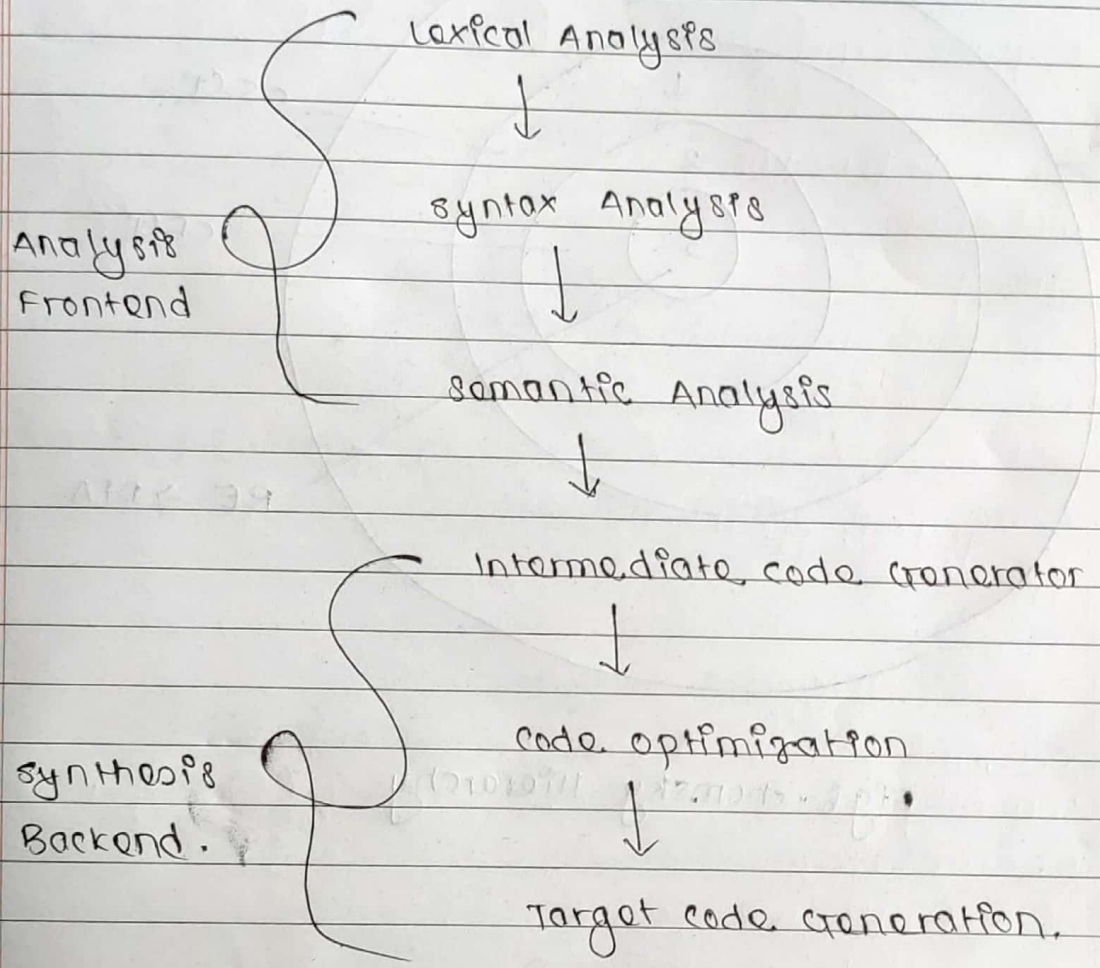
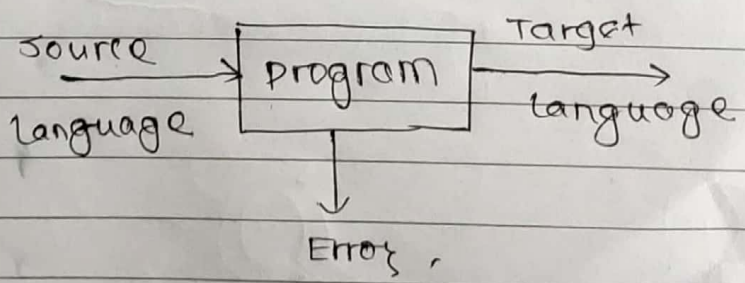
compiler:-


Lexical Analysis
Syntax Analysis
Analysis Frontend
Semantic Analysis
Intermediate code Generator
code optimization
Synthesis Backend.
Target code Generation.


Source
language
program
Target
language
Errors.

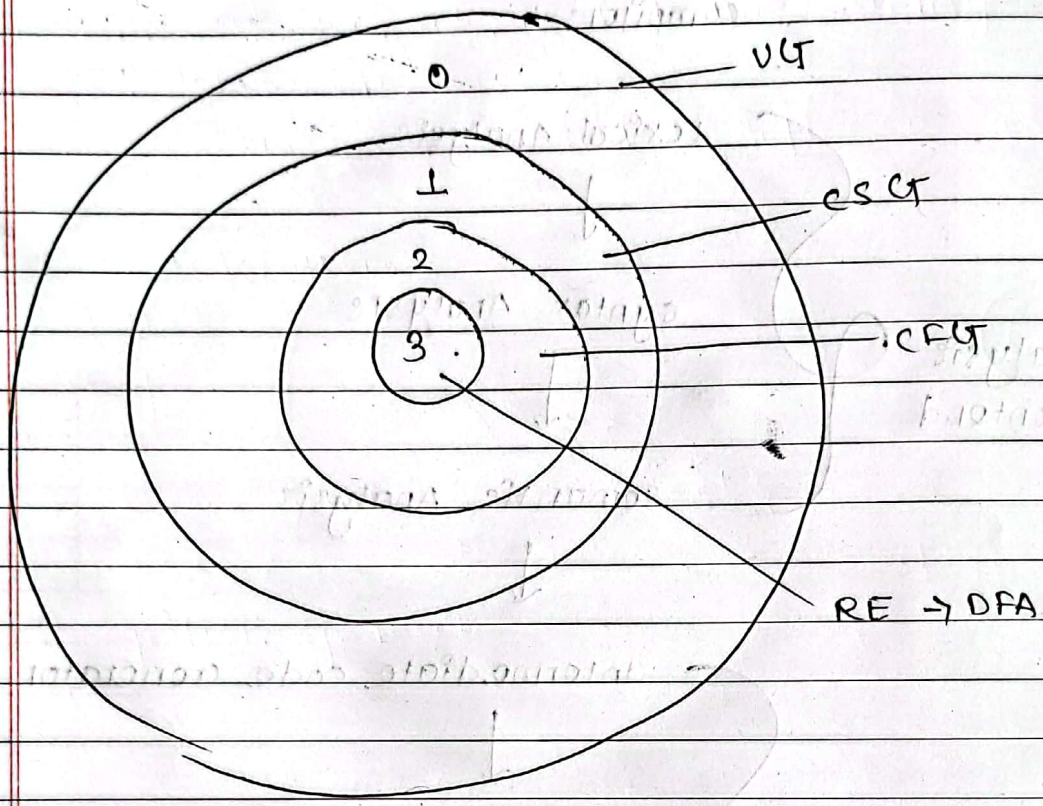
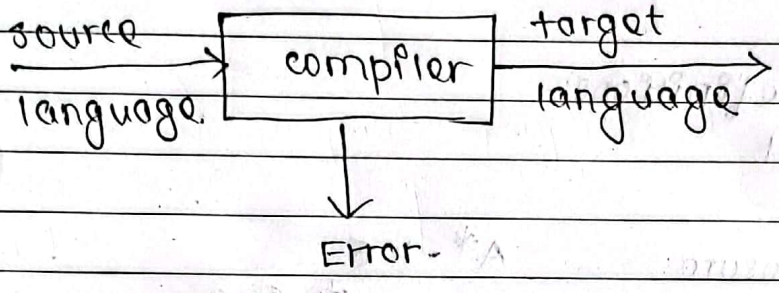


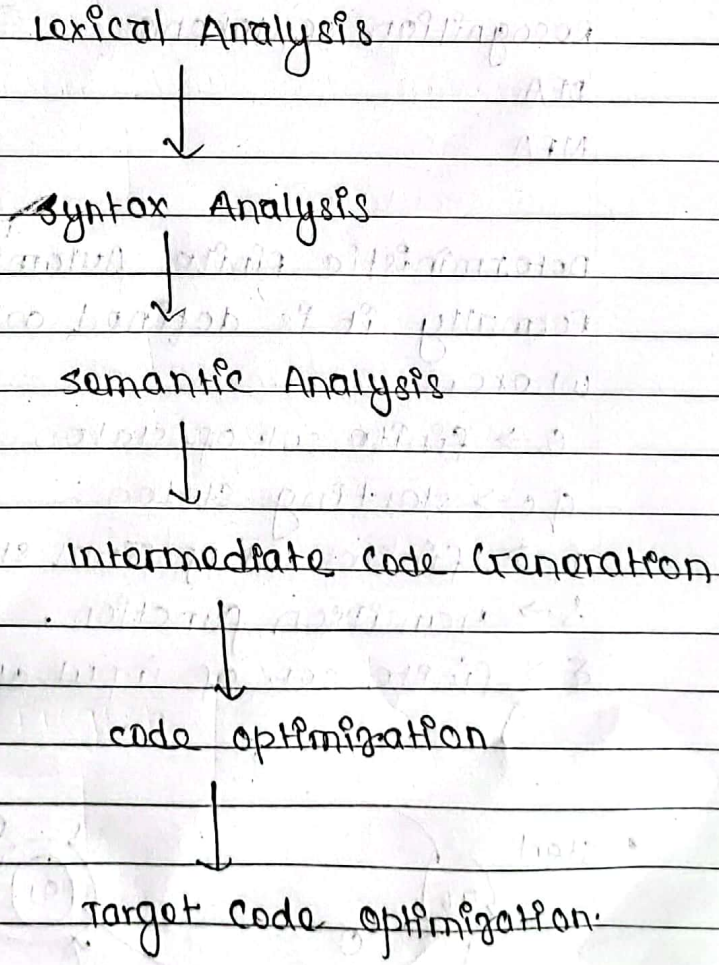
Fig :- chomsky Hierarchy

Unit:-1 Review of compiler structure:-

Compiler is a program that take the source language as input and produces target language as output.



phases of compiler.



Lexical Analysis.

Scans the input and tokenizes into a block of stream.

token, pattern and lexans.

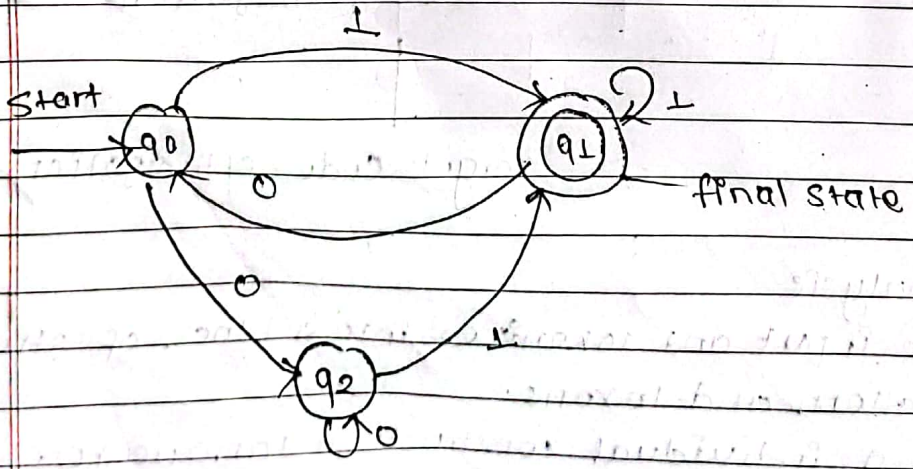
token is a individual construct a language.

patterns are the rule to test the validity of a token.
Valid tokens are lexemes.
Regular expression is used to define the patterns.

some definitions:-
alphabet
string
kleen closure $A^* \rightarrow \Sigma$
positive closure $A^+ \rightarrow A^* - \{\epsilon\}$

Recognition of tokens
DFA
NFA

Deterministic finite Automata (DFA)
formally it is defined as DFA = $(Q, \Sigma, q_0, F, \delta)$.
where,
$Q \rightarrow$ finite set of states
$q_0 \rightarrow$ starting states
$F \rightarrow$ finite set of final states
$\delta \rightarrow$ transition function, $\delta : Q \times \Sigma \rightarrow Q$
$\Sigma \rightarrow$ finite set of input alphabet.



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

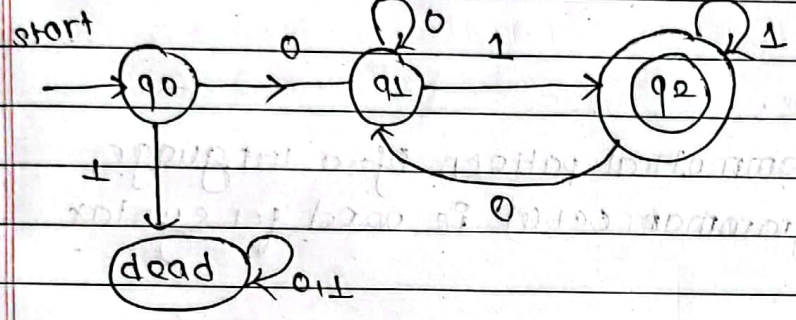
$q_0 = \{q_0\}$

$F = \{q_1\}$

$\delta = \delta(q_0, 1) = q_0$

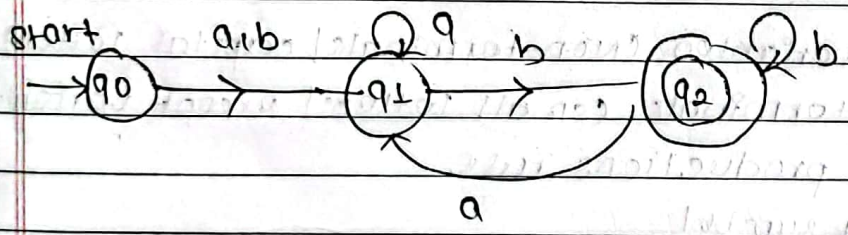
$\delta(q_0, 0) = q_2$

$0(0+1)^*1$

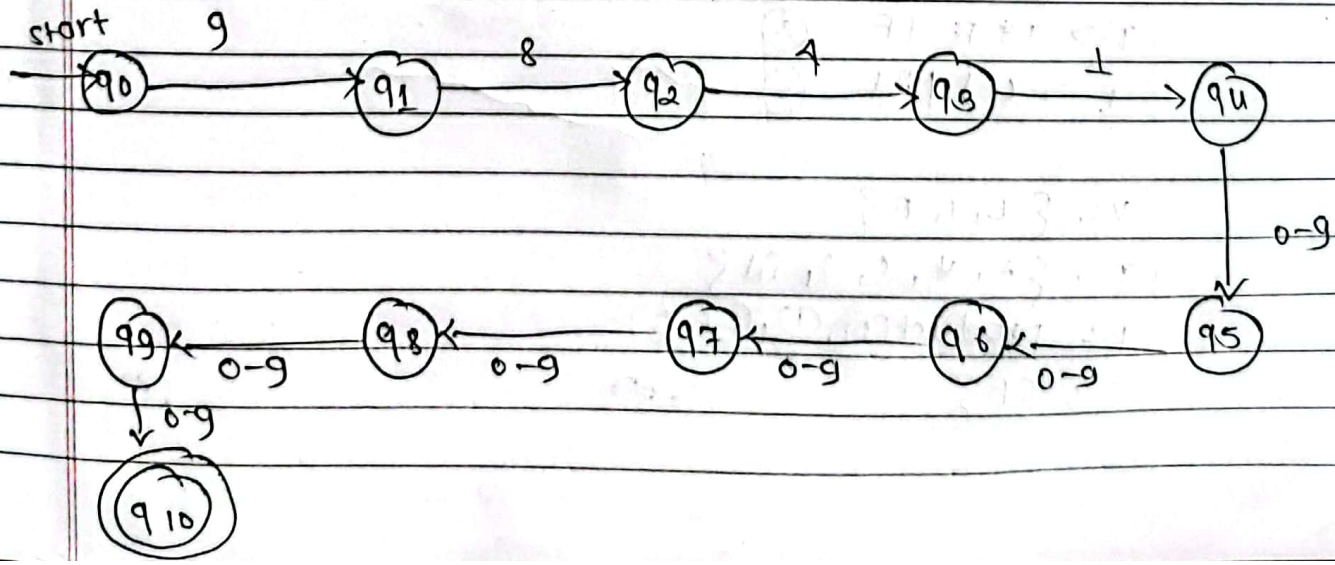


state.

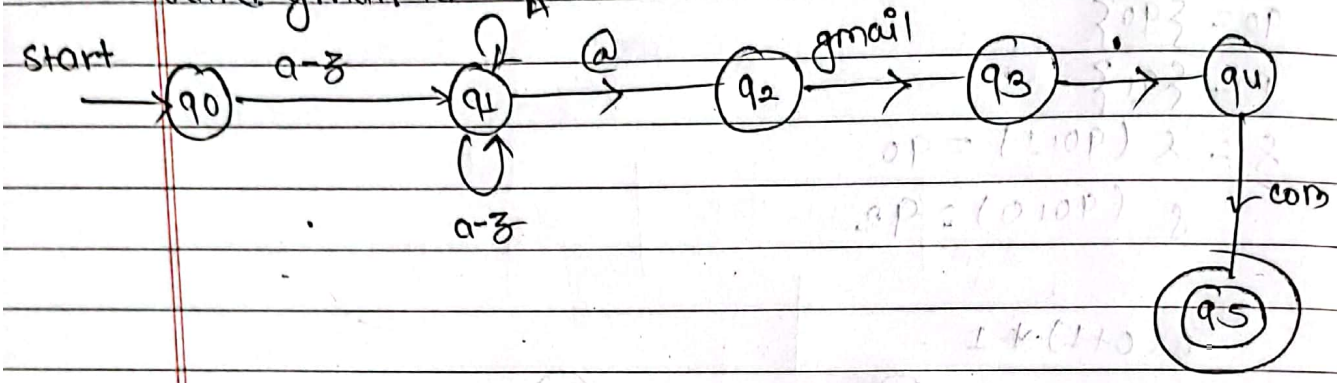
$(a+b)(a+b)^*b$



prepaid NTC mobile number.



Valid gmail id



Syntax Analyser.

* Define the grammatical pattern of a language.
* Context free grammar (CFG) is used for syntax definition.

CFG (Context Free Grammar)

It can be defined by $G = (V, T, P, S)$

where,

V = set of variables (Non-terminals/ capital letter)

T = set of terminals (small letters/ except variables)

P = set of productions rule

S = starting symbol

Example:

$E \rightarrow E+T \mid T$ ①

$T \rightarrow T*F \mid F$ ②

$F \rightarrow (E) \mid id$ ③

$V = \{E, T, F\}$

$T = \{+, *, (,), id\}$

P: production ①, ②, ③

$S = \{E\}$

parsing

Derive the string from grammar, to check whether that string is a language or not.

Types of parsing

1) Top down parser
2) Bottom up parser

} Leftmost derivation
Rightmost derivation.

Let, $w = id + id * id$

$E \to E+T$ $\quad \therefore E \to E+T$

$E \to T+T$ $\quad \therefore E \to T$

$E \to F+T$ $\quad E \to F$

$E \to id+T*F$ $\quad E \to T*F$

$E \to id+F*F$ $\quad T \to F$

$E \to id+id*id$ $\quad T \to id$

$w = id * (id+id)$

$E \to E+T$

$E \to E+F$

$E \to E+(E)$

$E \to E+(T)$

$E \to E+(T*F)$

$E \to E+(F*F)$

$E \to T+(id*id)$

$E \to F+(id*id)$

$E \to id+(id*id)$

$E \to E+T$

$E \to E+F$

$E \to E+(E)$ $\quad T*F+T$

$E \to F*(E)+T$

E

w = id * (id + id)

E → E+T

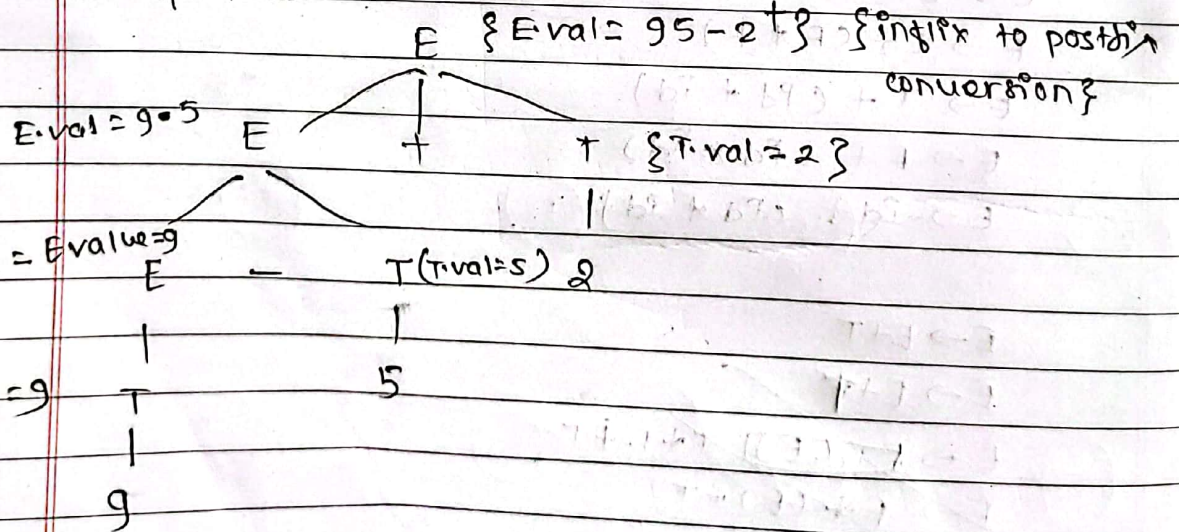
E →

Syntax Directed Translation

Defining the semantic action along with production rules in the grammar.

production	semantic action
E → E+T	E.value = E.val ‖ T.val ‖ "+"
E → E-T	E.value = E.val ‖ T.val ‖ "-"
E → T	E.val = T.val
T → 0	T.val = "0"
T → 1	T.val = "1"
⋮	⋮
T → 9	T.val = "9"

Example: 9-5+2



Semantic Analysis

concerns with the meaning of a language construct.

Example:- type checking

implicit / explicit type conversion.

Example :-

1. $A \rightarrow B + C$ { A.type == B.type == C.type }

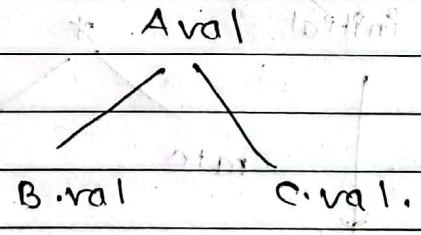
2. If (test) $S_1 \rightarrow A + B$ { if test == true S_1.type = A_1.type else C-D else S_1.type = C_1.type }

Synthesized and Inherited Attribute.

An attribute if depends upon child is synthesized attribute.

An attribute if depends on siblings or parent is called inherited attribute.

$A \rightarrow B + C$ $A.val = B.val + C.val$

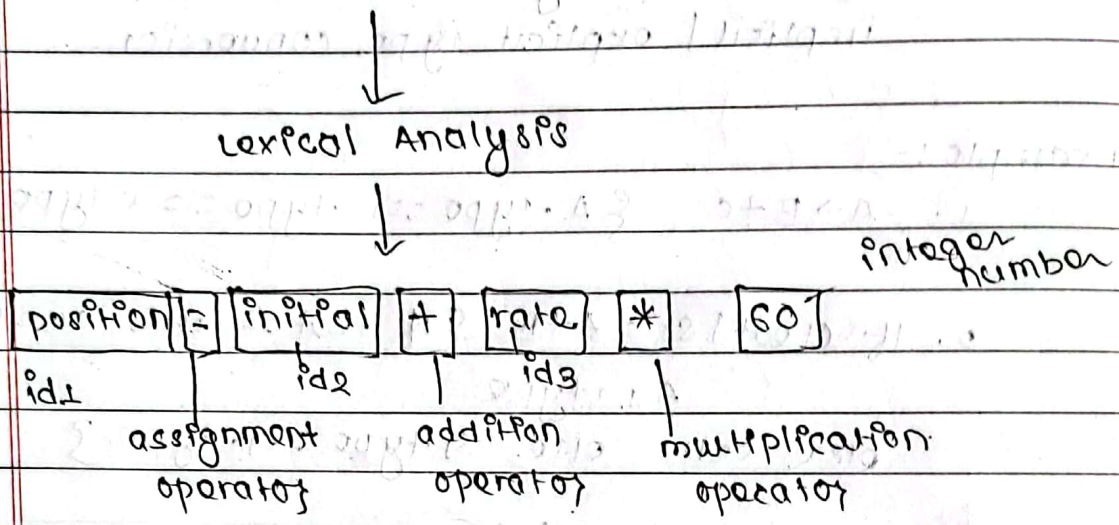


Example of synthesized attribute.

position = initial + rank * 60

Example:-

position = initial + rank * 60



Intermediate code Generation (three Address code)

↓

t1 = Int to real (60)
t2 = rate * t1
t3 = initial
tu = t3 + t2
position = t4

↓

code optimization.

t1 = Int to Real (60)
t2 = rate * t1
position = initial + t2

↓

Target code Generation.

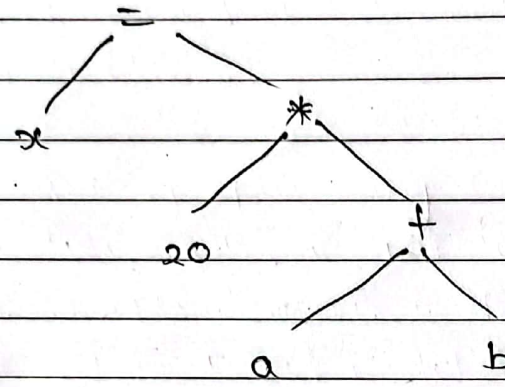
```
LOAD   F   R1 , #600
LOAD   F   R2 , rate
MUL    F   R1 , R2
LOAD   F   R3 , initial
ADD    F   R1 , R3
LOAD   F   position, R1
```

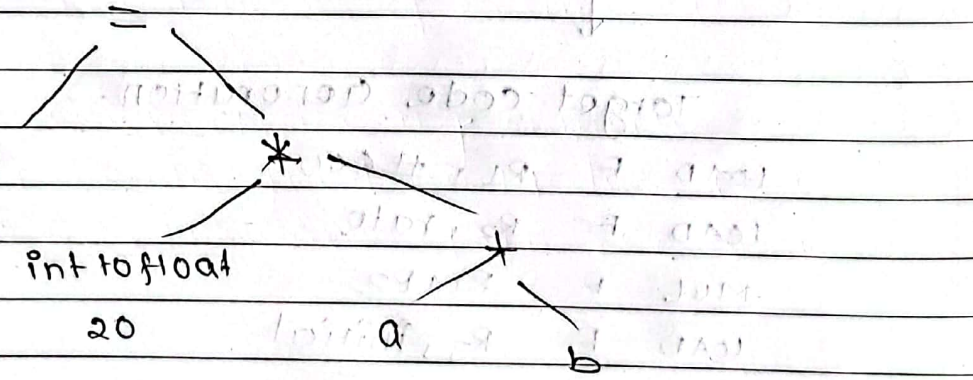
x = 20 * (a+b)

Ass. op op

x = 20 * (a + b)

id1 Integer id2 id3





Intermediate code generation.

t_1 = Int to real 20

t_2 = (a+b) a+b

t_3 = t_1 * t_2

x = t_3

```
t1 = int to float          t1 = a+b
t2 = a+b                   t2 = int to float
t3 = x                     t3 = x
                           t4 = t3+t1
                           x = t4
```

Importance of pipelining

Using a non-optimizing will usually result a non-efficient code when compared to more sophisticated compiles.

Example:-

```
int a,b,c,d
c = a+b;
d = c+1;
```

Fig:-1 C code.

```
LDW   a,R1
LDW   b,R2
ADD   R1,R2,R3

STW   R3,c
LDW   c,R3
ADD   R3,1,R4
STW   R4,d.
```

Fig:-2 SPARC code (7 cycle)

```
ADD a,b,c
ADD c,1,d
```

Fig:-3 Optimized SPARC code (2 cycle).

The non optimized SPARC code was 7 instructions, 5 of which are memory access instructions.

The optimized code is more efficient and use only 2 instructions.

structure of optimizing (There are two main modes)

1. structure code is translated into a low level code and all optimization is done on that form of code, called low level model of optimization.

2. The source code is translated to medium level intermediate code and the optimization is done, mixed model of optimization.

structure of optimizing compilers.
A compiler designed to produce fast object code includes optimizer components

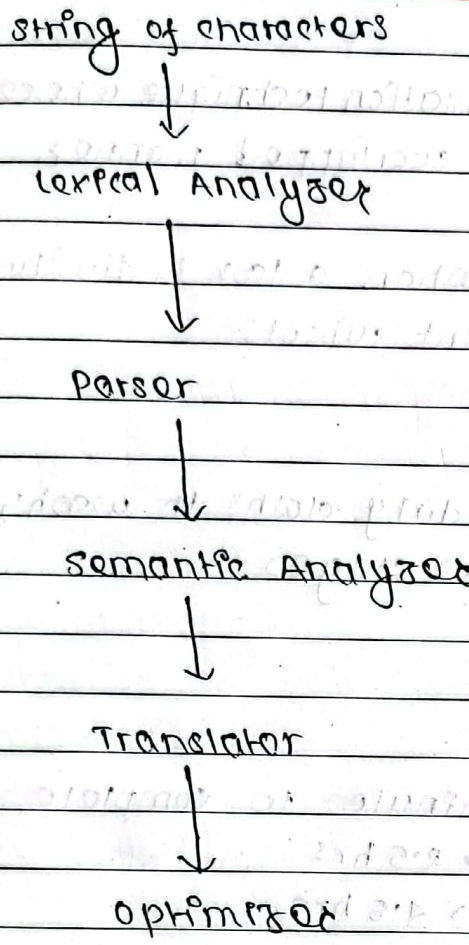


Fig 1:- Low level model

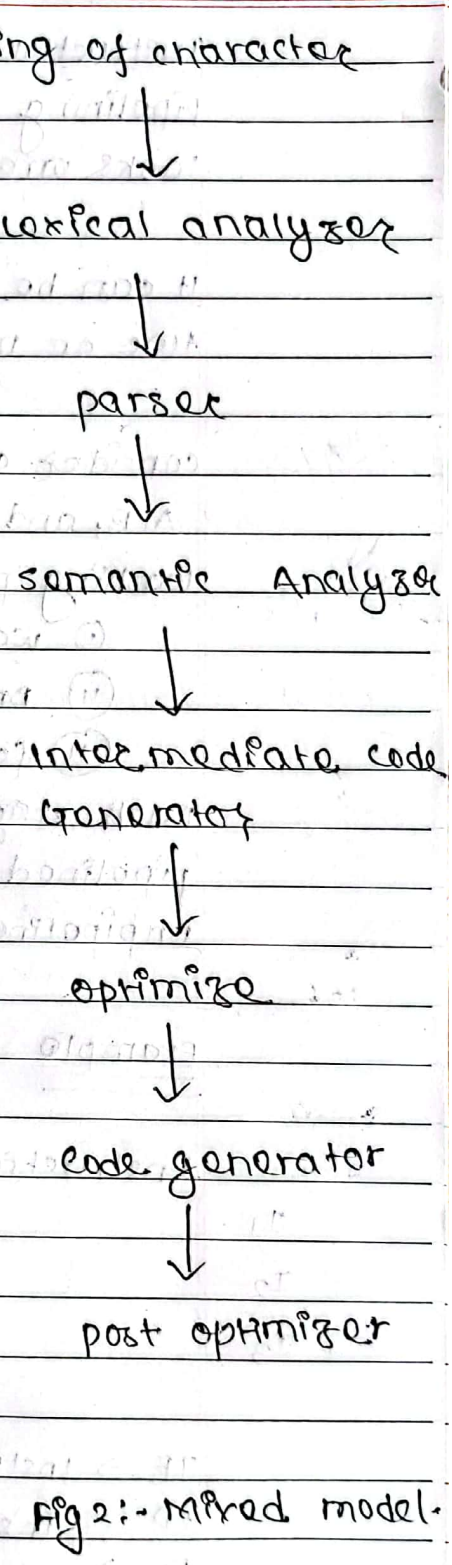


Fig 2:- Mixed model.

Instruction pipelining

Pipelining is the implementation technique where multiple tasks are performed in overlapped manner.

It can be implemented when a task is divided into two or more independent subsets.

Consider a scenario:-

A, B, and C each have dirty clothes to wash; the washing process has three stages.

(i) washing
(ii) drying
(iii) folding

Each stage takes 30 minutes to complete.

pipelined technique → 2.5 hrs

unpipelined technique → 4.5 hrs.

Example

Instructions	①	②	③	cycles ④	⑤	⑥	⑦
I_1	IF	ID	EX	MEM	WB		
I_2		IF	ID	EX	MEM	WB	
I_3			IF	ID	EX	MEM	WB

IF → Instruction Fetch

ID → Instruction Decode

EX → Execution

MEM → Memory access

WB → Write Back

Pipeline Hazards

condition or situation which does not allow the pipeline to operate normally.

Reduce the performance

Types of Hazards.

(a) control Hazard
(b) structural Hazard
(c) data Hazard

(a) control Hazard

- All instructions, who changes the program counter leads to control hazard.
- when to make a decision based on the results of one instruction while other are executing.

simulating pre-processor of a compiler.

```
#define PI 3.14
void main()
  {
  int PI, a,b
  float area = PI * r * r;
  printf ("PI = %of", PI);
  }
```

Fig:- Input file

```
void main()
{
int PI, a,b;
float area = 3.14 * r * r
printf ("PI = %of", 3.4)
{
```

Fig:- output file.

Data Hazards

- occurs when the pipeline changes in order of read/write access to operands so that the order differs from the order seen by sequentially executes instructions on the unpipelined machine types :-

1. RAW (Read After Write)
2. WAR (Write After Read)
3. WAW (Write After Write)

RAW

I1 : R2 = R2+R3

I2 : R5 = R2+R4

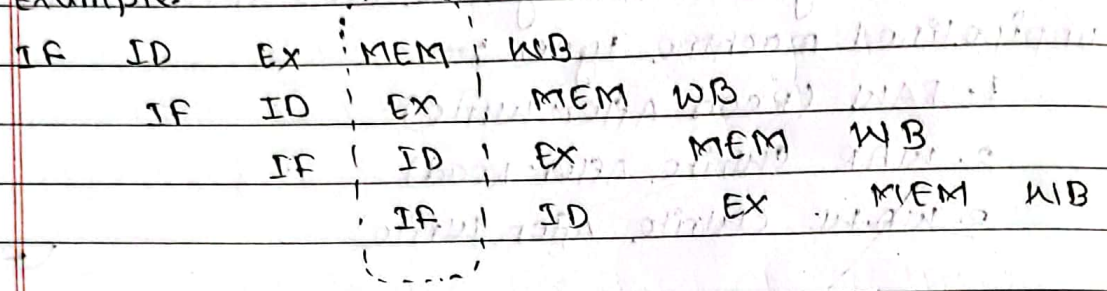
Phases	①	②	③	④	⑤
I1 :	IF	ID	EX	MEM	WB
I2 :		IF	ID	EX	MEM WB

(write back the data 5th phase)

I2 (fetch data in 3rd phase)

Structural Hazard
- Different instructions accessing same resource.

Example:



Multiple Issue Processor
A processor that can sustain more than one instruction per cycle

CPI (Instruction per cycle)

If CPI = 1 means one instruction per cycle

If we want to reduce CPI < 1, then we have to look out issuing multiple instructions.
i.e If two instructions are in one cycle,

$$CPI = \frac{1}{2} = 0.5 < 1$$

Types of multiple issue processor.
two variants
1. Superscalar processor
2. VLIW (very large instruction word)

Superscalar Processor
refer the machine that is designed to improve the performance of the execution of scalar instructions

Example:-

I_1	IF	ID	EX	MEM	WB
I_2	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB.

Assignment 1:-

VLSM

Processor parallelism

Due date
2079-08-15

Unit:-2 dependence and its properties

- For a given sequential program, a collection of the statement to statement execution ordering that can be used as a guide to select and apply transformations, that preserve the meaning of the problem.
- Each pair of statements in the graph is called a dependence.

Dependency represents two kinds of constraints:-

1. Data dependence
2. control dependence

Data dependence

Ensure that data is produced and consumed in correct order.

Example:-

S_1 : PI = 3.14

S_2 : R = 5.0

S_3 : Area = PI * R * R

- S_3 cannot be moved before S_1 or S_2.
- To prevent this we will construct data dependence from S_1 and S_2 to S_3
- No execution constraint between S_1 and S_2 because $\{S_1, S_2, S_3\}$ or $\{S_2, S_1, S_3\}$ both gives same result

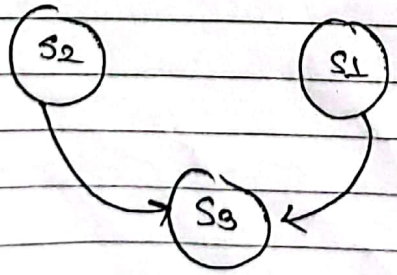


Fig:- Dependency in paths.

Lab no:-2

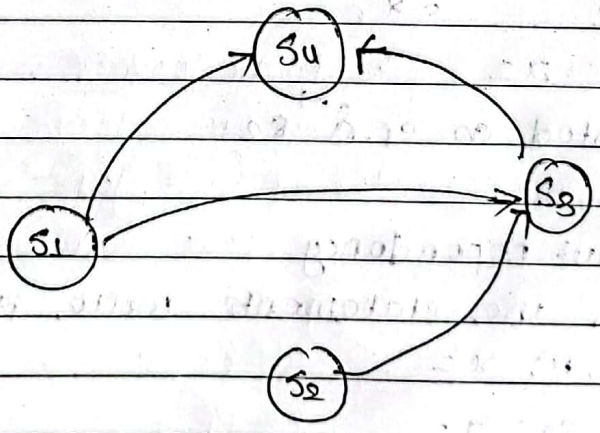
Control Hazard simulator.

I1 → 100
I2 → 102
I3 → 104
I4 → 106
I5 → 108
.
.
.
In

100
pc

S1 : a = 3+9
S2 : b = 5.0
S3 : c = a * b
S4 : d = a * c

Control Hazard :



Control Dependence:

Dependance due to control flow

1. S_1: if (CT == 0.0) goto S_3

S_2: A = A/T

S_3: CONTINUE

Here S_2 cannot be executed before S_1. divided by zero. Execution S_2 before S_1 could cause ~~control hazards~~.

Load store classification

1. True Dependance

The first statement stores into a location that is later read by the second statement.

S_1: x =

S_2: ... x

denoted by S_1 δ S_2

2. Anti Dependence

The first statement reads from a location into which the second statement later stores.

S_1: ... = x

S_2: x = ...

Denoted as S_1 δ^{-1} S_2

3. Output dependency

Both the statements write it into same location

S_1: x = ...

S_2: x = ...

Denoted by S_1 δ^{0} S_2

Example:

S_1 : x = 2

S_2 : - -

S_3 : x = 2

S_4 : w = x * y

Find dependency type between S_1, S_2 and S_3.

```
for (i=1; i<=n; i++)
for (j=1; j<=n; j++)

    S1: X[i,j] = A[i,j] + Y[i,j]
    S2: Y[i,j] = C[i,j] + X[i,j]
    S3: X[i,j] = Y[i,j] + D[i,j]
```

S_1, S_2 ⇒ True dependence.

S_2, S_3 ⇒ Anti dependency

S_1, S_3 ⇒ output dependency

Dependence in loop

Let us look at two different loops.

```
DO I=1, N
S1: A(I+1) = A(I) + B(I)
END DO
```

$A2 = A1 + B1$

$A3 = A2 + B2$

Fig: 1

```
DO I = 1,N
S1: A(I+2) = A(I) + B(I)
END DO
```

$A3 = A1 + B1$

$A4 = A2 + B2$

$A5 = A3 + B3$

Fig: (2)

In both cases S_1 depends on itself. However in fig(1) depends on previous statement and in fig(2) depend in two previous statement.

Iteration number.

For an arbitary loop in which the loop for index I runs from L to U in steps of S, the iteration number i of a specific iteration is equal to the value (I - L+S)/S where I is the value of the index on that iteration.

Example

```
DO I = 0, 10, 2
   S1 <some statement>
END DO
```

For $I = 4$,

$$\frac{4 - 0 + 2}{3}$$

$$= 3$$

Iteration vector

Given a nest of n loops, the iteration vector i of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.

The iteration vector for $i = \{i_1, i_2, \ldots, i_n\}$ where i_k, $1 \le k \le n$ represents the iteration number for the loop at nesting level k.

Example:-

```
DO I = 1,2
  DO J = 1,2
    S1 < some statements >
END DO
```

The iteration vector $S_1[(2,1)]$ denotes the instance of S_1 executed during the second iteration of I and first iteration of J.

Iteration space.

The set of a possible iterations vectors for a statement.

Example

In above, the iteration space of S_1 is $\{(1,1), (1,2), (2,1), (2,2)\}$

Loop dependence.

There exists a dependence from statement S_1 to statement S_2 in a common nest of loops if and only if, there exist two iteration vectors i and j for the nest, such that,

1. $i < j$ or $i = j$ and there is a path from S_1 to S_2 in the body of the loop.
2. S_1 accesses memory location M on iteration i and S_2 accesses M on iteration j
3. One of those accesses is a write.

- an iteration vector i precedes iteration vector j iff any statement executed on the iteration described by i is executed before any statement on the iteration desired by j.

Transformations

- We call a transformation safe if the transformed program has the same meaning as the original program.

- Two computations are equivalent if, on the same inputs, they produce same outputs in the same order.

Reordering Transformations:
changes the order of execution of the code, without adding or deleting any executions of any statements. Since a reordering transformation does not delet any statement executes any two executions that reference a common memory element before a reordering transformation will reference the same memory element after that transformations. Hence, if there is a dependence between the statements before the transformations, there will also be one afterward.
i.e a recording transformation does not eliminate dependent.