UNIT - I INTRODUCTION TO COMPILERS & LEXICAL ANALYSIS.

Introduction - Translators - Compilation and Interpretation - Language processors - The phases of Compiler - Lexical Analysis - Role of Lexical Analyzer - Input Buffering - Specification of Tokens - Recognition of Tokens - Finite Automata - Regular Expressions to Automata NFA, DFA - Minimizing DFA - Language for specifying Lexical Analyzers Lex tool.

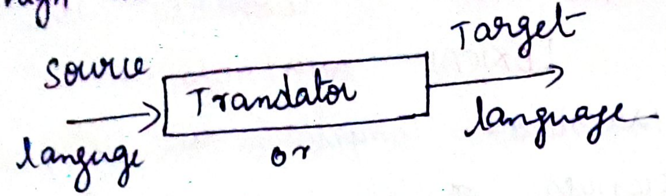
Introduction

Compilers are basically translators. how the source program is compiled with the help of various phases of compiler.

Translators:-

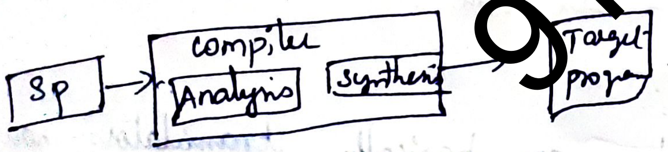
Translators is one kind of program that take source code as input and converts into another form.

low level language or assembly language or high level language high to low or machine



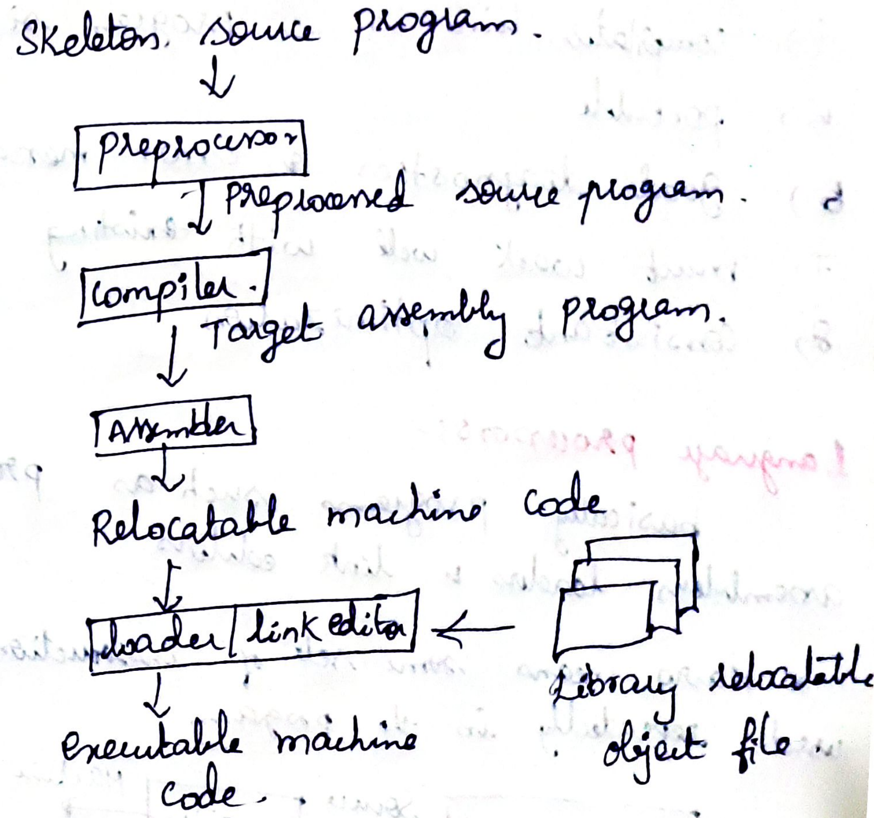
Translators (compilers and assemblers
high to machine   assembly to machine

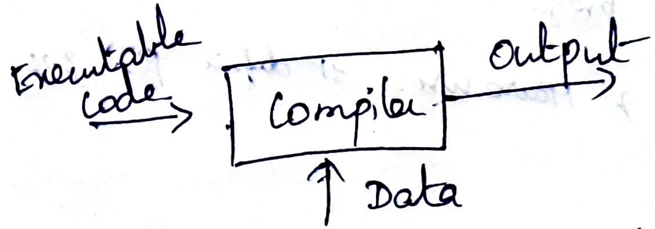
Analysis and Synthesis Model:
↳ intermediate form, converted into equivalent target program
↓
is read & broken into constituent pieces, intermediate code is created



Execution of program:
→ only compiler program is not sufficient
compiler source to highlevel language, target assembly code as input, relocatable machine code as output.

The task of loader, relocation of object code allocation of load time address which exits in memory & placement of load time address, and data in memory at proper locations.
The link editor link several files of relocate object modules to resolve the mutual reference. These files may be library files.





process of execution of program

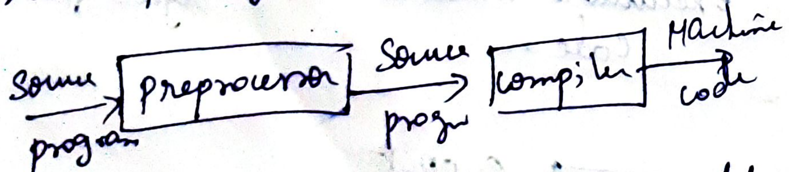
properties of compilers:

1) bug-free :
2) correct machine code
3) generate machine code run fast.
4) compilation time is ∝ program size
5) portable
6) good diagnostics & error message
7) must work well with existing debuggers.
8) consistant optimization

language processors:-

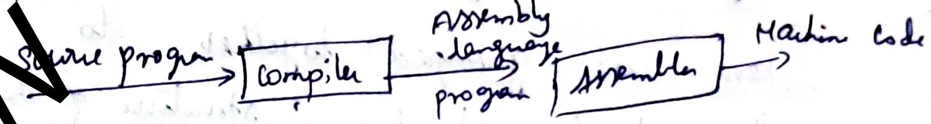
basically programs such as preprocessors assemblers loaders & link editors

* Macro means some set of instructions, used repeatedly in the program.



Macro definition & Macro use. # define PI 3.14

Assemblers




```
MOV a, R1
MUL #5,R1
ADD #, R1
MOV R1, b ;
```

} Binary language, Machine code. relocatable machine code. two passes → one pass input program end of second pass is relocatable machine code.

Loaders & link editors:-

relocatable machine code is read and the relocatable addresses are altered.

The phases of compiler:-

1) Lexical Analysis (scanning)

complete source code is scanned and broken up into group of strings called token.

A token is a sequence of characters having a collective meaning.

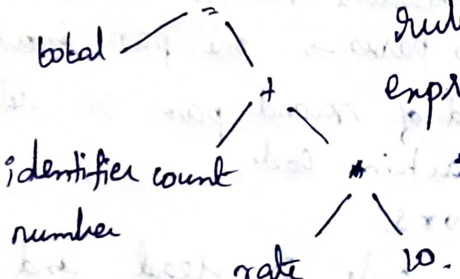
total = count + rate * 10.

$id_1 = id_2 + id_3 *$ constant no.

blank characters are eliminated during lexical analysis.

Syntax analysis:- parsing.

tokens are grouped together to form a hierarchical structure (structure of the source string). Called parse tree or syntax tree.



rules are usually expressed by context free grammar.

(1) E → identifier count

2, E → number

3, E → E1 + E2

4, E → E1 * E2

5, E → (E).

Semantic Analysis:- determines the meaning of the source string (matching of paranthesis or if else statements or performing arithmetic expressions, After these phases, intermediate code is generated.

Intermediate Code Generation:- Code can be easily converted to target code

three address code, quadruple, triple, posix

t_1 := int to float (10) t_3 := count + t_2

t_2 := Rate × t_1 total := t_3

order of operations devised by three address code

t_1 := int_to_float (10)

total := t_3

5, Code optimization.

faster executing code or less consumption of memory, optimizing the code, overall running time of target program can be improved.

6, Code generation

intermediate code instructions are translated into sequence of machine instructions.

MOV rate, R_1 — variable rate move R_1

MUL #10.0, R_1 — R_1 = R_1 * 10

MOV count, R_2 — variable count move R_2

ADD R_2, R_1 — add with R_1 R_2

R_1 = (Count) + (rate * 10) [R_1]

MOV R_1, total — R_1 moved to identifier total.

Symbol table Management:-

Store identifiers (variables) used in the program.

Stores information about attributes of each identifiers.

It is a data structure used to store the information about identifiers.

store & retrieve data from that record efficiently

Error detection & handling

errors are reported to error handlers, compilation can proceed, syntax analysis phase,

Syntax error

a = b + c * 60

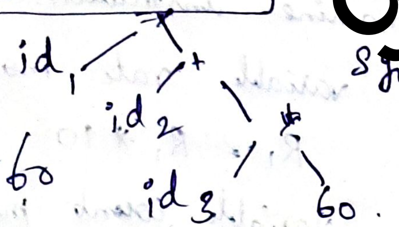
Lexical analyzer

↓

$t_3 := id_2 + t_2$ $id_1 = id_2 + id_3 * 60$ Tokens

$id_1 := t_3$

Syntax analyzer



syntax tree

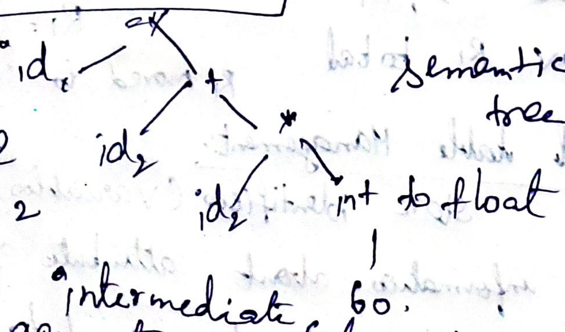
↓

Code optimize

$t_1 := id_3 * 60$

$id_1 := id_2 + t_1$

Semantic analyzer



semantic tree

Code generate

MOV F id_3, R_2

MUL F #60.0, R_2

MOV F id_2, R_1

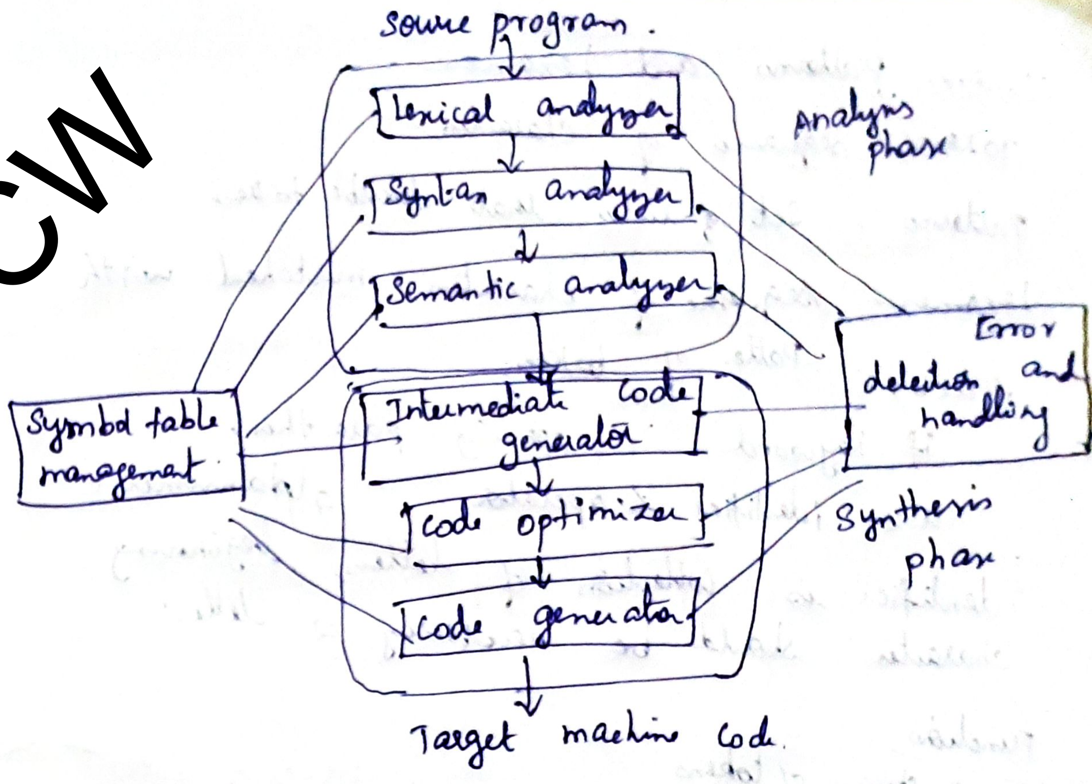
ADD F R_2, R_1

MOV F R_1, id_1

intermediate code generate ic

$t_1 :=$ int to float (60)

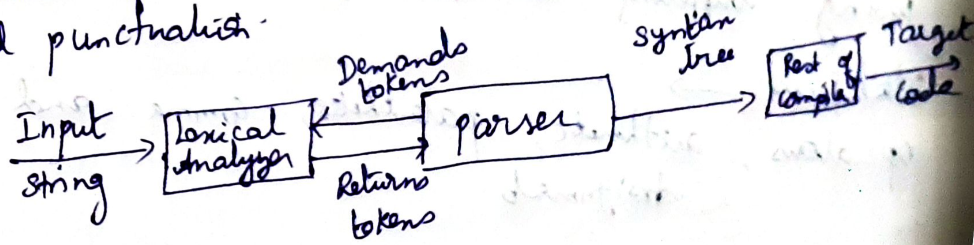
$t_2 := id_3 * t_1$



Lexical Analysis

Role of Lexical Analyzer (first phase of compiler)

Reads the input from source program left to right one character at a time and generates the sequences of tokens (identifiers, keyword operators and punctuation.



Tokens, patterns and Lexemes:-

Tokens: sequence of character.

patterns: set of rules, that describe tokens.

Lexemes: sequence of characters matched with pattern of token.

if(a<b)

if - Keyword '(' opening paranthesis

a ← identifier < operator , alphanumeric

identifier is collection of letters, beginning character should be necessary a letter.

Functions

1) Stream of tokens

Issues of lexical Analyzer.

1) Lexical Analysis and Syntax Analysis are seperated out, Reduces burden of on parsing phase,

using buffering techniques for efficient scan

identifiers →

operators, arithmetic, parathesis, comma and Assignment.

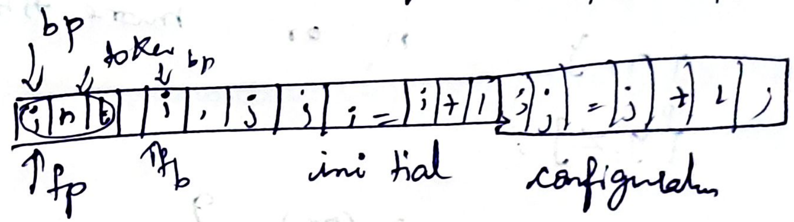
Keyword:: some special words, int, void are keywords

Input Buffering:-

is used to identify the lexeme correctly using two pointer method

scan left to right begin_ptr (bp)

forward - ptr (fp)



initial configuration

One buffer scheme: lexeme is very long, crosses buffer boundary, buffer has been refilled, overwriting the first part of lexeme.

Two buffer scheme: sentinel. identify the end of buffer

Regular expression:-

[abc] = a, b or c.

[^abc] - any character except a, b, c

[a-z] = a to z, [A-Z] = A to Z

[0-9] . 0 to 9.

[ ] ? -> 0 ~~or~~ 1 times

[ ] + occurs 1 or more

[ ] * " 0 or more.

{n} = occurs n times.

{n,} n or more times.

{y,z} .

1) mobile no/s 8 (or) 9

[8 9] [0-9]{9}.

2) upper case, contains lower case chars, one digit in between.

[A-Z] [a-z] * [0-9] [a-z]

\d [0-9]

\J [1 0-9]

\w [a-z, A-Z, 0-9]

[A\w]

Mail ID. abc123 @ gmail.com

[a-z A-Z 0-9 _ \- \.] + [@] [a-z] + [\.] [a-z]{2,3}

Recognition of TOKENS:

token representation

| TOKEN type | TOKEN value | [token attribute

↓ category. ↓ information regarding

1) Symbol table is maintained

2) identifier & constants (pointer to Symbol table)

TOKEN	code	value
if	1	-
else	2	-
while	3	-
for	4	-
identifier	5	ptr to ST
constant	6	ptr to ST
< <= > >= !=	7	1, 2, 3, 4, 5
( )	8	1, 2
+ -	9	1, 2
=	10	-

Steps to recognize token

1) stores input in input buffer

2) regular expression is built for corresponding token

3) N Deterministic finite automata is built.

4.

Finite Automata:-

5 tuples $(Q, \Sigma, \delta, q_0, F)$.

$Q \rightarrow$ finite set of states, non empty.
(duo)

$\Sigma \rightarrow$ input alphabet (finite set of i/ps.

δ - transition function, next state can be determined.

q_0 - initial state $q_0 \in Q$

$F \rightarrow$ final states.

Types of Automata $\rightarrow$

Finite Automata
- DFA
- NFA

$q_1 = \delta(q_0, a)$ Only for current i/p current state.

$q_1 \downarrow$ next state, $q_0 \downarrow$ current state, $a \downarrow$ input.

NFA with ε.
↳ empty symbol.

Regular NFA with 5 tuples.
$\{Q, \Sigma, q_0, F, \delta\}$.
where $\delta : Q \times \Sigma \rightarrow 2^Q$

ε-NF $\{Q, \Sigma, q_0, F, \delta\}$ $\delta : Q \times \Sigma \cup \varepsilon \rightarrow 2^Q$

eq

$\Rightarrow$ B is not seeing anything, but it goes to C.

here: every state on ε goes to it self.

A on ε goes to A.

Deterministic finite Automata.

* if the m/c is read on i/p string one symbol at a time. DFA $\xrightarrow{a}$ b

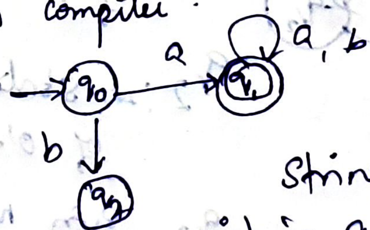
DFA uniqueness of the computation.

DFA only one path for specific i/p from the current state to the next state. (A) $\xrightarrow{0}$ B

DFA not accept null movie, does not change state without any i/p character.

DFA can have multiple final states. lexical Analysis in compiler.

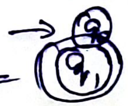
$\delta : Q \times \Sigma \rightarrow Q$ (current state, i/p symbol, next state)



String reaching final state it is acceptance.

$L\{\phi\} \Rightarrow$

$L\{\varepsilon\} =$



no/. of state = Min string + 1.

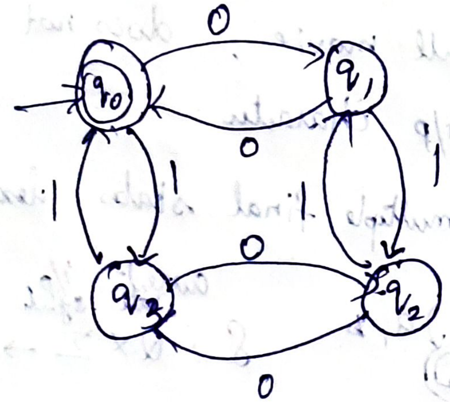
eg. DFA with $\Sigma\{0,1\}$.
Starts with 1 & ends with 0.

$L = \{10, 100, 1000, 1010, 1100, 1110 \ldots\}$

Min length = 2, No/. of states = 2+1 = 3

10 →
100 →
1010 →

2) FA with $\Sigma = \{0,1\}$ even no/. of 1's & even no/. of 0's.
$L = \{001, 110, 100, 110, \ldots\}$

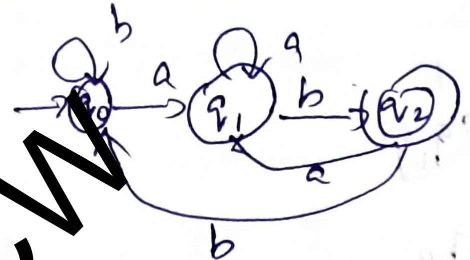


q0: even no/. of 1's, even no/. of 0's.

q1 = State of odd no/. of 0's, even no/. of 1's

q2 :→ State of odd no/. of 0's & 1's

q3. even no/. of 0's odd no/. of 1's



ab, baab, aab, bbaab

NFA Non Deterministic finite automata.

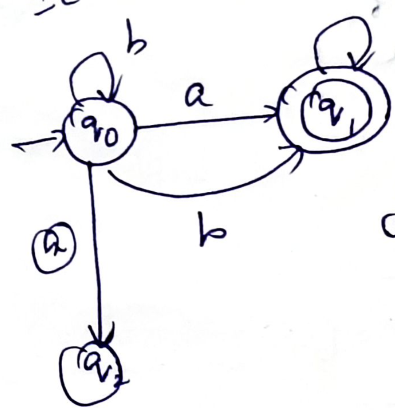
NFA is not DFA, each NFA translated to DFA

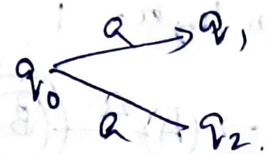
NFA is defined in the same way as DFA

but, It contains multiple next state.

It contains ε transitions.

eg





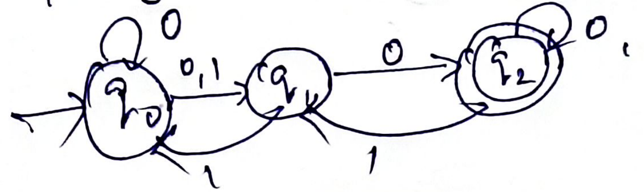
multiple next state

$\delta\ Q \times \Sigma \to 2^{Q}$
↓
Multiple states.

eg. $Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0,1\}$, $q_0 = \{q_0\}$ $F = \{q_2\}$

Soln

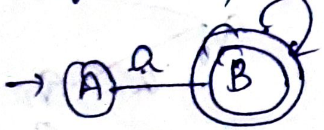


State	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_1\}$
q_1	q_2	q_0
(q_2)	q_2	$\{q_2, q_1\}$

Example NFA starts with 'a' $\Sigma = \{a, b\}$

$L = \{a, ab, abb, aba, aab, abab \ldots\}$

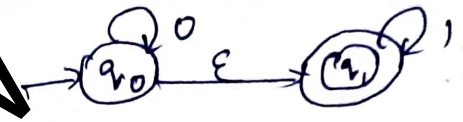
Min Length = 1, no. of states = 2.



Design NFA

State	0	1
q_0	$q_0 q_1$	$q_0 q_2$
q_1	q_3	ϵ
q_2	q_1, q_3	q_3
* q_3	q_3	q_3

NFA with ϵ to NFA without ϵ.



$M = \{Q, \Sigma, \delta, q_0, F\}$

$Q = \{q_0, q_1\}$ $\Sigma = \{0, 1, \epsilon\}$ $q_0 = q_0$, $F = q_1$.

Transition table

State \ Input	0	1	ϵ
q_0	$\{q_0\}$	-	$\{q_1\}$
q_1	-	$\{q_1\}$	

Step 2: find ϵ-closure.

ϵ-closure $(q_0) = \{q_0, q_1\}$

ϵ-closure $(q_1) = \{q_1\}$

Step 3. processing states q_0, q_1.

q_0 $\delta'(q_0, 0) = \epsilon\text{-closure}(\delta(q_0, q_1), 0)$

$= \epsilon\text{-closure}(q_0)$

$= q_0, q_1$

$\delta'(q_0, 1) = \epsilon\text{-closure}(\delta(q_0, q_1), 1)$

$= \epsilon\ closure(q_1)$

$= \{q_1\}.$

$\delta'(q_1, 0) = \epsilon - closure(\delta(q_1, 0))$

$= \epsilon - closure(\phi)$

$= \phi.$

$\delta'(q_1, 1) = \epsilon - closure(\delta(q_1, 1))$

$= \epsilon - closure\{q_1\}$

$= \{q_1\}.$

ε NFA to NFA

	0	1
→ A	{A, B, C}	{B, C}
B	{C}	{B, C}
* C	{C}	{C}

	ε*	0	ε*
A	A	A	A, B, C
	B	φ	–
	C	C	C

	ε*	1	ε*
A	A	φ	
	B	B	B, C
	C	C	C

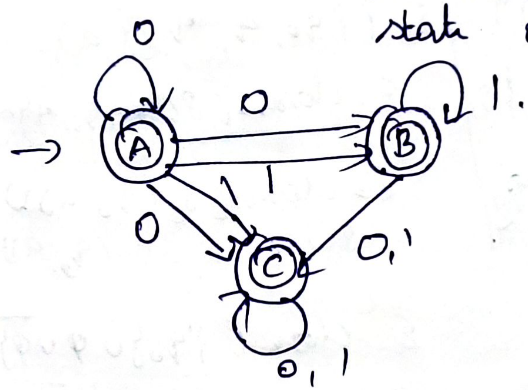
	ε*	0	ε*
B	B	φ	–
	C	C	C

	ε*	1	ε*
B	B	B	B, C
	C	C	C

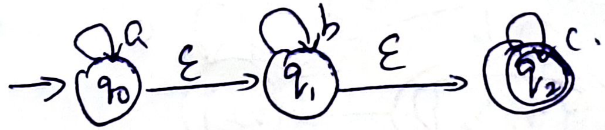
	ε*	0	ε*
C	C	C	C

	ε*	1	ε*
C	C	C	C

ε-closure (ε*) - All the states that can be reached from a particular state only by seeing ε symbol.



NFA with ε to DFA



ε-NFA	a	b	c	ε
→ q_0	{q_0}	φ	φ	{q_1}
q_1	φ	{q_1}	φ	{q_2}
* q_2	φ	{q_2}	{q_2}	φ

Set of all states that reach particular state, include that state

ε closure $(q_0) = \{q_0, q_1, q_2\}$

ε closure $(q_1) = \{q_1, q_2\}$

ε closure $(q_1) = \{q_2\}$.

Start state $\rightarrow \varepsilon$ closure of $\{q_0, q_1, q_2\}$

DFA δ_D	a	b	c
$\rightarrow q_0 q_1 q_2$	$[q_0, q_1, q_2]$	$\{q_1, q_2\}$	$\{q_2\}$
$\{q_1, q_2\}$	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	q_2

$q_0(a) \cup q_1(a) \cup q_2(a)$

$\delta_D(\{q_0, q_1, q_2\}, a)$

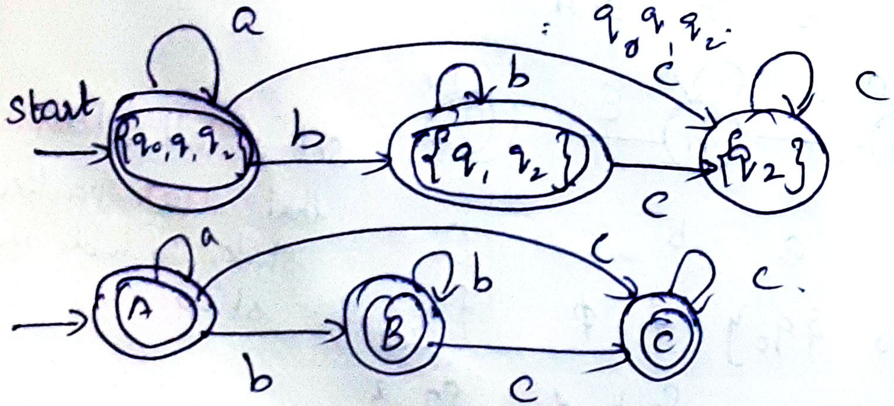
$= \varepsilon\text{-closure}(\delta_E(\{q_0, q_1, q_2\}, a))$

$= \varepsilon\text{-closure}(\delta_E(q_0, a) \cup \delta_E(q_1, a) \cup \delta_E(q_2, a))$

$= \varepsilon\text{-closure}(\{q_0\} \cup \phi \cup \phi)$

$= \varepsilon\text{-closure}(q_0)$

$= q_0 q_1 q_2$

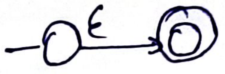


Regular Expression to ε-NFA (Thomson Construction)

RE to ε-NFA.

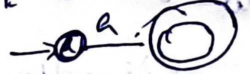
Basis:

1, RE $= \varepsilon$.



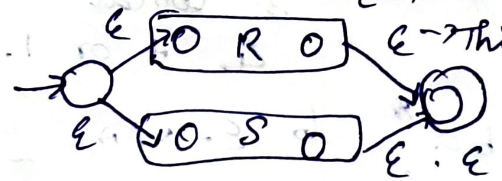
2, RE $= \phi$.

3, RE $=$ a



Induction: $\rightarrow$ Regular Expression.

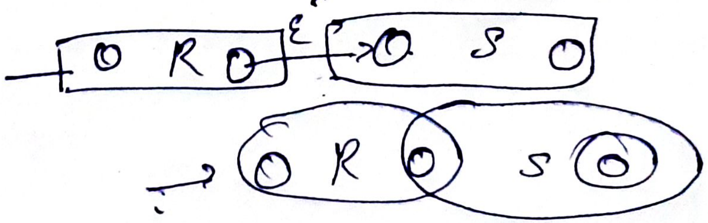
1) RE $= R+S$ or $R|S$.



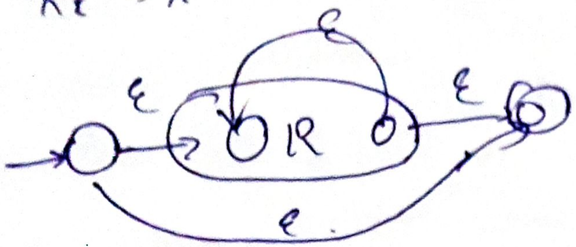
RE $= (0+1)^* | (01)$
R S

2, RE $=$ RS. [Concatenation]

identify element.



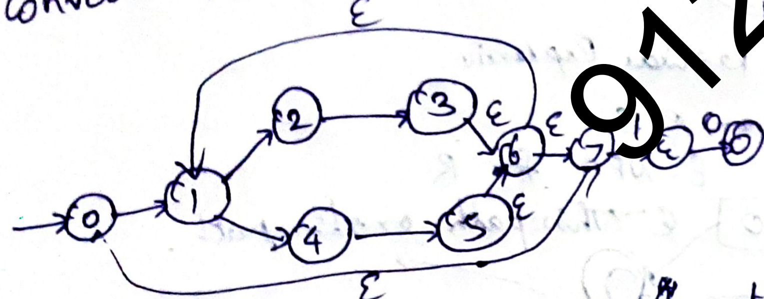
3) RE = R*



4) RE = (R).

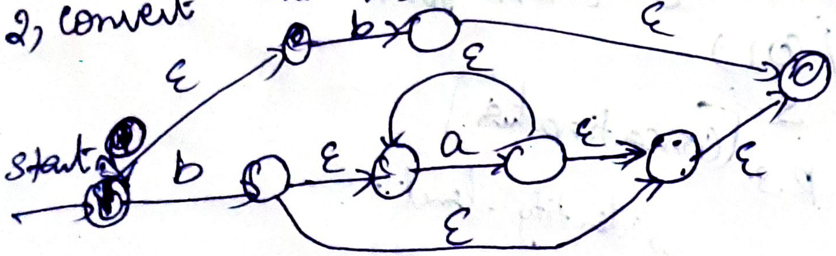
ε NFA for R = ε-NFA for (R)

convert the RE (0+1)* 10 to an ε-N.F.A.

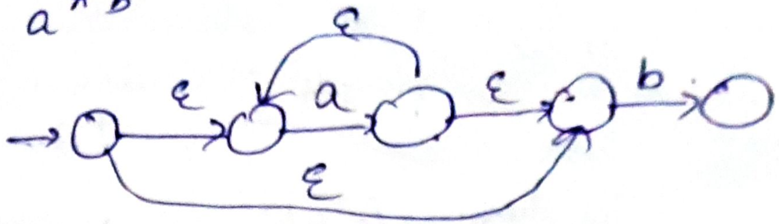


Precedence
1. ( )
2. *
3. concat
4. + or |.

2) convert the RE b+ba* to ε-NFA



3) r = a*b



Regular expression to DFA

RE to ε NFA to DFA.

RE to NFA with ε.

1) R-E → NFA with ε.
2) convert NFA with ε to NFA without ε
3) convert the obtained NFA to equivalent DFA.

R.E → NFA-ε → NFA → DFA.

1) Design FA for given R.E 10 + (0+11)0*1

10/9/2023

LEX = {tool}

⇓

→ break up an i/p streams into tokens.

→ automatically generating a Lexer (scanner)

lex.l → [Lex Compiler] → lex.yy.c

lex source program.

Step 2 lex.yy.c → [C compiler] → a.out } Lexical Analyzer LA.

Step 3:-

i/p stream → [a.out] → seq of tokens.

i/p → [LA] → tokens.

↳ character one by one.

check identifier, opers,

help of lex tool →

STRUCTURE of LEX Program:

{declaration} => declaration of variables.

%%

rules section {translation rules} => have the pattern {Action}

{i/p stream represent in lex program}

procedure section %

{auxilary function} => fun. can be compiled seperately.

lex program:- no. of vowel & constants.

i/p stream ⇓ grammer.

```
%{ # include <stdio.h>
int vowels = 0;
int cons = 0;
%}
%% {a e i o u A E I O U} {vowels ++;}
{a-z A-Z} {cons ++;}
%%
int yy wrap()
{
return 1;
}
main()
{
printf(" enter the string at end press ^\n");
yylex(); [find the vowels & cons
printf(" no. of vowels = %d\n
no. of cons = %d\n",
vowels, cons);
}
```

1) session starting & ending %, { & %}.

2) %%

3) two functions, main function & yywrap function.

yylex routine is given. lex.yy.c.

$ lex x.l ↳ program na

$ cc lex.yy.c [gcc can also used,

$ ./a.out.

yylex()

Starting point of lex from which scanning of source program starts.

yywrap()

This function is called when end of file is encountered. If yywrap returns 0 the scanner continues scanning if it returns 1 the scanner does not return tokens.

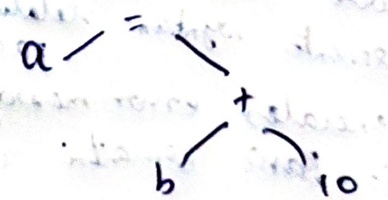
UNIT - II SYNTAX ANALYSIS.

Role of parser - Grammars - context free grammars - writing a grammar Top down parsing - General strategies - Recursive descent predictive parser - LL(1) Parser - Shift Reduce parser - LR parser - LR(1) Item construction of SLR parsing table - Introduction to LALR parser - Error Handling and Recovery in Syntax Analyzer - YACC tool - Design of a syntax analyzer for a sample language.

Introduction of Syntax Analyzer:

It is a second phase in compilation (parser) →

A parsing is a process which takes the input string w and produces either a parse tree (syntactic structure) or generates the syntactic errors. a = b + 10;



Basic issues of parsing

i) specification of syntax, (ii) Representation of input after parsing.

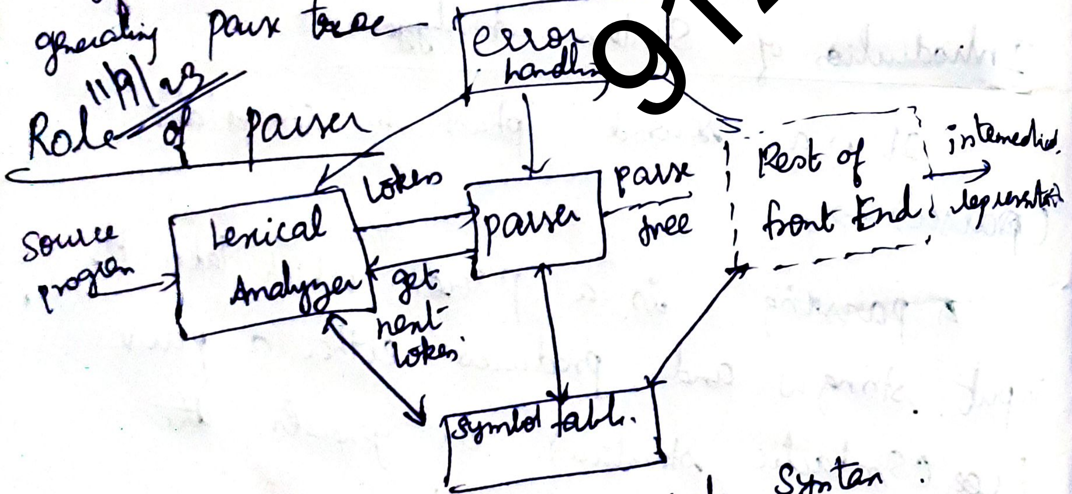
(i) precise & unambiguous, in detail complete.

Content free grammar.

(ii) information during parse tree has been generated, shall not be different after generating parse tree.

11/9/23

Role of parser



1) Syntax Analysis: check source code syntax.
2) parsing: Break code into tree-like structure
3) Data structure: Represents syntax relationships using parse trees.
4) Error handling: Generates error messages for developers
5) contextual Analysis: Basic semantic checks

6) Intermediate Representation: Eases further Compiler phases (process is optimized is easy, tokens are in content free.

Error Handling:

1. Lexical phase errors
   - spelling errors.
   - exceeding length of identifiers
   - illegal character appearance

normally typing mistake, the wrong spelling

2) Syntax analysis phase errors:
   - error in structure
   - Missing operators
   - unbalanced parenthesis

3) Semantic errors
   - Incompatible types of operands.
   - undeclared variables.
   - No matching of actual with formal parameters

Content free grammar: → formal grammar, used to generate all possible pattern of string in a given finite language

1) V → non terminals
2) T → terminals.
3) S → start symbol.
4) P → set of production rules

$G = (V, T, S, P)$. → set of production rules used to generate string of language.

Production rules

Non terminal $\longrightarrow (V \cup T)^*$

P → production rules which is used for replacing non-terminals symbols.

P → aSb. E → E+E

S → AS.

always non terminals S → b/a.

1) $L = a^n b^n$ where $n \geq 1$.

G. {V, T, S, P}

where V = {S}, T = {a, b}, S is start symbol.

P = { S → aSb

S → ab

1) Construct CFG for the language having any no. of a's over the set Σ = {a}

Σ = {a}

L = {ε, a, aa, aaa ... }

R.E = a^*

production rule.

S → aS -①

S → ε -②

i/p aaaaa

=> aS S → aS

=> aaS S → aS

=> aaaS S → aS

=> aaaaS S → aS

=> aaaaaS.

= aaaaa. S → ε

Type list terminat

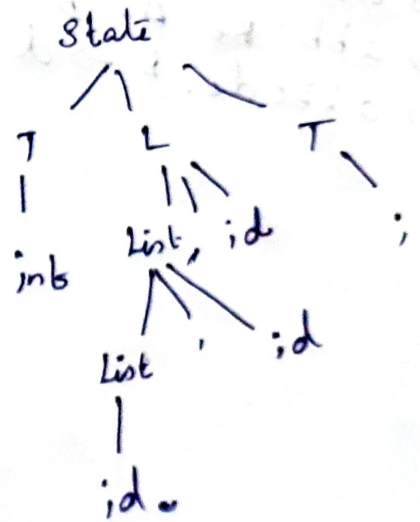
2) S → T L T

Type → int | float

List → List, id

List → id.

terminator → ;



int id, id, id ;

3) S → aABe

A → Abc | b

B → d.

i) Right most abbcde

ii) left most abbcde

3) parse tree

Soln right most abbcde

S → aABe. ←

S → aAde ; B → d

S → aAbcde. A → Abc

S → abbcde. A → b

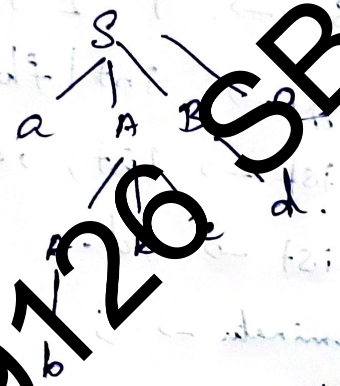
left most abbcde.

S → aABe.

S → aAbcBe. A → Abc

S → abbcBe A → b

S → abbcde. B → d



abbcde

Ambiguous grammar:-

more than one → left most, right most parse tree gives i/p string

If grammar is ambiguity, then it is not good for compiler construction.

must remove ambiguity

E → I   G = {V T P S}   V = {I, E}.

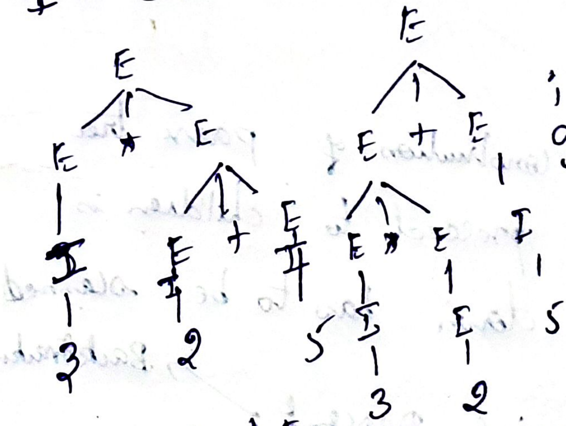
E → E+E   T = {+ * (,), ε, 0 1 ... 9}

E → E* E   3 * 2 + 5.

E → (E)

I → ε|0|1|2|... 9..

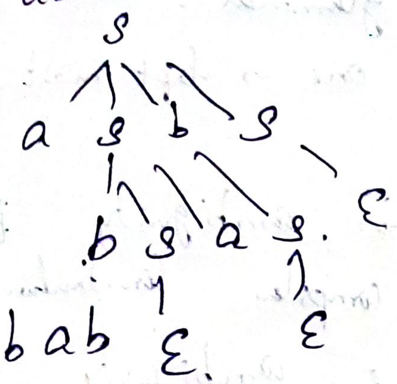
Since have two parse tree, so it is an ambiguous grammar



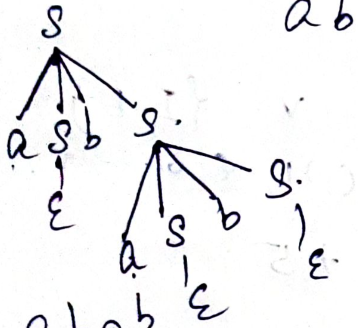
3 * 2 + 5.

$S \to aSbS$
$S \to bSaS$
$S \to \varepsilon$

abab



abab



abab

right most

Parsing Techniques:

Top-Down parser:

The process of construction of parse tree starting from root & proceed to children is called TDP. Top → down has to be scanned.

Top down → Recursive Descent → Backtracking / → Non-backtracking ↓ Predictive parser ≥ LL parser.

TDP internally uses left most derivation free from ambiguity & left recursion.

classification of TDP:

with backtracking → brute force technique

without backtracking → predictive parse → LL(1) / recursive descent parser.

Recursive Descent parsing:

1) parse construct from top to Bottom.

2) i/p is read from left to right.

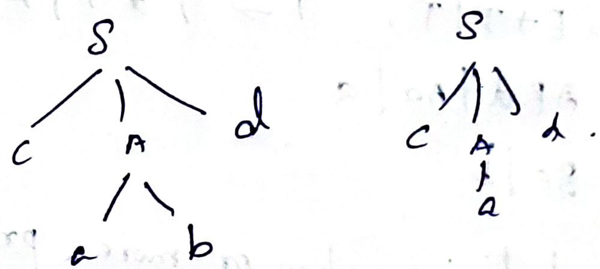
3) i/p is recursively parsed for preparing a parse tree with or without backtracking

Backtracking:

W = cad,

$S \to cAd$

$A \to ab \mid a$



Left recursion

$A \xrightarrow{+} A\alpha \mid \beta$. Keep calling itself.

$A \to A\alpha \mid \beta \implies A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$.

$S \to ABC$

$A \to Aa \mid Ad \mid b$

$B \to Bb \mid c$

$C \to Cc \mid g$

$A = Aa \mid b$.

$A = bA'$
$A' = aA' \mid \epsilon$

$A = Ad \mid b$

$A = bA'$
$A' = dA' \mid \epsilon$

$S \to ABC$

$A \to bA'$

$A' \to aA' \mid dA' \mid \epsilon$.

$B \to cB'$

$B' \to bB' \mid \epsilon$

$C \to gC'$

$C' \to eC' \mid \epsilon$.

$B = Bb \mid c$.

$B \to cB'$
$B' \to bB' \mid \epsilon$

$E \to E+T \mid T$, $T \to T*F \mid F$

$A \to ABd \mid Aa \mid a$

$B = Be \mid b$

Left factoring: two or more production are starting with same set of symbols

$A \to \alpha \beta_1 \mid \alpha \beta_2$.

$\alpha, \beta_1, \beta_2 \Rightarrow$ string.

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$.

$S \to iEtS \mid iEtSeS \mid a$

$E \to b$

$A = \alpha A'$
$A' = \beta_1 \mid \beta_2$.

$S \to iEtSS' \mid a$

$S' \to eS \mid a$.

$E \to b$.

ex

$A \to Ac \mid Aad \mid bd \mid c$.

left recursion

Left factoring $A \to aAB \mid aA \mid a$

$B \to bB \mid b$.

Removal of ambiguity.

(i) By adding precedence.

(ii) By adding associativeness

$E \to E+id \mid id$

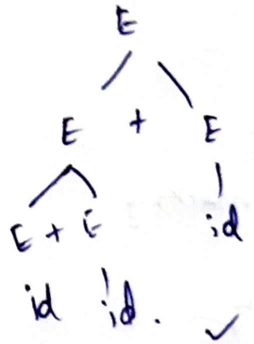
$E \to E+T \mid T$

$T \to T*F \mid F$

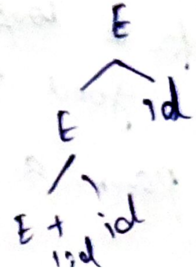
$F \to id$.

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

id + id + id.

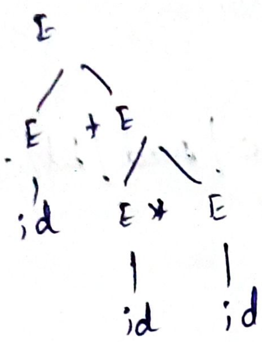


$E \to E+id \mid id$.



id+id+id.

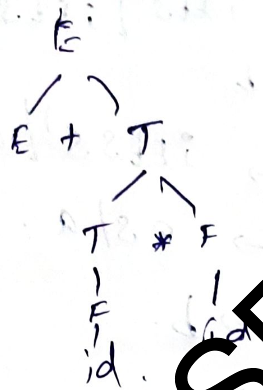
id+id * id :



E → E+T / T

T → T*F / F

F → id.



Recursive Descent Parsing:-

Collection of recursive procedures. a separate procedure non-terminal a separate procedure is written.

Advantages:-

1) Simple to build.
2) Constructed with the help of parse tree

Limitations:-

1) not very efficient
2) may enter infinite loop
3) not provide good error messaging
4) difficult to parse a string

Applications of FA (need)

1) Design of the lexical Analysis of a compiler.
2) Recognize the pattern by using regular Expressions
3) Helpful in text editors
4) used for spell checkers.
5) use of the Mealy & Moore machines for designing the combination & sequential circuits.

24/9/23

Predictive parser LL(1) parser:-

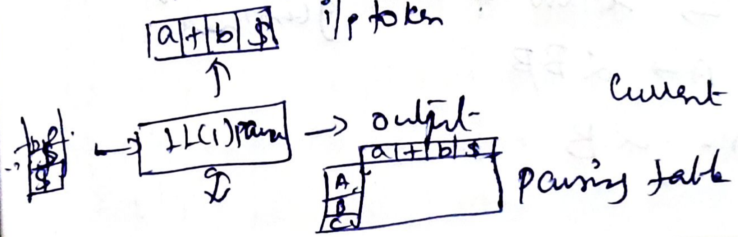
top-down parsing algorithm, non-recursive type. LL(1). first L → input is scanned from left to right. second L → left most derivation for i/p string.

Data structures: (i) i/p buffers → to store i/p tokens.

(ii) stack → hold left sentential form. pushed into stack in reverse order left to right

(iii) parsing table :→ row for non terminal, column for terminal M[A, a].

a ↓ terminal, current i/p symbol.



(i) Elimination of left Recursion
(ii) Left Factoring. E = E + T | T
(iii) First & Follow functions T = T * F | F
(iv) predictive parsing table. F = (E) | id
(5) parse the input string

Elimination of Left recursion

E → E + T | T (A = E, α = + T, β = T)

T = T * F | F

E → T E'

T → F T'

E' → + T E' | ε

T' → * F T' | ε

E → TE'

E' → +TE' | ε

T → FT'

T' → * FT' | ε

F = (E) | id

FIRST

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

Follow :-

1) $ → follow of start symbol of grammar
2) If A → α B β
3) A → α B

Follow(E) = {$,)}.

Follow(E') = {$,)}

Follow(T) = {+, $,)}

Follow(T') = {+, $,)}.

Follow(F) = {*, + $)}.

A → α

(i) M[A, a] = A → α
a is in FIRST(α)

M[A, b] = A → α
ε is in First(α),
b is in Follow(A).

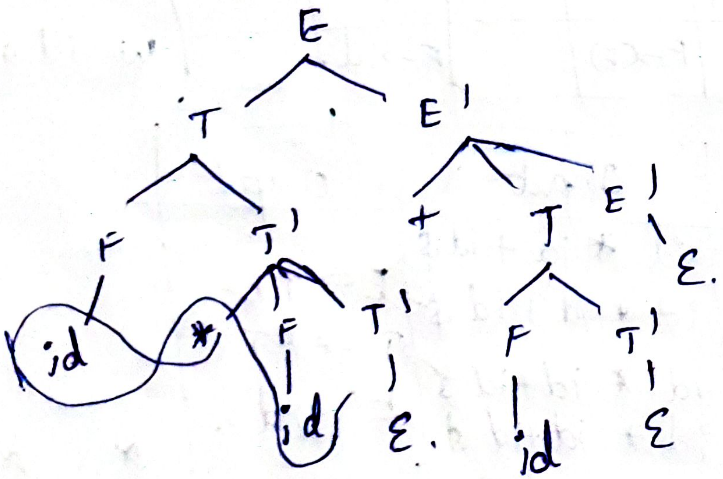
Parsing table Row → non terminals
column → terminals

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

w = id * id + id

Stack	Input	Output
$E	id * id + id $	
$ E' T	id * id + id $	E → TE'.
$ E' T' F	id * id + id $	T → FT'
$ E' T' id	id * id + id $	F → id.
$ E' T'	* id + id $.	
$ E' T' F *	* id + id $.	T' → *FT'
$ E' T' F	id + id $	

$ E'T'id	id+id$	F→id
$ E'T'	+id $	T'→ε
$ E'	+id $	E'→+TE'
$ E'T +	+id $	
$ E'T	id $	T→FT'
$ E'T'F	id $	F→id
$ E'T'id	id $	
$ E'T'	$	T'→ε
$ E'	$	E'→ε
$		



⇒ id * id + id

Follow:-

If following the variable, you have:

Terminal → write it as it is.

Non terminal → Write its First element.

last element → write Follow of LHS.

Follow set will never contains NULL.

S → iEtSS₁/a

$S_1 \to eS/\varepsilon$

E → b

First (S) = {i, a}

First (S₁) = {e, ε}

First (E) = {b}

Follow (S) = {e, $}

Follow (S₁) = {e, $}

Follow (E) = {t}

FOLLOW

1) Follow(S) = {$}, 2) A → αBβ, then.

FOLLOW(B) = FIRST(β). except ε.

3) If A → αB or A → αBβ where FIRST (β) contains ε (B → ε),

FOLLOW(B) = FOLLOW(A).

S → ABCDE

A → a/ε

B → b/ε

C → c

D → d/ε

E → e/ε.

FIRST

FIRST (S) = {a, b, c}.

FIRST (A) = {a, ε}

FIRST (B) = {b, ε}

FIRST (C) = {c}

FIRST (D) = {d, ε}

FIRST (E) = {e, ε}

FOLLOW(S) = \$
FOLLOW(A) = {b, c}
FOLLOW(B): {c}
FOLLOW(C): {d, e, \$}
FOLLOW(D) = {e, \$}
FOLLOW(E) = {\$}.

2)

S → ACB / CbB / Ba
A → da / BC
B → g / ε
C → h / ε

FIRST(S) - {d, g, h, ε, b, a}.
FIRST(A) - {d, g, h, ε}.
FIRST(B) - {g, ε}
FIRST(C) - {h, ε}

FOLLOW(S) - {\$}
FOLLOW(A) - {h, g, \$}
FOLLOW(B) - {\$, a, h, g}
FOLLOW(C) - {g, \$, b, h}.

Handle pruning:

bottom up parsing to find the sub-string that could be reduced by appropriate non-termi[nal]

3/10/23

Shift Reduce parsing bottom up parsing

Shift: Shift the next input symbol onto the top of the stack

Reduce: The right end of the string to be reduced must be at the top of the stack. Locate the left side of the string within the stack and decide with what non terminal to replace the string.

Accept: Announce successful completion of parsing.

Error: Discover a syntax error and call an error recovery routine.

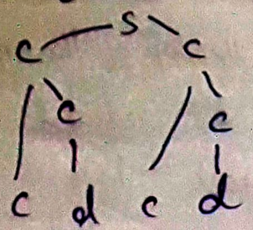
S → CC
C → cC / d

Stack	Input
\$	w\$
\$S	\$.

w → cdcd.

Stack	Input	Action
\$	cdcd \$.	Shift.
\$c	dcd \$	Shift
\$cd	cd \$	reduced by C → d.
\$cC	cd\$	reduced by C → cC
\$C	cd\$	shift.
\$Cc	d\$	shift
\$Ccd	\$	reduce by C → d.
\$CcC	\$	reduce by C → cC
\$CC	\$	red S → CC
\$S	\$	Accept

c d . c d



E → E+E

E → E*E

E → (E)

E → id

id * (id+id)

Stack	I/p string	Action
$	id*(id+id)$	shift
$id	*(id+id)$	reduced E → id
$E	*(id+id)$	shift
$E*	(id+id)$	shift
$E*(	id+id)$	shift
$E*(id	+id)$	reduce E → id
$E*(E	+id)$	shift
$E*(E+	id)$	shift
$E*(E+id	)$	~~shif~~ reduce E → id
$E*(E+E	)$	~~shift~~ reduce E → E+E
$E*(E	)$	shift
$E*(E)	$	reduce E → (E)
$E*E	$	reduce E → E*E
$E	$	accept

4/10/23

LR parsers, non recursive, shift reduce, bottom up parser

L → left to right scanning of input stream

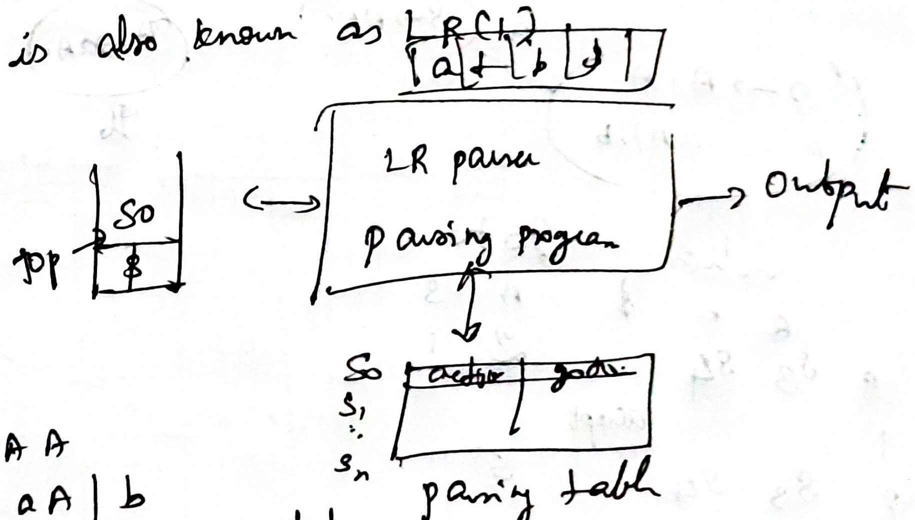
R → construction of right most derivation in reverse.

k → no. of lookaheads needed for decision

LR(0) < SLR(1) < LALR(1) < CLR(1)

SLR: Simple LR; LALR: lookahead LR; CLR: Canonical LR

(CLR(1)) is also known as LR(1)



LR(0)

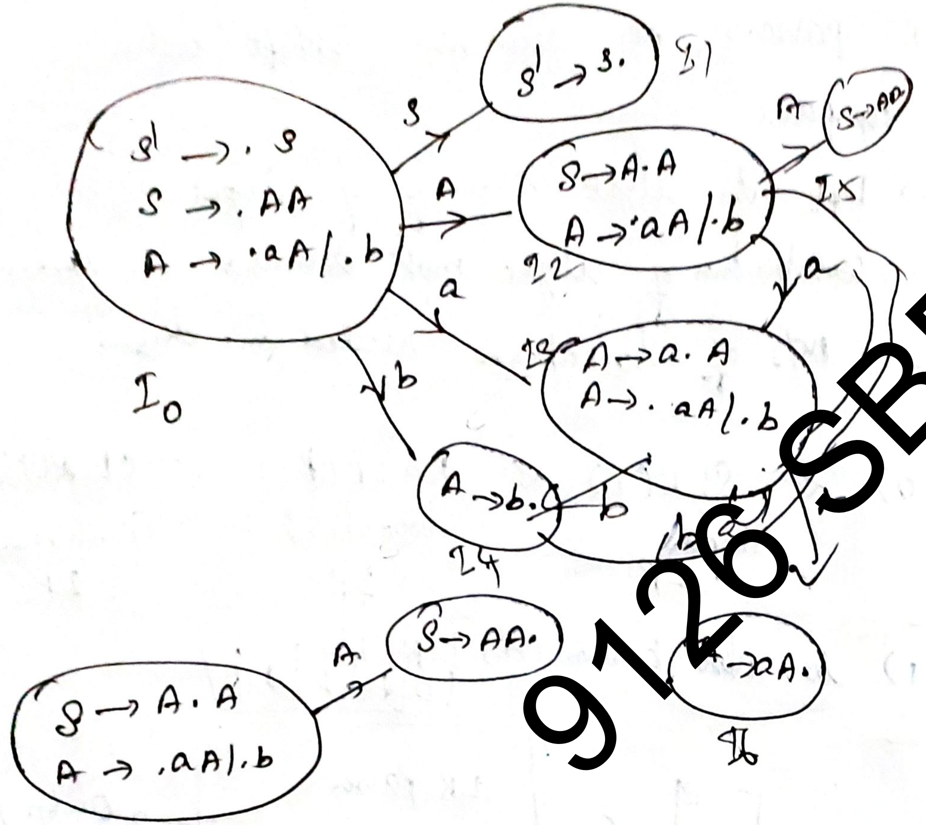
S → AA

A → aA | b

S' → .S → augmented production

S → .AA → dot in beginning.

A → .aA / .b. dot in end. my full production is parsed.



	action			goto	
	a	b	$	A	S
0	s3	s4		2	1
1			accept		
2	s3	s4		5	
3	s3	s4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

SLR

E → E+T 1
E → T 2
T → T*F 3
T → F 4
F → (E) 5
F → id. 6

T → · F
F → ·(E)
F → ·id.

goto(I0, id)
I5: F → id.

I0:
E' → ·E
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id

goto(I1, +)
I6: E → E+·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id.

goto(I0, E)
I1: E' → E·
E → E·+T

goto(I0, T)
I2: E → T·
T → T·*F·

goto(I2, *)
I7: T → T*·F
F → ·(E)
F → ·id

goto(I4, E)
I8: F → (E·)
E → E·+T

goto(I0, F)
I3: T → F·

goto(I0, ()
I4: T → (·E)
E → ·E+T
E → ·T
T → ·T*F

goto(I6, T)
I9: E → E+T·
T → T·*F

goto(I7, F)
I10: T → T*F·

goto(I8,))
I11: F → (E).

State, Action, goto.

State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2	r2	r2	S7	r2	r3				
3	r4	r4	r4	r4	r4				
4	S5			S4			8	2	3
5	r6.	r6	r6	r6	r6	~~r6~~			
6	S5			S4				9	3
7	S5			S4					10
8.		S6			S11				
9	r1	r1	r1	r1	r1				
10	r3	r3	r3	r3	r3				
11	r5	r5	r5	r5	r5				

stack	input	action
$0	id*id+id$	shift
$0id5	*id+id$	reduce F→id (1×2)
$0F3	*id+id$	reduce T→F (1×2)
$0T2	*id+id$	shift
$0T2*7	id+id$	shift
$0T2*7id5	+id$	reduce F→id. 1×2
$0T2*7F10	+id$	reduce T→T*F 3×2=6
$0T2	+id$	reduce E→T 1×2
$0E1	+id$	shift
$0E1+6	id$	shift
$0E1+6id5	$	reduce F→id.
$0E1+6F3	$	reduce T→F.
$0E1+6T9	$	reduce E→E+T
$0E1	$	accept

Canonical LR parsing:-

Similar to SLR parser. Only in reduce operation.

CLR → canonical collection of LR(1) items = LR(0) + look ahead.

LR(0)	SLR(1)	CLR < LALR
reduce is written in full row	reduce is written in follow of (P).	Redue is written only on look ahed.

E → BB

B → cB|d.

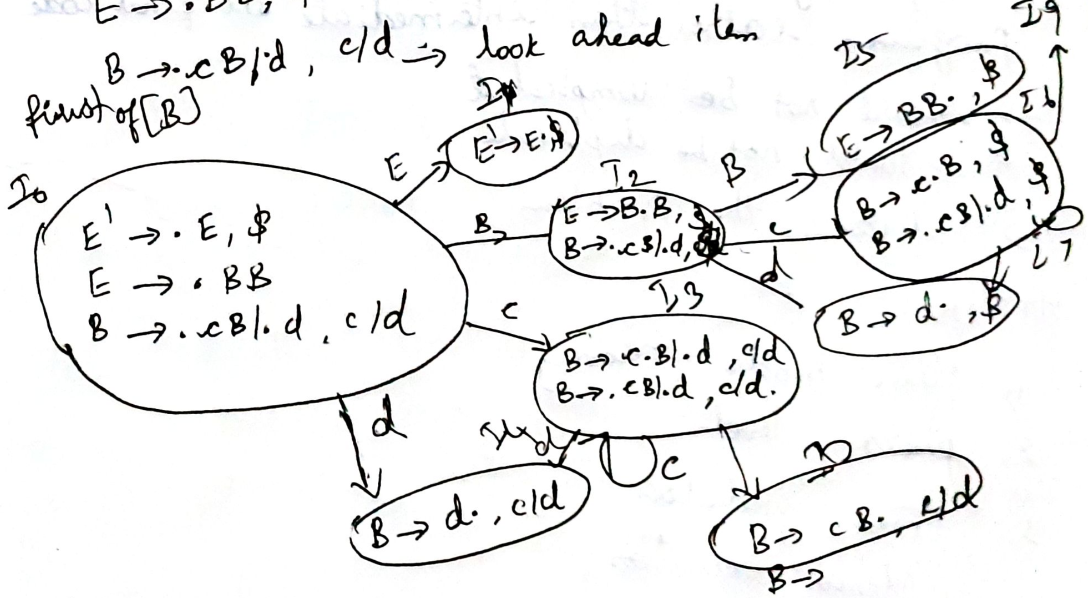
augument grammar along with look head items.

LR(0). E' → .E, $ → look ahead item.

E → .BB, $ → look ahead item

B → .cB|.d, c/d → look ahead item

first of [B]



	c	d	$	E	S
0	S3	S4		1	2
1			accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Error handing

1) important features detect & report errors
→ Reporting errors in original source program rather than intermediate or final code
→ should not be complicated
→ should not be duplicate
→ localize the problem.

strategies :

1) panic mode recovery
2) phrase level
3 Error productions
4) global correction

1) not specify ; y). Ex: a+b = c ;
d = e+f ;
2) by ; delete extra ; insert missing ;

3) good idea about common errors, find appropriate solutions is stored.
These productions detect the anticipated errors during parsing
Ex E → +E / -E / * E / /E

4) global corrections → incorrect input string required to transform x into y is expensive methods & not pratically used costly in terms of time & space

YACC → automatic tool for generating the parser program

Yet Another Compiler Compiler.
lex → lexical analyzer generator
yacc → parser generator
⇓
It is a tool which generate LALR parser

Lex → LA → grammar
YACC → Syntax Analysis → grammar.
Regular expression specification

Yacc working

Step 1 :- yacc specification parser.y
↓
Yacc compiler → y.tab.c

Step y.tab.c → C compiler → a.out syntax analy

Step :- i/p tokens → a.out → o/p parser

LA parse
grammar.y → YACC → y.tab.c { source program created by yacc
{ file containing desired grammar in yacc format
{ yacc program.
↓
C compiler
↓
i/p tokens generated by lex → a.out → Parse tree.
executable prog that will parse grammar given

parse tree
tokens grammar

Syntax :-
definitions { declaration of tokens, types of values used.
%%
Rules { list of grammar rules (grammar rules) with semantic actions.
%%
supplementary code

lex / yacc.

lexer.l token specification
↓ lex - tokens
lexer.c
type check.c
code generator

parser.y grammar spec
↓ yacc parser grammar
parser.c
others stuff
↓
My compiler

UNIT - III SYNTAX DIRECTED TRANSLATION & INTERMEDIATE CODE GENERATION

Semantic rules -

1) generate code
2) insert information into symbol table
3) perform semantic check.
4) Issue error messages.

Two notations of attaching semantic rules

1) Syntax Directed definitions - High level specification hiding many implementation details (Attribute grammars)

2) Translation schemes. More implementation oriented. Indicate the order in which semantic rules are to be evaluated

Semantic rule   production.

E.code = E1.code || T.code || '+'   E -> E1 + T

String val attribute differentiate.

we associate attributes to the grammar symbols representing the language constructs

values for attributes are computed by semantic rules associated with grammar productions

It is a generalization of context free grammars.

1) set of attributes
2) associated with semantic rules

Such formalism generates Annotated parse tree

each node of the tree is record with a field of each attribute.

L -> En -> numerical value.

F -> digit ., F.val => digit.lexval

Numerical value of token returned by lexical analyser

D → TL.   L.inh = T.type

T → int   T.type = integer

T → float   T.type = float

L → L1, id   L1.inh = L.inh

add Type(id, entry, L.inh)

L → id   add Type(id, entry, L.inh)

construction of Syntax tree

1) mknode(op, left, right)   x * y - 5 + z

2) mk leaf(id, entry)

3) mk leaf(num, val)

x * y

x p_1 = mk leaf(id, ptr to entry x)

y p_2 = mkleaf(id, ptr to entry y)

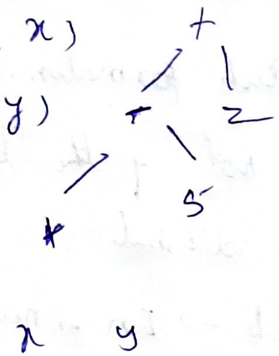
* p_3 = mknode(*, p_1, p_2)

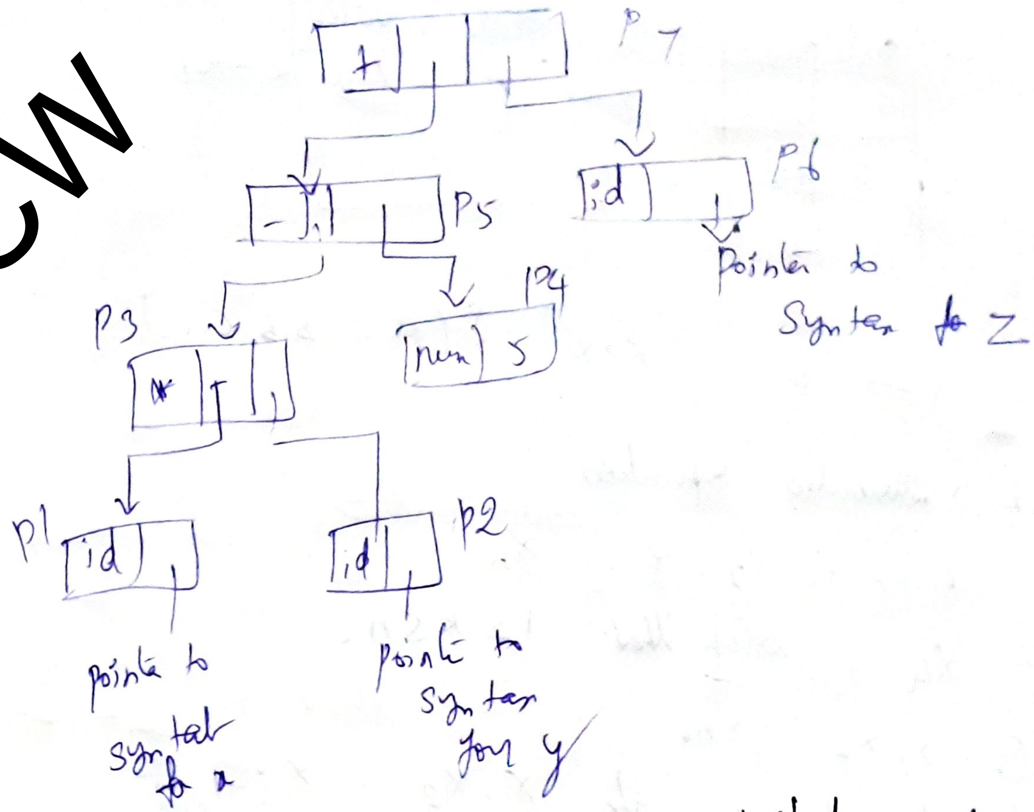
5   p_4 = mk leaf(num, 5)

\- p_5 mknode(−, p_3, p_4)

z p_6 mkleaf(id, ptr to entry z)

\+ p_7 = mknode(+, p_5, p_6)





* S attribute with synthesized attributes only

* evaluated using bottom up parse.

* purpose of stack is to keep track of values of the synthesized attributes associated with the grammar symbol, parse stack with the production X → ABC

State	Value
A	A.a
B	B.b
C	C.c

top → ⟹

State	Value
X	X.x

← top

$X \rightarrow ABC$, X.x = f [A, a, B.b, C.c]

L -> Attributed definitions

$A \rightarrow X_1, X_2, X \ldots X_n$.

X_4 is such that $1 \le k \le n$.

$A \rightarrow X_1 X_2 \ldots X_n$.

also depends upon the $X, X_2 \ldots X_{j-1}$ to the left of X'

depends upon inherited attribute A.

$A \rightarrow PQ$.   P.in := p(A, in)

Q.in := q(P, sy)

26/10/23 **Three address code:-**

* Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields,

* There are three representations used for three address code such as quadruples, triples and indirect triples.

Implementation of three address statements

1. Quadruple
2. Triples
3. Indirect triples

Quadruple:

op, arg1, arg 2, result.

x = y op z

x = op y (unary) } no arg 2.

x = y (copy statement)

Parameter x => not arg 2 & result.

goto L : unconditional Jump.

if x relop y goto L → Conditional jump.

$a = b * -c + b * -c.$

t_1 = uminus c.

$t_2 = b * t_1$

t_3 = uminus c

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Quadruples structure

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

Triples structure: (not using temp variables)

op, arg1, arg2

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Indirect triples

pointers to the triple structure.

			op	arg1	arg2
(0)	40	(40)	uminus	c	
(1)	41	(41)	*	b	(40)
(2)	42	(42)	uminus	c	
(3)	43	(43)	*	b	42
(4)	44	(44)	+	(41)	(43)
(5)	45	(45)	=	(a)	(44)

There are two ways to store two dimensional array in memory.

(i) column major order

(ii) row - major order.

Two dimensional 3X4 array.

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \end{array} \quad 3\times4$$

(i) row major order { elements are stored in memory row by row }

a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}
first row				second row			

(ii) Column major order { elements are stored in memory column by column.

a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}	a_{14}	a_{24}	a_{34}
1st column			2nd column								

The address of element first row & first column $A[a_{11}]$

base address of A : { base(A) }

column major order.

address of $A[j, K] = Base(A) + w(m(K-1) + j-1)$ (L_0)

w = size of element = datatype size.

m = no. of rows.

row major order.

address of $A[j, K] = Base(A) + w[n(j-1) + (K-1)]$

n = no. of columns.

eg

Single dimensional array.

Array — single / two: column MO, row MO

int A[5];

A[3] = 20;

Add(A[3]) = 10 + 3 * 2

	0	1	2	3	4
A	10	12	14	16	18

linear.

base add (L_0)

{add a[i] = base add + i * size of data type}

$= L_0 + i * w$

$= 10 + 3 * 2$

$= 10 + 6 = 16$

Some compiler will start the index from 1, instead of 0.

A[1...5] as integer.

	1	2	3	4	5
A	10	12	14	16	18

A[4] = 30

add(A[i]) = $L_0 + (i-1) * w$

$= 10 + (4-1) * 2$

$= 10 + 3 * 2 = 16.$

2D Arrays:

{In 2D Array compiler can follow row major / column major mapping}

eg int A[3][4];

A	0	1	2	3
0	100	102	104	106
1	108	110	112	114
2	116	118	120	122

3x4

Array always created in single dimension, in memory because memory is storing in linear, & to having single integer address.

row major

a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}	a_{20}	a_{21}	a_{22}	a_{23}
100	102	104	106	108	110	112	114	116	118	120	122

$A[2][1] = [L_0 + 2 * 4 + 1] * 2$

$= 100 + 9 * 2.$

add [A[i][j]] = $L_0 + [i * n + j] * w$ (indicates 0)

$= 100 + 2 * 4 + 1 * 2.$

$= (100 + 9) * 2 = 100 + 18 = 118.$

$A[i][j] = L_0 + [(i-1) * n + (j-1)] * w$ indicate '1'

Column major order:-

a_{00}	a_{10}	a_{20}	a_{01}	a_{11}	a_{21}	a_{02}	a_{12}	a_{22}	a_{03}	a_{13}	a_{23}
100	102	104	106	108	110	112	114	116	118	120	122

formula

$Add[A[i][j] = L_0 + [j * m + i] * w$ indicates '0'.

$Add[A[i][j] = L_0 + [(j-1) * m + (i-1)] * w$.

$Add[A[1][2] = L_0 + [2 * 3 + 1] * 2$

$= 100 + 7 * 2 = 114$

address of particular element in memory

UNIT-4 Simple Code Generator:-

1) Generates target code for a sequence of three address statements.

2, For each operator in a statement, there is a corresponding target language operator.

Register & Address Descriptors:

Register Descriptors:-

Keeps track of what is currently in each register.

- Initially all registers are empty. R_0 [ ]

2, **Address Descriptor:-** Keeps track of the location where the current value of the name can be found.

- Location may be register, a stack location or memory address. [a [ ]

A Code Generation Algorithm:-

For each three address statement of the form $x = y$ op z.

(i) Invoke a function get reg to determine the location L, where result of. y op z shall be stored.

getreg ⇒ empty register & name can be stored in memory L.

(name not currently used) Occupied register → memory location [L]

2, consult address descriptor for y to determine y, the current location of y. If y is not already in L, generate MOV y, L.

y < mem, reg

$x = y + z$. $L = R_0$.

MOV y, R_0

3, Generate the instruction op z, L. update address descriptor of x to indicate that x is in L. If L is a register, update its descriptor to indicate that it contains the value of x.

$x = y + z$ $L = R_0$

MOV y, R_0
ADD z, R_0

R_0 . y+z

If y & z have no next uses and not live on exit, update the descriptors to remove y & z

x is live on exit MOV R_0, x.

example:

$d = (a-b) + (a-c) + (a-c)$.

Three address code sequence

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$d = t_3 + t_2$

↳ live on exit

Statements	Code generated	Register descriptor	Address descriptor
		Registers are empty.	
$t_1 = a - b$	MOV a, R_0 SUB b, R_0	R_0 contains t_1	t_1 in R_0
$t_2 = a - c$	MOV a, R_1 SUB c, R_1	R_0 contains t_1 R_1 contains t_2	t_1 in R_0 t_2 in R_1
$t_3 = t_1 + t_2$	ADD R_1, R_0	R_0 contains t_3 R_1 contains t_2.	t_2 in R_1 t_3 in R_0
$d = t_3 + t_2$	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory.

Symbol table:

1) Symbol tables are data structures that are used by compilers to hold information about source program constructs.

→ It is used to store information about the occurrence of various entities such as.

objects, classes variable name, function etc.

It is used by both analysis phase & synthesis phase
Front end. Back end

LA SA SA IC IC CG IN.

The symbol table used for following purposes:

1) It is used to store the names of all entities in a structured format at one place.

2) It is used to verify if a variable has been declared

3) It is used to determine the scope of a name.

4) It is used to implement type checking by verifying assignments & expressions in the source code are semantically correct or not.

A symbol table can either be linear or hash table.

It mains the entry for each name as
<symbol name, type, attribute>.

eg <static, int, salary>.
Symbol table stores an entry in this format.

Use of symbol table:

1) A symbol table information is used by the analysis and synthesis phases.

2) To verify that used identifiers have been defined (declared).

3) To verify that expressions and assignments are semantically content - type checking.

4) To generate intermediate or target code

UNIT-5 Directed acyclic graph

DAG - optimization of Basic Block (transformation on Basic blocks)

1) Structure preserving transformations.

2, Algebraic transformations

I Structure preserving transformations.

1) Common subexpression elimination

if it previous computed, value cannot be changed

x + y - w (new computing)

x = y+z	x = y+z	y+z-w
y = x-w	y = x-w	y+z-y+
z = y+z	z = y+z	
w = x-w.	w = y+z	

2) Dead code elimination

a = a+2 → b = b+c

b = b+c

↓ ⟹ b = b * c

b = b * c   c = b+2.

c = b+2

3) Renaming of temporary variables:

$t_1 = x * y$ ⟹ $t_1 = x * y$

$t_2 = z - t_1$ ⟹ $t_2 = z - t_1$

$t_1 = t_1 * w$ ⟹ $t_3 = t_1 * w$

$w = t_2 + t_1$ ⟹ $w = t_2 + t_3$.

4) Interchanging of statements.

$t_1 = x * y$ → $t_1 = x * y$

$t_2 = z - t_1$ → $t_3 = t_1 * w$

$t_3 = t_1 * w$ (independent) → $t_2 = z - t_1$

$w = t_2 + t_3$. → $w = t_2 + t_3$.

Algebraic transformations:

Basic block, completed eliminate.

x = x+0. ✗

x = x-0.

a = a*1 ✗

b = b/1

c = d ** 2 ⟹ pow(d, 2)

⇓

c = d * d, for improving the basic blocks

Global Data flow Analysis:-

→ To do code optimization & code generation.

Compiler:- collect information about the whole program & distribute it to each block in the flow graph.

Data flow equations:-

out[S] = gen[S] ∪ [In[S] − kill[S]]

out[S] = Info at end of S

gen[S] - Info generated by S

in[S] - Info enters at the beginning of S

kill[S] - Info killed by S

points & paths:-

Within B, there is a point between 2 adjacent statements B, → 4 points.

A path from p_1 to p_n is a sequence of points $p_1, p_2 \ldots p_n$ such that for each i between 1 and n−1, either

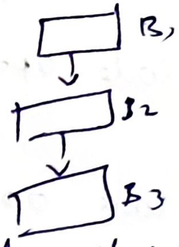
(i) p_i → point preceding the statement

p_{i+1} → p_i → end of block

p_{i+1} beginning of the successor block

Reaching definition:

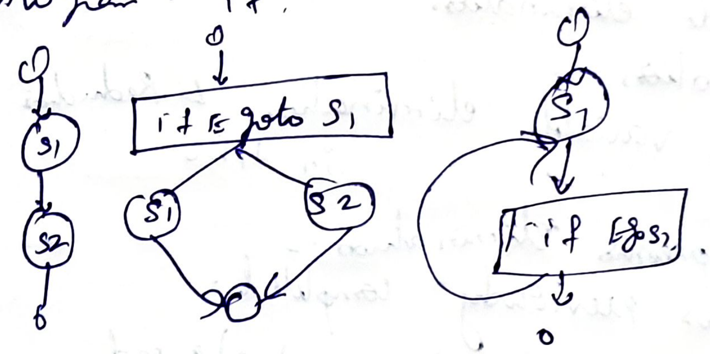
A definition of variable x is a statement that assigns a value of x.



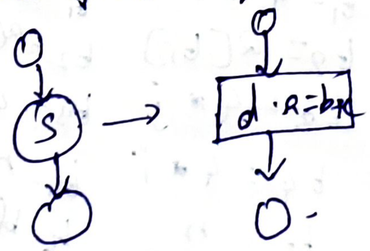
A definition of d reaches a point p, there is a path from d to p, such that d is not killed along the path.

Data flow analysis of structured program. if. dowhile.

a = 3
b = a+2
a = x+y
c = a+2



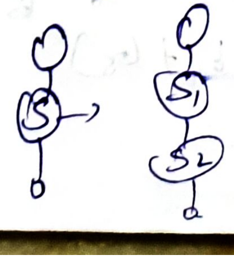
Data flow equation for reaching definition,



gen[S] = {d}

kill[S] = D_a − {d}

out[S] = gen[S] ∪ in[S] − kill[S])



gen[S] = gen[S2] ∪ (gen(S1) − kill[S2])

kill[S] = kill[S2] ∪ (kill[S1] − gen[S2])

in[S1] = in[S]

in[S2] = out[S1]

out[S] = out[S2]

Principal sources of optimization:

code optimization:- Improves the intermediate code
→ Less space & time

code optimization Technique

1. Common subexpression elimination
2. Constant folding
3. Copy propagation
4. Dead code elimination
5. Code motion
6. Induction variable elimination & Reduction in time

* preserves the meaning of the program

Common subexpression elimination:-
(i) if it was previously computed.
ii) values of variables have not changed

a = b+c	a = b+c	t_1 = 4 * i	t_1 = 4*i
b = a-d	b = a-d	t_2 = a[t_1]	t_2 = a[t_1]
c = b+c	c = b+c	t_3 = 4 * j	t_3 = 4*j
d = a-d	d = b.	t_4 = 4 * i	t_5 = n
		t_5 = n	t_6 = b[t_1] + t_5
		t_6 = b[t_4 + t_5].	

2> Constant folding:

value of a expression is constant, use the constant instead of expression

pi = 22/7, => 3.14.

Copy propagation:

f = g use g for f after f = g

x = a		x = a
y = x * b	=>	y = a * b
z = x * c		z = a * c

Dead code elimination.

variable is live, if its value can be used subsequently, otherwise it is dead

x = a	y = a * b
y = a * b	z = a * c.
z = a * c	

Code Motion:

moves code outside a loop.

```
while (j< 10)
{ x = y+z ;
  j = i+1 ;
}
```

```
x = y+z ;
while (i<10)
{ i = i+1 ;
}
```

loop invariant computation.

(loop control variable)
b) Induction variable Elimination &
Reduction in strength (Complex to simple)

```
i = 1;
while (i<10)
{ t = i * 4;
  i = i+1;
}
```

```
t = 4;
while (t<40)
{ t = t+4;
}
```

Loop optimization:
- Most execution time of a program is spent on loops.
- Decreasing the number of instructions in an inner loop improves the running time of a program.

Loop optimization techniques
1) code motion
2. Induction variable elimination & Reduction in Strength.

Loop unrolling:
duplicates the body of the loop Multiple times, inorder to decrease the number of times the loop condition is tested

```
for (i=0; i<100; i++)
{ display();
}
```

```
for (i=0; i<50; i++)
{ display();
  display();
}
```

4) Loop Jamming:
- combines the bodies of a adjacent loop that would iterate the same no. of times

```
for(i=0; i<100; i++)
  a[i]=1;
for (i=0; i<100; i++)
  b[i]=2;
```

```
for i=0; i<100; i++
{ a[i]=1
  b[i]=2
}
```

global

$gen[S] = gen[S_1] \cup gen[S_2]$
$kill[S] = kill[S_1] \cap kill[S_2]$

if else
$in[S_1] = in[S]$
$in[S_2] = in[S]$
$out[S] = out[S_1] \cup out[S_2]$

$gen[S] = gen[S_1]$

$kill[S] = kill[S_1]$

$in[S_1] = in[S] \cup gen[S_1]$

$out[S] = out[S_1]$.