushed in. This action is called 'Reduce action'.

(3) Accept : Announce successful completion of Parsing if the stack contains start symbol only & input buffer is empty at the same time.

(4) Error : A situation in which parser cannot either shift or reduce the symbols. It cannot even. Perform the accept action. This is called as error.

* Example :

(1) E → E-E
E → E*E
E → id

Perform shift-Reduce parsing of the input string "id1 - id2 * id3".

Sol :

Stack	Input Buffer	Parsing Action
$	id1 - id2 * id3 $	shift
$id1	- id2 * id3 $	Reduce by E → id
$ E	- id2 * id3 $	shift
$ E -	id2 * id3 $	shift
$ E - id2	* id3 $	Reduce by E → id
$ E - E	* id3 $	shift

Scanned by CamScanner

$ E-E↑	id3 $	shift
$ E-E*id3	$	Reduce by E→id
$ E-E*E	$	Reduce by E→E*E
$ E-E	$	Reduce by E→E-E
$ E	$	Accept

Here we have followed two rules:

1. If the incoming operator has more priority than in stack operator then perform 'shift'.
2. If the stack operator has same or less priority than the priority of incoming operator then perform 'reduce'.

* Conflicts during shift-Reduce Parsing:

Shift-reduce Parsing cannot be used for all types of CFG's. For some CFG's, shift-reduce parser can reach a configuration in which the parser knowing the entire stack contents & the next input symbol, cannot decide whether to shift to reduce (a shift/reduce conflict) or cannot decide which of several reductions to make (a reduce/reduce conflict).

* Example: Consider the following grammar where the productions are numbered as below:

E → E + T { Print '1' }

E → T { Print '2' }

Scanned by CamScanner

$T \rightarrow T * F$ { Print '3' }

$T \rightarrow F$ { Print '4' }

$E \rightarrow (E)$ { Print '5' }

$F \rightarrow id$ { Print '6' }

If a shift-reduce Parser writes the Production number immedi-tely after Performing any Production, what string will be Printed if the Parser input is id+id*id.

Sol:

Stack	Input	Operation
$	id+id*id$	shift
$id	+id*id$	Reduced by F→id Print '6'.
$F	+id*id$	Reduced by T→F Print '4'.
$T	+id*id$	Reduced by E→T Print '2'
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	Reduced by F→id Print '6
$E+F	*id$	Reduced by T→F Print '4
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	Reduced by F→id Print '6
$E+T*F	$	Reduced by T→T*F Print '3'
$E+T	$	Reduced by E→E+T Print '1'.
$E	$	

Scanned by CamScanner

∴ the final string obtained is 64264631.

* Operator Precedence Parsing :

A grammar 'G' is said to be operator Precedence if it Posses following Properties :

- No Production on the right side is 'ϵ' (epsilon).
- There should not be any Production rule Possessing two adjacent non-terminals at the right hand side.

* Example :

$$E \rightarrow EAE \mid (E) \mid -E \mid id$$
$$A \rightarrow + \mid - \mid * \mid / \mid \wedge$$

This grammar is not an operator Precedent grammar as in the Production rule $E \rightarrow EAE$.

→ It contains two consecutive non-terminals. Hence first we will convert it into equivalent operator Precedence grammar by removing 'A'.

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E$$
$$E \rightarrow (E) \mid -E \mid id$$

→ In operator Precedence Parsing, we will first define Precedence relations $<\cdot$, $=$, and $\cdot>$ between pair of terminals.

	means
• $P <\cdot q$	P gives more Precedence than q.
• $P = q$	P has same " as q.
• $P \cdot> q$	P takes Precedence over q.

Scanned by CamScanner

Operator Precedence (Relation) table :

	id	+	*	$
id		.>	.>	.>
+	<.	.>	.>	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Consider the string : id + id * id .

We will insert $ symbols at the start & end of the input string. We will also insert Precedence operator by referring the Precedence relation table.

$$\$ <\cdot \text{ id } \cdot> + <\cdot \text{ id } \cdot> * <\text{ id } \cdot> \$$$

* Steps to Parse the given string :

- Scan the input from left to right until first .> is encountered.
- Scan backwards over = until <. is encountered.
- The handle is a string between <. and .>.

Parsing is done as follows :

$ <. id .> + <. id .> * <. id .> $	Handle id is obtained b/w <. .> Reduce this by E → id
E + <. id .> * <. id .> $	Handle id is obtained b/w <. .> Reduce this by E → id
E + E * <. id .> $	Handle id is obtained b/w <. .> Reduce this by E → id

Scanned by CamScanner

E + E * E	Remove all the non-terminals
+ *	Insert $ at the beginning & at the end. Also insert the Precedence operators.
$ <· + <· * ·> $	The * operator is surrounded by <· ·>. This indicates that * becomes handle. That means we have to reduce E*E operation first.
$ <· + ·> $	Now '+' becomes handle. hence we evaluate E+E
$ $	Parsing is done.

* Advantage : It is simple to implement.

* Disadvantages :

- The operator like minus has two different Precedence (unary & binary). Hence it is hard to handle such tokens.
- This kind of Parsing is applicable to only small class of grammars.

* Application :

SNOBOL language uses operator Precedence Parsing.

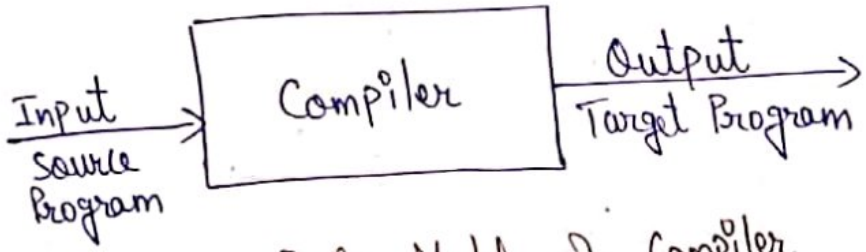
Scanned by CamScanner

COMPILER DESIGN

UNIT I : Overview of Compiler & Lexical Analysis

*** Introduction :** Compiler is a program which takes one language (source Program) as input and translates it into an equivalent another language (target Program).

→ Compilers basically act as translators. The basic model of compiler can be represented as follows :



Basic Model of Compiler

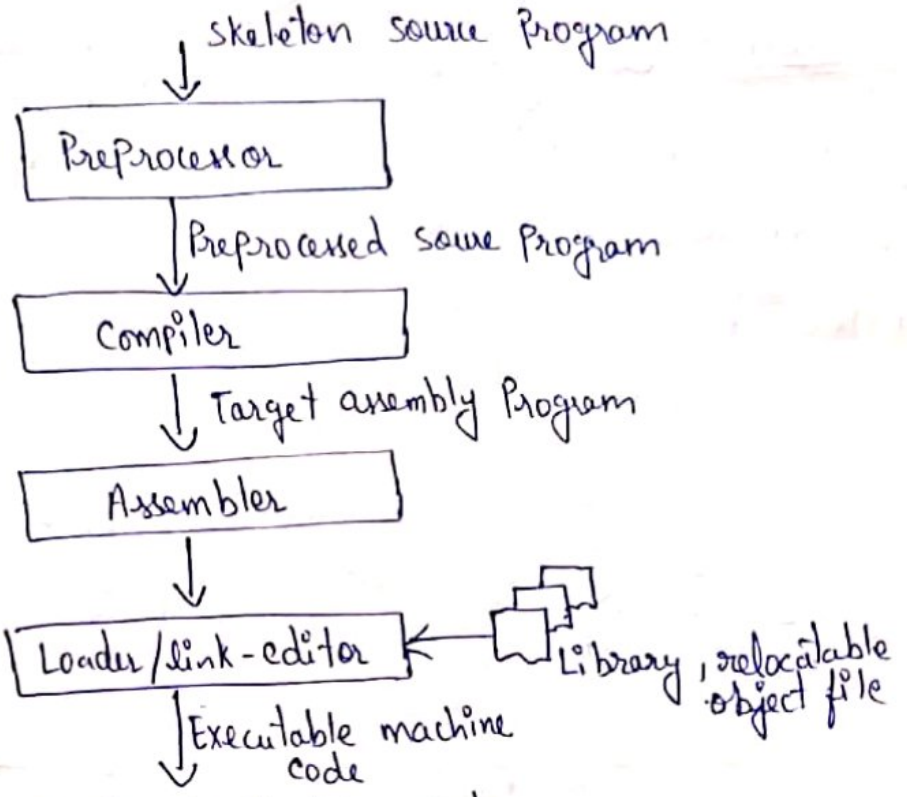
→ The compiler takes a source Program as higher level languages such as C, Pascal, Fortran & Converts it into low level language or a machine level language such as assembly language.

*** Properties of a Compiler :**

- The compiler itself must be bug-free.
- It must generate correct machine-code
- The generated machine code must run fast.
- The compiler must be portable.
- It must give good diagnostics & error messages.
- It must have consistent optimization.

Scanned by CamScanner

→ To create an executable form of your source Program only a compiler Program is not sufficient. we may require several other Programs to create an executable target Program



(or) Target Machine code.

Process of Execution of Program.

* Structure of a Compiler :

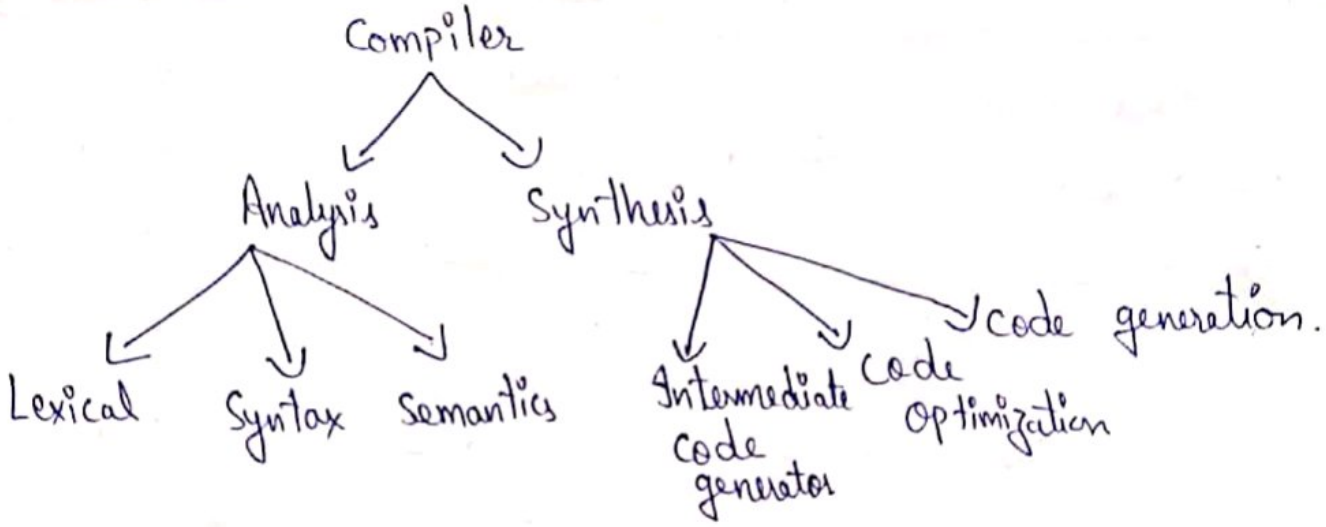
The Process of Compilation is carried out in two Parts namely: "Analysis & Synthesis".

→ Analysis Part is the front end of the compiler.

→ Synthesis Part is the back end of the compiler.

→ Compilation Process operates as a sequence of phases, each of which transforms one representation of the source program to another.

Scanned by CamScanner



Phases of compiler

(a) Lexical Analysis :

The lexical Analysis is also called "Scanning". The complete source code is scanned & the source program is broken up into group of strings called "lexemes". For each lexeme, the lexical analyzer produces as output a "token" of the form : (token-name, attribute-value).

sequence of characters having a collective meaning

→ The blank characters which are used in the Programming statement are eliminated during this phase.

Eg : c := a+10 ;

these are mapped to the tokens like this.

c, a ⇒ identifiers
:= ⇒ assignment
+ ⇒ operator
10 ⇒ constant.

tokens

<id, 1> <:=> <id, 2> <+> <10> ;

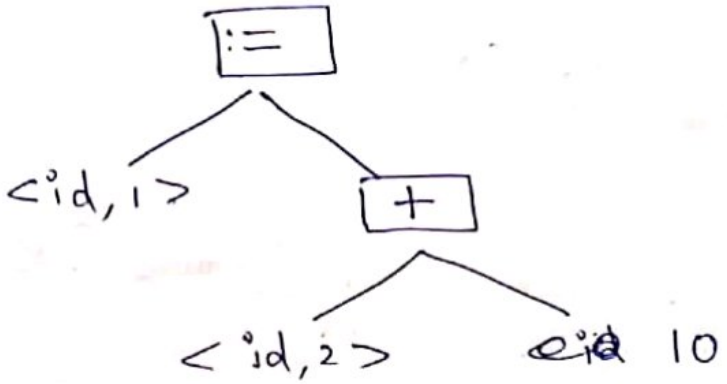
→ we use regular expressions to recognize tokens.

Scanned by CamScanner

(b) **Syntax Analysis** :- The syntax analysis is also called "Parsing". In this phase the tokens generated by the lexical analyzer are grouped together to form a hierarchical tree-like structure, called as Parse tree (or) Syntax tree. We use Context-free grammar (CFG) to recognize syntax or grammar.

Syntax tree for <id,1> <:=> <id,2> <+> ~~<id,3>~~ 10 is :-

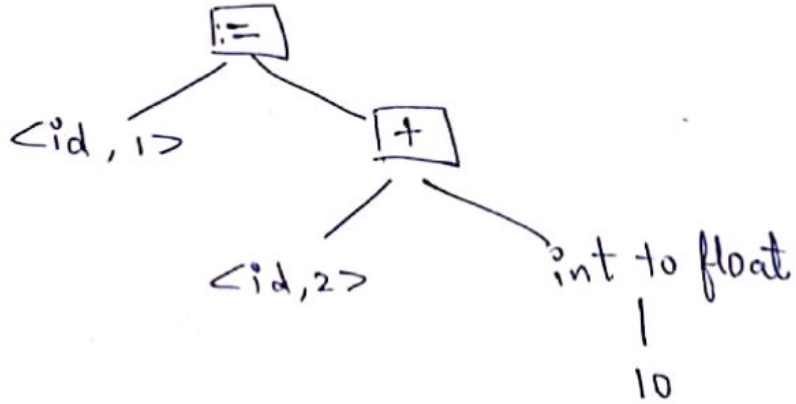
2 Parsing techniques in this phase
- Top-down
- Bottom-up



Parse tree (or) syntax tree.

(c) **Semantic Analysis** :-

Semantic Analysis determines the meaning of the source string. It also does "type checking" where the compiler checks that each operator has matching operands. For example, matching of Parenthesis in the expression, matching of else stmts, checking the scope of operation etc.



Scanned by CamScanner

(d) Intermediate code generation :

The intermediate code is a kind of code which is easy to generate & this code can be easily converted to target code. Intermediate code is of many forms such as three address code, quadruple, triple, Posix etc. For the above example let us convert it into three address form (consists of instructions each of which has at most three operands).

Eg :
$t_1 = \text{int to float}(10)$
$t_2 = id2 + t_1$
$id1 = t_2$

(e) Code optimization :

The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory. By optimizing the code the overall running time of the target program can be improved.

Eg :
$t_1 = id2 + 10.0$ // code is optimized here. unnecessary variables are removed.
$id1 = t_1$

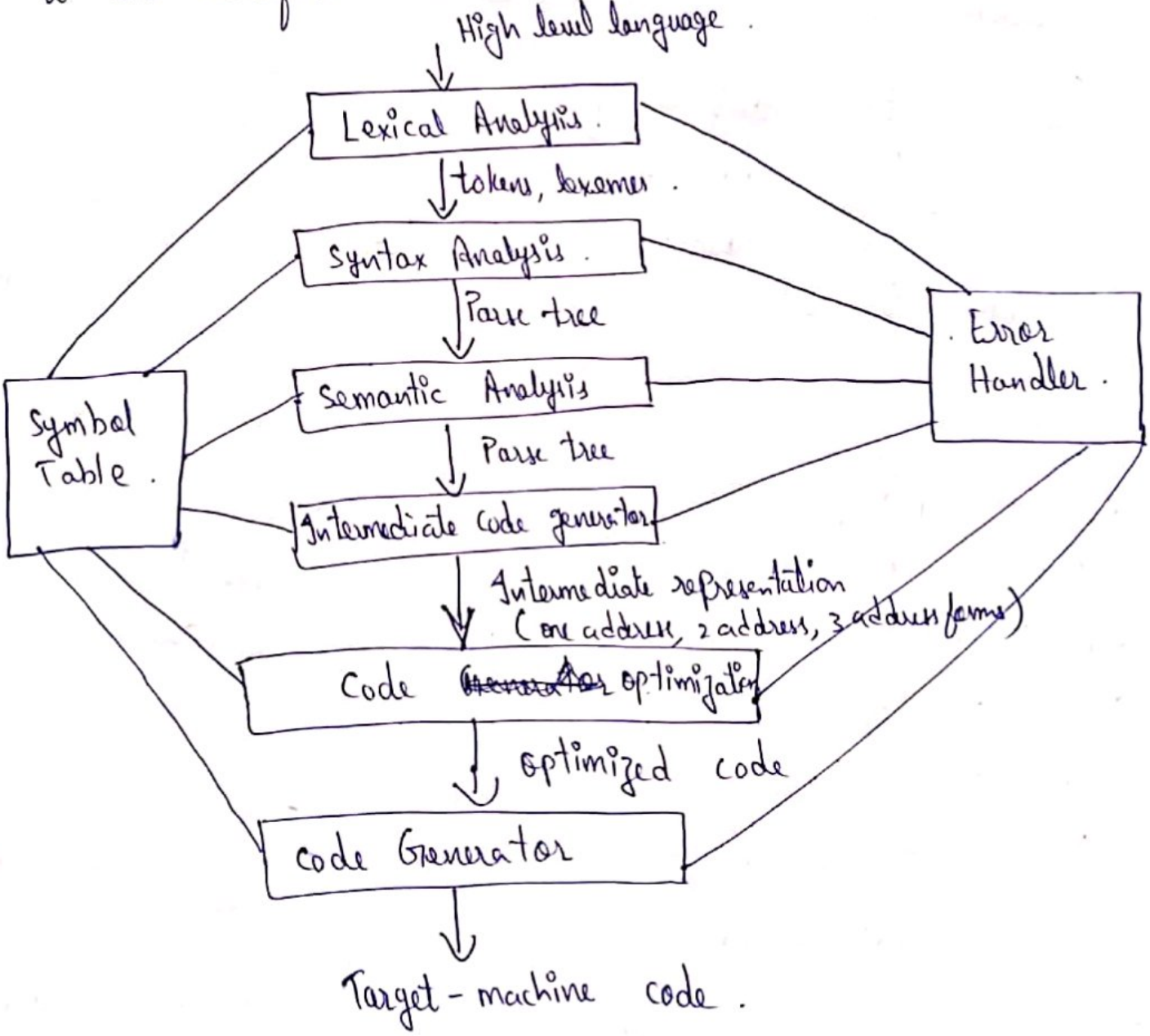
(c) Code Generation : In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions.

Scanned by CamScanner

Eg: Mov, id2, R1
ADDF R1, #10.0, R1

```
Mov   id2, R1
ADDF  #10.0, R1
Mov   R1, id3
```

we are moving value of id2 to 'R1' register. Then we add id2 & 10 & result is stored in 'R1'. Then value of R1 is assigned to id3 identifier.



Phases Of Compiler

Scanned by CamScanner

* Symbol Table

Symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. Symbol table also stores information about the subroutines used in the program.

→ It allows us to find the record for each identifier quickly & to store or retrieve data from that record efficiently.

* Error detection & Handling :

In compilation, each phase detects errors. These errors must be reported to error handler, whose task is to handle the errors so that the compilation can proceed. Errors are reported in the form of 'messages'.

→ Large no. of errors can be detected in syntax analysis phase. Such errors are called as "syntax errors".

* Pass : when several phases are grouped together, we call it as Pass.

* The Science of Building a Computer :

A compiler must accept all source programs that conform to the specification of the language. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the

Scanned by CamScanner

Compilers they create, but all the Programs that their compilers compile.

a) Modeling in Compiler Design & Implementation

The study of compilers is mainly a study of how we design the right mathematical models & choose the right algorithms, while balancing the need for generality & Power against simplicity & efficiency.

→ Some of most fundamental models are finite-state machines & regular expressions (for describing the lexical units of Programs), context-free grammars (to describe syntactic structure).

b) The Science of code optimization:

The term 'optimization' in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code.

→ Compiler optimizations must have the following design objectives:

- The optimization must be correct, i.e Preserve the meaning of the compiled Program.
- The optimization must improve the Performance of many Programs.
- The compilation time must be kept reasonable
- The Engineering effort required must be manageable.

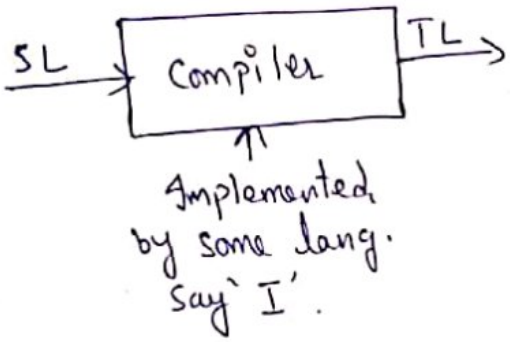
Scanned by CamScanner

* Bootstrapping :

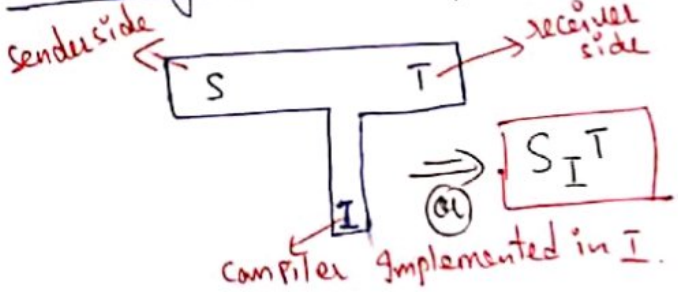
A Process by which simple language is used to translate more complicated Program, which in turn may handle an even more complicated Program.

→ There are three types of languages.

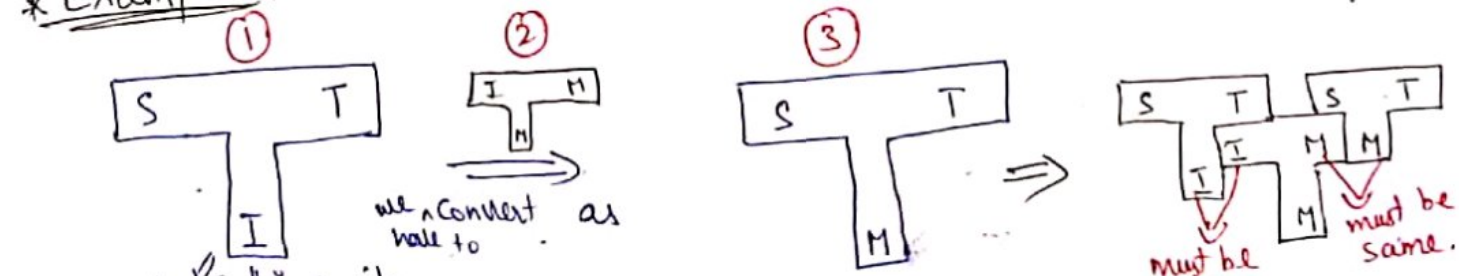
- Source language(SL) ; i.e the application program
- Target language(TL) in which machine code is written.
- Implementation language in which a compiler is written.



⇒ This can be represented by "T-diagram" as follows;



* Example :



Say we have this compiler

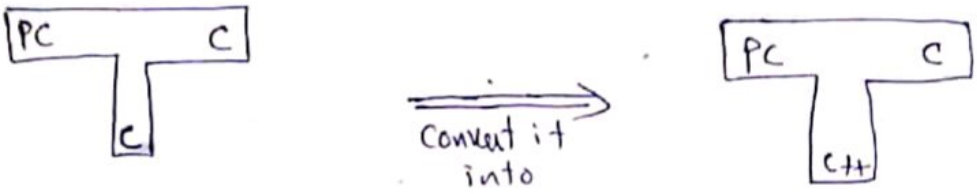
→ The Process of converting 'I' into 'M' lang. machine is Boot-strapping.

Scanned by CamScanner

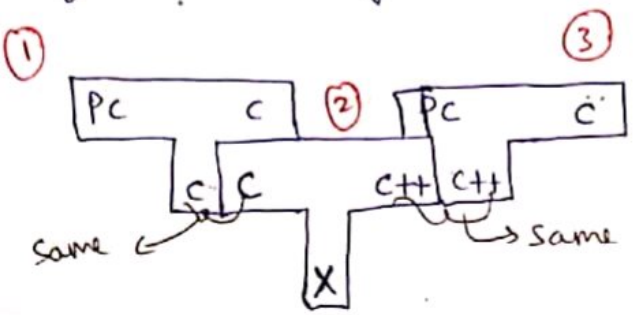
Example :

We have Pascal Translator written in → 'c' lang.
i/p is Pascal code(Pc), o/p – 'c'. create pascal translator in c++.

sol : Given :



with help of Bootstrapping :



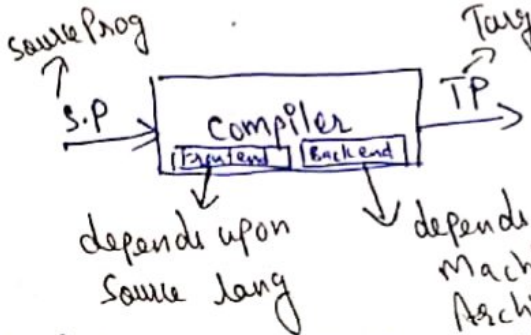
X ⇒ can be anything like java or any other lang

(or) $P_c c + C_x c++ \Rightarrow P_{c++} c$

* Cross Compiler :

Cross compiler is a compiler which is capable of creating executable code for a platform other than the one on which compiler is running.

eg : A compiler which runs on windows 7 but this generates a code which can run on Android smartphone. Then this compiler is known as Cross-compiler.



⇒ we need to change only the backend to implement Cross compiler.

→ using cross compilation, platform independency can be achieved.

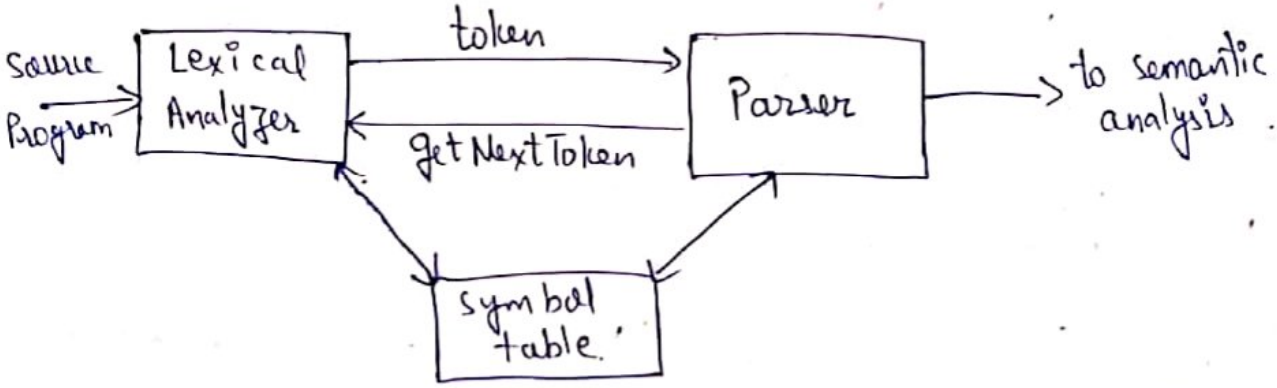
Scanned by CamScanner

LEXICAL ANALYSIS

* Role of Lexical Analyzer :-

Lexical Analyzer reads the input source program from left to right one character at a time & generates the sequence of tokens. It is the first phase of a compiler.

→ Each token is a single logical cohesive unit such as identifier, keywords, operators & punctuation marks. Then the Parser to determine the syntax of the source program can use these tokens.



Interactions between the lexical analyzer & the Parser

→ As the lexical analyzer scans the source program to recognize the tokens, it is also called as "Scanner".

* Functions of lexical Analyzer :-

- It Produces stream of tokens
- It eliminates whitespaces (blank, newline, tab) & comments.
- It generates symbol table which stores the information about identifiers, constants encountered in the i/p.
- It keeps track of line nos.
- It reports the error encountered while generating the tokens.

Scanned by CamScanner

* Tokens : A token is a pair consisting of a token name & an optional attribute value. It describes the class or category of input string. for example identifiers, keywords, constants are called tokens.

* Pattern : A pattern is a description of the form that the lexemes of a token may take. It is a set of rules that describe the token.

* Lexemes : Sequence of characters in the source program that are matched with the pattern of the token. For example int, i, num, choice;

Example : if (a < b)

if	(	a	<	b	)

⇒ tokens which are generated.

here "if", "(", "a", "<", "b", ")" ⇒ are all lexemes.

→ 'if' is a keyword
→ "(" is openening Parenthesis
→ "a" is identifier → collection of letter.
→ "<" is an operater
→ "b" is identifier
→ ")" is closing Parenthesis.

Scanned by CamScanner

* Attributes for tokens :

Eg :

Token		Lexemes
number	— 0, 1, 2, ...	0, 6.2, 1, ...

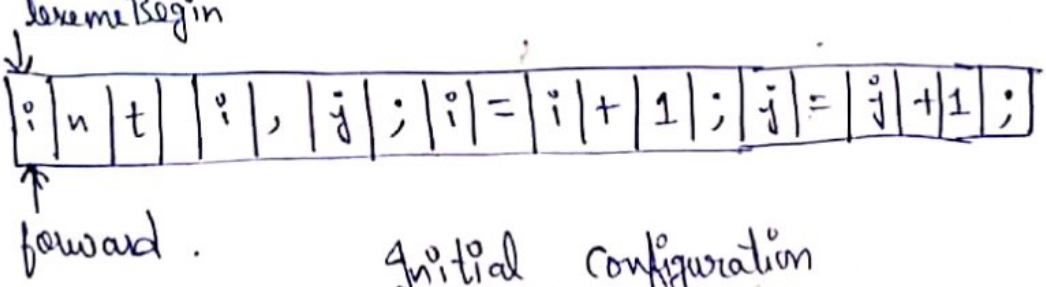
when more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.

Eg : Pattern for "token: Number" matches both '0' and '1'.

→ Lexical analyzer returns to the parser not only a token name but an attribute value that describes the lexeme represented by the token. Token name influences parsing decisions, while the attribute value influences translation of tokens after the ~~the~~ Parse.

* Input Buffering :

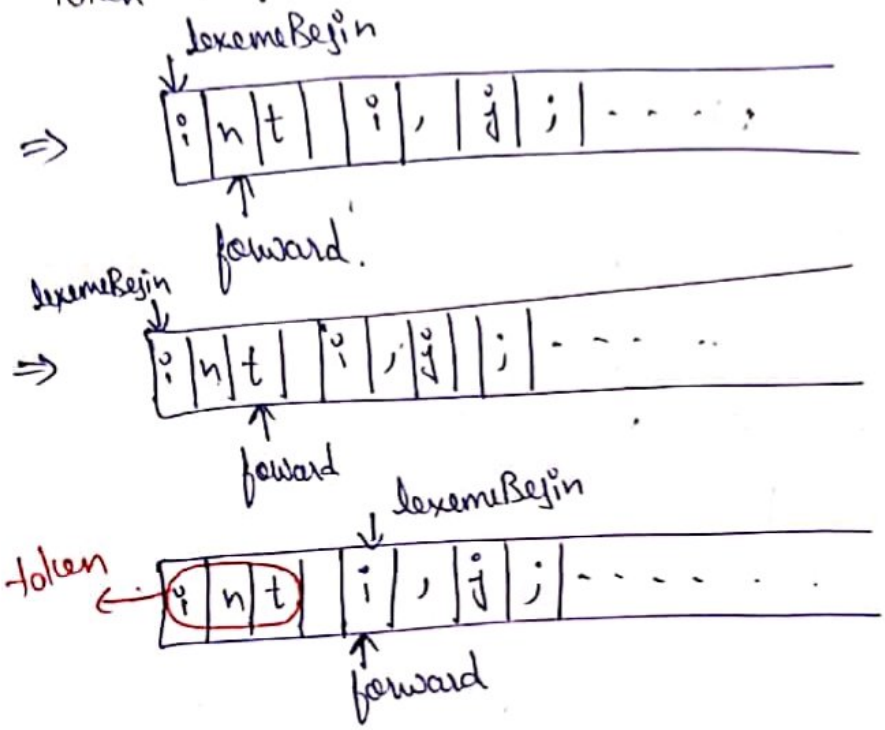
The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers: begin Pointer "lexemeBegin" and foward Pointer "foward" to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string as:



Initial Configuration

Scanned by CamScanner

The 'lexemeBegin' pointer remains at the beginning of the sting to be read & the 'forward' pointer moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. & then both 'lexemeBegin' & 'forward' is set at next token 'i'.



→ Reading i/p's from secondary storage is costly. so, a block of data is first read into a buffer & then scanned by lexical analyzer.

Input Buffering

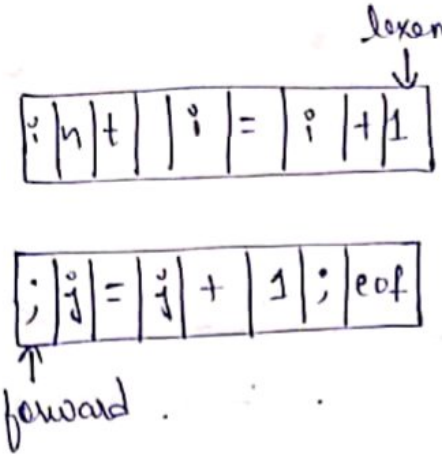
→ Before one buffer scheme was used but now we are using two buffer scheme.

* Two Buffer Scheme

→ to store i/p string.

Two buffers each of size 'N' (usually size of disk block) are used ~~A~~ Each buffer can read 'N' characters. In this method, first buffer & second buffer are scanned alternately. when the end of current buffer is reached the other buffer is filled.

Scanned by CamScanner



Two Buffer scheme

* Sentinel :
Both buffers have a special character "eof" at the end. when 'lexemeBegin' Pointer encounters 'eof' at Buffer 1, then it starts filling up second buffer. Similarly when 'eof' is encountered at Buffer2, it starts filling Buffer1 & so on.

→ This 'eof' character introduced at the end is called "Sentinel" which is used to identify the end of buffer.

* Specifications of tokens :

To specify tokens 'regular expressions' are used. when a pattern is matched by some regular expression then token can be recognized.

* Alphabet : Alphabet is a finite, non-empty set of symbols. It is represented by the symbol 'Σ'.

Eg : $\Sigma = \{0, 1\}$, $\Sigma = \{a, b\}$

$\Sigma = \{a, b, c, \cdots\}$, $\Sigma = \{e_0, e_1, \cdots e_9\}$

Scanned by CamScanner

* String : String is a finite sequence of symbols choosen from some alphabet.

Eg : $\Sigma = \{0, 1\}$.

0011 is a string of Σ , 102 is not a string of Σ.

1010 " " " " " , 234 " " " " " .

→ length of string is denoted by $|s|$..

→ Empty string can be denoted by 'ϵ'.

* Language : Set of strings which belong to some alphabet 'Σ' is called as a Language (L).

Eg : $\Sigma = \{0, 1\}$.

$L = \{00, 01, 10, 11\}$

$L = \{0^n 1^n \mid n \geq 1\}$.

* Operations on Language :

There are various operations which can be performed on a language as follows : below. Let 'L', M' be two languages.

Operation	definition & Notation.
• Union of L & M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
• Concatenation of L & M	$LM = \{ st \mid s \text{ is in } L \text{ \& } t \text{ is in } M \}$
• Kleen closure of 'L'	$L^* = \bigcup_{i=0}^{\infty} L^i$
• Positive closure of 'L'	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Definitions of operations on languages

Scanned by CamScanner

Eg :- $L = \{A, B, C, \ldots Z, a, b, c, \ldots\}$, $D = \{0, 1, 2, \ldots 9\}$ are two languages.

- LUD is a set of letters & digits
- LD is a set of strings consisting of letters followed by digits.
- L^5 is a set of strings having length of 5 each.
- L^* is " " " " all strings including 'ϵ'.
- L^+ " " " " except 'ϵ'.

* **Regular Expressions** :- Regular Expression is a notation to represent certain sets of strings in an algebraic fashion. This notation involves a combination of strings of symbols from some alphabet 'Σ', the symbol of null string 'ϵ', star operator (*) & plus operator (+).

* **Examples** :-

(1) write a Regular Expression (RE) for a language containing the strings of length two over $\Sigma = \{0, 1\}$.

Sol :- RE = (0+1)(0+1).

(2) write a RE for lang. containing strings which end with 'abb' over $\Sigma = \{a, b\}$

Sol :- $(a+b)^*abb$.

(3) write a RE for a recognizing identifier.

Sol :- For denoting identifier, we will consider a set of letters

Scanned by CamScanner

& digits because identifier is a combination of letters or letter & digits but having first character as letter always. Hence RE can be denoted as:

$$RE = letter\ (letter + digit)^*$$

where letter = (A, B, ... Z, a, b, ... z) & digit = (0, 1, 2, ... 9).

* Regular Definitions:

If 'Σ' is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_1 \longrightarrow r_1$$
$$d_2 \longrightarrow r_2$$
$$\vdots \qquad \vdots$$
$$d_n \longrightarrow r_n$$

where,

- Each 'di' is a new symbol, not in 'Σ' & not the same as any other of the d's &
- Each 'ri' is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

→ Some of the notations used for writing the regular expressions are as follows:

(1) One or more instance : To represent one or more instances "+" sign is used. Eg : r^+.

(2) Zero or more instances : To represent zero or more instances "*" sign is used. Eg : r^*.

(3) Character classes : A class of symbols can be denoted by [ ]. Eg : [012] means 0 or 1 or 2.

Scanned by CamScanner

* Examples :

(1) write Regular definition for 'C' language identifiers.

sol : 'C' identifiers are strings of letters, digits & underscores.

Regular definition is :

letter_ → [A-z a-z]
digit → [0-9]
id → letter_ (letter_ | digit)*

(2) write Regular definition for Unsigned numbers.

sol : unsigned nos (integer or floating point are strings like :
5080, 0.01234, 6.33458, 0.89E-4 etc.

Regular definition is :

digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?

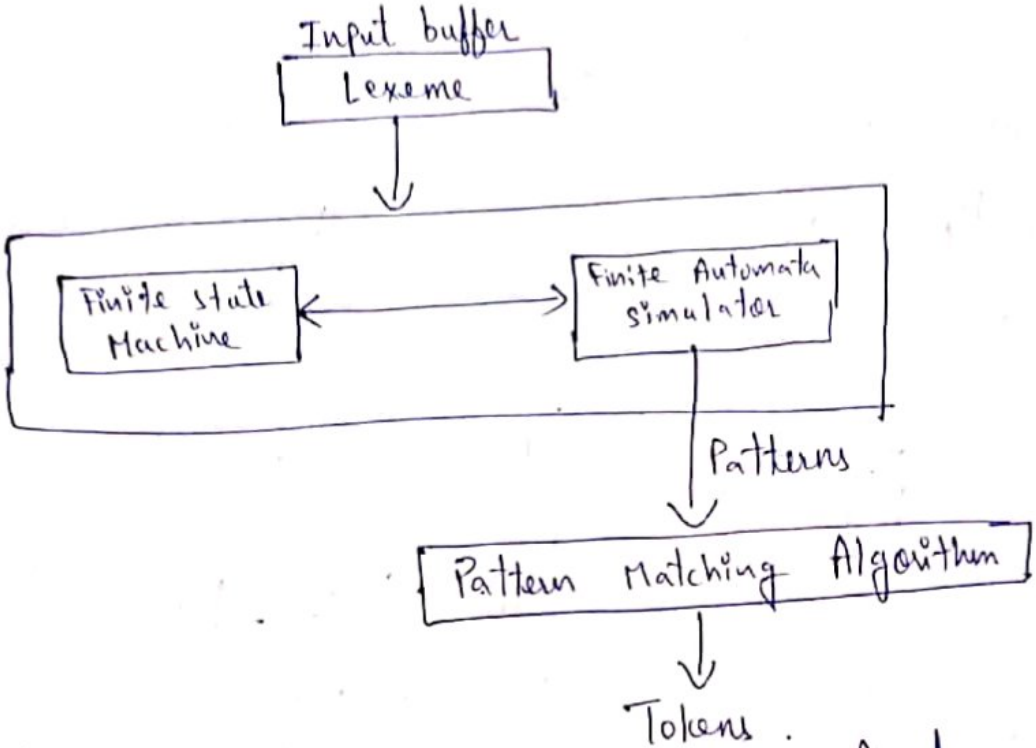
(? → zero or one instance)

* Recognition of Tokens :

The token is usually represented as: | Token type | Token value/attribute |

→ The token type tells us the category of token & token value gives us the information regarding token. During lexical analysis, Process, the symbol table is maintained. The token value can be a pointer to symbol table in case of identifiers & constants.

Scanned by CamScanner



Block diagram of Lexical Analyzer

* Example :

digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?
letter → [A-Z a-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relational operators ↷ relop → < | > | <= | >= | = | <>

Sample Regular definition

Scanned by CamScanner

Lexemes	Token Name	Attribute Value
Any ws	—	—
if	if	—
then	then	—
else	else	—
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry.
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Tokens, their patterns & attribute values

* Transition Diagrams:

In the middle of lexical analyzer Process we convert patterns into Transition diagrams. Transition diagrams have a collection of nodes or circles called states. Each state represents a condition that could occur during the Process of scanning the input looking for a lexeme that matches one of several patterns.

→ Edges are directed from one state to another. Each edge is labeled by a symbol or set of symbols.

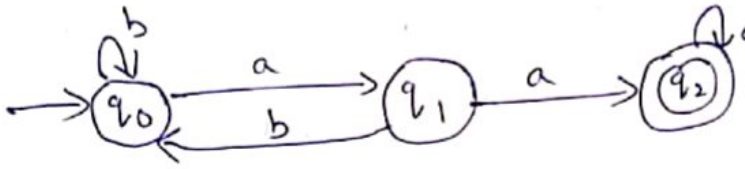
Scanned by CamScanner

- $\rightarrow (q_0)$ ⇒ initial state
- $\textcircled{\textcircled{q}}$ ⇒ final state

* Examples :-

(1) Design a transition diagram for the language consisting of all the strings containing at least one pair of consecutive a's over $\Sigma = \{a, b\}$.

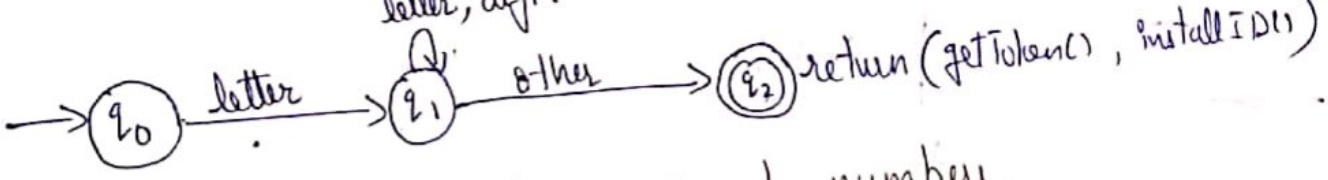
Sol :-



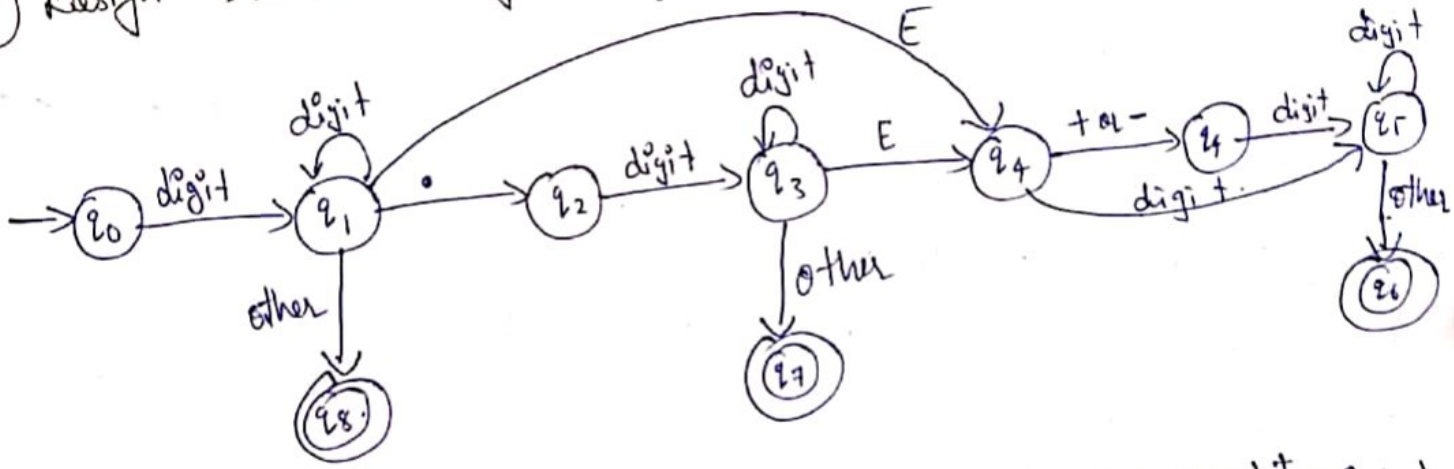
	a	b
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

(2) Design transition diagram for identifiers.

Sol :- RE for identifier :- r.e = letter (letter | digit)*



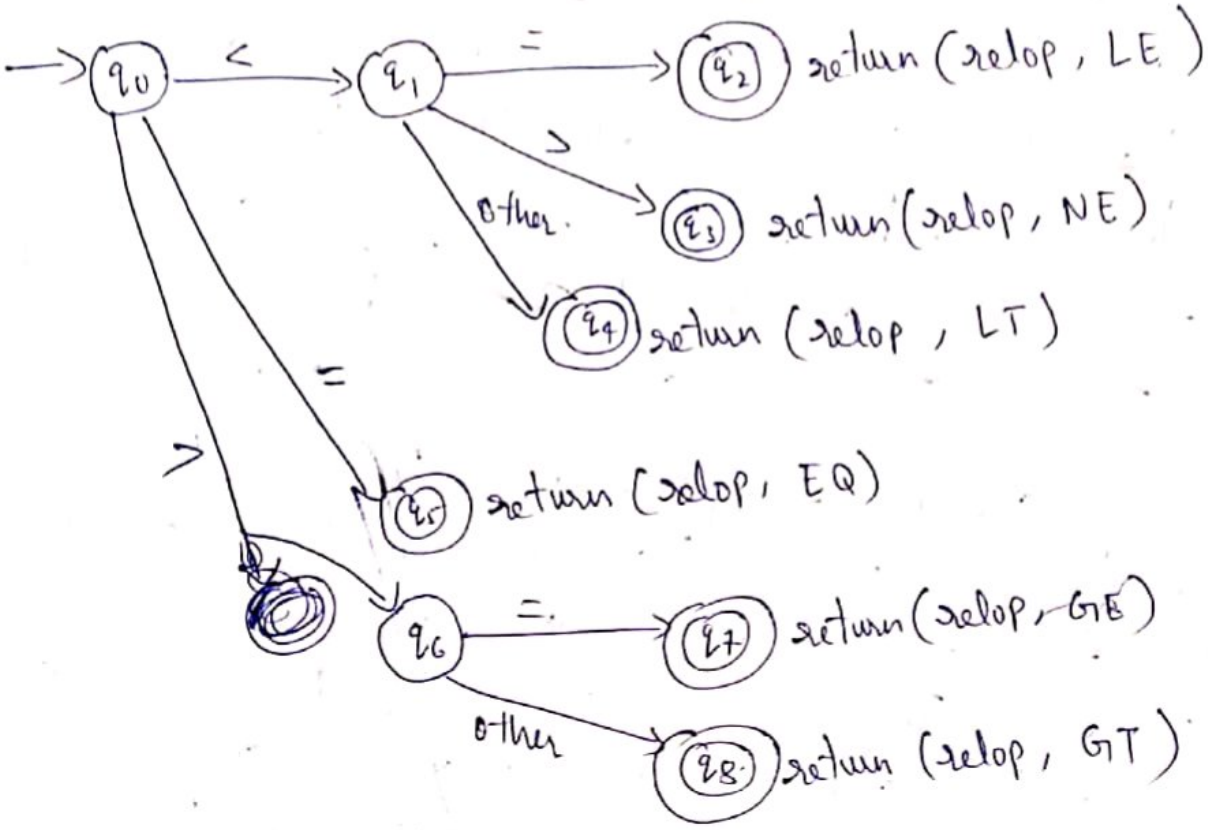
(3) Design transition diagram for unsigned numbers.



re = digit$^+$ | (digit)$^+$ (.) (digit)$^+$ | (digit)$^+$ (.) (digit)$^+$ E (+|-) (digit$^+$).

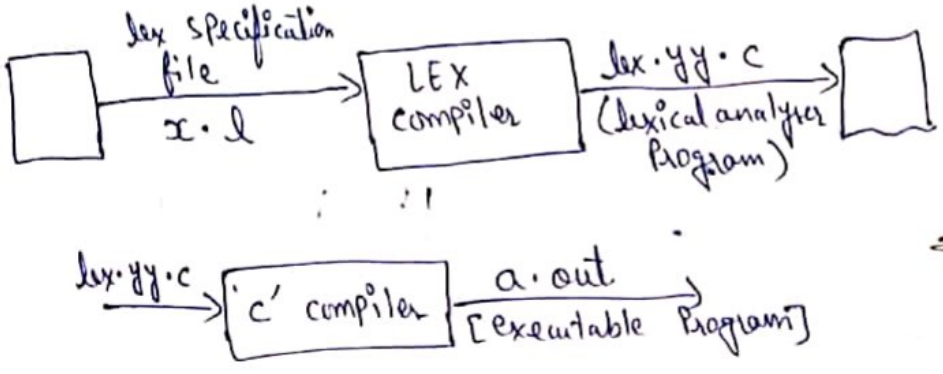
Scanned by CamScanner

④ Design transition diagram for relational operators (relop).



* LEX — Lexical Analyzer Generator: (or FLEX)

Regular expressions are used in recognizing the tokens. LEX is a tool called or FLEX (recent implementation) allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.



Generation of lexical analyser using LEX

Scanned by CamScanner

→ Specification file can be created using a file with extension ".l" (dot 'l'). Say for example "x.l". This x.l is then given to LEX compiler to Produce lex.yy.c. This lex.yy.c is a 'c' Program which is actually a lexical analyzer Program we know that the specification file stores the regular expressions for the tokens. The lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expressions of specification file say x.l.

→ The lexemes can be recognized with help of tabular transitions diagrams & some standard routines. In the specification file of LEX, actions are associated with each regular expression. These actions are simply pieces of 'c' code. These pieces of 'c' code are directly carried over to the lex.yy.c. Finally the 'c' compiler compiles this generated lex.yy.c & Produces an object Program a.out. When some input stream is given to a.out then sequence of tokens get generated.

→ LEX Program consists of three parts:
a) Declaration section, b) Rule section & c) Procedure section.

```
%{
   Declaration section
%}
%%
Rule section
%%
Auxiliary Procedure section
```

LEX Program format

Scanned by CamScanner

(a) Declaration section : In this section, declaration of variables, constants can be done. Some regular definitions can also be written in this section. The regular definitions are basically components of regular expressions appearing in the rule section.

(b) Rule section: It consists of regular expressions with associated actions. These translation rules can be given as :

R_1	{action 1}
R_2	{action 2}
⋮	
R_n	{action n}

where each R_i is a regular expression & each action; is a program fragment describing what action is to be taken for corresponding regular expression. These actions can be specified by piece of 'C' code.

(c) Auxiliary Procedure section: In this section, all the requi -red procedures are defined. Sometimes these procedures are required by the actions in the rule section.

→ The lexical analyzer or scanner works in co-ordination with Parser. when activated by the parser, the lexical analyzer begins reading its remaining input, character by character at a time. when the string is matched with one of the regular expressions R_i then corresponding action; will

Scanned by CamScanner

get executed & this action; returns the control to the Parser.

* Example LEX Programming :

```
// Program name : x.l
%{
%}  // Declaration section
%%  // Rule section
"Rama"  |
"seeta"  |
"Geeta"  |
"Neeta"  |  Printf ("In Noun");
"sings"  |
"dances"  |
"eats"      Printf ("In Verb");
%%
main()  // Auxiliary Procedure section.
{
   yylex();
}
int yywrap()
{
   return 1;
}
```

→ This is a simple program that recognizes noun & verb from the string.

Scanned by CamScanner

* Execution :

```
$ lex x.l ↵
$ cc lex.yy.c ↵
$ ./a.out ↵
```

o/p : ip1 : Rama eats ↵
o/p : Noun
Verb.

ip2 : seeta sings ↵
o/p : Noun
Verb

* LEX Actions :

(1) BEGIN : It indicates the start state. The lexical analyzer starts at state '0'.

(2) ECHO : It emits the input as it is.

(3) yylex() : As soon as call to yylex() is encountered, scanner starts scanning the source Program.

(4) yywrap() : The function yywrap() is called when scanner encounters end of file. If yywrap() returns '0'(zero) then scanner continues scanning. when " " '1'(one) that means end of file is encountered.

(5) yyin : It is the standard file that stores input source Program.

Scanned by CamScanner

*FLEX :

Flex is a tool for generating scanners. A scanner, sometimes called a 'tokenizer' is a program which recognizes lexical patterns in text. The flex program reads user-specified input files or its standard input if no file names are given for a description of a scanner to generate. The description is in the form of pairs of regular expressions & 'C' code called rules. Flex generates a 'C' source file named "lex.yy.c" which defines the function yylex(). The file lex.yy.c can be compiled & linked to produce an executable.

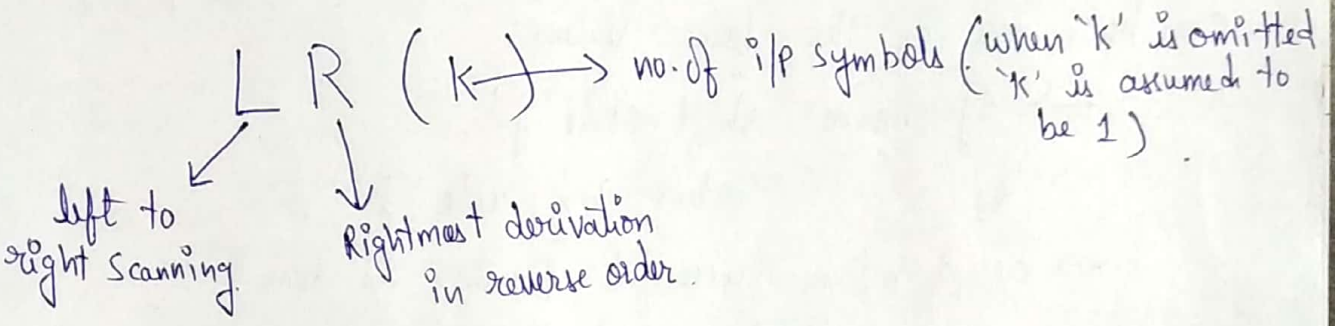
→ when the executable is run, it analyzes its i/p for occurances of text matching with the regular expressions for each rule. whenever it finds a match, it executes the corresponding 'C' code.

Scanned by CamScanner

UNIT-III : Syntax Analysis (Part-II)

* Introduction to LR Parsing :

LR Parsing is the most efficient method of bottom up Parsing which can be used to parse the large class of context-free grammars. This method is also called LR(k) Parsing.



→ LR Parsers are widely used for the following reasons :

(1) LR Parsers can be constructed to recognize most of the Programming languages for which context-free grammar can be written.

(2) The class of grammar that can be parsed by LR Parser is a superset of class of grammars that can be parsed using Predictive Parsers.

(3) LR Parser works using non-backtracking shift reduce technique yet it is efficient one.

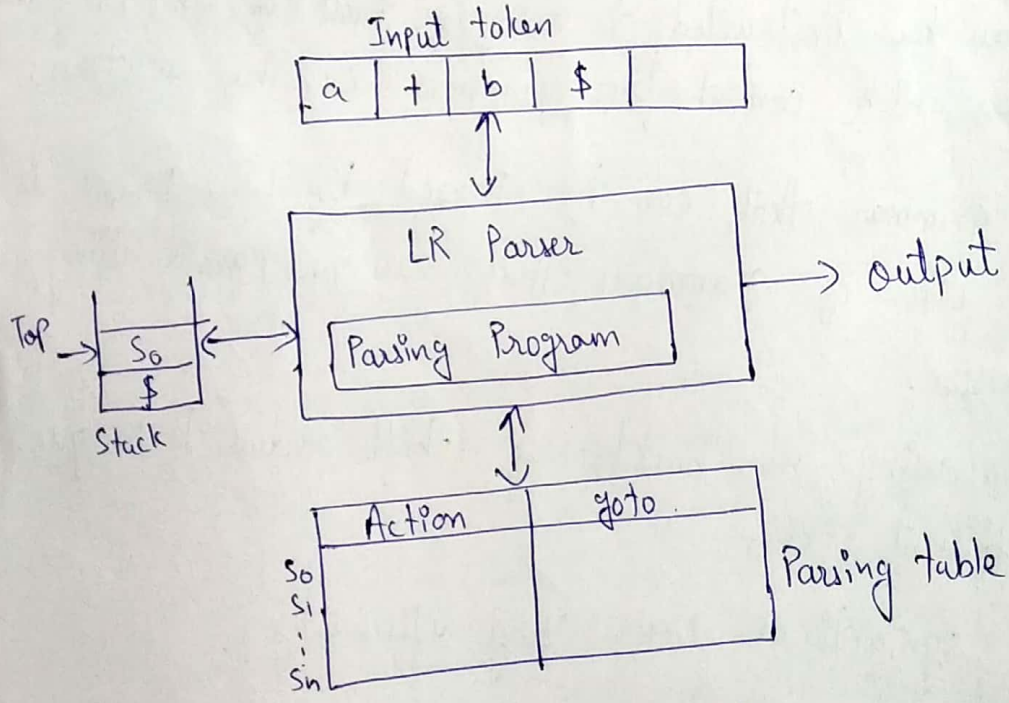
LR Parsers detect syntactical errors very efficiently.

* Structure of LR Parsers.

LR Parsers consists of input buffer for storing the input string, a stack for storing the grammar symbols, output & a Parsing table which has two parts : action & goto. There is one Parsing Program which reads the input symbol one at a time from

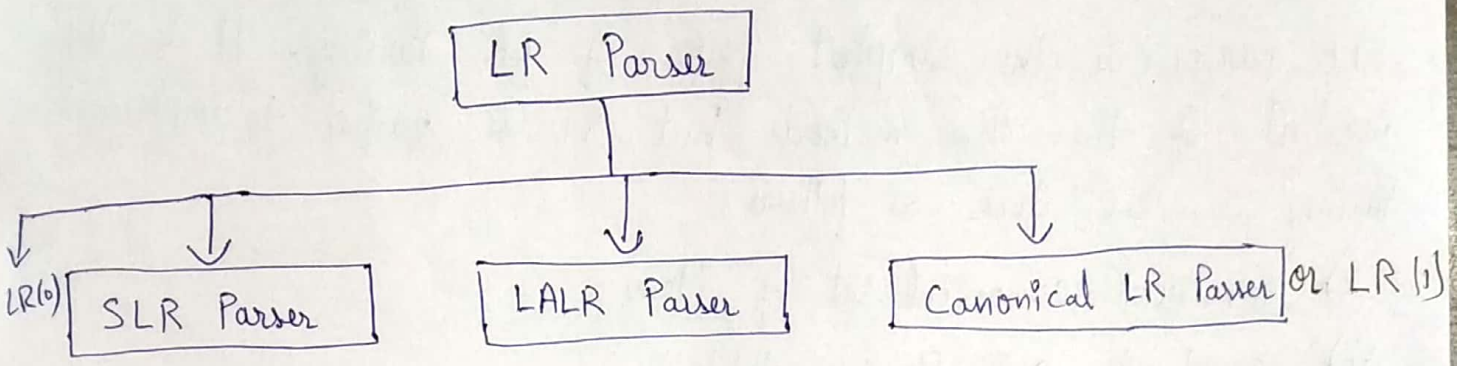
the input buffer. This Parsing Program works as follows:

- It initializes the stack with start symbol & invokes Scanner to get next token.
- It determines `s_j' the state currently on the top of the stack & a_i the current input symbol.
- It consults the Parsing table for the actions $[s_j, a_i]$ which can have one of the four values:
  - → s_j means shift state `j'.
  - → r_j " reduce by rule `j'.
  - → accept means successful Parsing is done
  - → error indicates syntactical error.



Structure Of LR Parser

*Types of LR Parsers :-



SLR Parser	LALR Parser	Canonical LR Parser (CLR)
(1) SLR Parser is the smallest in size.	(1) LALR & SLR have the same size.	(1) LR Parser (or) CLR Parser is largest in size.
(2) It is an easiest method based on FOLLOW function.	(2) This method is applicable to wider class than SLR.	(2) This method is most Powerful than SLR & LALR.
(3) Error detection is not immediate in SLR.	(3) Error detection is not immediate in LALR.	(3) Immediate error detection is done by LR Parser.
(4) It requires less time & space complexity.	(4) The time & space complexity is more in LALR but efficient methods exist for constructing LALR Parsers directly.	(5) The time & space complexity is more for canonical LR Parser.

Power :- SLR(1) ≤ LALR(1) ≤ CLR(1) (or) LR(1).

Size :- (SLR = LALR) < CLR .

* (a) SLR Parser (or) Simple LR.

SLR Parser is the simplest form of LR Parsing. It is the weakest of the three methods but it is easiest to implement. Parsing can be done as follows:

i) Construct canonical set of items
ii) Construct SLR Parsing table
iii) Parsing of input string.

→ A grammar for which SLR Parser can be constructed is called SLR grammar.

(Let $S \rightarrow \epsilon$ be a Production)

* Augmented grammar: If a gramar 'G' is having start symbol 'S' then augmented grammar is a new grammar G' in which S' is a new start symbol such that $S' \rightarrow S$. The Purpose of this grammar is to indicate the acceptance of input.

* Kernel items: It is collection of items $S' \rightarrow .S$ & all the items whose dots are not at the leftmost end of RHS of the rule.

* Non-Kernel items: The collection of all the items in which • are at the left end of RHS of the rule.

* Closure operation: For a CFG 'G', if 'I' is the set of items then the function closure(I) can be constructed using following rules:

- Consider 'I' is a set of canonical items & initially every item 'I' is added to closure(I).
- If rule $A \rightarrow \alpha . B\beta$ is a rule in closure(I) & there is another rule for 'B' such as $B \rightarrow \gamma$ then

closure (I) : $A \rightarrow \alpha \cdot B\beta$
$B \rightarrow \cdot\gamma$

This rule has to be applied until no more items can be added to closure (I).

* goto operation :-
If there is a production $A \rightarrow \alpha \cdot B\beta$ then goto $(A \rightarrow \alpha \cdot B\beta, B)$ = $A \rightarrow \alpha B \cdot \beta$. That means simply shifting of • one position ahead over the grammar symbol (may be terminal or non-terminal). The rule $A \rightarrow \alpha \cdot B\beta$ is in 'I' then the same goto function can be written as goto (I, B).

(i) Construction of Canonical set of items :-

- For the grammar 'G' initially add $S' \rightarrow \cdot S$ in the set of item 'C'.
- For each set of items I_i in 'C' & for each grammar symbol X (may be terminal or non-terminal) add closure (I_i, X)

This process should be repeated by applying goto (I_i, X) for each 'X' in I_i such that goto (I_i, X) is not empty & not in 'C'. The set of items has to constructed until no more set of items can be added to 'C'.

(ii) Construction of SLR Parsing table

By considering basic parsing actions such as shift, reduce, accept & error we will fill up the action table. The goto table can be filled up using goto function.

Input : An Augmented grammar G'.

output : SLR Parsing table.

Algorithm :

- Initially construct set of items $C = \{I_0, I_1, I_2, \ldots I_n\}$ where 'C' is a collection of set of LR(0) items for the input grammar G'.
- The parsing actions are based on each Item I_i. The actions are as given below :
  - → If $A \rightarrow \alpha . a\beta$ is in I_i & goto $(I_i, a) = I_j$ then set action $[i, a]$ as "shift j". Note that 'a' must be a terminal symbol.
  - → If there is a rule $A \rightarrow \alpha .$ in I_i then set action $[i, a]$ to "reduce $A \rightarrow \alpha$" for all symbols 'a', where $a \in$ FOLLOW (A). Note that 'A' must not be an augmented grammar s'.
  - → If $S' \rightarrow S$ is in I_i then the entry in the action table action $[i, \$]$ = "accept".
- The goto part of the SLR table can be filled as : The goto transitions for state 'i' is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto $[I_i, A] = j$.
- All the entries not defined by rule 2 & 3 are considered to be "error".

iii) Parsing the Input using Parsing Table:

Input: The input string 'w' that is to be parsed & Parsing table.

Output: Parse 'w' if $w \in L(G)$ using bottom up. If $w \notin L(G)$ then report syntactical error.

Algorithm:

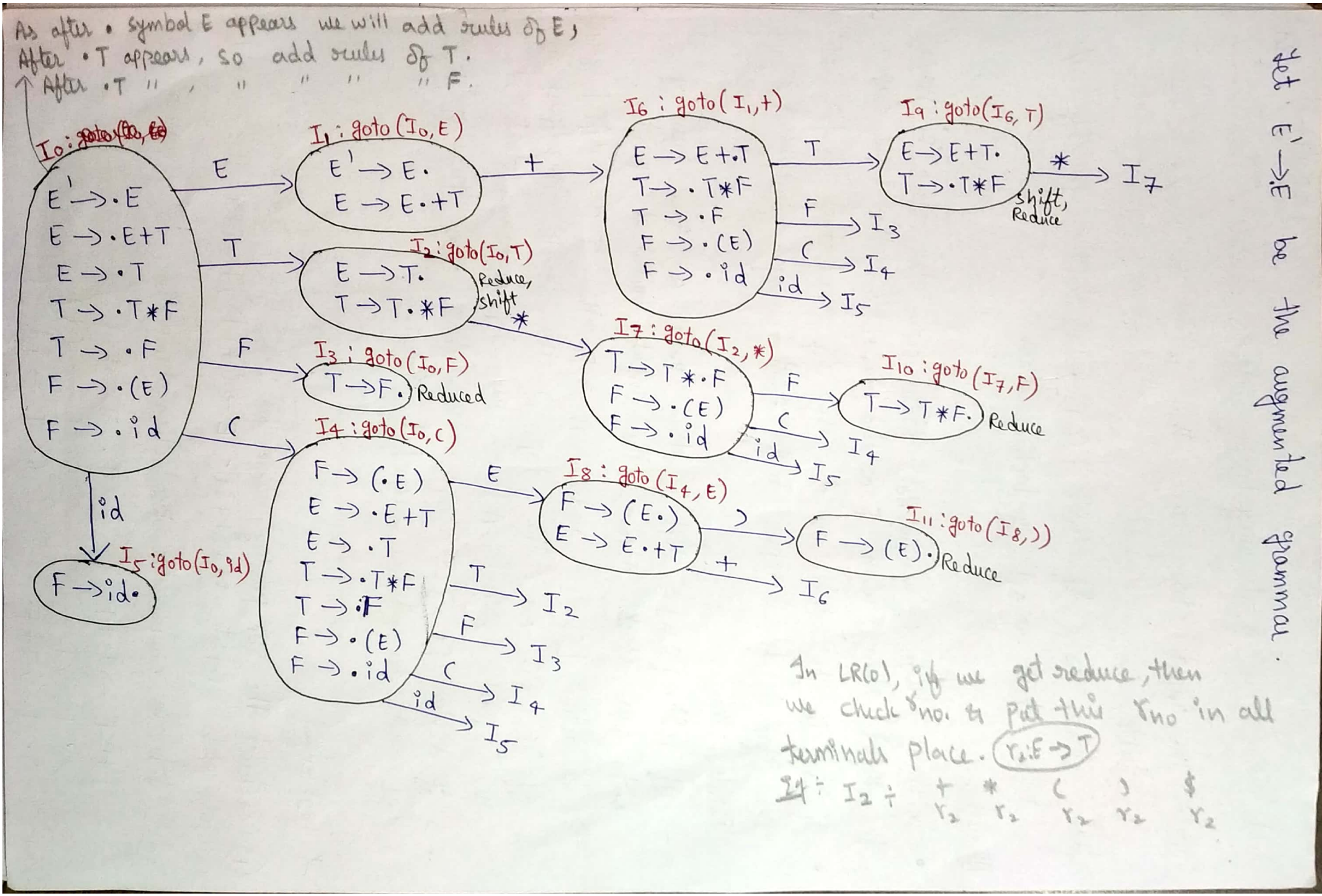
- Initially Push '0' as initial state onto the stack & place the input string with $ as end marker on the input tape.
- If 's' is the state on the top of the stack & 'a' is the symbol from input buffer pointed by a lookahead Pointer then
  - → If action [s,a] = shift 'j' then push 'a', then push 'j' onto the stack. Advance the input lookahead pointer.
  - → If action [s,a] = reduce $A \rightarrow \beta$ then pop $2*|\beta|$ symbols If 'i' is on the top of the stack then push 'A', then push goto [i,A] on the top of the stack.
  - → If action [s,a] = accept then halt the Parsing Process. If indicates the successful Parsing.

* Problems:

(1) Construct the SLR parsing table for the given grammar. Also Parse the input id*id+id. $E \rightarrow E+T \mid T$, $T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$.

Sol: Let us number the Production rules in the grammar.

1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$



- First(E) = First(T) = First(F) = {(, id} (here first & Follow do not do left recursion, left factoring)
- ~~First(E')~~ = ~~{+, ∈}~~
- ~~First(T')~~ = ~~{*, ∈}~~
- Follow(E) = {+, $,)}
- ~~Follow(E')~~ = ~~{$,)}~~
- Follow(T) = {+, *, $,)}
- ~~Follow(T')~~ = ~~{+, $,)}~~
- Follow(F) = {*, +, $,)}

* SLR Parsing Table: (Not LR(0) but it is SLR(1))

states	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2	LR(0) SLR(1) → r_2	r_2	S_7	r_2	r_2	r_2			
3	LR(0) r_4 SLR(1) → r_4	r_4	r_4	r_4	r_4	r_4			
4	S_5			S_4			8	2	3
5	LR(0) r_6 SLR(1) → r_6	r_6	r_6	r_6	r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9	LR(0): r_1 SLR(1) → r_1	r_1	S_7	r_1	r_1	r_1			
10	LR(0): r_3 SLR(1): → r_3	r_3	r_3	r_3	r_3	r_3			
11	LR(0): r_5 SLR(1) → r_5	r_5	r_5	r_5	r_5	r_5			

r_1 : E → E+T
r_2 : E → T
r_3 : T → T*F
r_4 : T → F
r_5 : F → (E)
r_6 : F → id

* → For Item 2, E → T. is reduced. check the list. it is numbered r_2. Now Follow(E) = ($,), +) in $,), + columns write r_2.

* → If it is shift operation just write S_{no}. If Item 0 on id it is going to I_5. So write "S_5".

* Parsing input "id * id + id".

Stack	Input	Action	output
$0	id * id + id $	shift s_5	whenever we do Reduce we need to pop out 2 symbols.
$0 id5	* id + id $	Reduce by F → id	
$0 F_3	* id + id $	Reduce by T → F	
$0 T_2	* id + id $	shift s_7	→ when you are shifting no need to pop.
$0 T_2 * 7	id + id $	shift s_5	
$0 T_2 * 7 id5	+ id $	Reduce by F → id	
$0 T_2 * 7 F10	+ id $	Reduce by T → T * F	∴ since there are no conflicts in SLR table. The given grammar is SLR(1).
$0 T_2	+ id $	Reduce by E → T	
$0 E1	+ id $	shift s_6	
$0 E1 + 6	id $	shift s_5	
$0 E1 + 6 id5	$	Reduce by F → id	
$0 E1 + 6 F3	$	Reduce by T → F	
$0 E1 + 6 T9	$	Reduce by E → E + T	
$0 E1	$	accept	

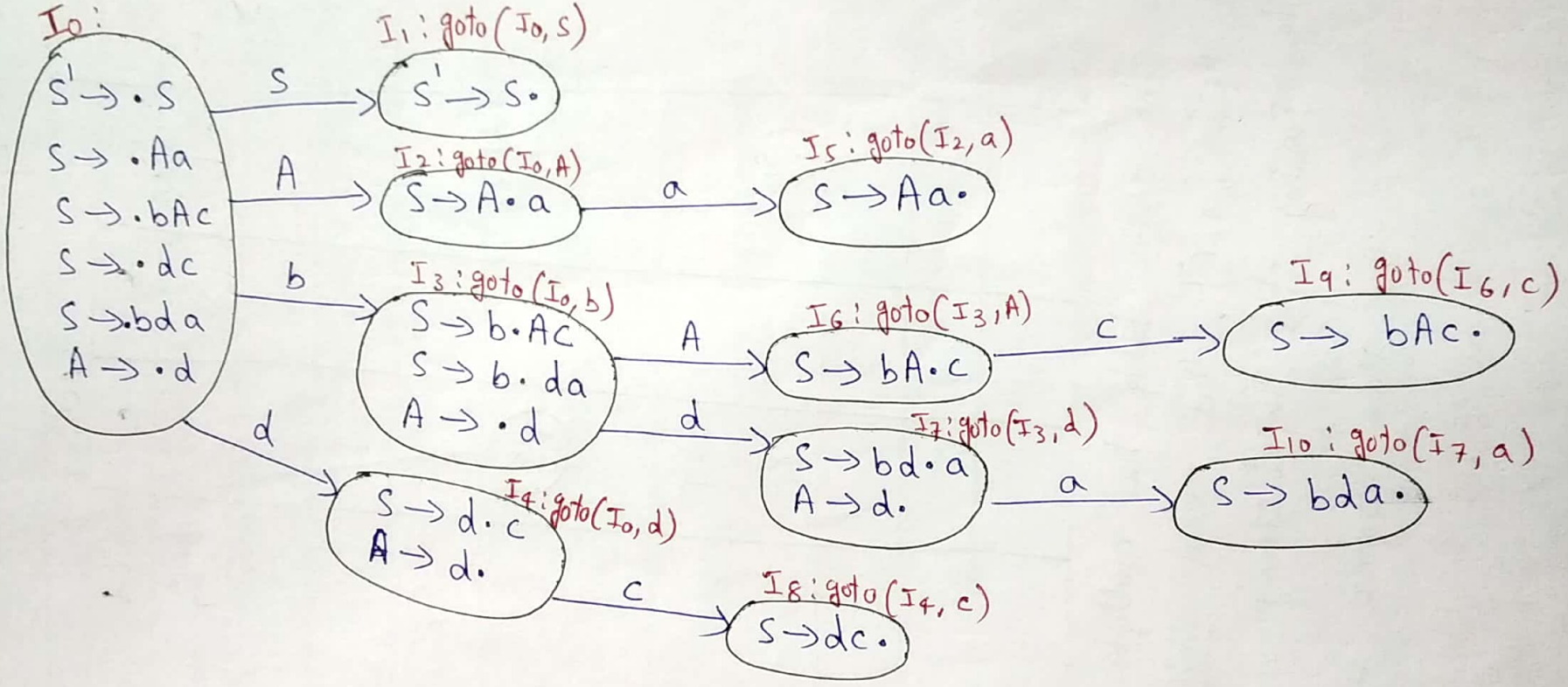
(2) Show that the following grammar is not SLR(1).

S → Aa | bAc | dc | bda

A → d

Sol: Let us number the Production rules in the grammar.

1. S → Aa
2. S → bAc
3. S → dc
4. S → bda
5. A → d



- Follow (S) = {$}
- Follow (A) = {a, c}
- First (S) = {b, d}
- First (A) = {d}

* SLR(1) Parsing table :

Not LR(0)

	Action					goto	
State	a	b	c	d	$	S	A
0		S_3		S_4		1	2
1					Accept		
2	S_5						
3				S_7			6
4	LR(0) r_5 SLR(1) r_5	r_5	S_8 r_5	r_5			
5					r_1		
6			S_9				
7	S_{10} r_5		r_5				
8					r_3		
9					r_2		
10					r_4		

Since the above table has multiple entries in Action [7,a] and Action [4,c], this means shift/reduce conflict will occur while Parsing the input string.

∴ The given grammar is not SLR(1).

LR(0) : in reduce write r_{no} in all terminals of that Item.
SLR(1) : ↓ we write r_{no} in follow " .

↓ only difference.

* More Powerful Parsers :

(a) Canonical LR : (or) LR(K) Parsers

The canonical set of items is the parsing technique in which a lookahead symbol is generated while constructing set of items. Hence the collection of set of items is referred as LR(1). The value 1 in the bracket indicates that there is one lookahead symbol in the set of items.

→ Steps to construct canonical LR :

- Construction of canonical set of items along with the lookahead.
- Building canonical LR Parsing table.
- Parsing the input string using canonical LR Parsing table.

* Construction of canonical set of items along with lookahead :

(1) For the grammar 'G' initially add $S' \rightarrow \cdot S$ in the set of item 'C'.

(2) For each set of items I_i in 'C' & for each grammar Symbol X (may be terminal or non-terminal) add closure (I_i, X). This process should be repeated by applying goto $((I_i, X))$ for each X in I_i such that goto (I_i, X) is not empty & not in 'C'. The set of items has to be constructed until no more set of items can be added to 'C'.

(3) The closure function can be computed as follows :
For each item $A \rightarrow \alpha \cdot X\beta a$ and rule $X \rightarrow \gamma$ & $b \in$ FIRST (βa) such that $X \rightarrow \cdot\gamma$ & 'b' is not in I

then add $X \to .\gamma, b$ to I.

④ Similarly the goto function can be computed as: For each item $[A \to \alpha . X\beta, a]$ is in I & rule $[A \to \alpha X . \beta, a]$ is not in I ~~then add~~ $X \to .\gamma, b$ goto items then add $[A \to \alpha X . \beta, a]$ to goto items.

This process is repeated until no more set of items can be added to the collection 'C'.

* Construction of canonical LR Parsing table:

① Initially construct set of items $C = \{I_0, I_1, I_2, \ldots I_n\}$ where 'C' is a collection of set of LR(1) items for the input grammar G'.

② The parsing actions are based on each item I_i. The actions are as given below:

- If $[A \to \alpha . a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then Create an entry in the action table action $[I_i, a]$ = shift j.
- If there is a production $[A \to \alpha ., a]$ in I_i then in the action table action $[I_i, a]$ = reduce by $A \to \alpha$. Here 'A' should not be S'.
- If there is a production $S' \to S., \$$ in I_i then action $[i, \$]$ = accept.

③ The goto Part of the LR table can be filled as: The goto transitions for state 'i' is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto $[I_i, A] = j$.

④ ~~All the entries not defined by rule 2 and 3 are considered to be "error"~~ All entries not defined by rule 2 & 3 are considered to be "error"

* Problems :

① Construct LR(1) or CLR set of items for the below grammar

$S \rightarrow CC$

$C \rightarrow aC \mid d$

Sol : Add $S' \rightarrow .S, \$$ as the first rule in I_0. The grammar is :

$S' \rightarrow .S, \$$

$S \rightarrow CC$

$C \rightarrow aC \mid d$

Add dot to every Production : $(A \rightarrow \alpha . X\beta, a)$; $A \in S'$, $X \in S$, $a \in \$$

I_0

$S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .aC, a/d$

$C \rightarrow .d, a/d$

$X \rightarrow \gamma, b$ then add $X \rightarrow .\gamma, b$

$b \in First(\beta a)$

$b \in First(\epsilon \$)$

$b \in First(\$)$

$b = \{\$\}$

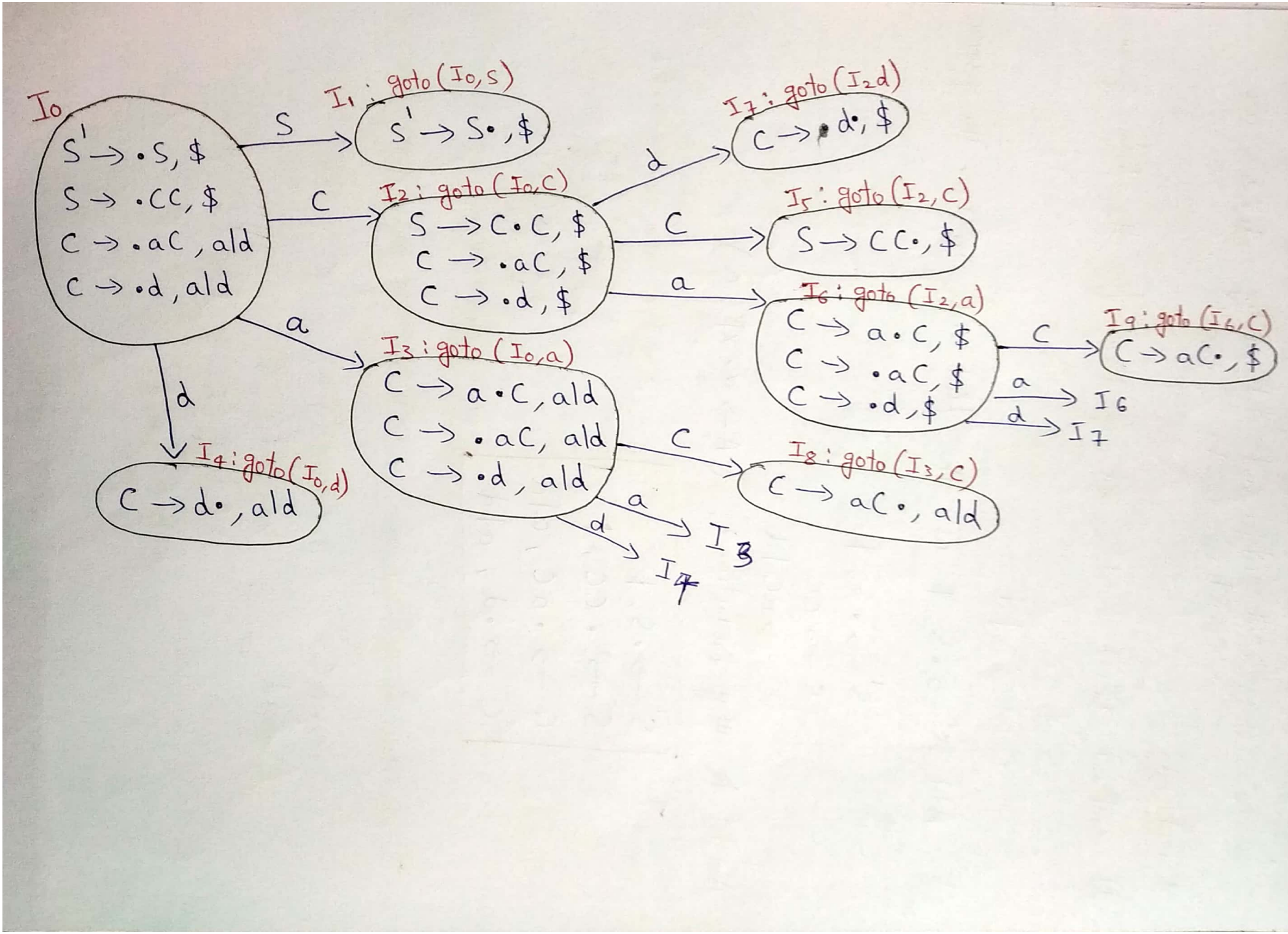
→ If there is a Production $X \rightarrow \gamma, b$ then add $X \rightarrow .\gamma, b$

$C \rightarrow .aC$ $\quad b \in First(\beta a)$

$C \rightarrow .d$ $\quad b \in First(C\$)$

$b \in First(C)$ as $First(C) = \{a, d\}$

$b = \{a, d\}$



* LR(1) Parsing table :-

	action			goto	
	a	d	$	S	C
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			
7			r_3		
8	r_2	r_2			
9			r_2		

r_1 : $S \rightarrow CC$

r_2 : $C \rightarrow aC$

r_3 : $C \rightarrow d$

I_0 :- on S it is shifted to I_1
C " " " " I_2
a " " " " I_3
d " " " " I_4

} write Sno in appropriate rows.

I_4 :- reduced check numbering : " r_3 ".

$C \rightarrow d\cdot$, (a/d) => write in a/d " r_3 ".

* Parsing the input using LR(1) Parsing table

Let us Parse for the i/p string "aadd" by using above Parsing table.

stack	Input buffer	action table	goto table	Parsing action
$0	aadd$	action [0,a] = S_3		
$0a3	add $	action [3,a] = S_3		shift
$0a3a3	dd$	action [3,d] = S_4		shift
$0a3a3d4	d$	action [4,d] = r_3	[3,C] = 8	Reduce by C→d
$0a3a3C8	d$	action [8,d] = r_2	[3,C] = 8	Reduce by C→aC
$0a3C8	d$	action [8,d] = r_2	[0,C] = 2	Reduce by C→aC
$0C2	d$	action [2,d] = S7		shift
$0C2d7	$	action [7,$] = r_3	[2,C] = 5	Reduce by C→d
$0C2C5	$	action [5,$] = r_1	[0,S] = 1	Reduce by S→CC
$0S1	$	accept		

Thus the given input string is succesfully Parsed using LR Parser or canonical LR Parser.

(2) show that the following grammar is LR(1).

S → Aa | bAc | Bc | bBa

A → d

B → d.

Sol: We will number out the Production rules in given grammar.

1. S → Aa
2. S → bAc
3. S → Bc
4. S → bBa
5. A → d
6. B → d

Let $S' \rightarrow .S, \$$.

* Constructing Canonical set of LR(1) items :-

* Parsing table :

State	Action					goto		
	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					Accept			
2	S_6						7	8
3				S_9				
4			S_{10}					
5	r_5		r_6					
6					r_1			
7			S_{11}					
8	S_{12}							
9	r_6		r_5					
10					r_3			
11					r_2			
12					r_4			

* Parsing the string "bda" using above Parsing table of LR(1) :

stack	Input Buffer	Action
$0	bda$	shift 3
$0b3	da$	shift 9
$0b3d9	a$	Reduce by B → d
$0b3B8	a$	shift 12

Stack	Input Buffer	Action
$ 0b3B8a12	$	Reduce by S → bBa
$ 0S1	$	Accept

The above string 'bda' is accepted. Since in the Parsing table we do not have multiple entries in the same cell, the above grammar is LR(1) or CLR(1).

* LALR :

The algorithm for construction of LALR Parsing table is as follows:

step 1 : Construct the LR(1) set of items.

step 2 : Merge the two states I_i and I_j if the first component (i.e the production rules with dots) are matching & create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$.

step 3 : The parsing actions are based on each item I_i. The actions are as given below:

a) If $[A \to \alpha . a\beta, b]$ is in I_i & goto $(I_i, a) = I_j$ then create an entry in the action table action $[I, a]$ = shift 'j'.

b) If there is a production $[A \to \alpha \cdot, a]$ in I_i then in the action table action $[I_i, a]$ = reduce by $A \to \alpha$ Here 'A' should not be S'.

c) If there is a production $S' \to S \cdot, \$$ in I_i then action$[i, \$]$ = accept.

step4 : The goto part of the LR table can be filled as: The goto transitions for state 'i' is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto $[I_i, A] = j$.

step5 : If the parsing action conflict then the algorithm fails to produce LALR Parser & grammar is not LALR(1). All the entries not defined by rule 3 & 4 are considered to be "error".

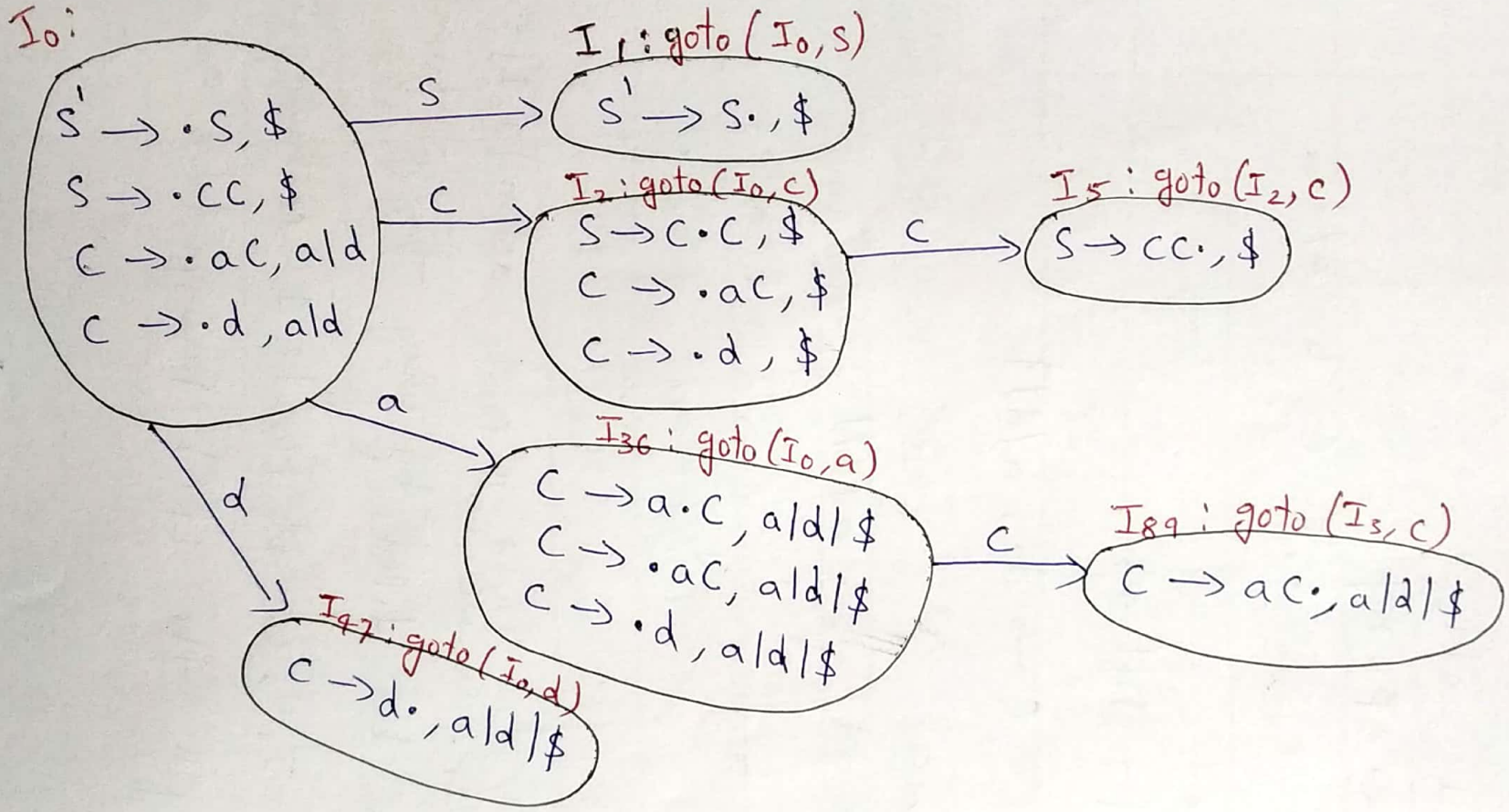
* Problems :

(1) Construct Parsing table for LALR(1) Parser for the below given grammar.

$S \to CC$
$C \to aC$
$C \to d$.

sol : First the set LR(1) items can be constructed as follows with merged states.

$r_1 : S \to CC$
$r_2 : C \to aC$
$r_3 : C \to d$.



Now consider state I_0 there is a match with the rule $[A \to \alpha . a\beta, b]$ and goto $(I_i, a) = I_j$.

$C \to .aC, a/d/\$$ and if the goto is applied on 'a' then we get the state I_{36}. Hence,

in I_0:

$$C \to .d \;, a/d$$

$$\boxed{A \to \alpha . a\beta, b}$$

where $A = C$, $\alpha = \epsilon$, $a = d$, $\beta = \epsilon$, $b = a/d$

goto $(I_0, d) = I_{47}$.

hence action $[0, d]$ = shift 47.

→ For state I_{47} : $C \to d.$, $a/d/\$$

$$A \to \alpha ., a$$

where $A = C$, $\alpha = d$, $a = a/d/\$$.

action $[47, a]$ = reduce by $C \to d$ i.e rule 3

action $[47, d]$ = " " " " "

action $[47, \$]$ = " " " " "

→ $S' \to S.$, $\$$ in I_1. so we will create action $[1, \$]$ = accept.

The goto table can be filled by using the goto functions. In this instance goto $(I_0, S) = I_1$, so goto $[0, S] = 1$. continuing in this manner we can fill the LR(1) Parsing table as follows:

States	Action			goto	
	a	d	$	S	C
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		

* Parsing string "adad" using LALR Parser :

Stack	Input buffer	Action table	goto table	Parsing action
$0	aadd $	action [0,a] = S36		
$0a36	add $	action [36,a] = S36		shift
$0a36a36	dd $	action [36,d] = S47		shift
$0a36a36d47	d$	action [47,d] = r_{36}	[36,C] = 89	Reduce by C→d
$0a36a36C89	d $	action [89,d] = r_2	[36,C] = 89	Reduce by C→aC
$0a36C89	d$	action [89,d] = r_2	[0,C] = 2	Reduce by C→aC
$0C2	d $	action [2,d] = S47		shift
$0C2d47	$	action [47,$] = r_{36}	[2,C] = 5	Reduce by C→d
$0C2C5	$	action [5,$] = r_1	[0,S] = 1	Reduce by S→CC
$0S1	$	accept		

(2) Show that the following grammar is not LALR(1).

Sol:-

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

Sol:- Let us number the Production rules of the grammar:

$r_1 : S \to Aa$

$r_2 : S \to bAc$

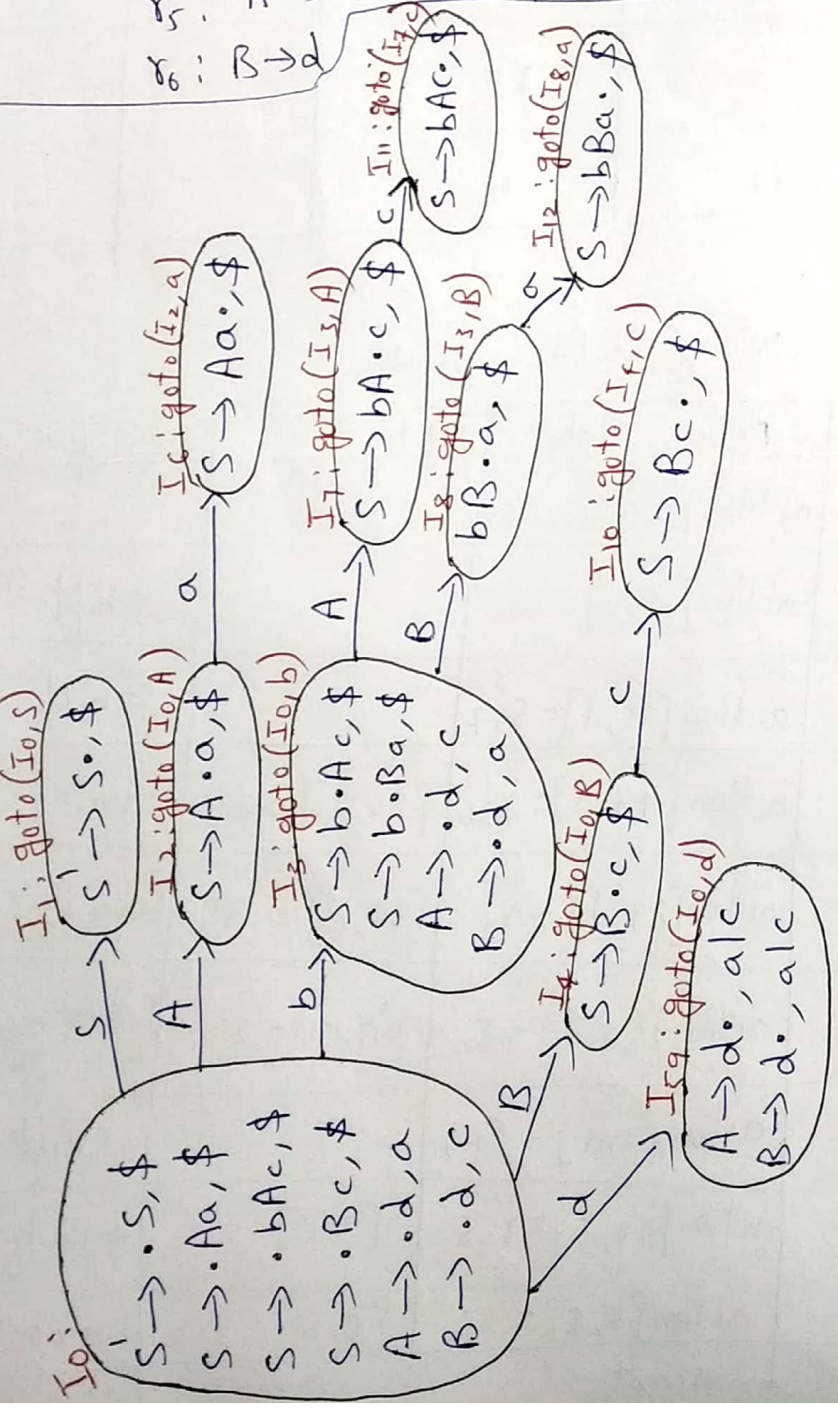
$r_3 : S \to Bc$

$r_4 : S \to bBa$

$r_5 : A \to d$

$r_6 : B \to d$

LALR(1) Set of items are as follows:-



states	Action					goto		
	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					Accept			
2	S_6							
3				S_9			7	8
4			S_{10}					
59	γ_5 γ_6		γ_5 γ_6					
6					γ_1			
7			S_{11}					
8	S_{12}							
10					γ_3			
11					γ_2			
12					γ_4			

The Parsing table shows multiple entries in Action [59, a] & Action [59, c]. This is called reduce/reduce conflict. Because of this conflict we cannot parse input.

∴ The given grammar is not LALR(1).

* Handling Ambiguous Grammar:

If all the grammar is ambiguous then it creates the conflicts & we cannot parse the input string with such ambiguous grammar. But for some languages in which arithmetic

expressions are given ambiguous grammar are most compact & Provide more natural specification as compared to equivalent unambiguous grammar.

→ while using ambiguous grammar for Parsing the input string we will use all the disambiguating rules so that each time only one Parse tree will be generated for that specific input. Thus ambiguous grammar can be used in controlled manner for Parsing the input.

* Using Precedence & Associativity to resolve Parsing Action Conflicts

Consider an ambiguous grammar:

$E \rightarrow E+E$

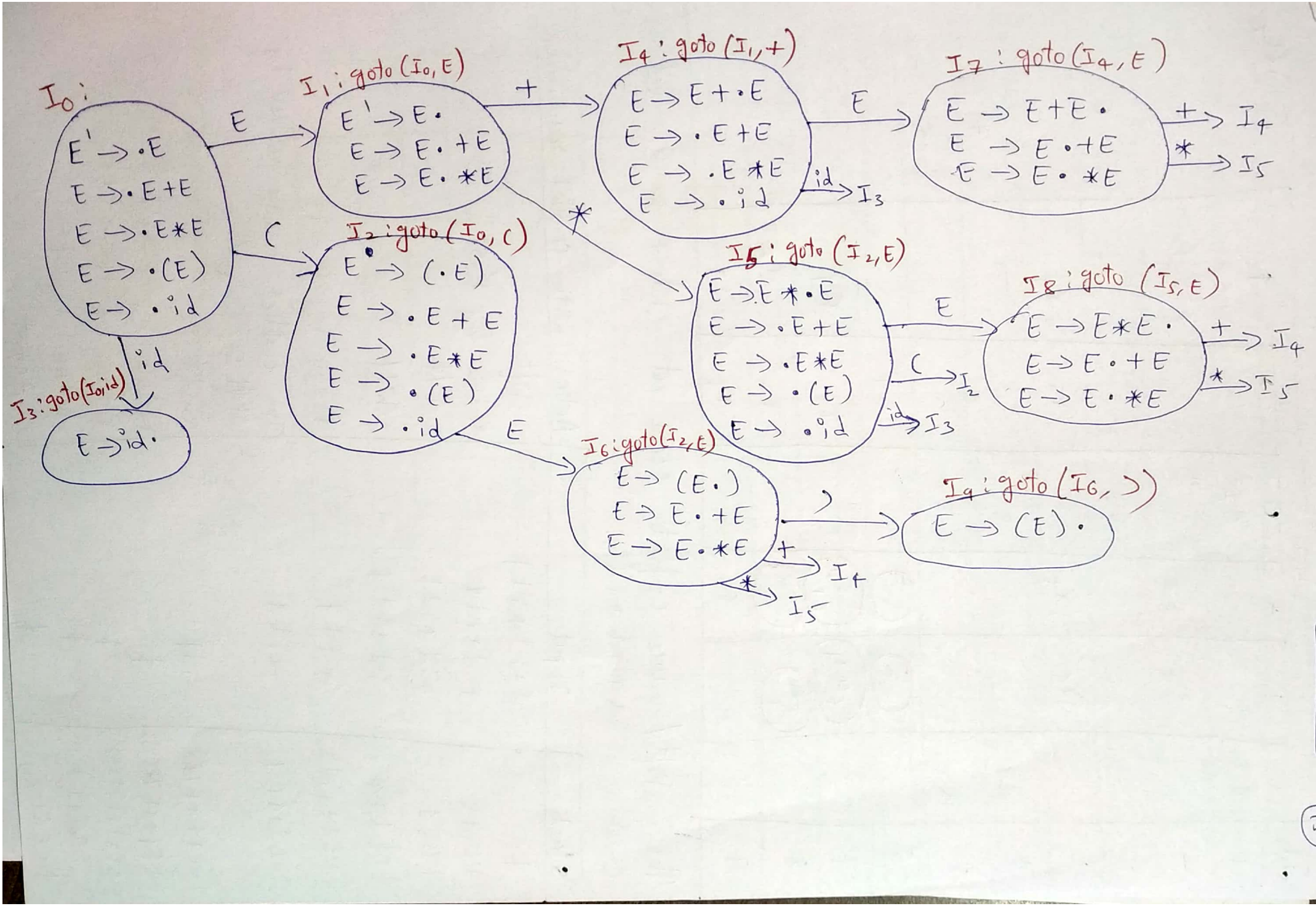
$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Now we will build the set of LR(0) items for this grammar.

- Follow(E) = $\{+, *,), \$\}$

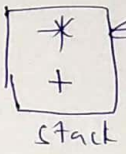


state	action						(goto) E
	id	+	*	(	)	$	
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		S_4 or r_1	S_5 or r_1		r_1	r_1	
8		S_4 or r_2	S_5 or r_2		r_2	r_2	
9			r_3		r_3	r_3	

The shift/reduce conflicts occur at state 7 & 8. Let us consider id+id*id.

Stack	Input	Action with conflict resolution
$0	id+id*id$	shift
$0id3	+id*id$	Reduce by E→id
$0E1	+id*id$	shift
$0E1+4	id*id$	shift
$0E1+4id3	*id$	Reduce by E→id
$0E1+4E7	*id$	The conflict can be resolved by shift 5
$0E1+4E7*5	id$	shift
$0E1+4E7*5id3	$	Reduce by E→id
$0E1+4E7*5E8	$	Reduce by E→E*E
$0E1+4E7	$	Reduce by E→E+E
$0E1	$	Accept

As * has precedence over '+' we have to perform multiplication operation first. And for that it is necessary to push * on the top of the stack. The stack position will be:



By this we can perform E*E first & then E+E. The Parsing conflict can be resolved by assigning shift operation. Hence action[7,*] = S_5.

state	Action						goto
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			
5	S_3			S_2			
6		S_4	S_5		S_9		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9			r_3		r_3	r_3	

* Using Dangling Else Ambiguity :

Consider the grammar :

$S \rightarrow iSeS \mid iS \mid a$

where ;seS = if expression then statement else statement

;S = if " " "

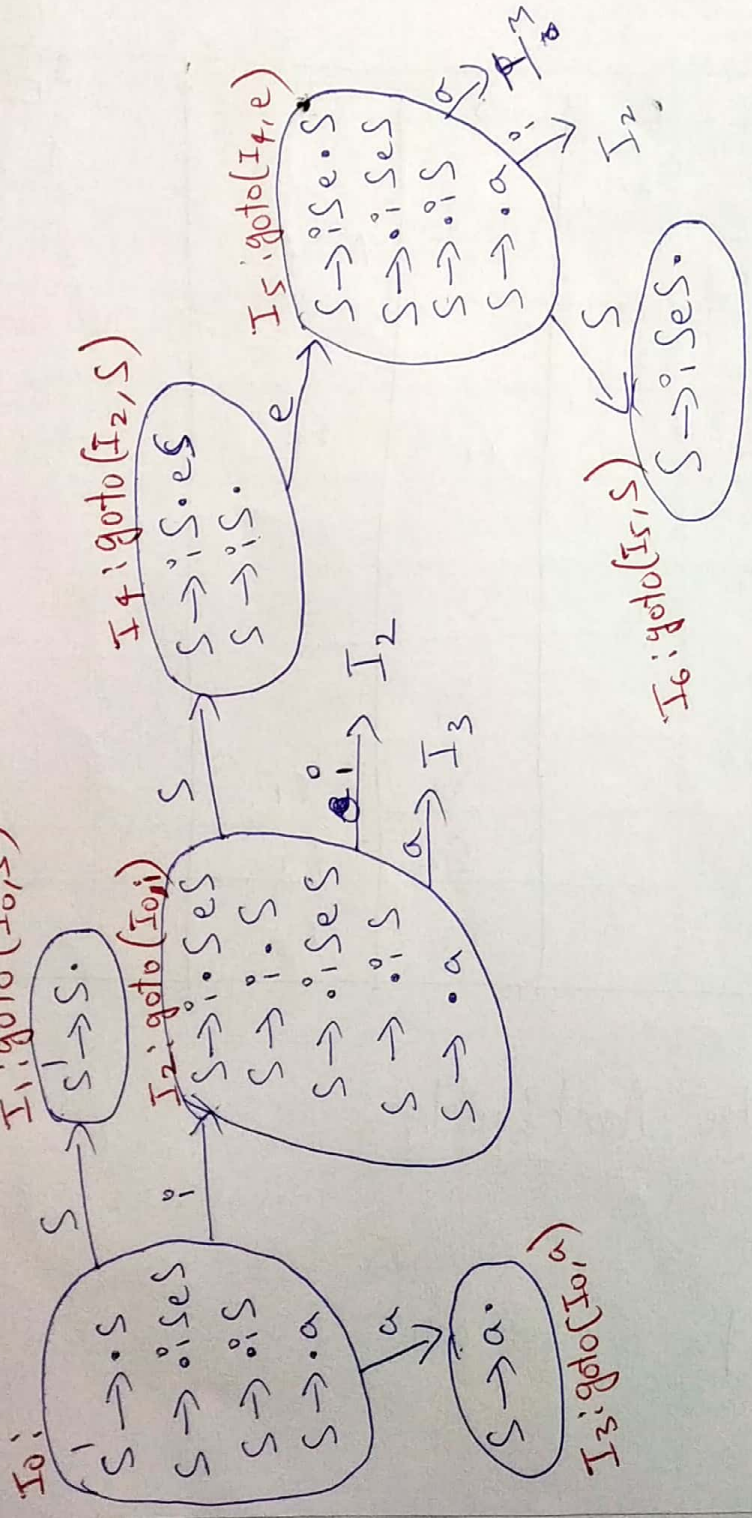
a = all other productions

Let the grammar be:

$S' \rightarrow S$

$S \rightarrow ;seS \mid ;S \mid a$

Let us construct LR(0) set of items as:



First(S) = {;, a}

Follow(S) = {e, $}

State	Action. i	e	a	$	goto S
0	S_2		S_3		1
1				Accept	
2	S_2		S_3		4
3		γ_3		γ_3	
4		S_5 or γ_2		γ_2	
5	S_2		S_3		6
6		γ_1		γ_2	

Action [5, e) has shift/reduce conflict.

Consider the input "iiaea$" for Procesing.

Stack	Input	Action with conflict resolution
$0	iiaea$	shift
$0i2	iaea$	shift
$0i2i2	aea$	shift
$0i2i2a3	ea$	Reduce S → a
$0i2i2S4	ea$	From the conflict we have chosen Reduce S → iS
$0i2S4	ea$	Reduce S → iS
$0S1	a$	Error!!

That means the choice of γ_2 in action [4, e] is not valid. Hence we will try it by choosing the shift action.

Stack	Input	Action with conflict resolution
$0	iiaea$	shift
$0i2	iaea$	shift
$0i2i2	aea$	shift
$0i2i2a3	ea$	Reduce S→a
$0i2i2S4	ea$	From the conflict we have chosen Reduce S→iS
$0i2i2S4e5	a$	shift
$0i2i2S4e5a3	$	Reduce S→a
$0i2i2S4e5S6	$	Reduce S→iSeS
$0i2S4	$	Reduce S→iS
$0S1	$	Accept

Logically also we should choose the shift operation as by shifting the else we can associate it with previous "if expression then statement". Therefore shift / reduce conflict is resolved in favour of shift.

→ After resolving the conflict we get Parsing table for dangling else problem as:

state	Action				goto
	i	e	a	$	S
0	S2		S3		1
1				Accept	
2	S2		S3		4
3		r3		r3	
4		S5		r2	
5	S2		S3		6
6		r1		r2	

* Error Recovery in LR Parser :-

The LR parser is a table driven Parsing method in which the blank entries are treated as errors. when we compile any Program we first get the syntactical errors. These errors are usually denoted by user friendly error messages.

*Example :- $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$.

(we have already constructed LR(0) items for this grammar in handling Ambiguous grammar topic).

state	Action						goto
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			Accept	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9			r_3		r_3	r_3	

Following are the rules to fill the table:

(1) If there are entries for r_j in some particular state then fill up all the blank entries r_j only.

(2) If there are shift entries only in some particular state then do not replace blank entries. Keep them as it is.

During the error detection & recovery process we have replaced some blank (error) entries by particular reduction rules. This change means we are postponing the error detection until one or more reductions are done & error will be introduced before any shift move take place.

Consider the set of items generated for obtaining the error messages.

$I_0 : E' \rightarrow \cdot E$

It means that there is no symbol before the dot. And being the initial state the stack is empty. In such a case if + (or) * (or) $ comes in the input string then we say that operand is missing for these operator.

Stack	Input	Error could be
$	+	Missing operand
$	*	missing operand
$	$	Missing operand
$	)	Unbalanced right Parenthesis

I_1 : goto (I_0, E)
$E' \rightarrow E \cdot$

If we ultimately reduce $E \rightarrow id$ then id = missing operator, id (= missing operator, id) = unbalanced right Parenthesis.

Stack	Input	Error could be
$...id	id	Missing operator
$...id	(	" "
$...id	)	unbalanced right Parenthesis

I_6 : goto (I_2, E)
$E \rightarrow (E \cdot)$

If we eventually reduce $E \rightarrow id$ then the rule becomes $E \rightarrow (id \cdot)$. That means after id we expect ')' & if $ comes then the error will be missing right Parenthesis.

Stack	Input	Error could be
$.... (id	id	Missing operator
$.... (id	(	" "
$... (id	$	Missing right Parenthesis

From all these situations, some error messages are:

(1) E1 : These errors are in states I_0, I_2, I_4 & I_5. This indicates that the operand should appear before operator. Hence the error message will be "missing operand".

(2) E2 : This error is in ')' column & from states I_0, I_1, I_2, I_4 & I_5 indicating unbalancing in right parenthesis. Hence error message will be "unbalanced right Parenthesis".

(3) E3 : The operator is expected in this case of error as it is from state I_1 or I_6. The error message will be "missing operator".

• E_4 : This error occurs at state 6 in the $ column. The state 6 expects ')' Parenthesis at the end of expression. Hence the error message will be "missing right Parenthesis"

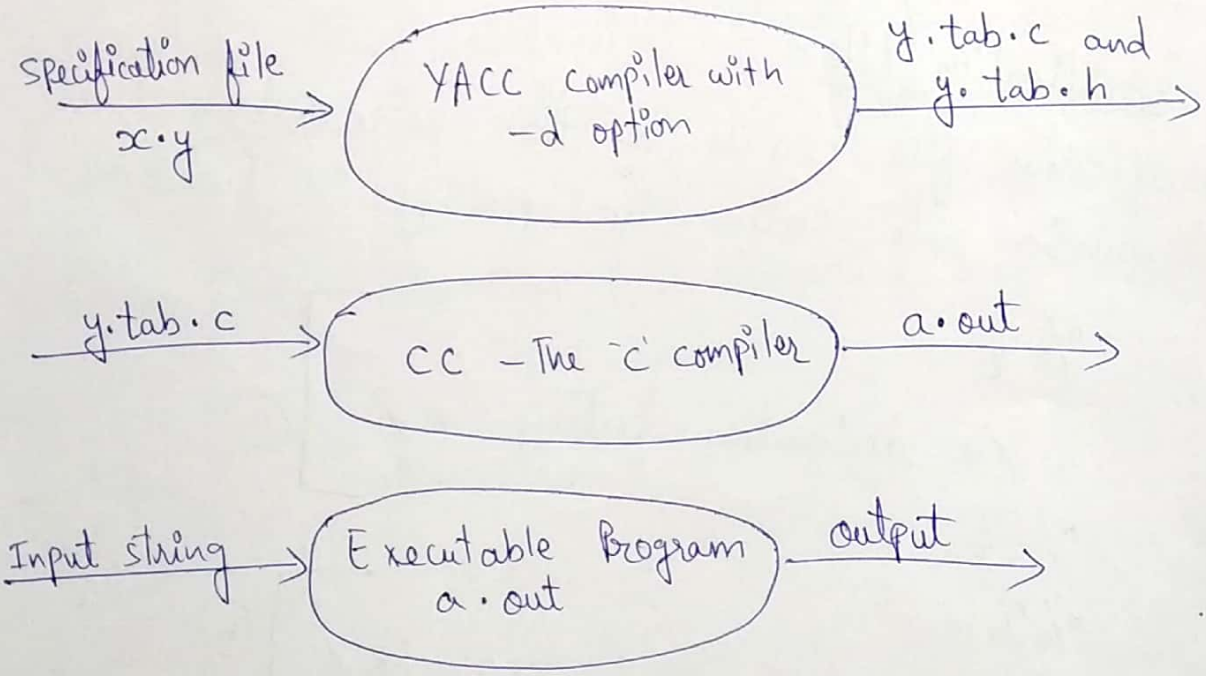
→ Modified table with included error messages :-

state	Action						goto
	id	+	*	(	)	$	E
0	S_3	E_1	E_1	S_2	E_2	E_1	1
1	E_3	S_4	S_5	E_3	E_2	Accept	
2	S_3	E_1	E_1	S_2	E_2	E_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	S_3	E_1	E_1	S_2	E_2	E_1	7
5	S_3	E_1	E_1	S_2	E_2	E_1	8
6	E_3	S_4	S_5	E_3	S_9	E_4	
7	r_1	r_1	S_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

While Parsing the input, LR Parser will refer the parsing table if it detects the error entry then respective error message will be reported. This is how we get syntax errors when we compile our Program.

* YACC – Automatic Parser Generator :

YACC is an automatic tool for generating the Parser Program. YACC stands for "Yet Another Compiler Compiler", which is basically the utility available from Unix. Basically YACC is LALR Parser generator. The YACC can report conflicts or ambiguities in the form of error messages. LEX & YACC work together to analyse the program syntactically.



YACC : Parser generator model

YACC specification file need to be written first, for example "x.y" This file is given to the YACC compiler by Unix command as:

```
$ yacc x.y
```

Then it will generate a Parser Program using your YACC specification file. This Parser Program has a standard name as "y.tab.c". This is basically Parser Program in 'C'

generated automatically.

```
$yacc -d xc.y
```

By -d option two files will get generated one is "y.tab.c" & other is "y.tab.h". The header file y.tab.h will store all the tokens & we need not have to create "y.tab.h" explicitly. The generated y.tab.c program will then be compiled by 'C' compiler & generates the executable a.out file. We can test YACC program with the help of some valid & invalid strings.

* YACC specification file :-

YACC specification file consists of three parts: (a) declaration section, (b) translation rule section & (c) supporting 'C' functions.

Eg :-

```
%{
   /* declaration section */        (a)
%}
%%
   /* Translation rule section */   (b)
%%
/* Required 'c' functions */        (c)
```

YACC specification file

(a) Declaration section :- In this section ordinary 'C' declarations can be written. We can also declare grammar tokens in this section. The declaration of tokens should be after %{ %}.

(b) Translation Rule Section: It consists of all the Production rules of context free grammar with corresponding actions.

Ex:

Rule 1	action 1
Rule 2	action 2
⋮	⋮
Rule n	Action n

→ If there are more than one alternatives to a single rule then those alternatives should be seperated by "|" character. The actions are typical 'C' statements. If CFG is:

LHS → alternative 1 | alternative 2 | .... | alternative n

then:

LHS : alternative 1 {action 1}
| " 2 {" 2}
⋮
| alternative n {action n}

(c) 'C' functions section: This section consists of one main function in which the routine yyparse() will be called. And it also consists of required 'C' functions.

*Program : Write a YACC Program that will take arithmetic expression as input & Produce the corresponding Postfix expression as output.

Sol:

```
/* LEx Program to convert Infix expression to Postfix form */
%{
  #include <stdlib.h>
  #include <stdio.h>
  #include "y.tab.h" /* contain token definitions */
%}
%%
/* Valid digit */
[0-9]+ {
  /* convert from character form to integer form */
  yylval.no = atoi(yytext);
  /* Return the appropriate token */
  return(DIGIT);
}
/* Valid Identifier / variable */
[a-z|A-Z][a-z|A-Z|0-9|_]* {
    strcpy(yylval.str, yytext);
    return(ID);
}
/* Operators */
```

```
"+"   { return (PLUS); }
"-"   { return (MINUS); }
"*"   { return (MUL); }
"/"   { return (DIV); }
"^"   { return (EXPO); }
"("   { return (OPEN); }
")"   { return (CLOSE); }
"\n"  { return 0; }

/* Ignore white spaces */
[\t]  ;

/* Ignore remaining characters */
.     ;
%%
```

* YACC Program :

```
/* YACC Program to convert Infix expression to Postfix form */
%{
   #include <stdio.h>
%}
/* The identifier can be a digit (number) or a Variable (str) */
%union
  { int no;
    char str[10];
  }
```

```
/* Declaring the tokens */
%token <no> DIGIT
%token <str> ID
/* + and - are left associative & have the least Precedence */
%left PLUS MINUS
/* then comes * & / which are also left associative */
%left MUL DIV
/* And lastly exponent operator ^ which has higher Precedence
   & is right associative */
%right EXPO
/* Brackets which has the highest Precedence */
%left OPEN CLOSE
%%
STMT : EXPR          { Printf ("\n"); }
EXPR : EXPR PLUS EXPR    { Printf ("+"); }
     | EXPR MINUS EXPR   { Printf ("-"); }
     | EXPR MUL EXPR     { Printf ("*"); }
     | EXPR DIV EXPR     { Printf ("/"); }
     | EXPR EXPO EXPR    { Printf ("^"); }
     | OPEN EXPR CLOSE
     | DIGIT     { Printf ("%d", yyval.no); }
     | ID        { Printf ("%s", yyval.str); }
     ;
%%
```

```
/* user sub-routine section */
int main(void)
{
    Printf("\n");
    yyparse();
    Printf("\n");
    return 0;
}
```

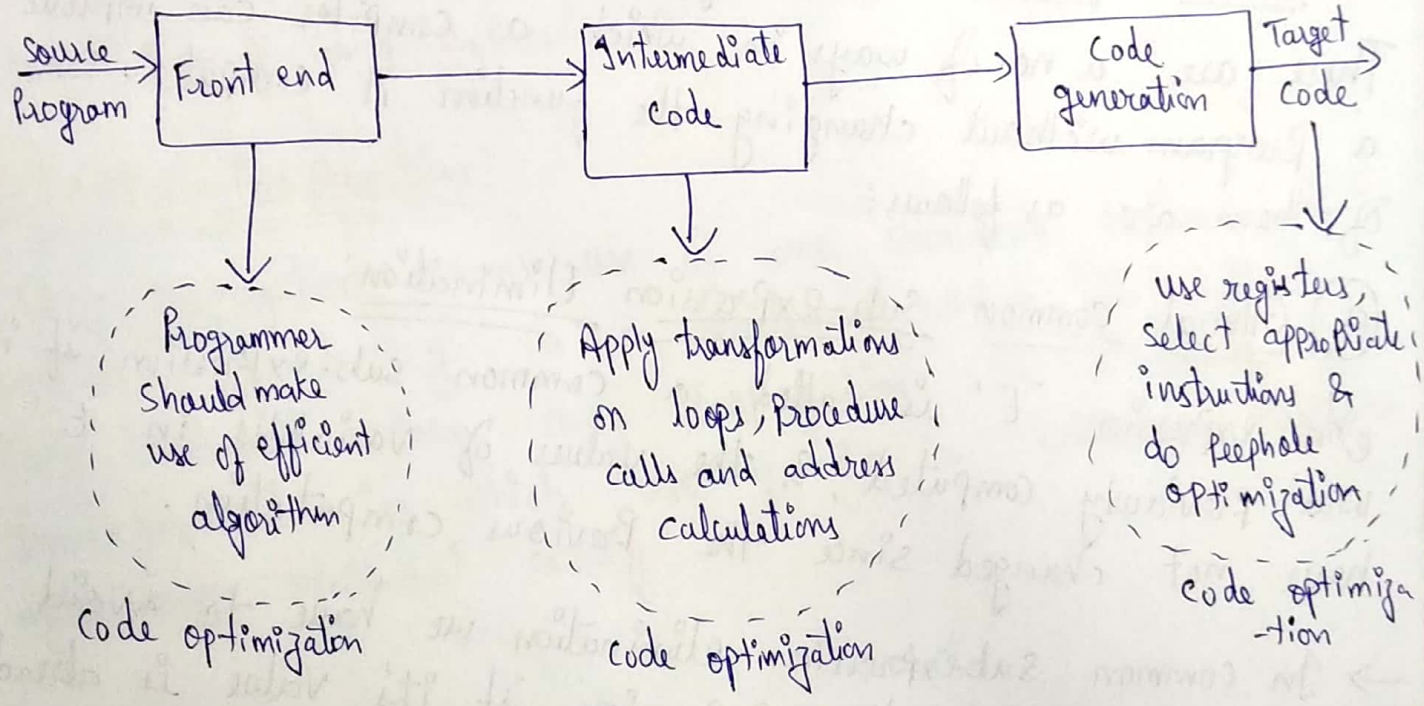
* output :-

```
$ lex postfix.l
$ yacc postfix.y
$ gcc lex.yy.c y.tab.c -ll -ly
$ ./a.out
(2+3)*(3/4+4) - 3^2
23+34/4+*32^-
```

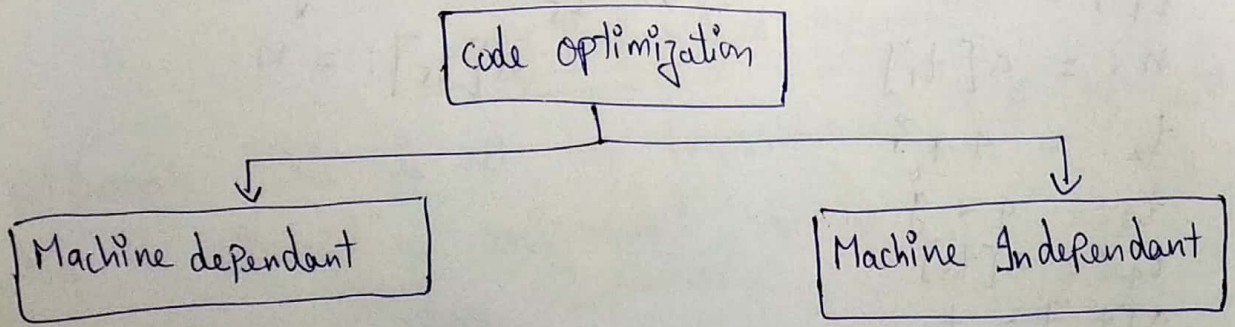
UNIT - V : Machine Independent Optimization and Code Generation

The code optimization is required to produce an efficient target code. while optimizing code :

- The semantic equivalence of the source program must not be changed.
- The improvement over the program efficiency must be achieved without changing the algorithm of the program.



code optimization



→ The machine dependant optimization is based on characteristics, of the target machine for the instruction set used & addressing modes used for the instructions to produce the efficient target code.

→ The machine independant optimization is based on the characteristics of the programming languages for appropriate programming structure & usage of efficient arithmetic properties in order to reduce the execution time.

* Various function Preserving transformations :

There are a no. of ways in which a compiler can improve a program without changing the function it computes. Some of them are as follows :

(a) Global common Sub-expression Elimination :

An expression 'E' is called a common sub-expression if it was previously computed, & the values of variables in 'E' have not changed since the previous computation.

→ In common subexpression elimination we have to avoid recomputing of the expression if it's value is already been computed.

```
t1 := 4*i
n  := a[t1]
t2 := 4*i
t3 := 4*j
t4 := a[t3]
a[t2] := t4
```

```
t5 := 4*j
a[t5] := n
```

After common subexpression elimination, the code becomes

```
t1 := 4*i
n  := a[t1]
t2 := 4*j
t3 := a[t2]
a[t1] := t3
a[t2] := n
```

In the above code 4*i and 4*j are common subexpressi -ons. They are eliminated by eliminating t_2 & t_4. Hence we replaced the corresponding variables in order to Preserve the value of each expression.

(b) Copy Propagation:-

Copy Propagation means use of one variable instead of another. If x := y is a statement which is called copy statement, then copy Propagation is a kind of transformation in which use 'y' for 'x' wherever possible after copy statement x := y.

```
n := t1                    n := t1
a[t1] := t2   ⟹           a[t1] := t2
a[t3] := n                 a[t3] := t1
```

copy Propagation example

Although this is not an improvement but it will definitely help in eliminating assignment to 'n'.

(c) Dead Code Elimination :-

A Variable is said to be live in the Program if its value can be used subsequently otherwise it is dead at that point.

→ The dead code is basically an useless code. An optimization can be performed by eliminating such dead code.

Eg1:-
```
i = j ;
...
...
x = i+10 ;
....
```

The dead code elimination can be done by eliminating the assignment statement i=j. This assignment statement is called dead assignment.

Eg2 :-
```
i = 0 ;
if (i == 1)
{ a = x+5 ;
}
```
→ dead code we can eliminate it.

(d) Constant Folding :-

Constant Folding is a technique in which the constant expressions are calculated during compilation.

Eg :-
```
const max = 3 ;
...
i := 2 + max ;
j := i * 3 + a ;
```
This can be optimized as :-
```
i := 5 ;
j := 15 + a ;
```

Eg2: m := -1 ;
n = 6 + m * k - 2 + t ;

this can be written as : n = 4 - k + t.

* Loop Optimization :

Loop optimization is a technique in which the code optimi-zation is performed on inner loops. Inner loops is a place where program spend large amount of time. Hence if number of instructions are less in inner loop the running time of the program will get decreased to a large extent.

→ loop optimization is carried out by following methods :

(a) Code Motion :

code Motion is a technique which moves the code outside the loop. If there lies some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop.

Eg

```
while (i <= Max-1)
{
Sum = Sum + a[i];
}
```

optimized as

```
n = max-1 ;
while (i<=n)
{
Sum = sum + a[i];
}
```

(b) Induction Variables & Reduction in strength:

A variable 'x' is called an induction variable of loop 'L' if the value of variable gets changed every time. It is either decremented or incremented by some constant.

Eg:

B1

```
i := i+1
t1 := 4*j
t2 := a[t1]
if t2 < 10 goto B1
```

In above code the values of i & t_1 are in locked state. i.e when value of 'i' gets incremented by 1 then t_1 gets incremented by 4. Hence i & t_4 are induction variables

→ when there are two or more induction variables in a loop, it may be possible to get rid of all but one.

(c) Reduction in strength:

The strength of certain operators is higher than others.

Eg: strength of "*" is higher than "+".

→ In strength reduction technique the higher strength operators can be replaced by lower strength operators.

Eg:

```
for (i=1; i<=50; i++)
{
  ....
  count = i*7;
  ....
}
```

Here we get values of count as 7, 14, 21 .... 50.
This code can be replaced by using strength reduction as follows:

```
temp = 7
for (i=1; i<=50; i++)
  { ...
    count = temp
    temp = temp + 7;
  }
```

The replacement of multiplication by addition will speed up the object code. Thus the strength of operation is reduced without changing the meaning of above code.

→ The strength reduction is not applied to the floating Point expressions because such a use may yield differen -t results.

(d) Loop unrolling:

In this method the no. of jumps & tests can be reduced by writing the code two times.

eg:

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

can be written as →

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

(e) Loop Fusion:

In loop fusion method several loops are merged to one loop.

Eg:

```
for i:=1 to n do
for j:=1 to m do
a[i,j]:=10
```

can be written as

```
for i:=1 to n*m do
a[i]:=10.
```

* Basic Blocks:

The basic block is a sequence of consecutive statements in which flow of control enters at the beginning & leaves at the end without halt or possibility of branching.

Eg:
$t_1 := a*5$
$t_2 := t_1+7$
$t_3 := t_2-5$
$t_4 := t_1+t_3$
$t_5 := t_2+b$

* Algorithm for Partitioning into Blocks:

Any given Program can be partitioned into basic blocks by using following algorithm. We assume that an intermediate code is already generated for the given Program.

(1) First determine the leaders by using following rules:
a) The first statement is a leader.
b) Any target statement of conditional or unconditional goto is a leader.
c) Any statement that immediately follow a goto or unconditional goto is a leader.

(2). The basic block is formed starting at the leader statement & ending just before the next leader statement appearing.

*Example: Consider the following program code for computing dot product of two vectors 'a' & 'b' of length 10 & Partition it into basic blocks.

```
Prod = 0;
i = 1;
do
{
    Prod = Prod + a[i] * b[i];
    i = i+1;
} while (i<=10);
```

Sol: First let us write the equivalent three address code for the above program.

1. Prod := 0
2. i := 1
3. t_1 := 4*i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4*i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := Prod + t_5
9. Prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i<=10 goto (3)

According to the algorithm :

- statement 1 is a leader by rule 1(a)
- statement 3 " also " " " 1(b)

Hence statement 1 & 2 form the basic block. similarly statement 3 to 12 form another basic block.

Block 1 :

```
1. Prod := 0
2. i := 1
```

Block 2 :

```
3. t1 := 4*i
4. t2 := a[t1]
5. t3 := 4*i
6. t4 := b[t3]
7. t5 := t2*t4
8. t6 := Prod + t5
9. Prod := t6
10. t7 := i+1
11. i := t7
12. if i<=10 goto (3)
```

* Flow Graphs :

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flow graph are represented by basic blocks.

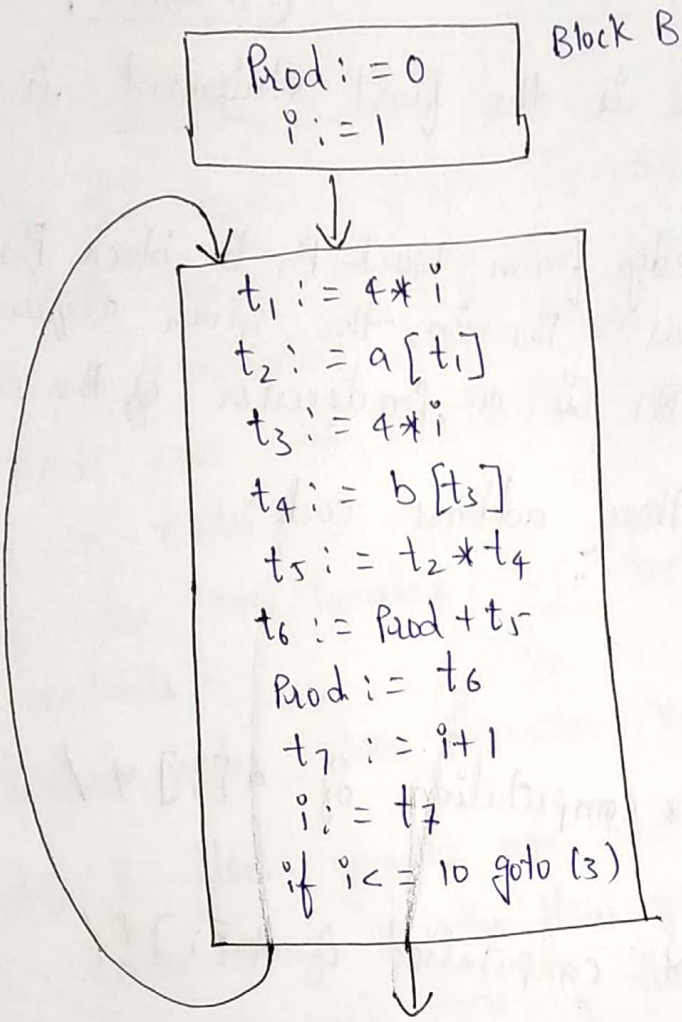
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B_1 to block B_2 if B_2 immediately follows B_1 in the given sequence. we can say that B_1 is a predecessor of B_2.

Example: Consider the three address code:

```
1. Prod := 0
2. i := 1
3. t1 := 4*i
4. t2 := a[t1]   /* computation of a[i] */
5. t3 := 4*i
6. t4 := b[t3]   /* computation of b[i] */
7. t5 := t2 * t4
8. t6 := Prod + t5
9. Prod := t6
10. t7 := i+1
11. i := t7
12. if i<=10 goto (3).
```

The flow graph for the above code can be drawn as follows:

→ In this flow graph block B_1 is a initial block.



Block B_1 : the initial block

Flow graph

*Loop : Loop is a collection of nodes in the flow graph such that

- All such nodes are strongly connected. That means always there is a path from any node to any other node within that loop.
- The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.
- The loop that contains no other loop is called inner loop.

* DAG Representation :

The directed acyclic graph is used to apply transformations on the basic block. To apply the transformations on basic block a DAG is constructed from three address statement.

→ A DAG can be constructed for the following type of labels on nodes.

- Leaf nodes are labeled by identifiers or variable names or constants.
- Interior nodes store operator values.

The DAG & flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

* Algorithm for construction of DAG :

Three address code can be in the following formats :

Case (i) $x : y \text{ op } z$

Case (ii) $x := \text{op } y$

Case (iii) $x := y$

With the help of following steps DAG can be constructed:

step 1 : If 'y' is undefined then create node (y). similarly if 'z' is undefined create a node (z).

step 2 : For the case (i) create a node (op) whose left child is node (y) & node (z) will be the right child.

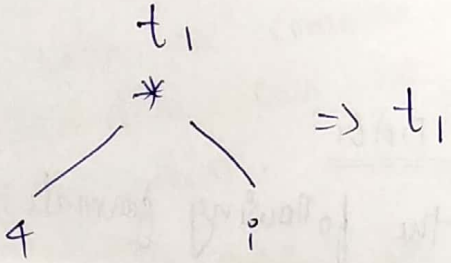
Also check for any common sub-expressions. For the case(ii) determine whether it is a node labeled op, such node will have a child node (y). In case (iii) node 'n' will be node (y).

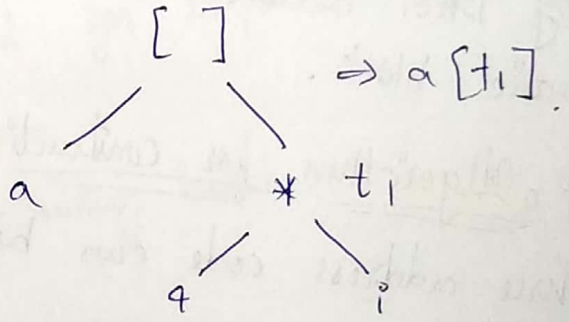
step3 : Delete 'x' from list of identifiers for node (x). Append 'x' to the list of attached identifiers for node 'n' found in step2.

* Example : construct DAG for below three address code.

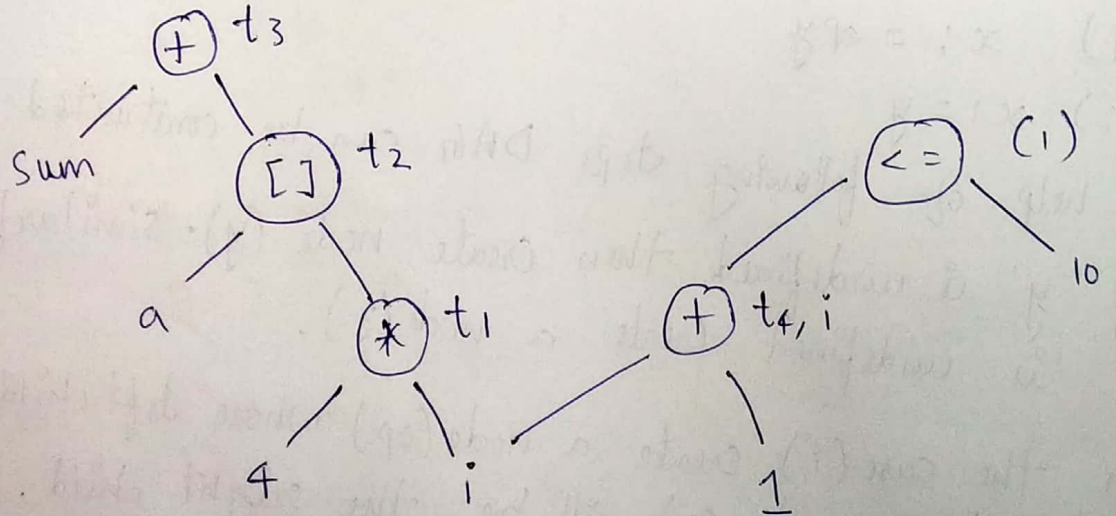
1. $t_1 := 4 * i$
2. $t_2 := a[t_1]$
3. $t_3 := sum + t_2$
4. $sum := t_3$
5. $t_4 := i + 1$
6. $i := t_4$
7. if $i < 10$ goto (1)

sol :





like this if we construct we get :



DAG for above 3-address code

* Applications of DAG:

DAG is used in:

(1) Determining the common sub-expressions
(2) " which names are used inside the block & computed outside the block.
(3) Determining which statements of the block could have their computed value outside the block.
(4) simplifying the list of quadruples by eliminating the common sub-expressions & not performing the assignment of the form x := y unless & until it is a must.

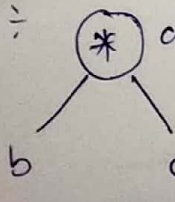
* DAG Based Local Optimization:

DAG can be constructed for a block and certain transfor-mations such as common subexpression elimination, dead code elimination can be applied for performing the local optimization.

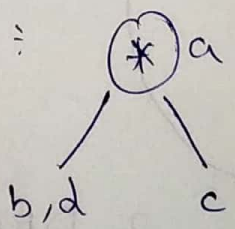
Eg: Construct DAG for the following block

```
a := b * c
d := b
e := d * c
b := e
f := b + c
g := f + d
```

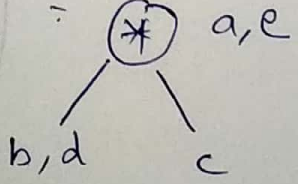
Step1:



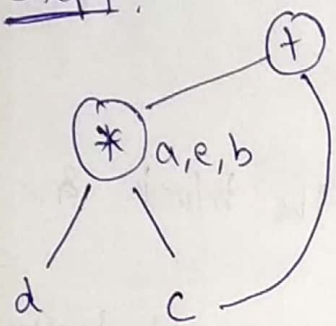
Step2:



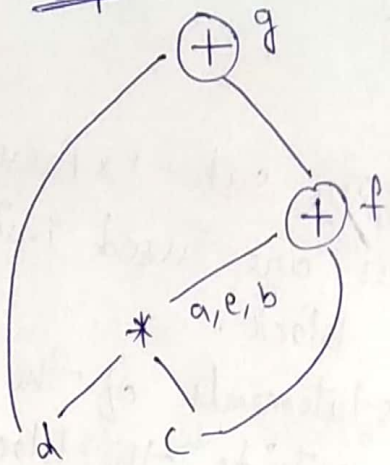
Step3:



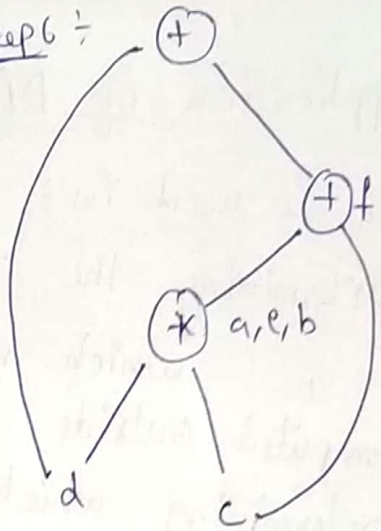
Step4 :



step5 :



Step6 :



The optimized code can be generated by traversing the DAG

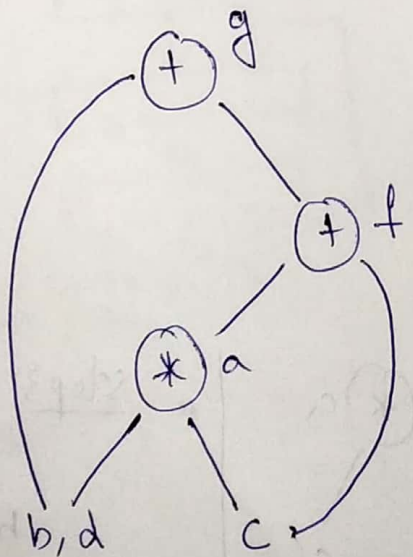
→ The local optimization on the above block can be done as

(1) A common subexpression $e = d * c$ which is actually $b * c$ (since $d := b$) is eliminated.

(2) A dead code for $b = e$ is eliminated.

The optimized code of basic block is :

```
a := b * c
d := b
f := a + c
g := f + d
```



Optimized Code & DAG

* Data Flow Analysis :

Data flow analysis determines the information regarding the definition & use of the data in the Program. Analysis is made on flow of data.

→ Data flow analysis is basically a Process in which the values are computed using data flow properties.

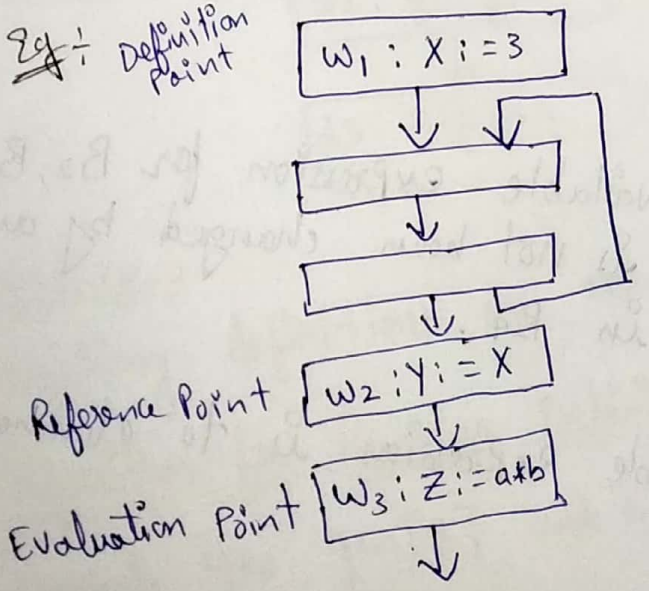
* Data flow Property : It represents certain information regarding usefulness of data items for the purpose of optimization.

→ A Program Point containing the definition is called "definition Point".

→ A Program point at which a reference to a data item is made is called "reference Point".

→ A Program point at which some evaluating expression is given is called "evaluation point".

Eg :



Program Points

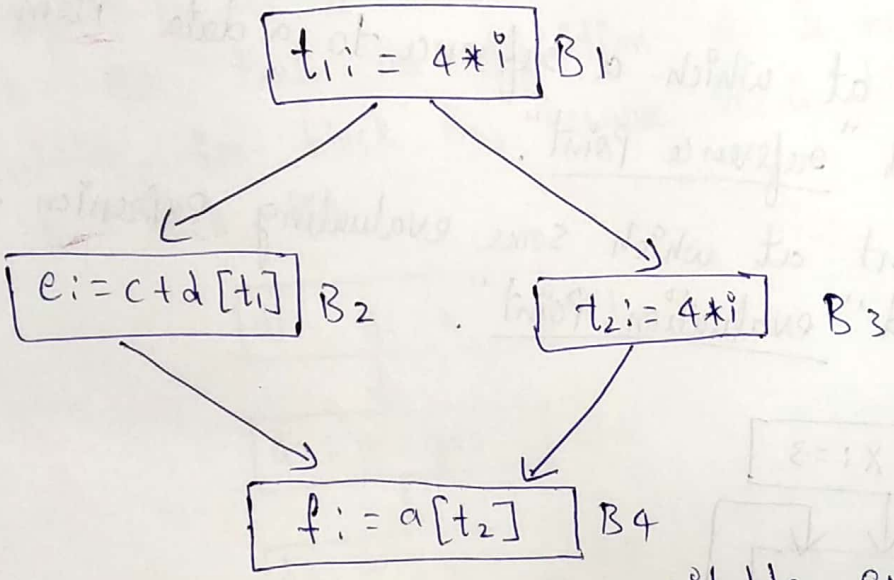
Data flow properties are as follows :-

(a) Available Expressions :-

An expression x+y is available at a program point 'w' if & only if all paths are reaching to 'w'.

- The expression x+y is said to be available at its evaluation point.
- The expression x+y is said to be available if no definition of any operand of expression follows its last evaluation along the path.

Example :-



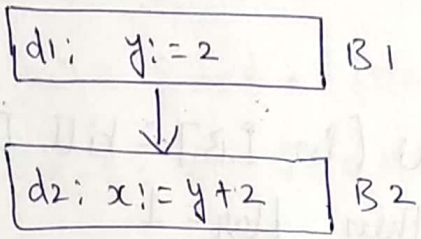
The expression 4*i is the available expression for B_2, B_3 & B_4 because this expression is not been changed by any of the block before appearing in B4.

*Advantage :- The use of available expression is to eliminate common sub-expressions.

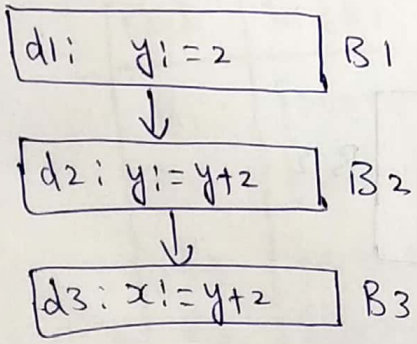
(b) Reaching Definitions : (10)

Reaching definitions is one of the most common & useful data-flow schemas. A definition 'D' of variable 'x' is killed when there is a redefinition of x. A definition D reaches at Point 'P' if there is a path from 'D' to P along which 'D' is not killed.

Eg :-



The definition d_1 is said to a reaching definition for Block B_2. But the definition d_1 is not a reaching definition in block B_3, because it is killed by definition d_2 in block B_2.



Advantage :-

Reaching definitions are used in constant & Variable Propagation.

* Algorithm for Reaching Definition :-

1. Compute the gen[B] and kill [B] for each block of flow graph.

2. Initialize $in[B] = \phi$ for all B. similarly $out[B] = gen[B]$

3. flag = 1

4. if flag = 1 then repeat steps 5 to 7.

5. flag = 0;

6. for each block B do step 7.

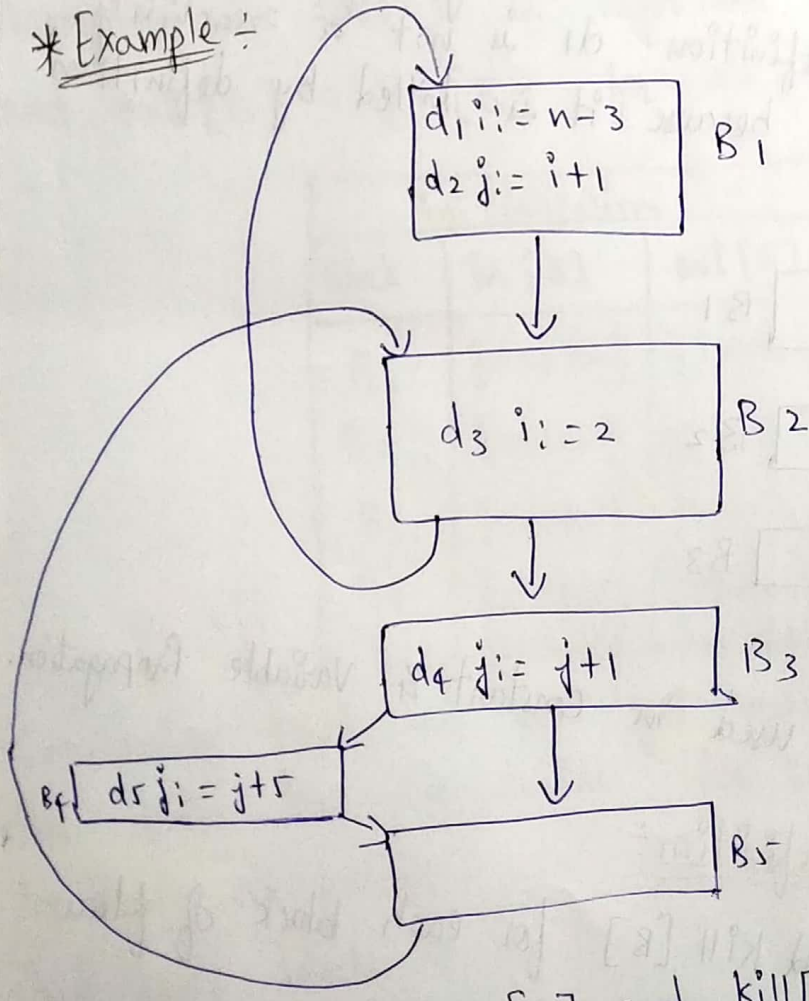
7. $in[B] := \bigcup_p out[P]$ where 'P' is a Predecessor of block B

old := out [B]

out[B] := gen[B] ∪ (in [B] − kill [B]);

if out [B] ≠ old then flag = 1

*Example :-



We can represent gen [B] and kill [B] by bit vectors. That means we can represent each di by 0's & 1's. The line(1)

Of algorithm is for computing gen[B] and kill[B].

Block	gen[B]	kill [B]
B_1	gen $[B_1] = \{d_1, d_2\} = \{11000\}$	kill $[B_1] = \{d_3, d_4, d_5\} = \{00111\}$
B_2	gen $[B_2] = \{d_3\} = \{00100\}$	kill $[B_2] = \{d_1\} = \{10000\}$
B_3	gen $[B_3] = \{d_4\} = \{00010\}$	kill $[B_3] = \{d_2, d_5\} = \{01001\}$
B_4	gen $[B_4] = \{d_5\} = \{00001\}$	kill $[B_4] = \{d_2, d_4\} = \{01010\}$
B_5	gen $[B_5] = \phi = \{00000\}$	kill $[B_5] = \phi = \{00000\}$

According to line(2) of algorithm we initialize in $[B] = \phi$ and out $[B]$ = gen $[B]$ for all the blocks B.

initialization		
Block	in [B]	out [B]
B_1	{00000}	{11000}
B_2	{00000}	{00100}
B_3	{00000}	{00010}
B_4	{00000}	{00001}
B_5	{00000}	{00000}

Considering that we will get more than one iterations for the while loop computation for in [B] and out [B] can be done as follows. in $[B_1] = \bigcup$ out [P] where 'P' is Predecessor blocks.

Predecessor of B_1 is B_2.

$in[B_1] = out[B_2]$

$= 00100$

$out[B_1] = gen[B_1] \cup (in[B_1] - kill[B_1])$

$= 11000 \cup (00100 - 00111)$

$= 11000$

Now consider for block B_2.

$in[B_2] = out[B_1] + out[B_5]$ $\because B_1$ & B_5 are Predecessor to B_2.

$= 11000 + 00000$

$= 11000$

$out[B_2] = gen[B_2] \cup (in[B_2] - kill[B_2])$

$= 00100 \cup (11000 - 10000)$

$= 01100$

consider for block B_3 $\because B_2$ is Predecessor to B_3

$in[B_3] = out[B_2]$

$= 01100$

$out[B_3] = gen[B_3] \cup (in[B_3] - kill[B_3])$

$= 00010 + (01100 - 01001)$

$= 00110$

similarly

$in[B_4] = out[B_3]$ $\because B_3$ is Predecessor to B_4

$= 00110$

$out[B_4] = gen[B_4] \cup (in[B_4] - kill[B_4])$

$= 00001 + (00110 - 01010)$

$= 00101$

similarly

to

$$in[B_5] = out[B_3] \cup out[B_4]$$

(1) $= 00110 + 00101 = 00111$

and $out[B_5] = gen[B_5] \cup (in[B_5] - kill[B_5])$

(2) $= 00000 + (00111 - 00000)$

$= 00111$

Then the next two iterations can be computed in this way,

* as we are considering that this while loop will execute for more than one.

Pass 1		
	in [B]	out [B]
B_1	{00100}	{11000}
B_2	{11000}	{01100}
B_3	{01100}	{00110}
B_4	{00110}	{00101}
B_5	{00111}	{00111}

Pass 2		
Block	in [B]	out [B]
B_1	{01100}	{11000}
B_2	{11111}	{01111}
B_3	{01111}	{00110}
B_4	{00110}	{00101}
B_5	{00111}	{00111}

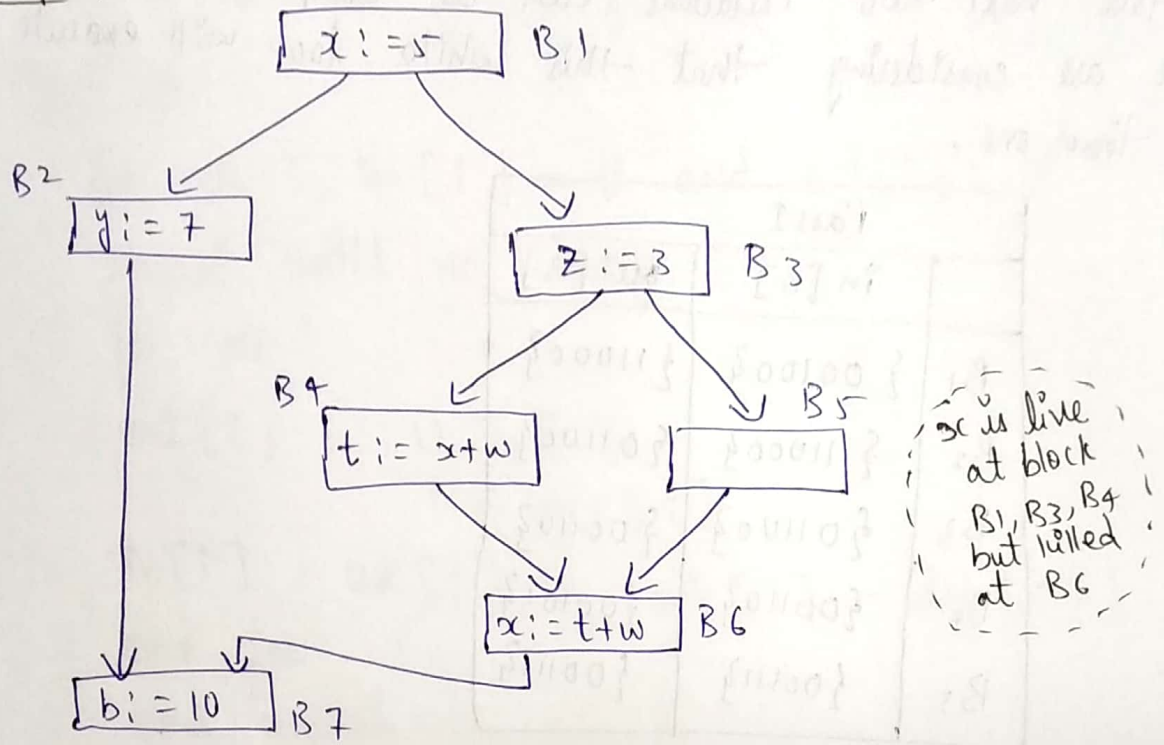
Pass 3		
Block	in [B]	out [B]
B_1	{01111}	{11000}
B_2	{11111}	{01111}
B_3	{01111}	{00110}
B_4	{00110}	{00101}
B_5	{00111}	{00111}

After Pass 3 same iterations are coming for in [B] & out [B].
Hence Pass 3 is the final iteration.

(c) Live Variables :

A Variable 'x' is live at some point 'p' if there is a Path from 'p' to the exit, along which the value of x is used before it is redefined. Otherwise the variable is said to be dead at that point.

*Example:



Live Variables

*Advantages :

→ Live variables are useful in register allocation

→ Live " " " for dead code elimination.

→ we define data-flow equations directly in terms of IN[B] and OUT[B] which represent the set of variables live at the Points immediately before & after block B respectively. These equations can also be derived by first defining the transfer

functions of individual statements & composing them to create the transfer function of a basic block. Define

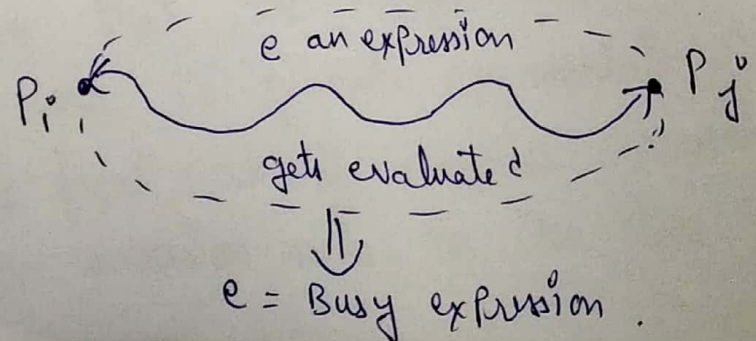
① def_B as the set of variables 'defined' in B prior to any use of that variable in B.

② "use_B" as the set of variables whose values may be used in 'B' prior to any definition of the variable.

* Algorithm:-

for all i, $in[i] = \emptyset$ and $out[i] = \emptyset$
repeat until no change
for all i
$out[i] = \bigcup_{i \in succ[i]} in[i]$
$in[i] = use[i] \cup (out[i] - def[i])$
end for
end repeat.

(d) Busy Expression:- An expression 'e' is said to be a busy expression along some path $P_i \ldots P_j$ if and only if an evaluation of 'e' exists along some path $P_i \ldots P_j$ and no definition of any operand exists before its evaluation along the path.

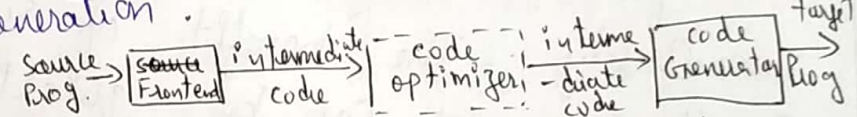


Advantage: Busy expressions are useful in performing code movement optimization.

* Code Generation:

Code generation is the final activity of compiler. Code generation is a process of creating assembly language / machine language statements which will perform the operations specified by the source program when they run.

→ Object code generation must have properties like correctness, High quality, Efficient use of resources of the target machine, Quick code generation.



Position of code generator

* Issues in Code Generation:

Some of the common issues in design of code generator are as follows:

(a) Input to the code generator:

The code generation phase takes intermediate code as input. This intermediate code along with the symbol table information is used to determine the runtime addresses of the data objects.

→ The intermediate code generated by the front end should be such that the target machine can easily manipulate it, in order to generate appropriate target machine code. In the front end of compiler necessary type checking & type conversion needs to be done. The detection of the

Semantic errors should be done before submitting the input to the code generator. The code generation phase requires the complete error free intermediate code as input.

(b) Target Programs:-

The output of code generator is target code. The target code comes in three forms such as: absolute machine language, relocatable machine language and assembly language.

→ The advantage of Producing target code in absolute machine form is that it can be placed directly at the fixed memory location & then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

→ The advantage of Producing the relocatable machine code as output is that the subroutines can be compiled separately. Relocatable object modules can be linked together & loaded for execution by a linking loader. This provides great flexibility of compiling the subroutines separately.

→ The advantage of Producing assembly code as output makes the code generation process easier. The symbolic instructions & macro facilities of assembler can be used to generate the code. It is advantageous to have assembly language as output for the machines with small memory.

→ out of these three forms of output, target code it is always preferable to have relocatable machine code as target code.

(c) Instruction selection:-

The uniformity & completeness of instruction set is an important factor for the code generator. The selection of instruction depends upon the instruction set of target machine. The speed of instruction & machine are important factors in selection of instructions.

→ For each type of three address code, the code skeleton can be prepared which ultimately gives the target code for the corresponding construct.

Eg: $x := a+b$ for this code sequence can be generated as:

```
MOV  a, R0  /* loads the 'a' to register R0 */
ADD  b, R0  /* Performs the addition of b to R0 */
MOV  R0, x  /* stores the contents of register R0 to x */
```

Here the code is generated line by line. Such a line by line code generation process generates the poor code because the redundancies can be achieved by subsequent lines & those cannot be considered in the process of line by line code generation.

Eg: $x := y+z$
$a := x+t$

The code for the above statements can be generated as follows:

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
MOV  x, R0
ADD  t, R0
MOV  R0, a
```

⇒ This code is a poor code because MOV R0, a is not used and statement MOV a, R0 is redundant. Hence the efficient code can be:

```
MOV y, R0
ADD z, R0
ADD t, R0
MOV R0, a
```

The quality of generated code is decided by its speed & size. Simply line by line translation of three address code into target code leads to correct code but it can generate unacceptably non efficient target code.

(d) Register allocation: If the instruction contains register operands then such a use becomes shorter & faster than that of using operands in the memory. Hence while generating a good code efficient utilization of register is an important factor. There are two activities done while using registers.

(1) Register allocation: During register allocation, select appropriate set of variables that will reside in registers.

(2) Register Assignment: During register assignment, pick up the specific register in which corresponding variable will reside. Certain machines require register pairs such as even odd numbered registered for some operands & results. Obtaining minimum assignment of registers to variable is difficult.

Eg: IBM systems integer multiplication requires register pair.

Eg: $t_1 := a + b$
$t_1 := t_1 * c$
$t_1 := t_1 / d$

$\Rightarrow$ efficient machine code sequence will be:

```
MOV a, R0
ADD b, R0
MUL c, R0
DIV d, R0
MOV R0, t1
```

(e) Choice of Evaluation Order:

The evaluation order is an important factor in generating an efficient target code. Some orders require less number of registers to hold the intermediate results than the others. Picking up the best order is one of the difficulty in code generation. We can avoid this problem by referring the order in which the three address code is generated by semantic actions.

* Simple Code Generation Algorithm:

In this method computed results can be kept in registers as long as possible.

Eg: x := a + b;

The corresponding target code is:

```
ADD  b, Ri
```

Here Ri holds value of 'a' have cost = 2

or

```
Mov. b, R1
ADD  R1, R0
```

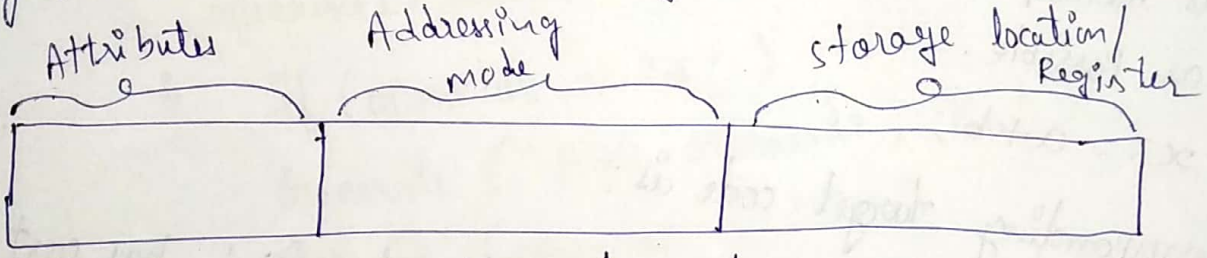
here R0 holds value of a have cost = 2

The code generator algorithm uses descriptors to keep track of register contents & addresses for names.

→ A register descriptor is used to keep track of what is currently in each register. The register descriptors show that initially all the registers are empty. As the code generation for the block progresses the registers will hold the values of computations.

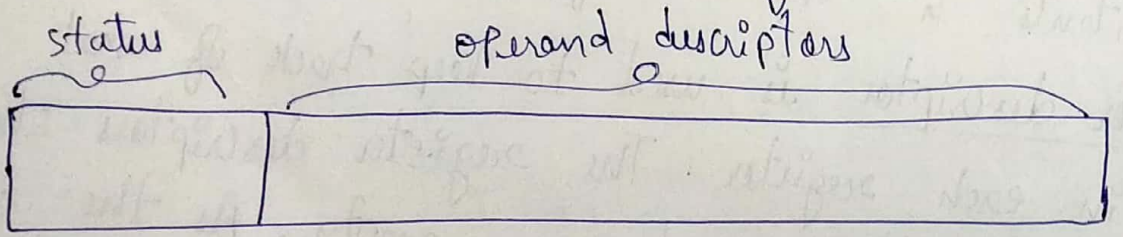
→ The address descriptor stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table & is used to access the variables.

→ The modes of operand addressability are as follows;
'S' is used to indicate value of operand in storage.
'R' " " " " " " " " register
IS indicates that the address of operand is stored in storage i.e indirect accessing
IR indicates that the address of operand is stored in register i.e indirect accessing.



Address descriptor

The attributes mean type of the operand. It generally refers to the name of temporary variables. The addressing mode indicates whether the addresses are of type 'S', 'R', 'IS', 'IR'. The third field is location field which indicates whether the address is in storage location or in register.



Register descriptor

By using register descriptors we can keep track of the registers which are currently occupied. The status field is of Boolean type which is used to check whether the register is occupied with some data or not. when the status field holds the value 'True' then operand descriptors field contains the pointer to the operand descriptor who is having the latest value in the register.

* Algorithm for code Generation :-

```
Gen_Code (operator, operand1, operand2)
{ if (operand1.addressmode = 'R')
  { if (operator = '+')
    Generate ('ADD operand2, R0');
    else if (operator = '-')
      Generate ('SUB operand2, R0');
    else if (operator = '*')
      Generate ('MUL operand2, R0');
    else if (operator = '/')
      Generate ('DIV operand2, R0');
  }
  else if (operand2.addressmode = 'R')
  { if (operator = '+')
      Generator ('ADD operand1, R0');
    else if (operator = '-')
      Generate ('SUB operand1, R0');
```

```
else if (operator = '*')
    Generate ('MUL operand1, R0');
else if (operator = '/')
    Generate ('DIV operand1, R0');
}
else
{
    Generate ('MOV operand2, R0');
        if (operator = '+')
    Generate ('ADD operand2, R0');
        else if (operator = '-')
        Generate ('SUB operand2, R0');
    else if (operator = '*')
        Generate ('MUL operand2, R0');
    else if (operator = '/')
        Generate ('DIV operand2, R0');
}
```

* Example :

Generate code for the following expression :

x := (a+b) * (c-d) + ((e/f) * (a+b))

The corresponding three address code can be given as :

$t_1 := a+b$

$t_2 := c-d$

$t_3 := e/f$

$t_4 := t_1 * t_2$

$t_5 := t_3 * t_1$

$t_6 := t_4 + t_5$

Using simple code generation algorithm the sequence target code can be generated as :

Eg:

Three Address code	Target code sequence	Register Descriptor	Operand Descriptor
$t_1 := a+b$	MOV a, R_0 ADD b, R_0	Empty R_0 contains t_1	t_1 | R | R_0
$t_2 := c-d$	MOV c, R_1 SUB d, R_1	R_1 contains c R_1 " t_2	t_2 | R | R_1
$t_3 := e/f$	MOV e, R_2 DIV f, R_2	R_2 contains e R_2 " t_3	t_3 | R | R_2
$t_4 := t_1 * t_2$	MUL R_0, R_1	R_0 contains t_1 R_1 " t_2 R_1 " t_4	t_4 | R | R_1
$t_5 := t_3 * t_1$	MUL R_2, R_0	R_2 contains t_3 R_0 " t_1 R_0 " t_5	t_5 | R | R_0
$t_6 := t_4 + t_5$	ADD R_1, R_0	R_1 contains t_4 R_0 " t_5 R_0 " t_6	t_6 | R | R_0

* Register Allocation & Assignment

Use of register operands instead of memory operands is always the faster way. This means registers help in generating the good code.

* Various strategies used in Register Allocation & Assignment

(1) Global Register Allocation:-

while generating the code the registers are used to hold the values for the duration of single block. All the live variables are stored at the end of each block. For the

Variables that are used consistently we can allocate specific set of registers. Hence allocation of variables to specific registers that is consistant across the block boundaries is called global register allocation.

→ The global register allocation has a strategy of storing the most frequently used variables in fixed registers through out the loop.

→ Another strategy is to assign some fixed no. of global registers to hold the most active values in each inner loop.

→ The registers not already allocated may be used to hold values local to one block.

→ In certain languages like 'C' or Bliss Programmer can do the register allocation by using register declaration.

(2) Usage Count :

The usage count is the count for the use of some variable 'x' in some register used in any basic block. The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation. The approximate formula for usage count for the loop 'L' in some basic block 'B' can be given as:

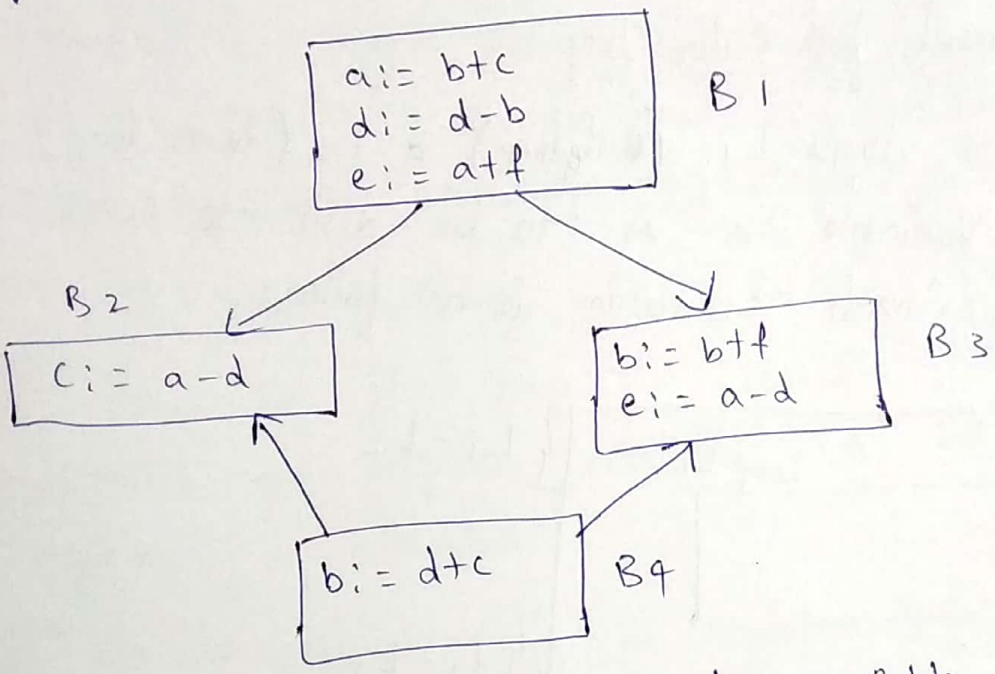
$$\sum_{\text{block } B \text{ in } L} (use(x,B) + 2 * live(x,B))$$

where use(x,B) is no. of times 'x' used in block 'B' prior to any definition of 'x' & live(x,B) = 1 if 'x' is live on exit from 'B', otherwise live(x) = 0.

B in L

Eg:

Consider a block B_1, B_2, B_3, B_4 & count the usage count for block 'B' in following loop 'L'.



The usage count for block B_1 for variable 'a' is

use $(a, B_1) = 0$ ∵ 'a' is defined in B_1 before use

$2 *$ live $(a, B_1) = 2$ ∵ 'a' " live on exit of B_1 hence live $(a, B_1) = 1$

∴ (use $(a, B_1) + 2 *$ live $(a, B)) = 2$

The usage count for block B_2 and B_3 for variable 'a' is

use (a, B_2) = use $(a, B_2) = 1$ ∵ 'a' is used in B_1 & B_2 before definition.

$2 *$ live (a, B_1) = $2 *$ live $(a, B_2) = 0$ ∵ a is not live on exit of B_1 & B_2

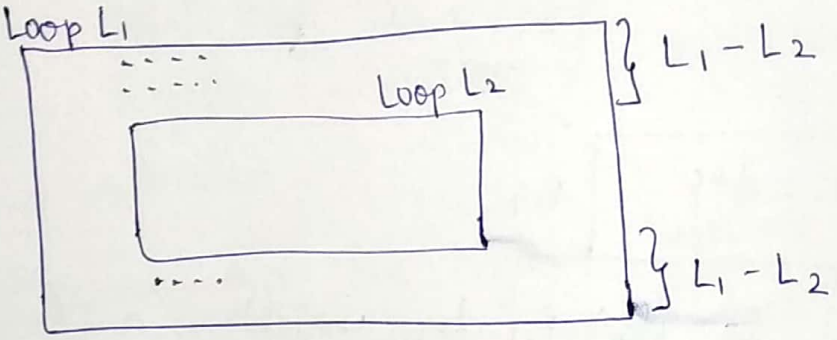
$$\sum_{B \text{ in } L} \text{use}(a, B) = 2$$

$$\sum_{B \text{ in } L} \text{use}(a, B) + 2 * \text{live}(a, b) = 2 + 2 = 4.$$

hence the usage count of 'a' is 4. That means compiler can save 4 units of cost by selecting a for the global register allocation.

(c) Register Assignment for Outer loop :

Let there be two loops L_1 (outer loop) & L_2 (inner loop). And allocation of variable 'a' is to be done to some register. The approximate scenario is as follows:



Nested loops

Following rules must be followed for register assignment of outer loop:

1) If 'a' is allocated in loop L_2 then it should not be allocated in $L_1 - L_2$

2) If 'a' is allocated in L_1 & it is not allocated in L_2 then store 'a' on 'a' entrance to L_2 & load 'a' while leaving L_2.

3) If 'a' is allocated in L_2 & not in L_1 then load 'a' on entrance of L_2 & store 'a' on exite from L_2.

(d) Graph Coloring for Register Assignment :

When variable is encountered, register is allocated for it.

But a time may come when a register is needed for computation but all the registers are occupied. In such a situation we may need to make some registers free for reusability. Again from the heap of allocated registers which register can be freed is another problem. To solve this, a graph coloring technique is used.

→ The graph coloring works in two passes as below:-

(1) In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.

(2) In the second pass, the register inference graph is prepared. In register inference graph each node is a symbolic register & an edge connects two nodes where one is live at a point where other is defined.

(3) Then a graph coloring technique is applied for this register inference graph using k-color. The k-colors can be assumed to be no. of assignable registers. In graph coloring technique no two adjacent nodes can have same color. Hence in register inference

* Peephole Optimization

Peephole optimization is a simple & effective technique for locally improving target code. This technique is applied to

improve the Performance of the target Program by examining the short sequence of target instructions & replacing these instructions by shorter or faster sequence.

→ Peephole optimization can be applied on the target code using the following:

① Redundant instruction Elimination:

Redundant loads & stores can be eliminated in this type of transformations.

Eg: Mov R0, x
Mov x, R0

we can eliminate the second instruction since 'x' is already in R0. But if (Mov x, R0) is a label statement then we cannot remove it.

→ we can eliminate the unreachable instructions also.

Eg:
```
sum = 0
if (sum)
    Printf ("%d", sum);
```

Now this "if" stmt will never get executed hence we can eliminate such unreachable code.

Eg2:
```
int fun (int a, int b)
{   c = a+b;
    return c;
    Printf ("%d", c); /* unreachable code
                         & so it can be
                         eliminated */
}
```

(2) Flow of control optimization :-

Using Peephole optimization unnecessary jumps can be eliminated.

Eg :-

```
goto test
...
test: goto done
...
done
```

can be changed as ⟹

```
goto done
...
test: goto done
...
done:
```

Thus we reduce one jump by this transformation.

Eg2 :-

```
if a<b goto test
...
test: goto done
...
done:
```

⟹

```
if a<b goto done
...
test: goto done
...
done:
```

(3) Algebraic simplification :-

Peephole optimization is an effective technique for algebraic simplification.

Eg :- statements such as :

x := x + 0

(or) x := x * 1 can be eliminated by peephole optimization.

(4) Reduction in strength :-

Certain machine instructions are cheaper than the other. In order to improve the performance of the intermediate code we can replace these instructions by equivalent cheaper instruction.

Eg :- x^2 is cheaper than $x * x$. similarly addition & substraction are cheaper than multiplication & division.

(5) Machine Idioms :-

The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Eg :- Some machines have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.