— Aho ullman.

Compiler design

Translator (HLL → LLL)

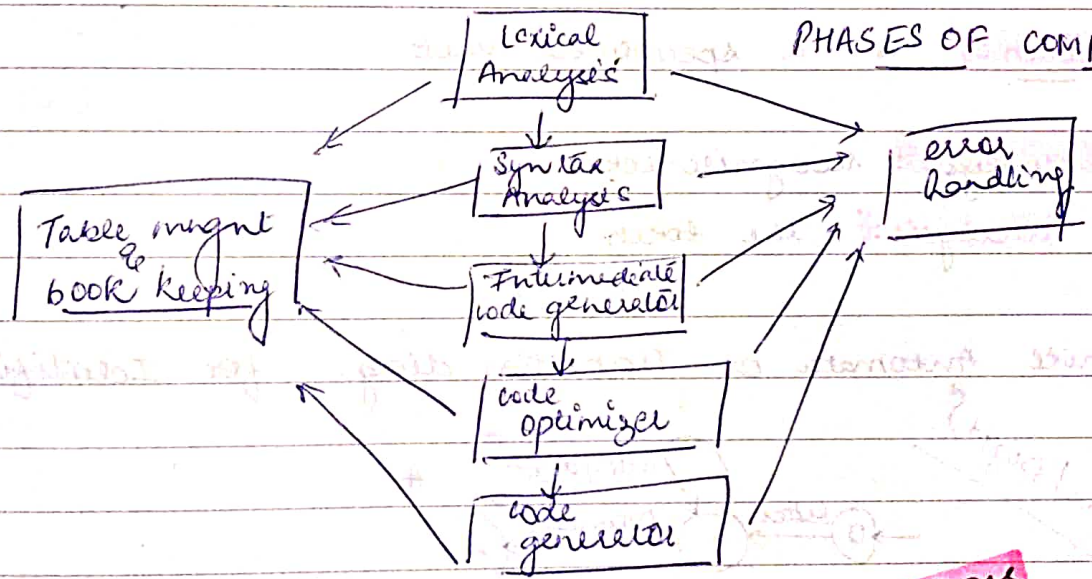
diff + egs
1) compiler ✓
2) Interpreter ✓
3) Assembler ✓

→ Q. Why do we need translator?
– HLL → LLL (ML)

Q why C middle lang?

→ Assignment (I)

PHASES OF COMPILATION



Passes →
Boot Strapping

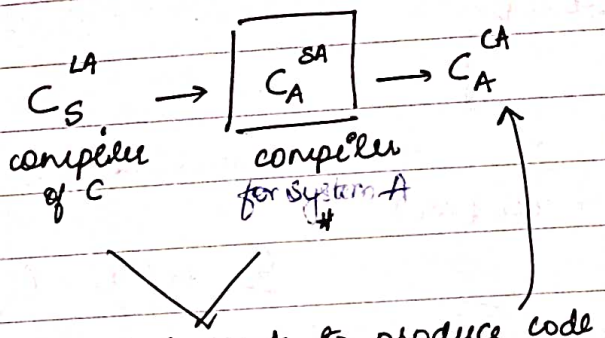
C → Multipass compiler
1st 3 stages are compiled.
1 2 3 →

unix → single pass compiler
(C flex) → C compiler of unix

pass & phase
(we pass in many phases).

Bootstrapping

cross compiler → produce obj code for one M/C but run on other



Compiler is characterised by 3 lang
- source lang
- target lang
- lang in which compiler is written

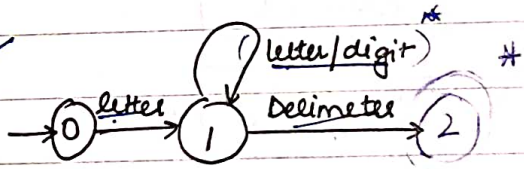
token: is a specialised unit

analyser: recognise-token

analysis: the token

UNIT-2

* Finite Automata or Transition diag. for Identifier:



letter/letter/digit*

A B C D 1 2 D (7) → identifier ∴ retract 1 position back.

I/P Buffering

Need of 'LA':
The purpose of splitting the analysis of the source prog into 2 phases, is to simplify the overall design of compiler.

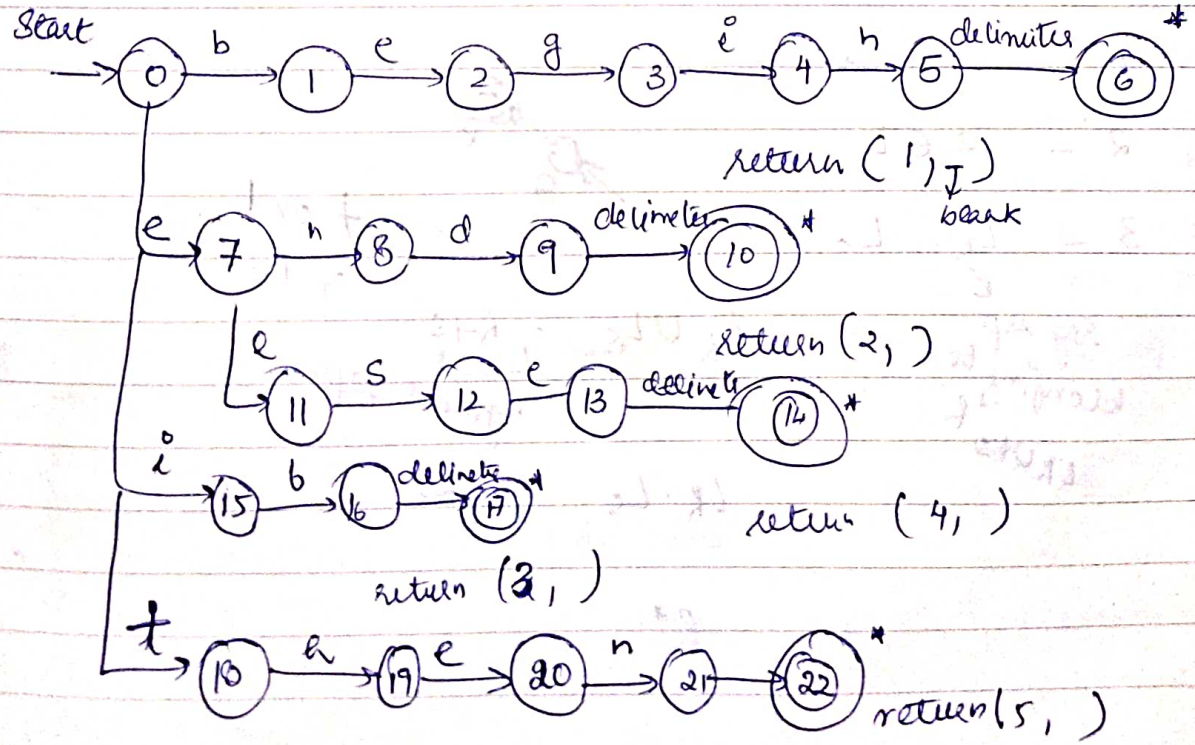
many tokens may hv to be examined b4 next token is determined ∴ I/P buffer is used.

Token		Code	Value
Keywords	begin	1	-
	end	2	-
	if	3	-
	else	4	-
	then	5	-
	identifier	6	Pointer to symbol table
	constants	7	"
Relops	<	8	1
	<=	8	2
	=	8	3
	>	8	4
	<>	8	5
	>=	8	6

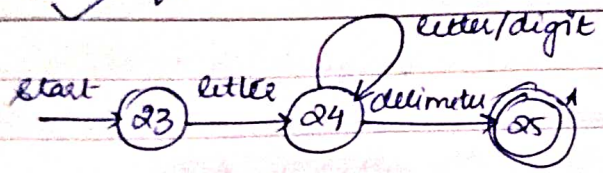
RETRACT()

to identify integer value independently

return (id. INSTALL())
(0-127)

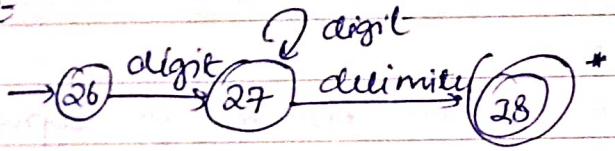


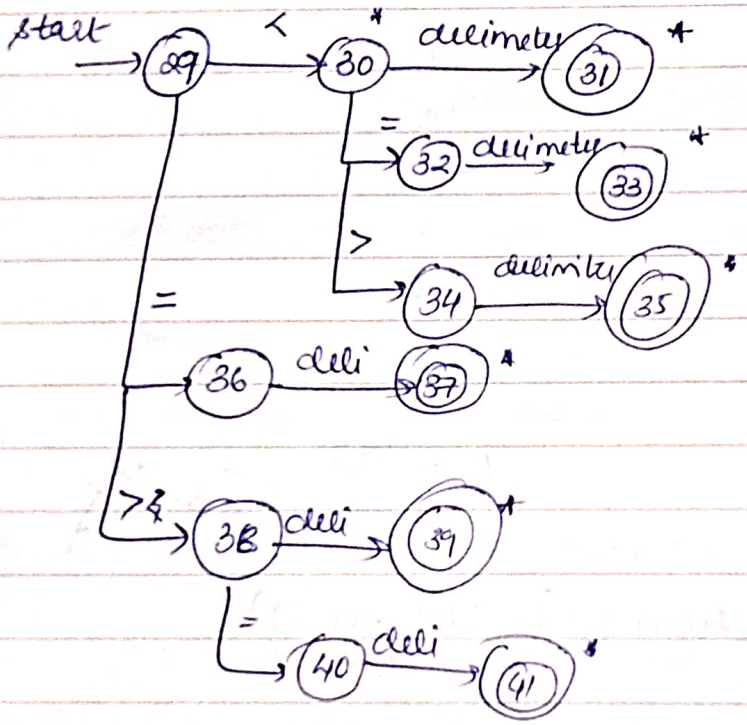
identifier



return (6, INSTALL())

constant





1 - $\in \rightarrow$

2 - $a \in L$

3 - $L_R \quad L_S$

+ or |
eg R|S

reg exp to belonging R

$L_R \cup L_S = \underbrace{R+S}_{\text{symbol reg.}}$

$L_R \cdot L_S = L_S$

R^*

keywords :- BEGIN | END | IF | THEN | ELSE

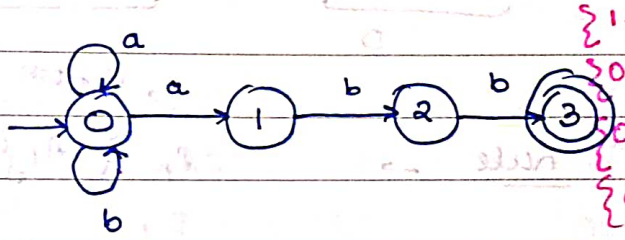
identifier :- LETTER (LETTER | DIGIT)*

- equi of NDFA & DFA
- Reg exp to DFA
- DFA to Reg. Exp.
- Min. of FA
- $\wedge$ moves

NDFA

	a	b
0	{1,0}	{0}
1	—	{2}
2		{3}
(3)	—	{3}

Q Construct DFA equi to NDFA



	a	b
{1,0}	{1,0}	{0,2}
{0,2}	{1,0}	{0,3}
{0,3}	{1,0}	{0}
{0}	{1,0}	{0}
{2}	—	{3}
{3}	—	—

Q construct minimized DFA

state	a	b
→ A	→ B	C
B	B	D
C	→ B	C
D	B	E
(E)	B	C

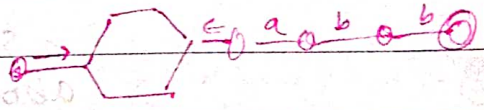
$\pi_0 = \{E\}\{A,B,C,D\}$

$\pi_1 = \{E\}\{A,C\}\{B,D\}$

$\pi_2 = \{E\}\{A,C\}\{B\}\{D\}$

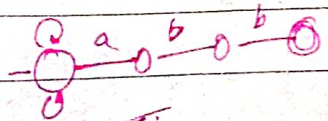
	a	b
{E}	B	E
{A,C}	B	C
{B}	B	D
{D}	B	E

Q construct NFA for



R.E

$R = (a|b)^* abb$



LEX

lexeme

Lang. used to specify lexical Analyser.

it consists of

(1) Auxiliary defina" → Distinct name

$D_1 = R_1$ → regular exp

$D_n = R_n$

eg: IDENTIFIER (D) = letter (letter | Digit)* (R)

(2) Transi" Rule → Pattern

$P_1 \{ A_1 \}$

$P_i \{ A_i \}$

eg. Begin (P) → { return(1, -) } (A)

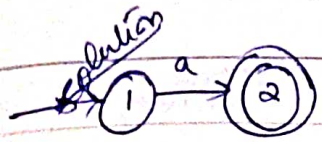
Q. Consider the lex source prog.

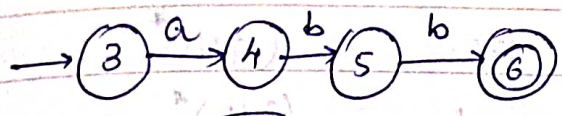
AUX Def.
(none)

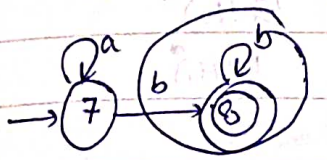
Trans" rule.

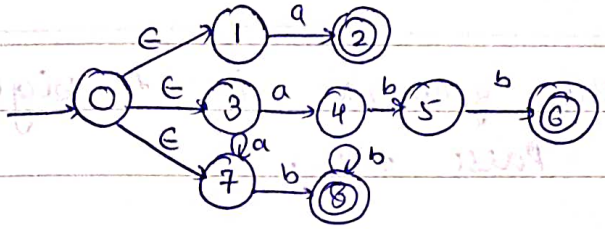
```
a {}
abb {}
a* b+ {}
```

/* Actions are omitted here */



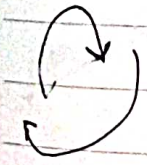






Present state	a	b	Token formed
0137	247	8	—
final 247	7	58	a
state 8	—	8	$a^* b^+$
7	7	8	—
58	—	68	$a^* b^+$
68	—	8	abb

Q Consider lex prog. given as.



AD

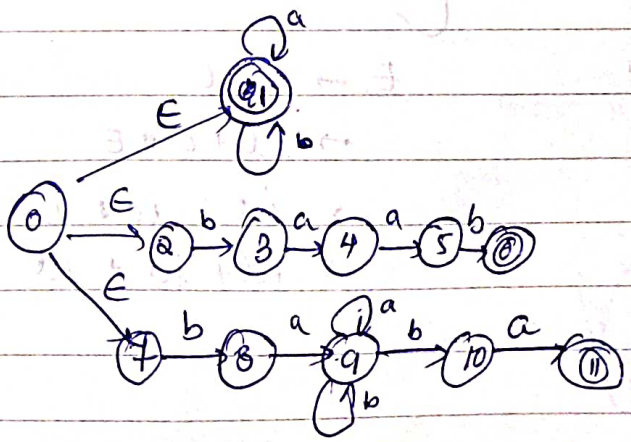
(none)

TR

$(a+b)^*$ {}

baab {}

ba $(a+b)^*$ ba {}



Present state	a	b	Token format
0127	1	138	$(a+b)^*$
138	149	1	$(a+b)^*$
149	159	1910	$(a+b)^*$
159	19	16910	"
1910	1911	1910	"

UNIT:3

Q Show that the grammar is Ambiguous, Also construct Parse tree.

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

Q What is CFG.

Q Find the LMD, RMD for and construct Parse tree for the given CFG

$S \rightarrow iCtS$

$S \rightarrow iC_t S_e S$

$S \rightarrow a$

$C \rightarrow b$

where i, t, e, S stands for if, then else

C & S stand for "Conditional" & "Statements".

A:

$E \rightarrow E+E$

$\rightarrow id+E*E$

$\rightarrow id+id*E$

$\rightarrow id+id*id$

$E \rightarrow E*E$

$\rightarrow E+E*E$

$\rightarrow id+id*id.$

Author: Created:

UNIT: 4

Parsing

- Top-down
- bottom-up.

start → end

- ~~Predictive~~ Predictive
- Recursive
- shift reduce Parsing. (consists of shifting of i/p symbols onto a stack until the right side of production appears on top)
- operator precedence Parsing.

book (148)

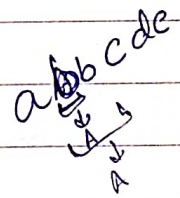
consider grammar

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$.

abbcde



aAbcde

aAcde

aAcBe

S.

Right most derivation.

RMD

LR

book (151)

bottom-up parsing → canonical reduction sequence is obtained

SHIFT REDUCE PARSING

Handle Pruning

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

Right Sentential form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E*E$	$E \rightarrow E*E$
$E+E$	$E+E$	$E \rightarrow E+E$
E		

Handle Pruning

Q consider the foll grammar for list structure.

$$\begin{cases} S \to a \mid \wedge \mid (T) \\ S \to T, S \mid S \end{cases}$$

(T) terminal non-terminal

right sent. form	handle	Reducing Prod
$(a_1,(a_2,a_3))$	a_3	$S \to a_3$
$(a_1,(a_2,S))$	a_2	$S \to a_2$
$(a_1,(S,S))$	a_1	$S \to a_1$
$(S,(S,S))$	S	$T \to S$
$(S,(T,S))$	T,S	$T \to T,S$
$(S,(T))$	(T)	$S \to (T)$
(S,S)	S	$T \to S$
(T,S)	T,S	$T \to T,S$
(T)	(T)	$S \to (T)$

Jisme Badda — Is Badda

a)

$S \to (T)$
$S \to (T,S)$
$S \to (S,(T))$ ✓
$S \to (S,(T,S))$
$S \to (S,(S,S))$
$S \to (a,(S,S))$
$S \to (a,(a,a))$

b)

$S \to (T)$
$S \to (T,S)$
$S \to (S,S)$ ✓
$S \to ((T),S)$
$S \to ((T,S),S)$
$S \to (((T,S,S),S)$
$S \to (((S,S,S),S)$
$S \to (((T),S,S),S)$
$S \to (((T,\varepsilon),S,S),S)$
$S \to (((S,\varepsilon),\varepsilon,S),S)$
$S \to (((a,a),\wedge,(T)),S)$
$S \to (((a,a),\wedge,(S)),S)$
$S \to (((a,a),\wedge,(a)),a)$

LMD

$(a_4, (a_2, a_3))$

$(S, (a_2, a_3))$

$(T, (a_2, a_3))$

$(T, (S, a_3))$

$(T, (T, a_3))$

$(T, (T, S))$

$(T, (T))$

(T, S)

(T)

S.

RMD

a_1	$S \to a_1$
S	$T \to S$
a_2	$S \to a_2$
S	$T \to S$
a_3	$S \to a_2$
T, S	$T \to T, S$
(T)	$S \to (T)$
T, S	$T \to T, S$
(T)	$S \to (T)$

Book (154)

OPERATOR PRECEDENCE PARSING

1- $a = b$

$\alpha a \beta b \gamma$

$\beta \to \epsilon$ or single NT

$S \to i\ C\ t\ S\ a$

$C \to b \quad i = t$

$S \to i t S$

$i = t$.

2- $<\cdot$

$\alpha a A \beta$

$A \to \gamma b \delta$

$\gamma \to \epsilon$ or Single NT

$a <\cdot b$

8. ⇒

$\alpha A b \beta$

$A \to \gamma a \delta$

$\delta \to \epsilon$ or single NT

$a \Rightarrow b$

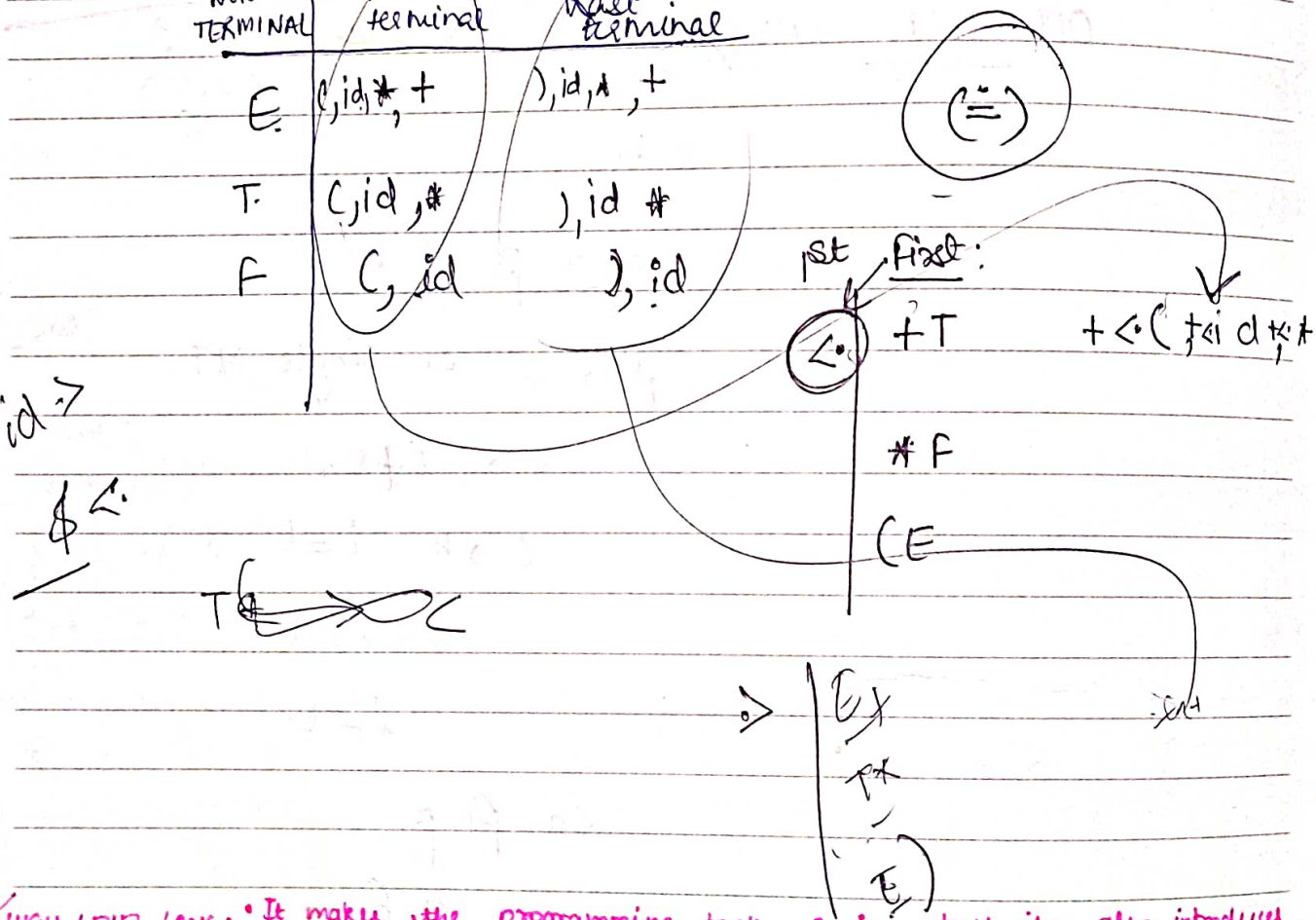
eg. $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

leading — trailing

NON TERMINAL	1st terminal	Last terminal
E	(, id, *, +	), id, *, +
T	(, id, *	), id, *
F	(, id	), id



✓ HIGH LEVEL LANG: • It makes the programming task easier, but it also introduces some prob.
ie There is a need to translate HLL into M/c L.
• It enables prog. to express algo. in more natural notations.

c

adv: Makes prog. simple
disadv: need to translate

Source prog → compiler → obj Prog.

obj prog i/p → object Prog → obj prog o/p

COMP" & EXECU"

COMPILER:

It is a Prog. that takes I/P as High level lang → low level lang.

eg. FORTRAN, COBOL etc → eg. Assembly or M/C lang.

Source lang → target lang (O/P)

Execution of a prog. is done in 2 steps:

- compilation of source prog. into target prog.
- loading resulting obj. prog. into M/M & executing.

INTERPRETER:

smaller than compilers. but slower execu"

Prog lang → simplified lang ≡ (Intermediate code) → directly executed by INTERPRETER.

- execution time is more (slower)

ASSEMBLER:

Source → target

Assembly lang → Mc lang.

PREPROCESSORS:

High level lang of I form → high level lang of other form.

TRANSLATORS:

* NEED OF TRANSLATOR *

① coz M/c lang is in terms of bits, registers etc.
② complex algorithm. is diff. to Program
③ M/c lang Prog is cryptic & impossible to modify in convenient manner.

SYMBOLIC ASSEMBLY LANGUAGE:

In this a prog uses mnemonic names for both operation codes & data addresses. eg. ADD X, Y.

comp cannot execute a prog in assembly lang thus, it is translated into M/clang with the help of assembler.

MACROS: (Text replacement capability)

Translate into a sequence of assembly lang stmnts.

eg. MACRO DEFINATION

```
MACRO
ADD2 (X,Y)
LOAD X
ADD Y
STORE X
END MACRO
```

gives ADD2 name to macro

formal parameters.

now after defining we can use the macro ADD2 anywhere in Prog without defining its functionality.

eg. ADD2 (A,B) → actual parameters.

which will be translated into →

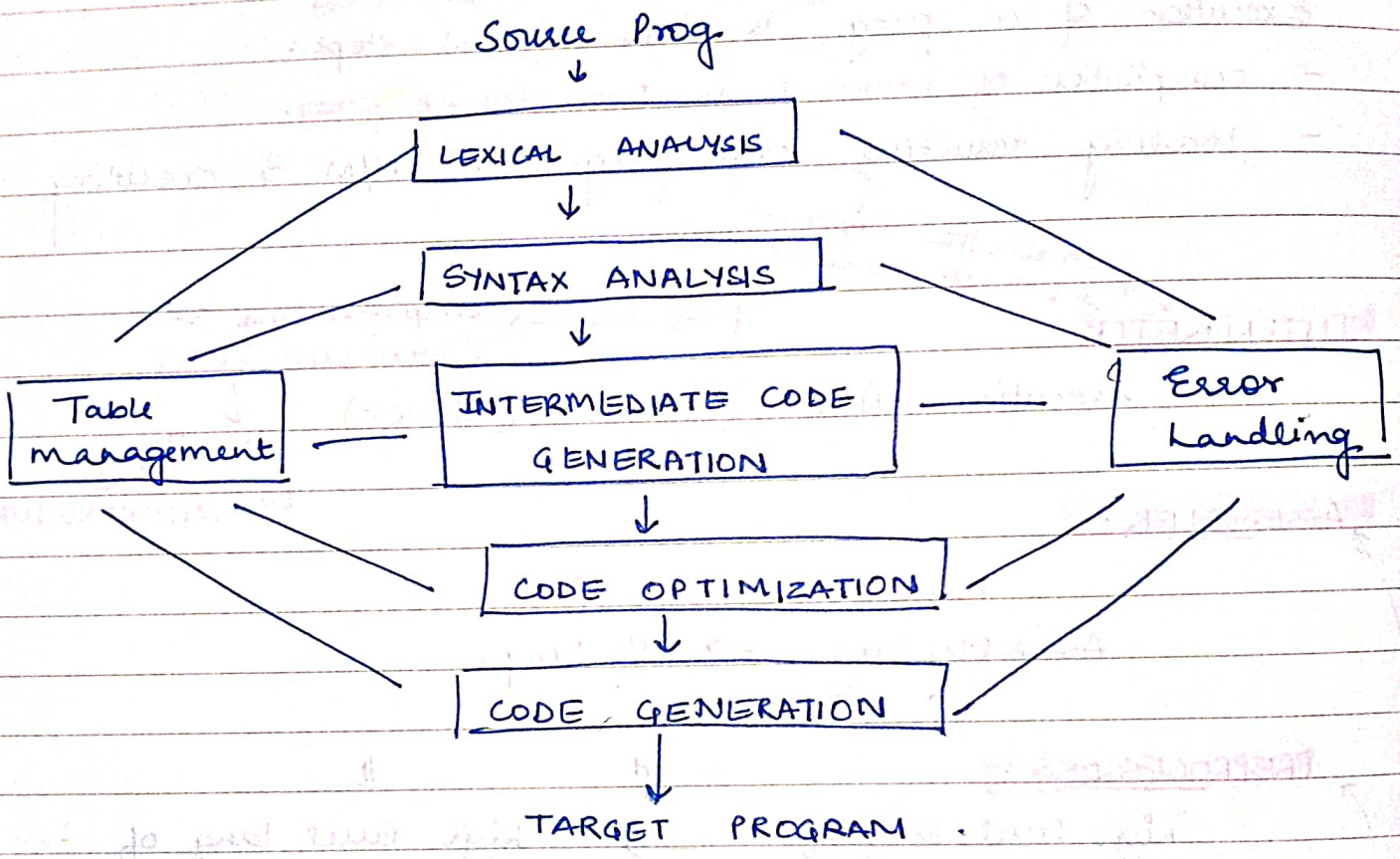
```
LOAD B
ADD A
STORE B
```

source prog → target prog.

It is a very complex process.

STRUCTURE OF A COMPILER:

compilation process is partitioned into a series of subprocesses called PHASES. (It is a logically cohesive operation, it takes 1 i/p of source prog & produces o/p.)



① LEXICAL ANALYSER:

Seperates characters of source lang into groups that logically belong bcg., these groups are called TOKENS.

The usual tokens are keywords eg. DO or IF.

'LA' is a stream of tokens, which is passed to next phase.

Token (2 types) → { specific string eg. if / semicolon } ; { class of string eg. constant or identifier }

Token as a pair has 2 parts
↳ token type
↳ token value.

Finding token:

To find the next token the 'LA' examines successive characters in source prog.

② SYNTAX ANALYSIS:

It identifies which part of i/p stream should be group
eg. A/B *C
Interpretations:
A ÷ B & then *C
→ or B *C & then ÷ A
after this a parse tree is generated.

It groups tog. tokens into syntatic structures
eg. 3 tokens representing A+B might be grouped tog. into a data class called EXPRESSION. that may be combined to form STATEMENTS

③ ICG:

It uses structure produced by the 'SA' to create a Assembly code. stream of simple instructions.

The diff. b/w IC & AC is that in IC we need not specify the registers.

3-Address code:

It is a popular type of intermediate lang.
A typical '3AC' statement is -

operands ↑ ↑ operator

A := B op C

④ OPTIMIZATION:

(*) loop optimiza': move a computation that produces the same result each time around the loop to a point in prog. just b4 loop is entered.

It is an optional phase designed to improve the intermediate code so that the ultimate obj prog. runs faster or takes less space.

⑤ CODE GENERATION:

It converts intermediate code into a sequence of M/c instructions.

& selects registers in which computation will be done.

↓

⑥ BOOKKEEPING: portion of compiler keeps track of names of used by the prog. & records essential info.
eg. types of integers, variables. etc.

⑦ ERROR HANDLING: it is invoked when a flaw in the source prog. is detected.

PASS: reads o/p of previous pass → make transformation → write o/p into intermediate file.

COMPILER WRITING TOOLS:
eg. Scanner, parser generator etc.

Transition diag + finite automata
ways of designing token recognizers.

Regular exp: suitable for describing tokens.
string: seq of symbols.
lang: set of strings.

---

use of cross compiler: They are usually found in use for generating compiles for embedded system or multiple platforms.

It is a tool used for a platform where it is inconvenient to compile.

It is used in →
- Embedded comp
- compiling for multiple M/c
- compiling on same form
- Bootstrapping.

BOOTSTRAPPING :

A compiler is characterised by 3 lang:

- Source lang
- Obj lang
- Lang. in which it is written.

These all languages are quite different.

CROSS COMPILERS : They run on one M/c & produce obj. code for another M/c.

Suppose, we have new lang 'L', which we want to make available on M/c 'A' & 'B'.

We write compiler C_S^{LA} in simple lang 'S'. It runs through C_A^{SA}. Produces obj code for A.

$$C_S^{LA} \longrightarrow \boxed{C_A^{SA}} \longrightarrow C_A^{LA}$$

Bootstrapping a compiler

$$C_L^{LB} \rightarrow \boxed{C_A^{LA}} \rightarrow C_A^{LB}$$

$$C_L^{LB} \rightarrow \boxed{C_A^{LA}} \rightarrow C_B^{LB}$$

Bootstrapping a compiler to 2 M/c

COMPILER CONSTRUCTION TOOLS :

A no. of tools have been developed specifically to help construct compilers.

They range from — Scanner generator
parser generator
to.
complex systems.

The i/p specification of such tools contain:

- lexical & Syntatic Struc of source lang.
- discription about o/p.
- description of target M/c

• Parser generator : These produce syntax analyzers, ~~Parser analyzers~~, normally from the i/p is based on a context free grammar.
They utilize powerful algorithms, that are too complex to be carried out by hand.

• Scanner generator : They automatically generate, lexical Analyzer normally, from the specification based on regular expression.

• Syntax - directed translation engine: They produce collection of routines that walk the parse tree generating intermediate code

• Automatic code generators : They take collection of rules for each operation of
Intermediate lang → M/c lang.

PASS :

It is an implementation of a compiler.

It reads source prog → makes transformations specified
↓
writes out the output.

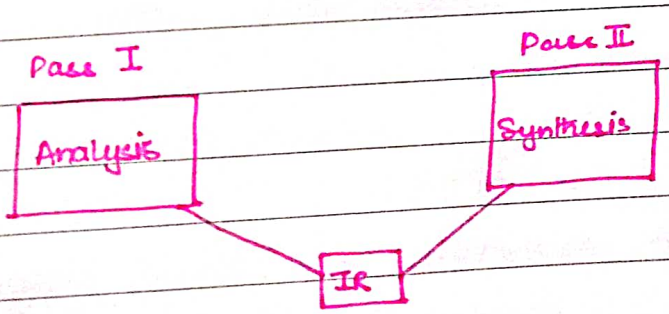
A no. of phases are grouped into a pass.

The environment in which the compiler operates can effect the no. of passes.

There are 2 passes:

- Single Pass
- Multipass.

A multipass compiler uses less space than single pass but.
MPC is slower than SPC as each file reads & writes an intermediate file.



MPC is better than SPC.

A one-pass assembler, generate a skeleton of the GOTO M/c instruction the 1st time it sees it &

↓

then append M/c address.

eg. unix compiler

A two-pass assembler uses its 1st pass to enter into its symbol table a list of all identifiers together with M/c address.

↓

2nd pass replaces mnemonic operation code eg GOTO by their M/c lang equivalent.

eg. c compiler

Token

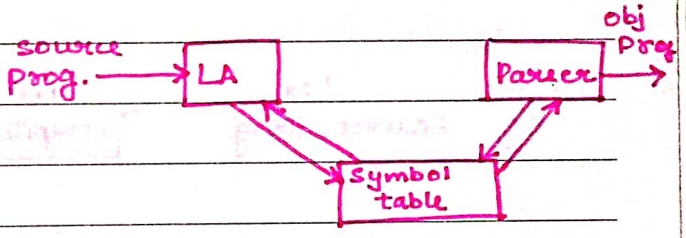
UNIT2 :

LEXICAL ANALYSER:

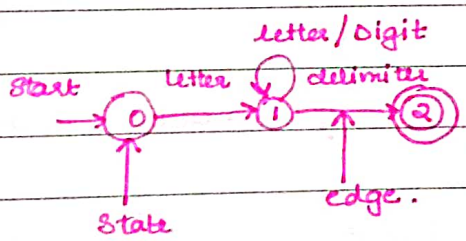
It could be a seperate pass, placing its O/P on an intermediate file from which parser may then take its I/P.

It acts as a subroutine, which is called by the parser whenever it needs a token.

The purpose of splitting the analysis of the source prog. into two phases, ie lexical analysis & syntax analysis is to simplify the overall design of the compiler.

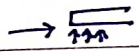


Transition diagram



letter (letter | digit)*

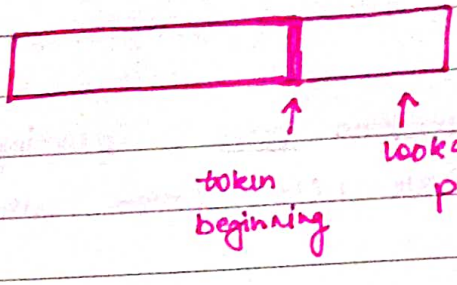
Input buffering :



The lexical analyser scans the characters of source prog. one at a time to discover tokens.

Often, characters beyond the next token may have to be examined before the next token can be determined.

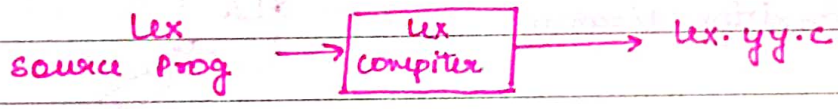
For this reason, it is desirable for LA to read its I/P from an I/P buffer.

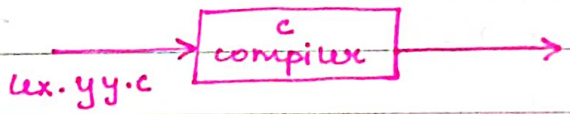


LEX

It is a tool, that has been widely used to specify lexical analysers.

lex like specifications can be used even if lex compiler is not available.





It consists of:

1) Declarations

2) Auxiliary defination.: $D_1 = R_1$

$D_n = R_n$

eg. Identifier → letter (letter | digit)*

D R

3) Transition Rules:

$P_1 \{A_1\}$

⋮

$P_n \{A_n\}$

eg. Begin → { return (1, _) }

P A

Lexem: It is I/P string or grammar.

NDFA

RE → Recognizer

A better way to convert a regular expression to a recognizer is to construct a generalised transition diagram.

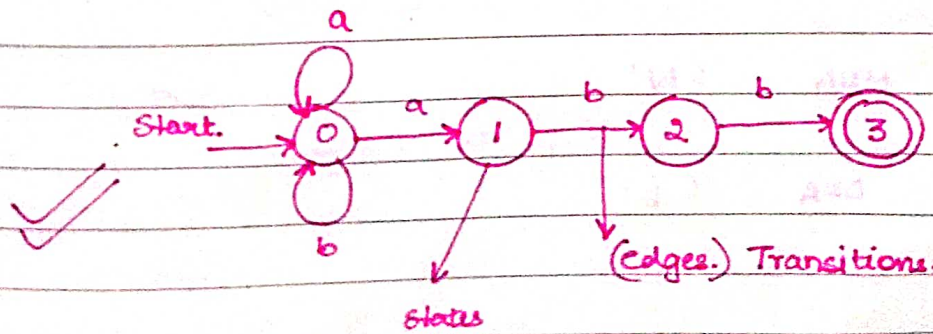
This diagram is called a NDFA.

A NDFA recognising lang (a|b)* abb

ie.

State	I/P symbol a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}

transition table.



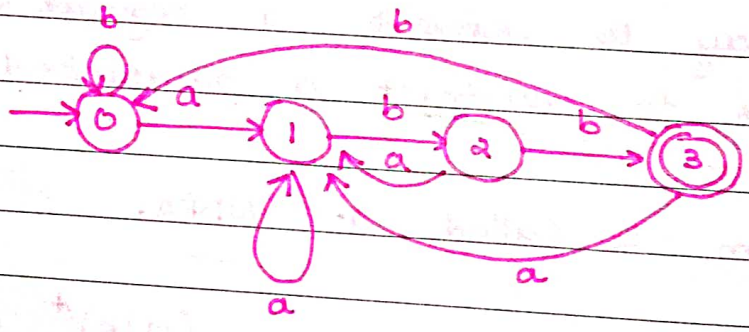
The lang defined by NFA is set of i/p strings it accepts.

DFA.

In deterministic FA:

- There are no transitions on i/p ε
- for each state (s) and i/p symbol (a), there is at most one edge labled a leaving s.

DFA accepting $(a|b)^*$ abb



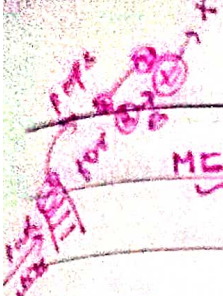
DFA is easier to simulate than NFA.

States of DFA represent subsets of all sets of NFA.

NDFA → DFA.

I/P: An NFA 'N'

O/P: A DFA 'D'



METHOD:

ε-CLOSURE

begin

push all states onto STACK

ε-CLOSURE (T) := T ;

while STACK not empty

do

begin

pop 's' → topelement from stack

for each

edge from s to t labeled ε

do

if t is not in ε-CLOSURE (T)

do

begin

add t to ε-CLOSURE(T);

push t onto STACK

end

end

end.

DFA is a 5- tuple $(Q, \Sigma, \delta, q_0, F)$

Q → finite set of states

Σ → " " " i/p symbols

δ → transition func^n

q_0 → a start state

F → a set of final states.

DFA on i/p string $S_1 \text{ --- } S_n$

Q_1 → initial state

If Q is accepting then return "yes"

For $j = 1$ to n do

Begin

Q_j = goto $[Q, S_j]$

If Q is accepting then return "yes"

End.

Return "NO"

End.

$R = Q^+ RP$

$R = QP^+$

eg. BA → bA
CA → AB etc.

Type 0 → no restrictions
$\phi A \psi \rightarrow \phi \alpha \psi$
C (context sensitive) Type 1 →
(context free) Type 2 → $A \rightarrow \alpha$ LHS has no right & left context
eg. $S \rightarrow Aa$
$A \rightarrow a$
$A \rightarrow \Lambda$

UNIT-3

[Regular grammar] Type 3 → $A \rightarrow a, A \rightarrow aB$

CFG:

For the syntatic specification of a prog.ng lang we use a notation called a CONTEXT-FREE GRAMMAR.

- It gives a precise, yet easy to understand, syntatic specification.

- An efficient parser can be constructed automatically.

A grammar imparts a structure to a prog. that is useful for its ~~representation~~
↳ Translation into obj code.
↳ detection of errors.

A grammar involves:

- Terminals

they are basic symbols in a string.

They are → a, b, c (i) +, - (i), (i) 1, 2 (i) id etc.

(i), A/S

- Nonterminals

Special symbols that denote sets of string.

They are → expression (i) E, A (i) S etc.

- Start Symbol

- productions

Types

CFG

CNF GNF

CNF
GNF

Chomsky Normal form: A 'CFG' G is in CNF if every production is of form:

$$\begin{cases} A \to a & \text{or} \\ A \to BC & \text{and} \\ S \to \Lambda \end{cases}$$

Theorem:

1) Elimination of NULL and UNIT production.
   $A \to \Lambda$ $A \to B$
2) Elimination of terminals on RHS.
3) Restrict no. of variables on RHS.

eg. $S \to ABC$

$S_1 \to AB$

$\therefore S \to S_1C$

Greibach Normal form: A 'CFG' is in GNF if every produc is of form:

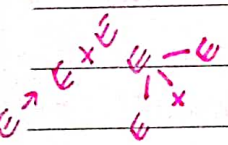
$$\begin{cases} S \to \Lambda \\ S_1 \to aBCD \cdots \\ A \to a \end{cases}$$

Theorem:

1) Remove NULL and UNIT production.
2) Renaming Variables.
3) Substituting by looking at the index.
   i.e if LHS index > RHS
4) Apply LEMMA THEOREM.

Regular expr" & CFG:

- For 'RE' lexical rules are simple & we don't need powerful notations.
- 'RE' notation is easier to understand
- It is easier to construct efficient recognisers from 'RE' than 'CFG'.
- Seperating syntatic structure into { lexical part / non lexical part } provides a managable - sized components.



PARSE TREE

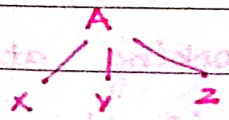
It is a graphical representation for derivations.

This representation is called a "Parse tree"

It makes hierarchical syntatic structure of sentences.

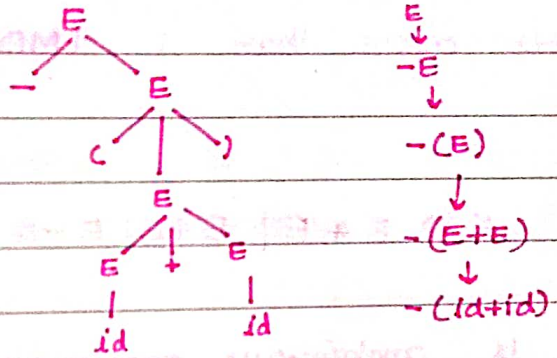
Each interior node of parse tree is labeled by some nonterminal A. & nodes are labeled from left to right by the symbols.

eg. $A \rightarrow XYZ$



The leaves of the tree are labeled by non-terminals or terminals, read from left to right.

eg. Parse tree for -(id + id)



E
↓
-E
↓
-(E)
↓
-(E+E)
↓
-(id+id)

eg.

id + id * id

has 2 LMD

E → E+E

E → id + E

E → id + E * E

E → id + id * E

E → id + id * id

E → E * E

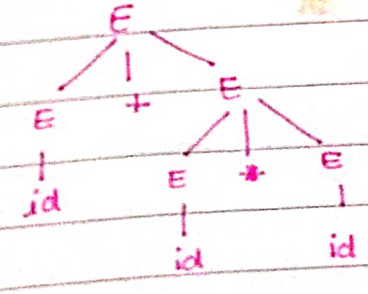
E → E + E * E

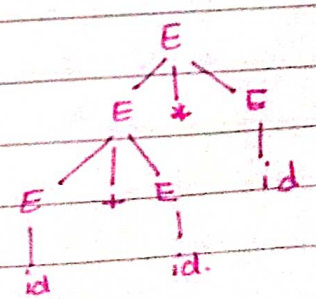
E → id + E * E

E → id + id * E

E → id + id * id.

Their parse tree:





Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be "ambiguous."

It produces more than 1 LMD or RMD. for same sentence.

eg. $E \rightarrow E+E \mid E+E \mid E-E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$

This is ambiguous grammar. but we can

↓

disambiguate

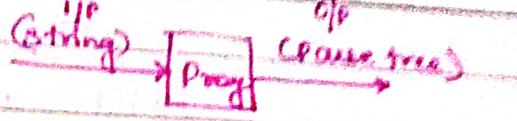
all these grammars by specifying

- Associativity
- precedence of operators.

Ambiguous grammar fails to be LR.

UNIT-4

PARSERS:

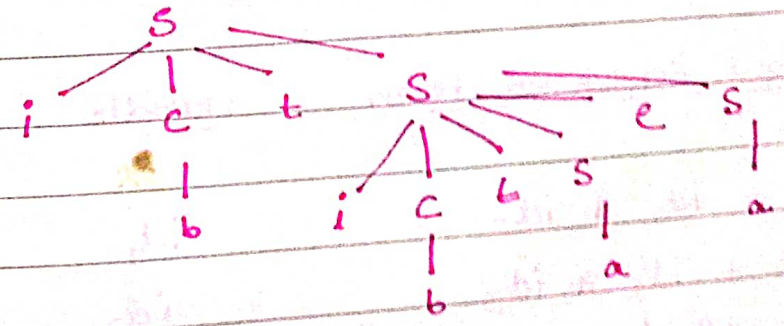


A parser for a grammar is a prog that takes I/P as a string & produces O/P as a parse tree if I/P is a sentence or leaves an error msg.

- **Bottom up** — build parse tree from bottom to top.
  - shift red
  - operator precedence
- **Top-down** — It works with root & then moves to the leaves.
  - Predictive
  - Recursive

eg.
$S \rightarrow iCtS$
$S \rightarrow iCtSeS$
$S \rightarrow a$
$C \rightarrow b$

$\therefore$
$S \Rightarrow iCtS$
$\Rightarrow iCtiCtSeS$
$\Rightarrow ibtibtaeS$
$\Rightarrow ibtibtaea$



SHIFT REDUCE PARSING

This parsing method attempts to construct a parse tree for an i/p string beginning at the leaves and working up towards root.

- Each replacement of Right Side product^{n} by left side is called "Reduction".

- A "handle" → it is the element ie reduced. of right sentential form γ is a production $A \to \beta$ & a position of γ ↓ where string β may be found & replaced by A

- If grammar in unambiguous then it has only 1 handle

- "Handle Pruning" : A RMD in reverse is called 'canonical reduction sequence' & is obtained by HP.

Q. i/p → $id_1 + id_2 * id_3$
reduces to → E

Right Sentential Form	Handle	Reducing product^{n}
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
→ $E + E * id_3$	id_3	$E \to id$
$E + E * E$ ←	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

There are 2 problems:

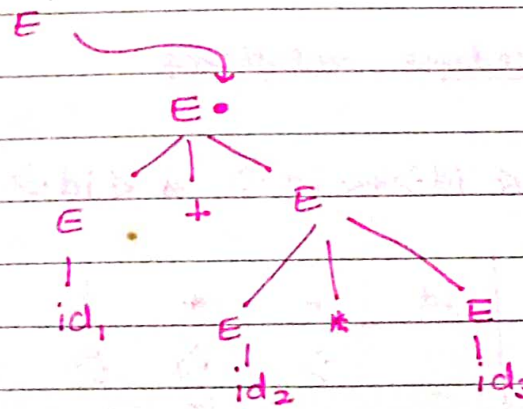
① To locate handle in RSF

② what production to choose if more than 1 production.

∴ we use 'STACK'

stack	i/p	Action
$	$id_1 + id_2 * id_3$ $	shift
$ id_1	$+ id_2 * id_3$ $	reduce by $E \to id$
$ E	$+ id_2 * id_3$ $	shift
$ E +	$id_2 * id_3$ $	shift
$ E + id_2	$* id_3$ $	reduce by $E \to id$
$ E + E	$* id_3$ $	shift
$ E + E *	id_3 $	shift
$ E + E * id_3	$	reduce by $E \to id$
$ E + E * E	$	reduce by $E \to E * E$
$ E + E	$	reduce by $E \to E + E$
$ E	$	accept

Parse tree



OPERATOR PRECEDENCE PARSING

* These grammars have property:

- no production Right side is $\in$ ($A \to \in$, $A \to AA$)
- no 2 adjacent non-terminals

* disadv of OP :

① hard to handle tokens eg (-) has 2 precedences.

② Only small class of grammars can be parsed.

* 3 dispint 'precedence relations':

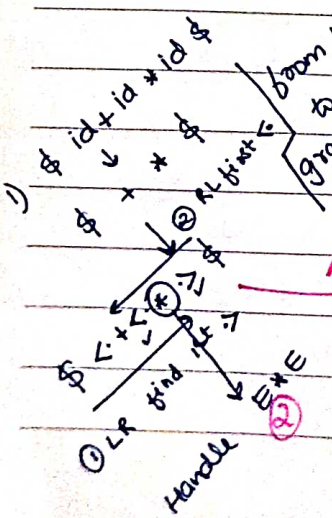
$a \lessdot b$ a yields precedence to B

$a \gtrdot b$ a takes precedence over B

$a \doteq b$ same precedence.

* Operator precedence relations

eg. ① $\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$



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

④ $+ \lessdot *$

⑤ $* \gtrdot +$

⑥ $+ \gtrdot +$, $* \gtrdot *$ left associative

② $\$ \lessdot * \gtrdot \$$ (1st priority)

③ $\$ \lessdot + \gtrdot \$$

OPR (Associativity + Precedence)

* Asso. + Prece. rules :

① If Θ_1 has higher precedence than Θ_2 make

$\Theta_1 \cdot> \Theta_2$ & $\Theta_2 <\cdot \Theta_1$

eg. $(* \cdot> +)$ & $(+ <\cdot *)$

② If Θ_1 & Θ_2 have equal precedence. make

$\Theta_1 \cdot> \Theta_2$ & $\Theta_2 \cdot> \Theta_1$] left associative

or

$\Theta_1 <\cdot \Theta_2$ & $\Theta_2 <\cdot \Theta_1$] right associative.

③ $\Theta <\cdot id$, $id \cdot> \Theta$

$\Theta <\cdot ($, $(<\cdot \Theta$

$) \cdot> \Theta$, $\Theta \cdot>)$

$\Theta \cdot> \$$, $\$ <\cdot \Theta$

eg. $E \rightarrow E+E \mid E-E \mid (E) \mid \cancel{EE} \mid id.$

$\$ <\cdot E \cdot> + <\cdot E \cdot> \$$

$\$ <\cdot id \cdot> + <\cdot id \cdot> - <\cdot id \cdot> \$$ (E) E+E-E

(id + id)

(<+>)
(+)
$ E R $
(-)

	+	-	id	(	)	$
+	·>	·>	<·	<·	·>	·>
-	·>	·>	<·	<·	·>	·>
id	·>	·>			·>	·>
(	<·	<·	<·	<·	=	
)	·>	·>			·>	·>
$	<·	<·	<·	<·		

[(=) + ·> +, - ·> -]
[$ <· (,) ·> $]
[$ <· id, id ·> $]
[(<· (,) ·>)]
[(<· id, id ·>)]

←) ·> +

← (<· +

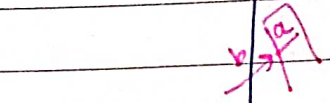
id ↑ $ → lowest.

OP. PRE. PARC. ALGO

repeat forever
if only $ on stack & $ on i/p then
accept
and break
else
begin
let (a) be topmost terminal on stack
& (b) be current i/p
if a <· b or a ≐ b then shift (b)
onto stack.
else if
a ·> b then
repeat pop
until top stack symbol
is related by <·
to terminal popped.
else
call error corrector routine
end

OPERATOR PRECEDENCE GRAMMAR :

① $a \doteq b$



if in $\alpha a \beta b \gamma$

β ↓

$\in$ or single non-terminal

② $a <\cdot b$

if in $\alpha a A \beta$ $\quad A \Rightarrow \gamma b \delta$

$\therefore$ $\quad$ γ ↓

$\in$ or single nonterminal

③ $a \cdot> b$

if in $\alpha A b \beta$ $\quad A \Rightarrow \gamma a \delta$

$\therefore$ $\quad$ δ ↓

$\in$ or single nonterm

LEADING :

I/P : A CFG & G'

O/P : Boolean Array L [A, a]
in which A & a is true
if & only if a is in LEADING [A].

Method :

INSTALL (A, a) ⟶ It sets L[A,a] = true
if it was not true
↓
Place (A, a) on STACK

```
procedure INSTALL (A, a);
    if not L[A,a] then
        begin
            L[A,a] := true;
            push (A,a) onto STACK
    end.
```

Main Procedure:

begin.

1) for each non terminal A & terminal a
   do
   L(A,a) := False;

2) For each produc" of form
   $A \rightarrow a\alpha$
   or $A \rightarrow B a\alpha$ do
   INSTALL (A,a);

while STACK not empty do
begin

3) Pop top pair (B,a) from STACK
4) for each produc" of form
   $A \rightarrow B\alpha$ do
   INSTALL (A,a)

end
end.

→ Trailing is Analogous to this

COMPUTING OPERATOR PRECEDⁿ RELAⁿ:

I/P: operator grammar 'G'

O/P: The relations $<\cdot$, $\doteq$, $\cdot>$ for G.

METHOD: 1) Compute LEADING (A)
& TRAILING (A)

2) Set $ $<\cdot$ a for all a in LEADING(S)
& b $\cdot>$ $ for all b in TRAILING(S)

for
$A \rightarrow X_1 X_2 \cdots X_n$ do
for i := 1 to n-1 do

begin

if X_i & X_{i+1} are terminals then set $X_i \doteq X_{i+1}$

if X_i is terminal & X_{i+1} is non-terminal
then set for all Leading (X_{i+1})
$X_i <\cdot a$

if X_i is non-terminal & X_{i+1} is terminal
then set for all trailing (X_i)
$a \cdot> X_{i+1}$

End.

8 $E \to E+T \mid T$

$T \to T * F \mid F$

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

Nonter.	Leading	Trailing
E	+, *, (, id	+, *,), id
T	*, (, id	*,), id
F	(, id	), id.

for E (+) & T → (*) & F → (() & (id))

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	
)	⋗	⋗		⋗		⋗
id	⋗	⋗		⋗		⋗
$	⋖	⋖	⋖		⋖	

✓

① (≐)

② FIRST: (⋖) 1) +T

ie

+ ⋖ * | + ⋖ (| + ⋖ id

2) * F

* ⋖ (| * ⋖ id

3) (E

ie

(⋖ id , (⋖ * , (⋖ + & (⋖ (

⑧ LAST:

⑦

1) E+

* ⋗ + | + ⋗ + |) ⋗ + | id ⋗ +

2) T*

* ⋗ * |) ⋗ * | id ⋗ *

3) E)

* ⋗) | + ⋗) |) ⋗) | id ⋗)

we add.

* ⋗ $
+ ⋗ $
) ⋗ $
id ⋗ $

$ ⋖ *
$ ⋖ +
$ ⋖ (
$ ⋖ id

(≐)

UNIT: 7

SYMBOL TABLE

* A symbol table is merely a table with 2 fields, ie.
  - Name
  - information

* We need to be able to:

- determine whether a given name is in table.
- add new name
- access info. associated with given name
- add new info for a name.
- delete a name.

* The info. collected about a name includes:

- Type of characters (eg. integer)
- form (eg. a structure)
- other attributes.
- offset describing position in storage.

* Each time a name is encountered, the symbol table is searched to see whether that name has been seen previously.

If name is new, it is entered into symbol table.
Info. is entered into table during - lexical
- syntactic analysis.

① Info in symbol table is:
- used during several phases of compilation
- detect error & correct it.

② A linear list → simple but slow
A hash table → fast but more complex.
Tree → intermediate performance.

③ SYMBOL TABLE RECORDS :

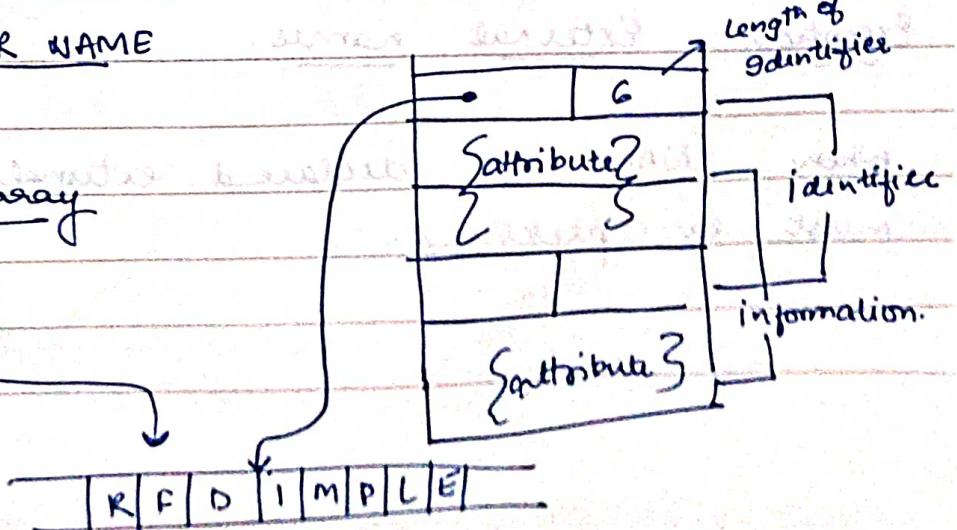
The simplest way to implement a symbol table is as linear array of RECORDS.
↓
It consists of known no. of words of memory.

RECORD

identifier information.

STORING IDENTIFIER NAME

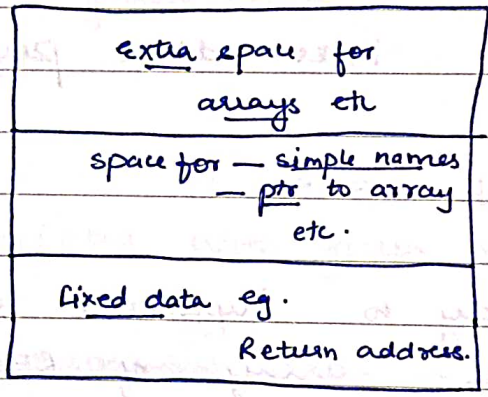
In seperate array



INDIRECTION : It can save space in representing identifiers.

The most sig. adv of this is when we have a type of info that is applicable to only a minority of entries.

⑧ A sensible way to allocate space is :
"ACTIVATION RECORD"



REUSING SYMBOL TABLE SPACE :

The identifier used by programmer to denote a particular name in symbol table must be preserved.

It is essential as all uses of the identifier can be associated with the same symbol-table entry & hence same name.

Exception : External names.

When NAME is declared extend to Prog. its IDENTIFIER must be preserved.

A compiler when finds a name ie external,

↓

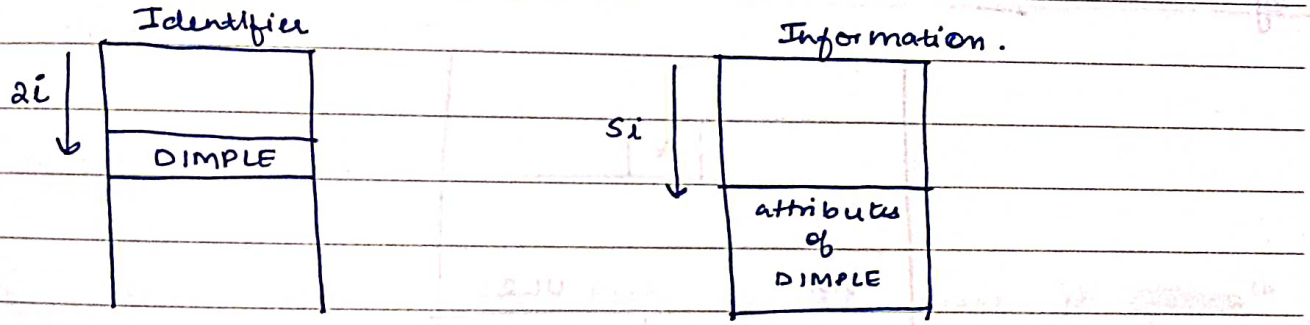
makes a copy of NAME in another small array

↓

Then POINTER to the identifier is made to point the new copy of NAME.

→ we can reclaim the space used to store identifiers

eg. TWO-ARRAY SYMBOL TABLE



DATA STRUCTURES :

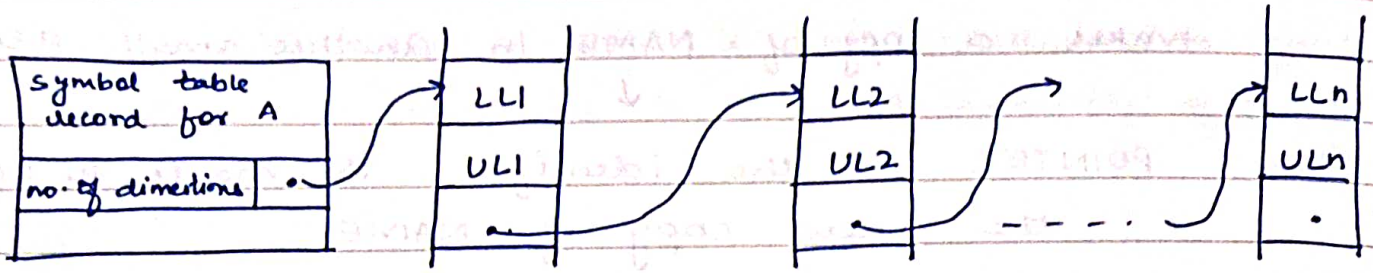
① ARRAYS: Data stru. of same kinds.

If a language places a limit on no. of dimentions then all subscript info. is placed in symbol table.

If an expression is constant its value can be stored in symbol table.

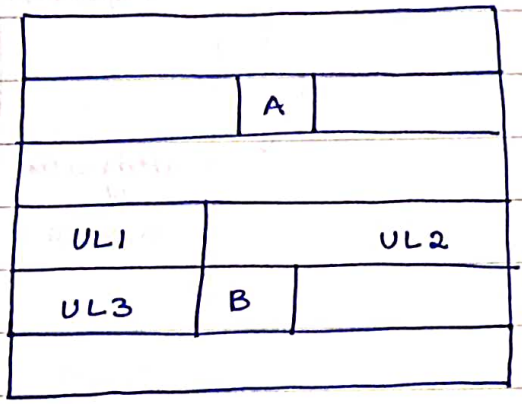
We may store,
- no. of dimentions
- pointer to linked list
- element holding upper limit & lower limit.

as in,



no. limitation of dimention.

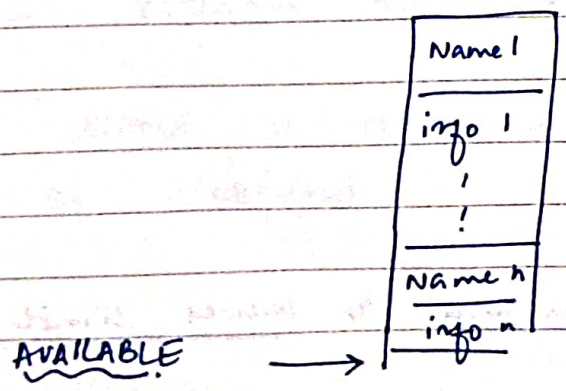
eg. FORTRAN limits to 3 dimentions.



Symbol table record for FORTRAN

② LISTS : They are simplest & easiest to implement data stru. for symbol table.

we use single array or a group of arrays to store names & associated info.



To retrieve info about a name :

we search from beginning of array till position marked by AVAILABLE.
↓
(indicate beginning of empty portion of array)

when name is located, its associated info. can be found

but, if we reach AVAILABLE WITHOUT FINDING THE NAME we have a FAULT

To insert a name :

we scan down the list to be sure it is not already there.

if it is there then we hv a fault of → multiply defined name

If not, then
↓
we store new name immediately after AVAILABLE & UPDATE the pointer.

To insert n names & m enquires

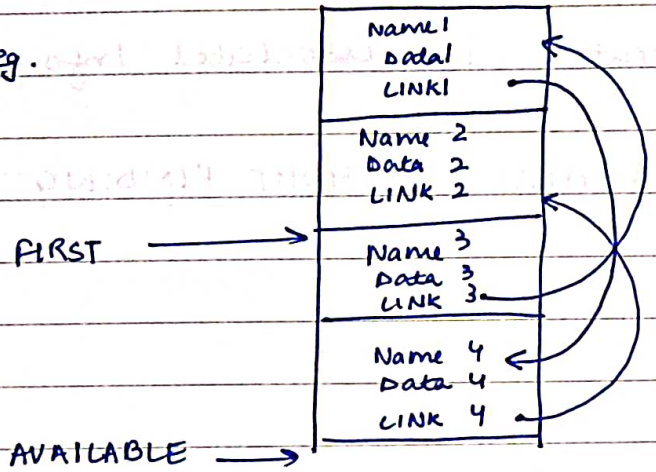
Total work = $O(n+m)$
↓
constant

self organised list

It saves fraction of time spend in searching the symbol table.

There is a LINK field in each record.

eg.



Order is 3, 1, 4, 2.

(8) SEARCH TREES :

It is a more effecient approach to symbol table organisation.

It has 2 link fields :

LEFT RIGHT

to each record.

We use these fields to link the records in Search tree.

If names are encountered randomly → The avg. length of path will be proportional to $\log n$

Av length $\propto \log n$

where n → no. of names.

Since each search follows 1 path from root

∴

TIME for n names & m inquiries is

$\propto$

$(n+m) \log n$

(3) Take P → pointer , ∴

(1) while P ≠ NULL do

(2) if NAME = NAME (P)

∴ "Name found"

(3) else if NAME < NAME (P)

then

P := LEFT (P)

(4) else if

NAME (P) < NAME

then

P := RIGHT (P)

(4) HASH TABLES : It is a superior method.

It gives us capability of performing

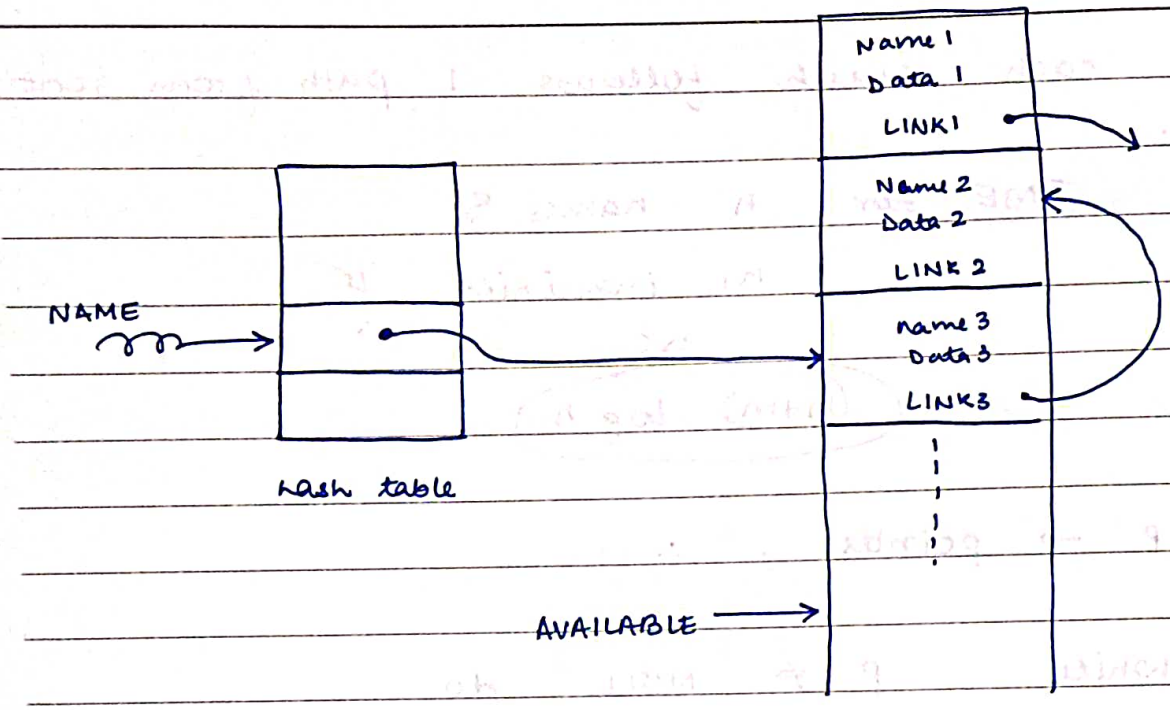
m → accesses on

n → names

in

Time $\propto \frac{n(n+m)}{k}$

k → constant.



storage table.

It has 2 tables — Hash, Storage

Hash table consists of K words

0 ↓ --- K-1

they are ptr into storage table.

easy to work with

(*) we apply Name to Hash table

& apply hash function h(NAME)

it is an integer between 0-K-1

SEARCH

To inquire about NAME, we compute h(NAME) & search list

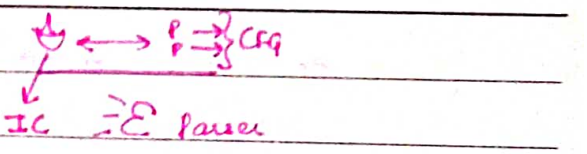
(*) create a record in the available space in storage table

ENTER

link record to beginning of h(NAME) list.

UNIT-6

SYNTAX DIRECTED TRANSLATION :



It allows subroutines to be attached to a productio of a context-free grammar.

These subroutines generate intermediate code when called by a parser.

They are useful beco2 it enables the compiler designer to express the generation of IC directly in terms of syntatic structure.

Semantic action:

A syntax directed transla^n scheme is merely a CFG in which an OUTPUT ACTION is associated with each produc^n.

OUTPUT ACTION may involve:

- computation of values.
- generation of IC.
- Printing errors etc.

Produc^n & Semantic action

$E \rightarrow E^{(1)} + E^{(2)} \quad \{ E.VAL := E^{(1)}.VAL + E^{(2)}.VAL \}$

value associated with grammar symbol is TRANSLATION.

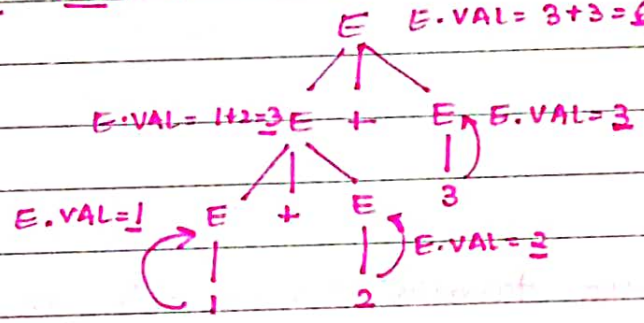
eg. $A \rightarrow XYZ \{ Y.VAL := 2 * A.VAL \}$

In this translation of non-terminal on right side is defined in terms of translation of non-terminal on left.

This is INHERITED TRANSLATION.

$E \cdot VAL \rightarrow$ "is integer valued translation"

values of translations are determined by constructing a PARSE TREE.



Parse tree for 1+2+3

eg. $E \cdot VAL := E^{(1)} \cdot VAL + E^{(2)} \cdot VAL$

IMPLEMENTATION:

It is a convenient discription of what we would like done.

The O/P defined is independent of kind of Parser used.

It is easy to modify.

eg. Specify 'Desk Calculator' for i/p 23 * 5 + 4 $
& o/p 119

The productions :

$S \rightarrow E \$$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow I$

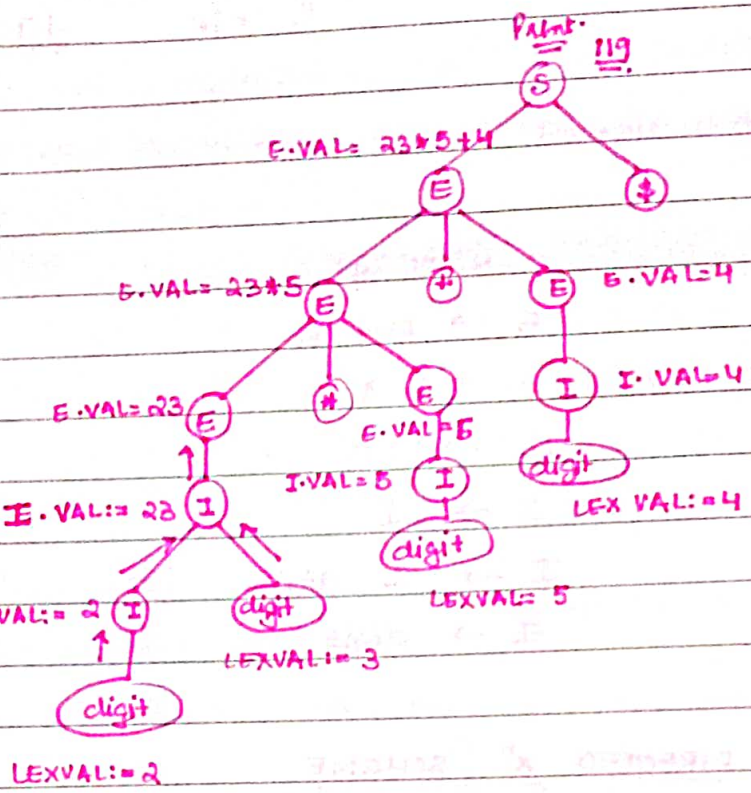
$I \rightarrow I$ digit

$I \rightarrow$ digit

SYNTAX DIRECTED x^n SCHEME

Production	Semantic Action
$S \rightarrow E \$$	{ print E.VAL }
$E \rightarrow E^{(1)} + E^{(2)}$	{ E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL }
$E \rightarrow E^{(1)} * E^{(2)}$	{ " := " * " }
$E \rightarrow (E^{(1)})$	{ E.VAL := $E^{(1)}$.VAL }
$E \rightarrow I$	{ E.VAL := I.VAL }
$I \rightarrow I^{(1)}$ digit	{ I.VAL := 10 * $I^{(1)}$.VAL + LEXVAL }
$I \rightarrow$ digit	{ I.VAL := LEXVAL }

Parse tree



PARSE TREE AND SYNTAX TREE

graphical representation of source prog.

It is very useful sply in OPTIMISATION PHASE

as in this the code needs to be extensively restructured.

It is a variant of Parse tree, in which:

each leaf represents operand

&

Each interior node rep. operator

E.VAL is a translation whose value is a pointer to node in syntax tree.

④ The function NODE (OP, LEFT, RIGHT) take 3 arguments.

1st → name of operator.

2nd & 3rd → ptr's to root of subtrees.

It creates a new node & makes 2nd & 3rd Arg. the left & right children.

eg.

$E \rightarrow E^{(1)}$ op $E^{(2)}$ $\{E.VAL := NODE(op, E^{(1)}.VAL, E^{(2)}.VAL)\}$

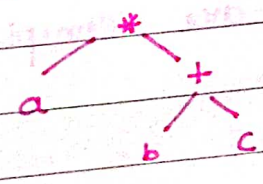
④ The function UNARY (OP, CHILD)

makes new node labeled OP & CHILD its child.

④ The function LEAF (ID)

creates new node labeled ID & returns ptr to that node.

eg. Syntax tree



THREE-ADDRESS CODE, QUADRUPLES AND TRIPLES:

① 3 add. code:

It is a sequence of statements, typically of general form

$$A := B \text{ op } C$$

where A, B & C are either → prog. defined names
or temp names etc.

& op stands for operator ie → fixed
→ arithmetic etc.

It is called "3-add code" as each statement usually contains

→ 3 addresses — 2 for operands
1 for result

eg. "X + Y * Z" yield

$T_1 := Y * Z$

$T_2 := X + T_1$

T_1 & T_2 are compiler generated Temp names.

② Some Statements are:

① A := B op C

② A := op B (unary op)
eg. Shift

③ unconditional Jump ie goto L

④ condisional jump ie
if A relop B goto L

⑤ Indx stmts
eg. A := B[i] &
A[i] := B

QUADRUPLES:

It has 4 fields ie → OP
ARG 1
ARG 2
RESULT

This representation of 3-add code is Quadruples.

eg. A := −B * (C+D)

∴ T_1 := −B
T_2 := C+D
T_3 := T_1 * T_2
result → A := T_3

	OP	ARG 1	ARG 2	RESULT
(0)	uminus	B	-	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3	-	A

TRIPLES:

It enables us to avoid entering temporary names.

It has 3 fields ie → OP
ARG 1
ARG 2

We use parenthesized nos to represent pointers.

eg.

	OP	ARG1	ARG2
(0)	uminus	B	-
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	(A)	(2)

$T_1 \to 0$
$T_2 \to 1$
$T_3 \to 2$
$A \to 3$

also for A[I] := B

	OP	ARG1	ARG2
(0)	[]=	A	I
(1)	-	B	-

INDIRECT TRIPLES:

It lists pointers to the triples rather than listing triples themselves.

	STATEMENT
(0)	(14) ← our choice.
(1)	(15)
(2)	(16)
(3)	(17)

	OP	ARG1	ARG2
(0)	uminus	B	-
(1)	+	C	D
(2)	*	(14)	(15)
(3)	:=	(A)	(16)

ERRORS

Repair & Recovery:

* Error repair → repairing error that has occured in the prog.

Error recovery → removing error from prog.

* Users of HLL are apt to make many errors. Thus, various phases of a compiler communicate with — error handler & error reporter which are invoked when error are detected.

* It is desirable that compilation of erroneous Prog continue, so that user can clean several errors out of the source before they recombine.

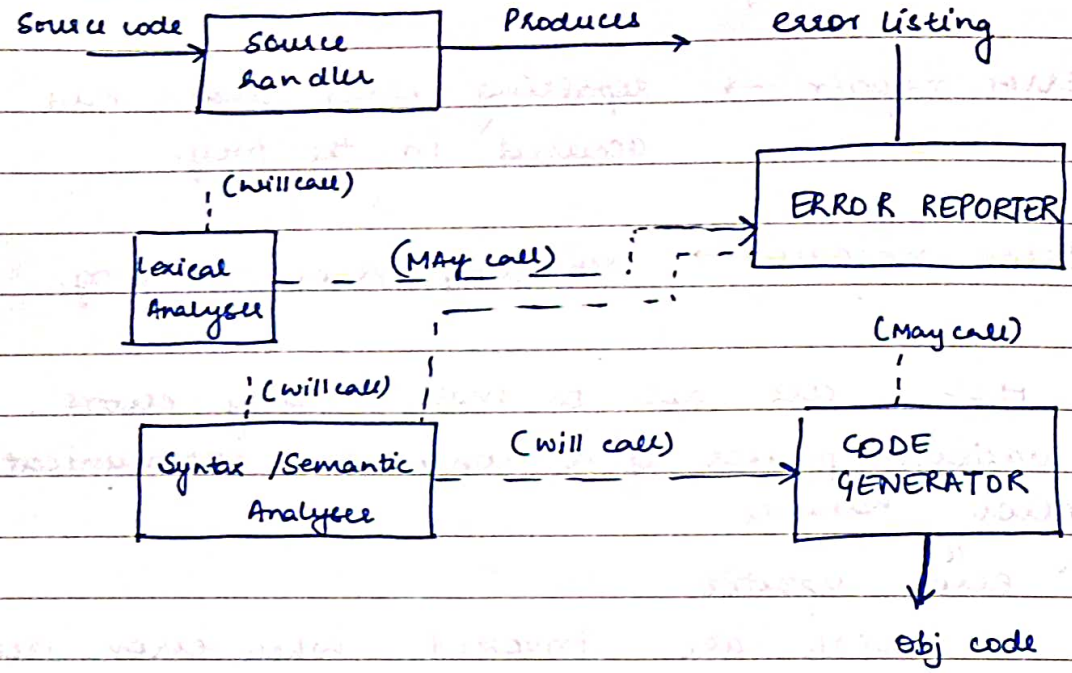
* Error can be detected at:

- compile time (in source code)
- Run time (when executing obj prog)

* Sometimes, mistakes are called:

Errors (are mistakes), that show at COMPILE TIME

& Exceptions at RUN TIME



④ On an error :

A simple compiler may STOP all activities.

A complex compiler may REPAIR the error.

⑤ Diff types of errors :

→ lexical analyser may be unable to proceed bcoz token is misspelled.

→ Syntatic error.

→ ICG may detect error on operator whose operands have incompatible types.

etc

MINIMUM DIST. MATCHING :

There are a no. of situation eg. – misspelling. for which we may use MDM to correct spelling of token.

if we have suspect 'x' ie misspelled bcoz no declaration for x is present in symbol table, we could try to find out an identifier in symbol table ie closest to x.

The prob. of finding a word from a given collec' that is closest to string 'x' has been studied in detail.

A no. of Algorithms are known :- ie–

fewest no. of insertions
" " " deletions etc.

SYNTATIC PHASE ERRORS :

Much of errors detection & correction in compiler is centered around syntax analysis.

From a grammar we can generate a parser that recognises exactly the lang. specified by the grammar.

↓
violation of this will be caught automatically by the parser.

A parser detects an error when it has no legal move from its current configuration.

To recover from the error, the parser should locate the position of error, correct it & revise its current configuration.

Time Detection :

LL(1) & LR(1) have valid prefix property :

they will announce errors as soon as a prefix of i/p as been seen for which there is no valid continuation.

The time taken by them is the shortest.

PANIC MODE

Effective systematic method of error recovery is called the panic mode.

In this mode, The parser discards i/p symbol until a synchronizing token eg. ; is encountered

The parser deletes the stack entries until it finds an entry such that it can continue parsing.

ER in Operator Precedence parsing:

The problems are:-

- There is no precedence relation
- Handle is found but no production is there.

In handle match we check that:

- terminals are same
- positions occupied by non-terminal is also the same.

ER during reduction:

Suppose abc is popped & there is no production right side consisting a, b & C tog. Then we can:

1) Delete one of a, b or c:

ie aAcB & issue

ILLEGAL b ON LINE

2) Insert a terminal
ie abAdc & issue

→ MISSING d ON LINE

Handling shift reduce errors :

suppose a & (b) → top are 2 top stack symbols &
c & d are next symbols & there is no
precedence rela^n b/w c & b

↓ then :

eg. if a ⋖ c we pop 'b' from stack

or b ⋖ d we delete 'c'.

ER in LR parsing :

It detects an error when it consults the parsing action table & finds an error entry.

Exhaustive method

It is to examine each error entry in parsing table & decide the most likely error that would give rise to that error.

Systematic method

Scan down the stack until state 's' till nonterminal 'A' is found.
The parser stacks the state GOTO [s, A] & resumes.

Adhoc: Design a handling routine for an LR parser.

ER for YACC :

user adds grammar error prodn of form

$A \rightarrow$ error α

YACC encounters errors & finds underlying set of items + item of form $A \rightarrow$ error α.

↓

It then shifts error

↓

Then invokes semantic action associated with

$A \rightarrow$ error

↓

It discards i/p symbols until it finds a symbol from where parsing can proceed.

ER in LL Parsing :

It has "PANIC MODE".

On finding error we skip i/p

↓

we pop the stack until synchroniza token (X) is on top of stack

↓

If (X) is not synch. token then X will be expanded & symbols will be popped till syn. Token comes on top of stack.

SEMANTIC ERRORS:

- The 1° source of SE are:
  - undeclared names
  - type incompatibility.

- Recovery from undeclared name is straightforward.

1st time we encounter an undeclared name we make an entry for that name in the symbol table

A flag in symbol table entry is set to indicate the entry was made in response to semantic error.

- Each time a name is used incorrectly, we check to determine whether previously this erroneous usage was ~~declared~~ detected or not.

If detected → no new error msg is printed

If not → error msg printed & new erroneous usage is added to list.

- eg. ARAY [A , B]

↙

it is not a valid Array name.

UNIT 5

LR(0)

LR(0) item of grammar 'G' is production of G with a dot at some point of right side.

eg. $A \rightarrow XYZ$

So $A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$ etc.

consider: G → Grammar G' → Augmented grammar

S → Start Symbol S' → new start symbol

& $S' \rightarrow S$.

CLOSURE: Let 'I' be set of items for 'G'. Then closure is.

1) Every item in 'I' is in CLOSURE (I).

2) If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE (I) & $B \rightarrow \gamma$ is a production then add it to I.

GOTO:

GOTO (I, X).

where I → set of items

X → grammar symbol.

if $A \rightarrow \alpha X \cdot \beta$ is in I

then add all productions

in which X. will be present.

SLR Parsing table : | CLR parsing table

I/P : canonical collection of sets of items for an augmented grammar G'

O/P : LR Parsing table containing — ACTION & GOTO functions.

METHOD: Let

$C = \{ I_0 \text{ --- } I_n \}$

the states of parser from 0 to n.

state i is being constructed from I_i

The parsing action is:

1) If $[A \to \alpha \cdot a\beta]$ is in I_1 & GOTO $(I_i, a) = I_j$ (a → terminal.)
set ACTION $[i, a]$ to 'shift j'

2) If $[A \to \alpha \cdot]$ is in I_1, set ACTION $[i, a]$ to 'reduce $A \to \alpha$' for all a in FOLLOW (A)

3) If $[S' \to S \cdot]$ is in I_1, Set ACTION $[i, \$]$ to 'accept'

4) If GOTO $[I_i, a] = I_j$ then GOTO $[i, A] = j$

5) All entries not defined by rules (2) & (3) are made "ERROR".

6) The initial state of parser is one constructed from set containing item

$(S' \to .S, \$)$

LR(1):

I/P: Grammar G

O/P: set of LR(1) items.

METHOD:

```
procedure CLOSURE (I);
begin
  repeat
    for [A → α·Bβ, a] in I,
    each production B → γ & each terminal b
    such that [B → ·γ, b] is not in I
    do add [B → ·γ, b] to I;
  return I
end;

Procedure GOTO (I, X)
begin
  let J be set of Items [A → αX·β, a] such that
      [A → α·Xβ, a] is in I;
  return closure (J)
end;

begin
  C := { CLOSURE ({S' → ·S, $}) };
  repeat
    for each set of Items I
      & symbol X
      such that GOTO (I,X) is not empty
      & not in C do
        add GOTO (I,X) to C
  until no more sets of items can be added to C
end.
```

LALR Parsing table:

I/P: grammar G augmented by $S' \rightarrow S$

O/P: LALR Parsing table.

METHOD:

1) Construct $C = \{ I_0 \text{ -- } I_n \}$
sets of LR(1) items.

2) Find all sets having core & replace these sets by their union.

3) Let $C' = \{ J_0 \text{ --- } J_m \}$ be resulting LR(1) items.

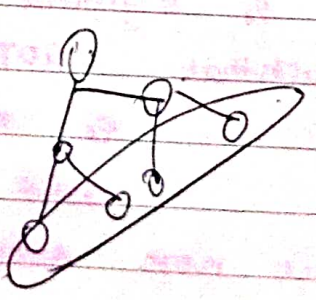
4) GOTO table is constructed as follows.

if $J = I_1 \cup I_2 \cup \text{---} I_m$ then

cores of GOTO (I_1, X)
GOTO (I_2, X) ---
are same

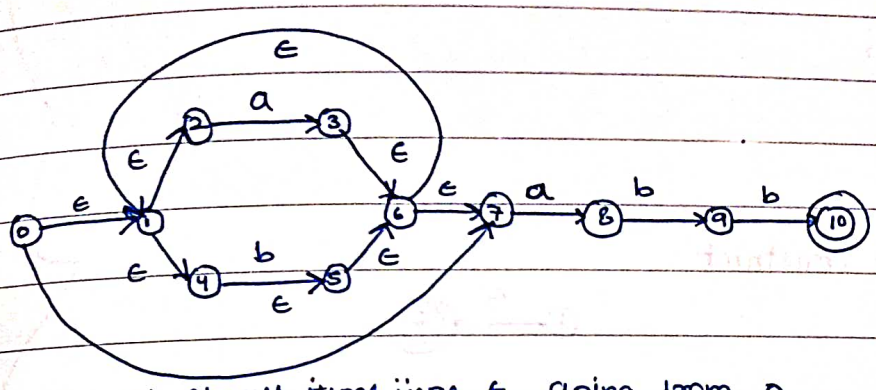
since I_1 -- I_k have same cores.

Thus GOTO $(J, X) = K$



UNIT:2

NDFA: $(a|b)^* abb$



set of all items jispe ϵ going from 0

$A = \{0, 1, 2, 4, 7\}$

from 1 to all elements via (a)

$B = \{1, \underset{\epsilon}{2}, 3, 6, 7, 8, 4\}$

from 1 to all elements via (b)

$C = \{1, 4, 5, 6, 7, 2\}$

now reach (b) on 9 & (b) on 10

$\therefore D = \{1, 4, 5, 6, 7, 2, 9\}$

$E = \{1, 4, 5, 6, 7, 2, 10\}$

now,

	I/P	
State	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C

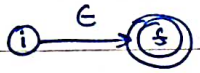
RE → NFA

I/P: RE 'R'

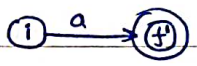
O/P: NFA 'N'

METHOD:

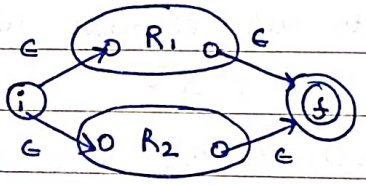
1) For ∈ we construct



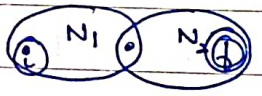
2) For a we construct



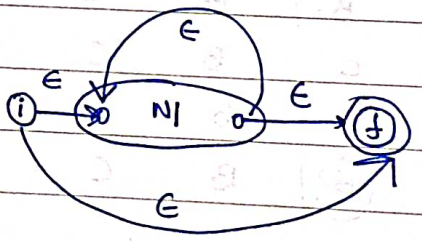
3) For R_1/R_2 we construct



4) For R_1R_2 we construct



5) For R_1^* we construct



Minimizaⁿ of FA :

I/P: A DFA 'M' with set of states 'S'

O/P: DFA 'M' with reduced set of states.

METHOD:

eg.

	I/P	
States	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C

$\pi_0 = \{E\}\{A, B, C, D\}$ ← subgroups

$\pi_1 = \{E\}\{A, C\}\{B\}\{D\}$

$\therefore \pi_f = \{E\}\{A, C\}\{B\}\{D\}$

$\therefore$

	I/P	
States	a	b
→ A	B	A
B	B	D
D	B	E
(E)	B	A

we omit state C as C = A.

goto func^n

LR(0)

I_0 = $E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot id$

closure(I_1) = $E' \to E\cdot$
$E \to E\cdot+T$

I_2 = $E \to T\cdot$
$T \to T\cdot *F$

I_3 = $T \to F\cdot$

I_4 = $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E) \mid \cdot id$

goto(0,5)

I_5 = $F \to id\cdot$

I_6 = $E \to E+\cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E) \mid \cdot id$

I_7 = $T \to T*\cdot F$
$F \to \cdot(E) \mid \cdot id$

I_8 = $F \to (E\cdot)$
$E \to E\cdot+T$

I_9 = $E \to E+T\cdot$
$T \to T\cdot *F$

I_{10} = $T \to T*F\cdot$

I_{11} = $(E)\cdot$

SLR: not as shift-reduce conflict.

0) $S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

I_0 = $S' \to \cdot S$
$S \to \cdot L=R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot id$
$R \to \cdot L$

I_1 = $S' \to S\cdot$

I_2 = $S \to L\cdot =R$
$R \to L\cdot$

I_3 = $S \to R\cdot$

I_4 = $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

I_5 = $L \to id\cdot$

I_6 = $S \to L=\cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

I_7 = $L \to *R\cdot\cdot$

I_8 = $R \to L\cdot$

I_9 = $S \to L=R\cdot$

Table same as LALR | CLR but we dont add $ to production → we just calculate follow & add $

Not Ambi grams + LR(0)

8. $S \to CC$ (1)
$C \to cC$ (2)
$C \to d$ (3)

Follow{S} →

we add with start symbol.

$I_0 =$ $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$ → all terminals
$C \to \cdot d, c/d$

$I_1 =$ $S' \to S\cdot, \$$

$I_2 =$ $S \to C\cdot C, \$$ match.
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

$I_3 =$ $C \to c\cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

$I_4 =$ $C \to d\cdot, c/d$

$I_5 =$ $S \to CC\cdot, \$$

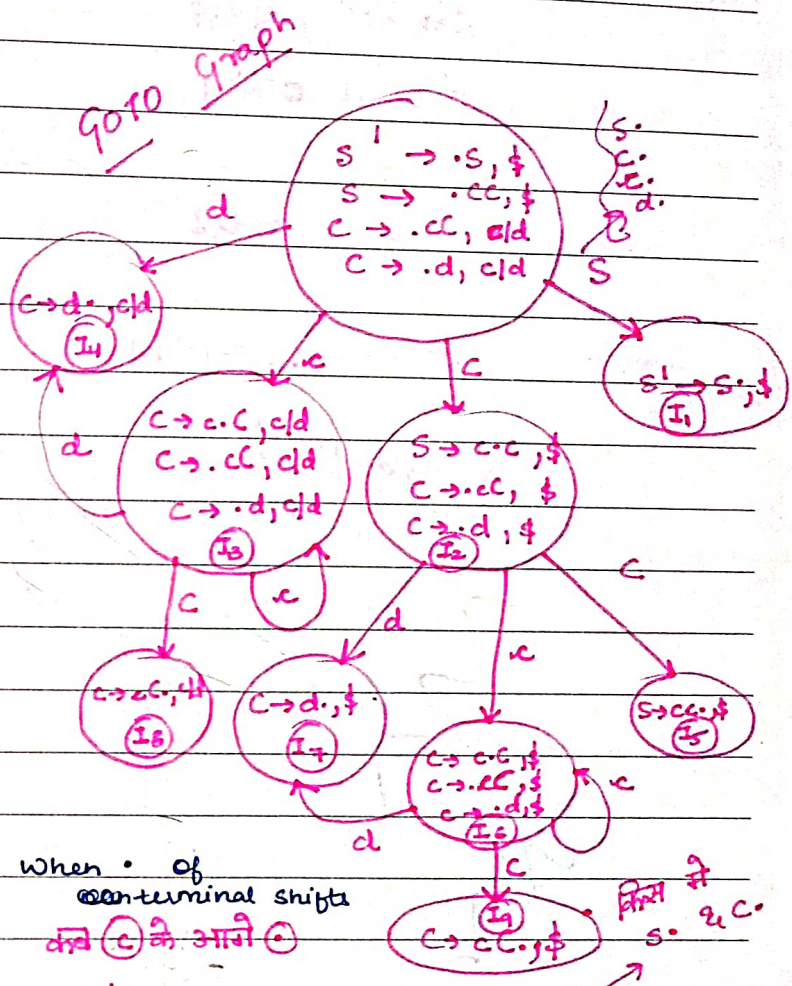
$I_6 =$ $C \to cC\cdot, \$$ choose only $ we take.
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

$I_7 =$ $C \to d\cdot, \$$

$I_8 =$ $C \to cC\cdot, c/d$

$I_9 =$ $C \to cC\cdot, \$$

goto graph



When • of non-terminal shifts दाएं (c) के आगे •

State	Action c	Action d	Action $	goto S	goto C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5	r_1	r_1	r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

$S' \to S\cdot, \$$

If ~~any~~ ? check (1) (2) (3) ... with • at last. (end)

When • is at end

OVEMBER 2008

	Mo	Tu	We	Th	Fr	Sa	Su
						1	2
0	4	5	6	7	8	9
7	11	12	13	14	15	16
4	18	19	20	21	22	23
	25	26	27	28	29	30

wk DECEMBER 2008

wk	Mo	Tu	We	Th	Fr	Sa	Su
49	1	2	3	4	5	6	7
50	8	9	10	11	12	13	14
51	15	16	17	18	19	20	21
52	22	23	24	25	26	27	28
1	29	30	31	*	*	*	*

- DFA ⟷ NDFA
- Minimizaⁿ of FA

SATURDAY 04

* { refer Pg 141 } not Imp.

Regular Exp to FA :

eg. RE → $(0+1)^*\ (00+11)\ (0+1)^*$

1. $q_0 \xrightarrow{(0+1)^*(00+11)(0+1)^*} q_f$

2. $q_0 \xrightarrow{(0+1)^*} q_1 \xrightarrow{(00+11)} q_2 \xrightarrow{(0+1)^*} q_f$

3. $q_0 \xrightarrow{(0+1)^*} q_1$; $q_1 \xrightarrow{0} q_3 \xrightarrow{0} q_2$; $q_1 \xrightarrow{1} q_4 \xrightarrow{1} q_2$; $q_2 \xrightarrow{(0+1)^*} q_f$

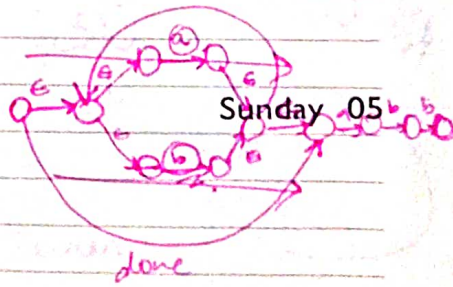
PB

4. $q_0 \xrightarrow{\wedge} q_5 \xrightarrow{\wedge} q_1$; $q_5 \xrightarrow{(0,1)}$ self; $q_1 \xrightarrow{0} q_3 \xrightarrow{0} q_2$; $q_1 \xrightarrow{1} q_4 \xrightarrow{1} q_2$; $q_2 \xrightarrow{\wedge} q_6 \xrightarrow{\wedge} q_f$; $q_6 \xrightarrow{(0,1)}$ self

5. $q_0 \xrightarrow{(0,1)}$ self; $q_0 \xrightarrow{0} q_3 \xrightarrow{0} q_f$; $q_f \xrightarrow{(0,1)}$ self; $q_0 \xrightarrow{1} q_4 \xrightarrow{1} q_f$

State	0	1
→q_0	q_0, q_3	q_0, q_4
q_3	q_f	q_f
q_4	q_f	q_f
(q_f)		

$(a|b)^*abb$



Sunday 05

PRIORITIES

★ Pg 156

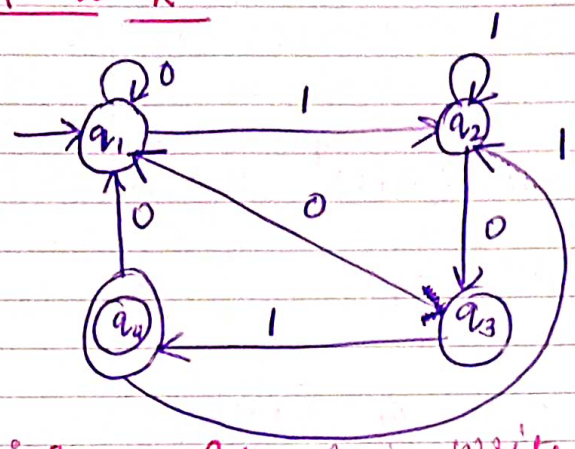
Pg 156.

The wise does at once what the fool does at last.

06 MONDAY

FA to RE :

pg 149



q_1 → jinke 0 data hai write them.

$\therefore \quad q_1 = q_1 \cdot 0 + q_4 \cdot 0 + q_3 \cdot 0 + \wedge$

$q_2 = q_1 \cdot 1 + q_2 \cdot 1 + q_4 \cdot 1$

$q_3 = q_2 \cdot 0$

$q_4 = q_3 \cdot 1$

now substitute $\therefore \quad q_4 = q_2 \cdot 0 \cdot 1$

$q_2 = q_1 1 + q_2 1 + q_4 1$

$R = Q + RP$

$q_2 = q_1 \cdot 1 + q_2 \cdot 1 + q_2 \cdot 0 \cdot 1 \cdot 1$

$q_2 = q_1 \cdot 1 + q_2 (1 + 011)$

$R = Q + RP = QP^*$

$\therefore \quad q_2 = q_1 \cdot 1 (1+011)^*$

now $q_1 = q_1 \cdot 0 + q_2 \cdot 0 \cdot 1 \cdot 0 + q_2 \cdot 0 \cdot 0 + \wedge$

$= q_1 \cdot 0 + q_1 \cdot 1 (1+011)^* \cdot 0 \cdot 1 \cdot 0 + q_1 1 (1+011)^* \cdot 0 \cdot 0 + \wedge$

FA → RE

PRIORITIES

$q_1 = q_1 (0 + 1(1+011)^* \cdot 0 \cdot 1 \cdot 0 + 1(1+011)^* \cdot 0 \cdot 0) + \wedge$

$R \quad R \quad P \quad Q$

$q = \pi P^*$

$q = P^*$

$q_1 = \pi_{q_1} \{ (0 + 1(1+011)^* \cdot 0 \cdot 1 \cdot 0 + 1(1+011)^* \cdot 0 \cdot 0) \}^*$

$q_1 = (0 + 1(1+011)^* \cdot 0 \cdot 0 \cdot 1 + 1(1+011)^* 0 \cdot 0)$

$q_4 = q_2 \cdot 0 \cdot 0 = q_1 1 (1+011)^* 01 = (0 + 1(1+011)^* \cdot 0 \cdot 0 \cdot 1 + 1(1+011)^* \cdot 0 \cdot 0)^* (1+011)$