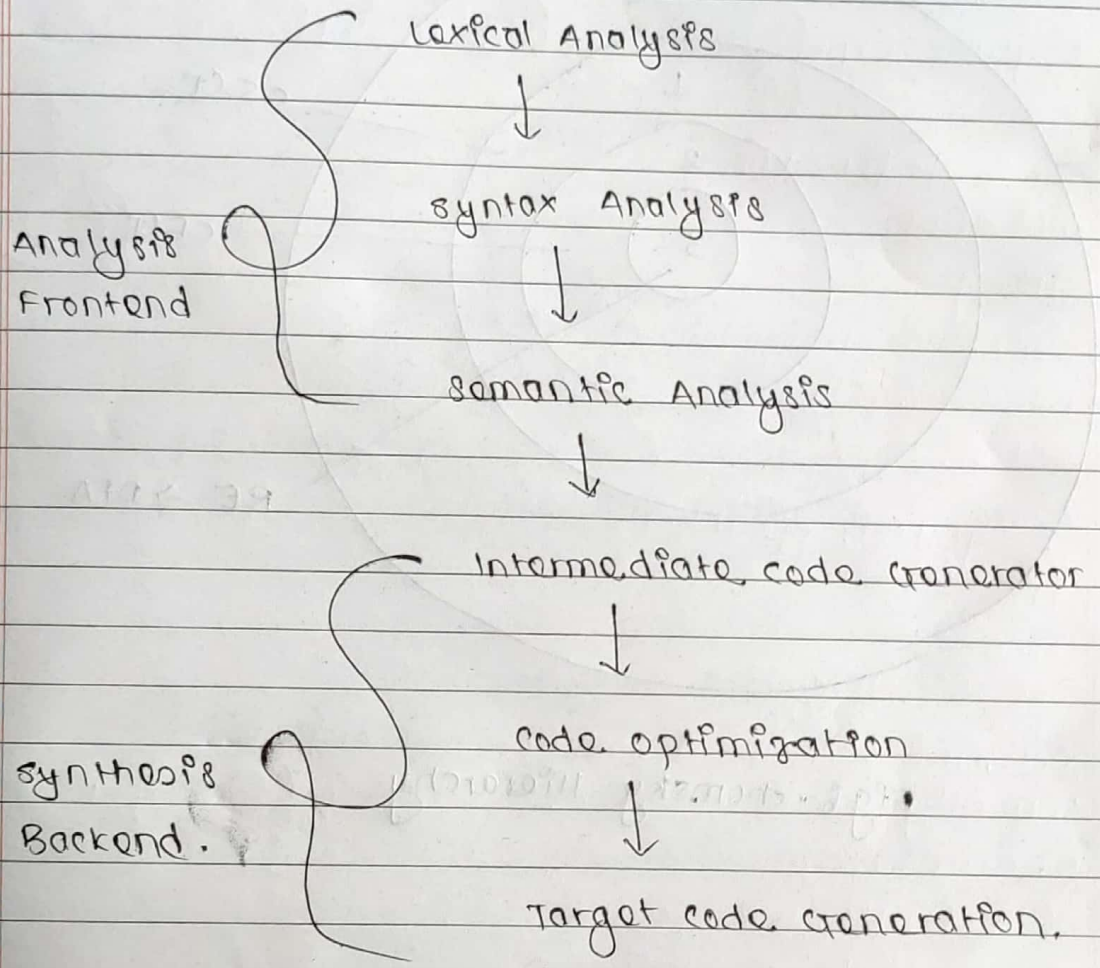
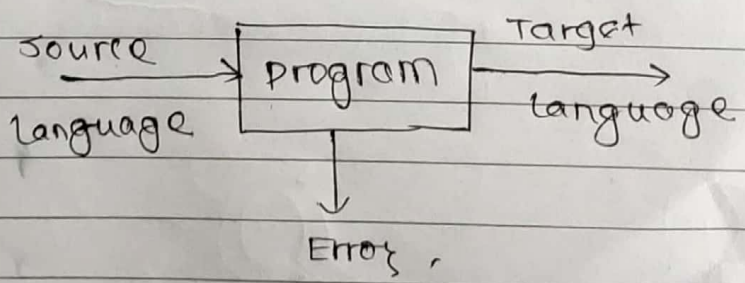
compiler:-


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


Source
language
program
Target
language
Errors.

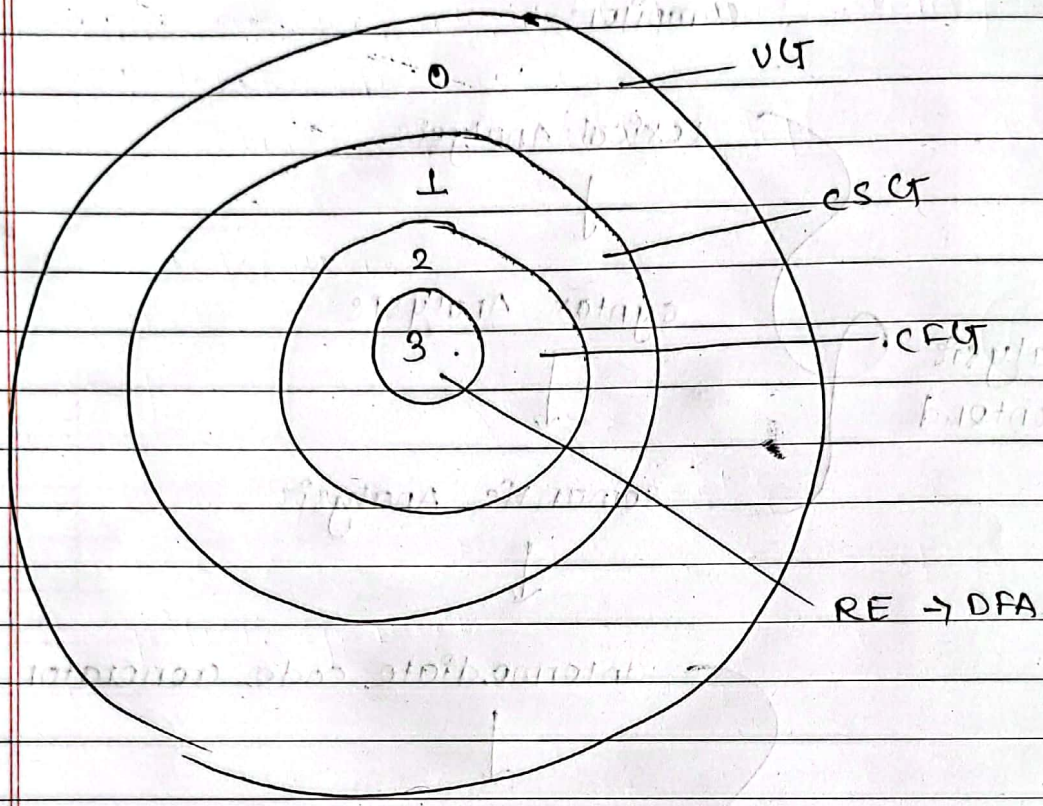
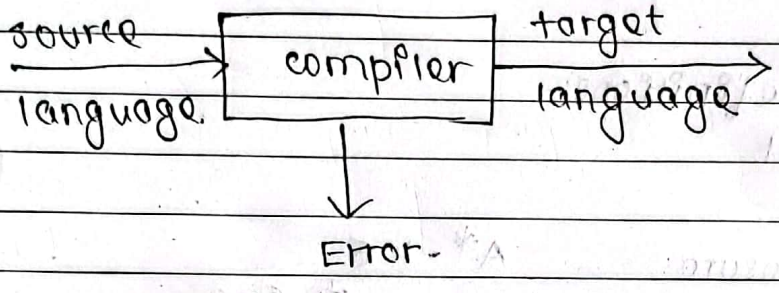


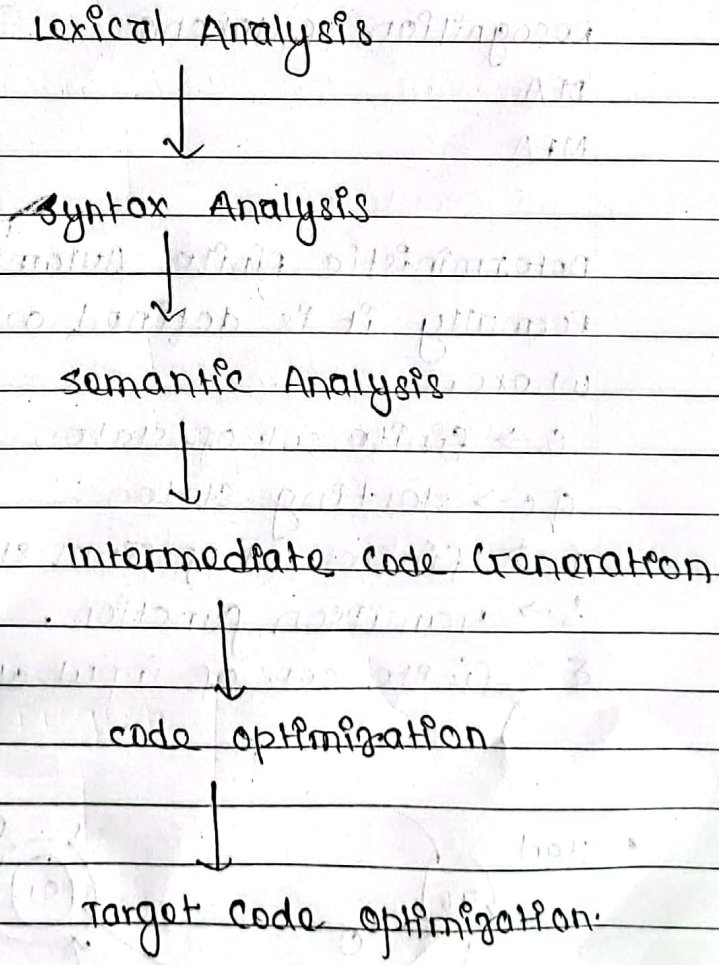
Fig :- chomsky Hierarchy

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.

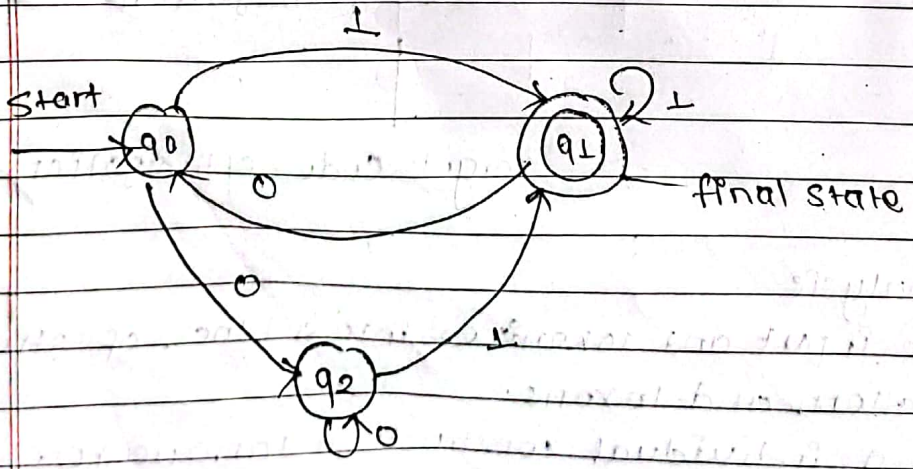
token is a individual construct a language.

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.



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

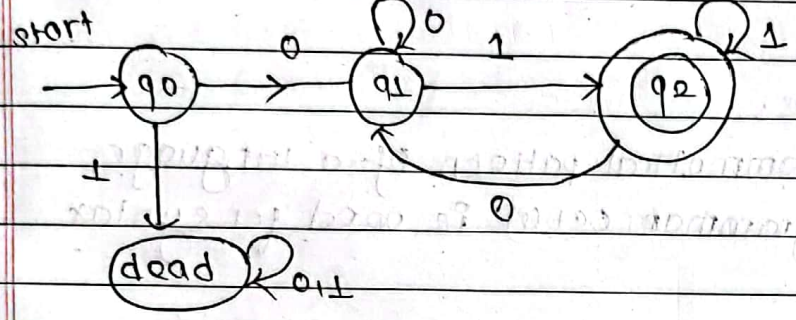
$q_0 = \{q_0\}$

$F = \{q_1\}$

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

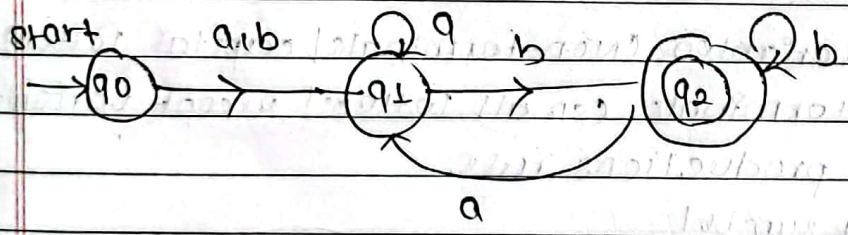
$\delta(q_0, 0) = q_2$

$0(0+1)^*1$

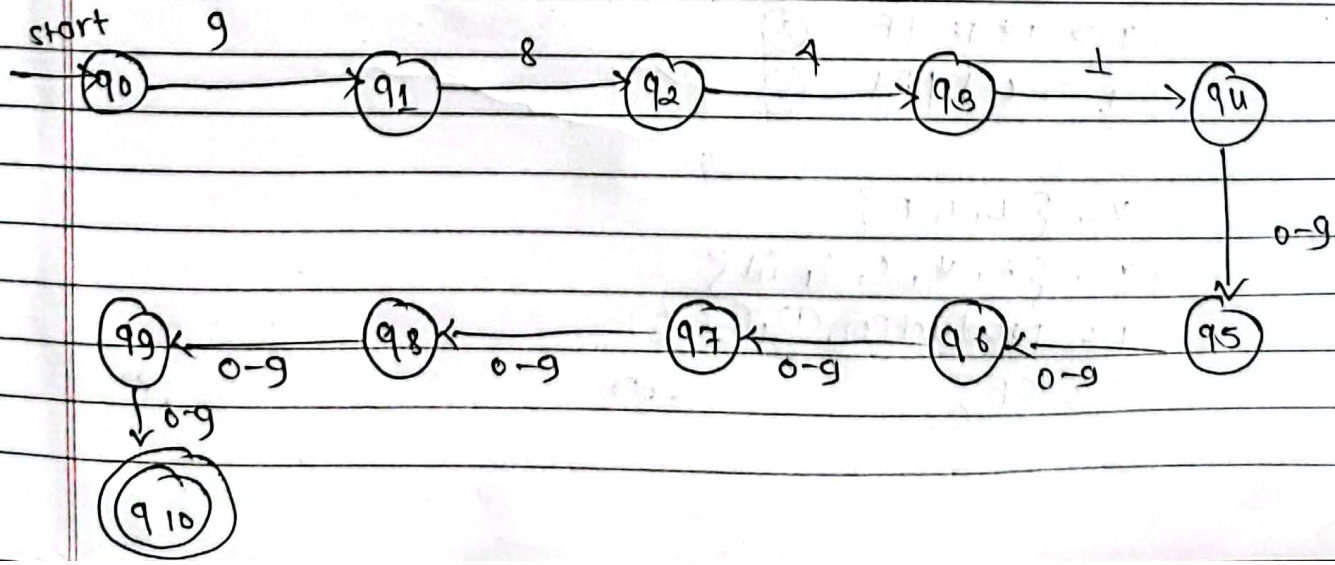


state.

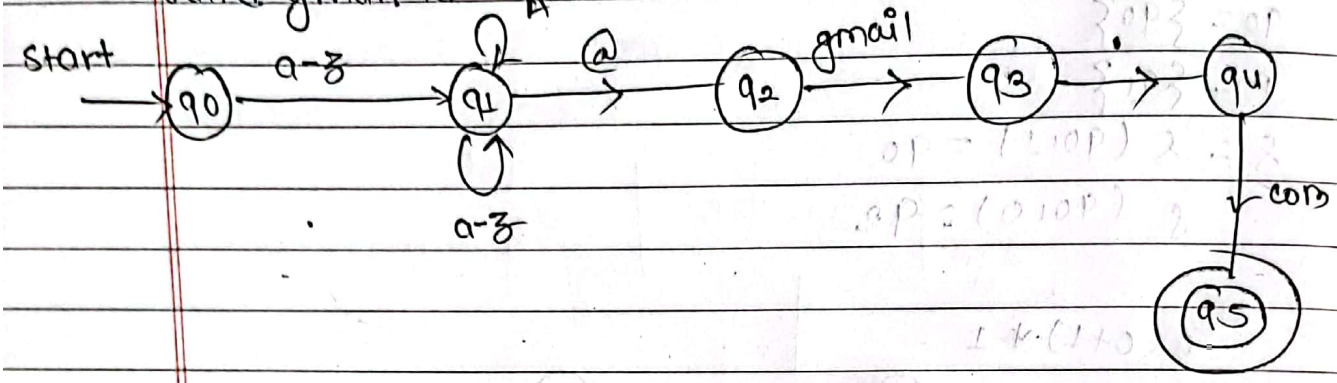
$(a+b)(a+b)^*b$



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

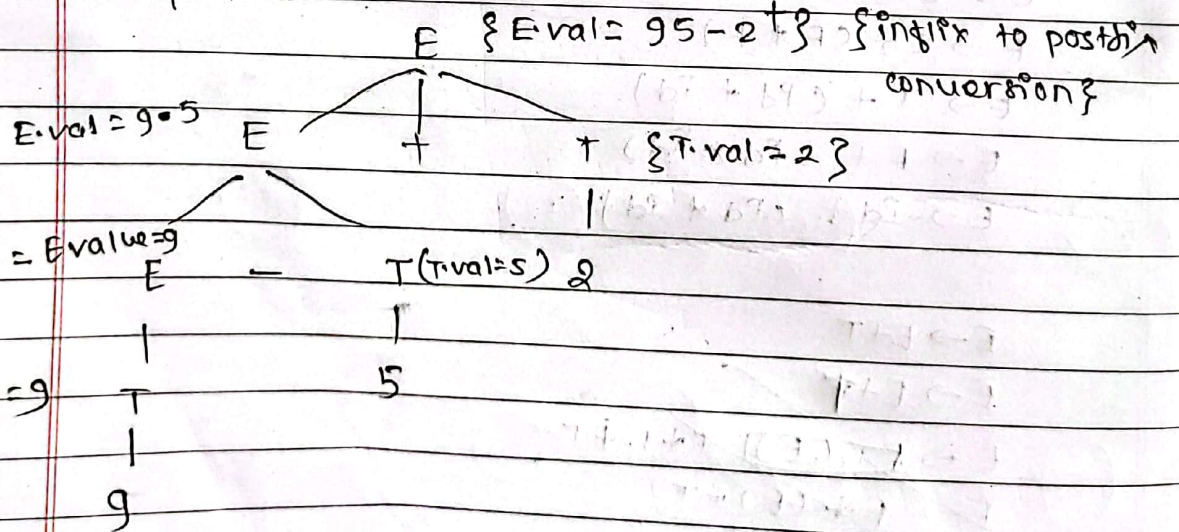
E →

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 }

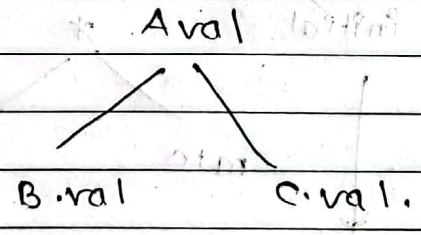
2. If (test) $S_1 \rightarrow A + B$ { if test == true S_1.type = A_1.type else C-D else S_1.type = C_1.type }

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.

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

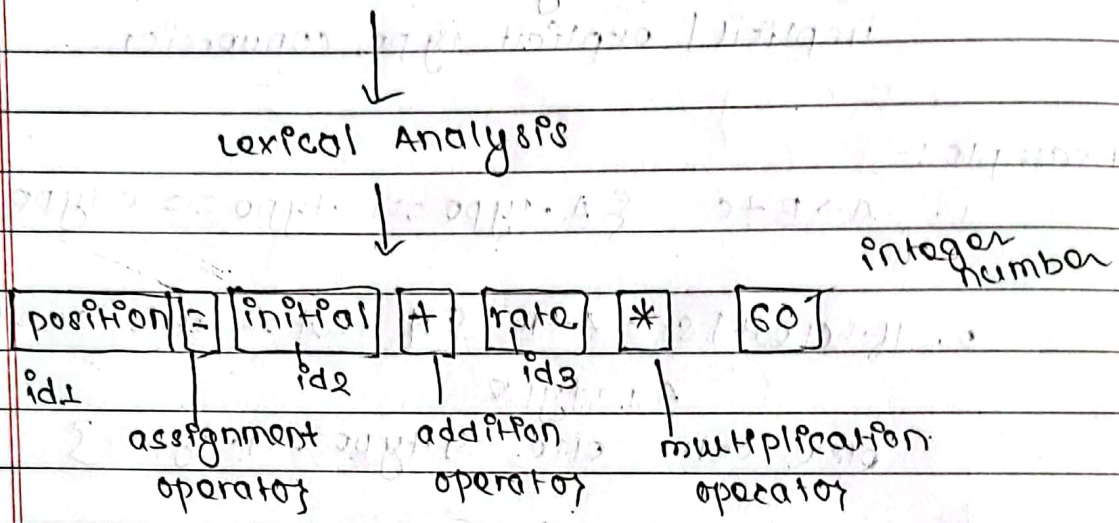


Example of synthesized attribute.

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.

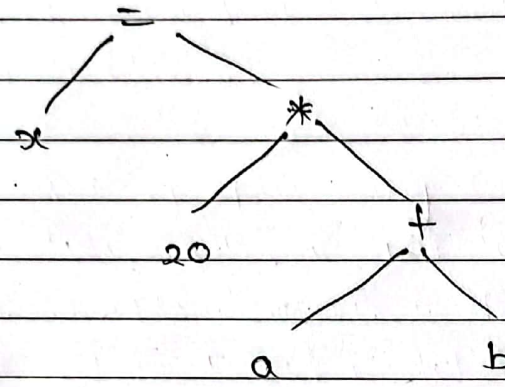
```
LOAD   F   R1 , #600
LOAD   F   R2 , rate
MUL    F   R1 , R2
LOAD   F   R3 , initial
ADD    F   R1 , R3
LOAD   F   position, R1
```

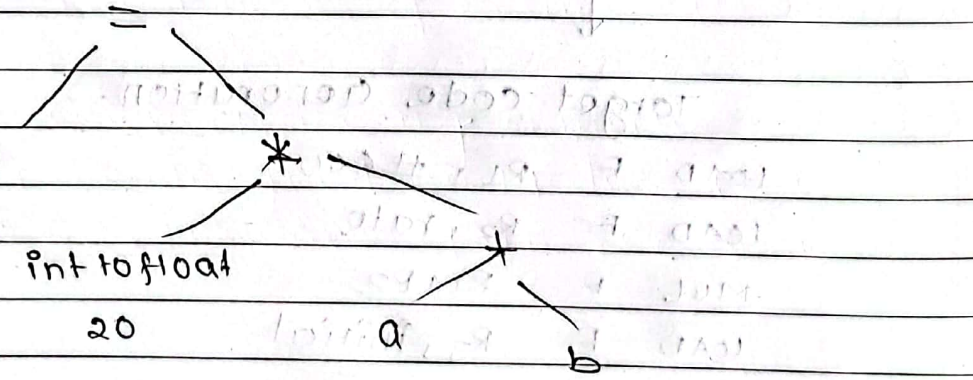
x = 20 * (a+b)

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 models)

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

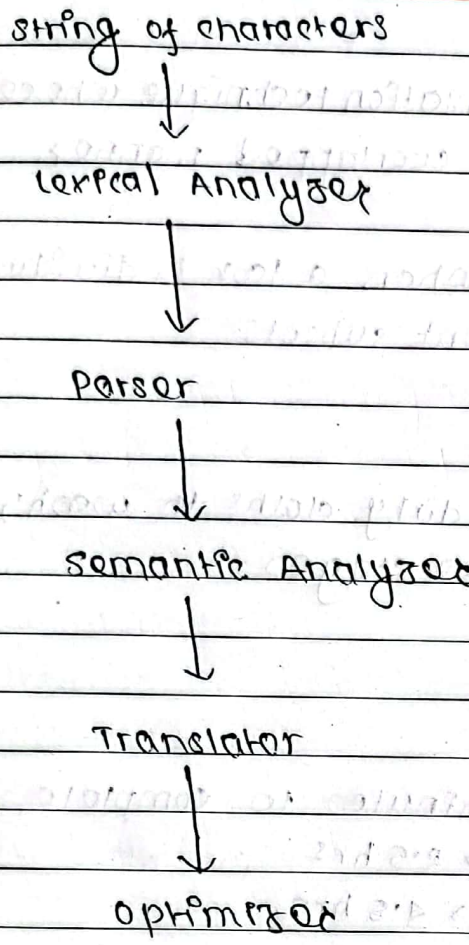


Fig 1:- Low level model

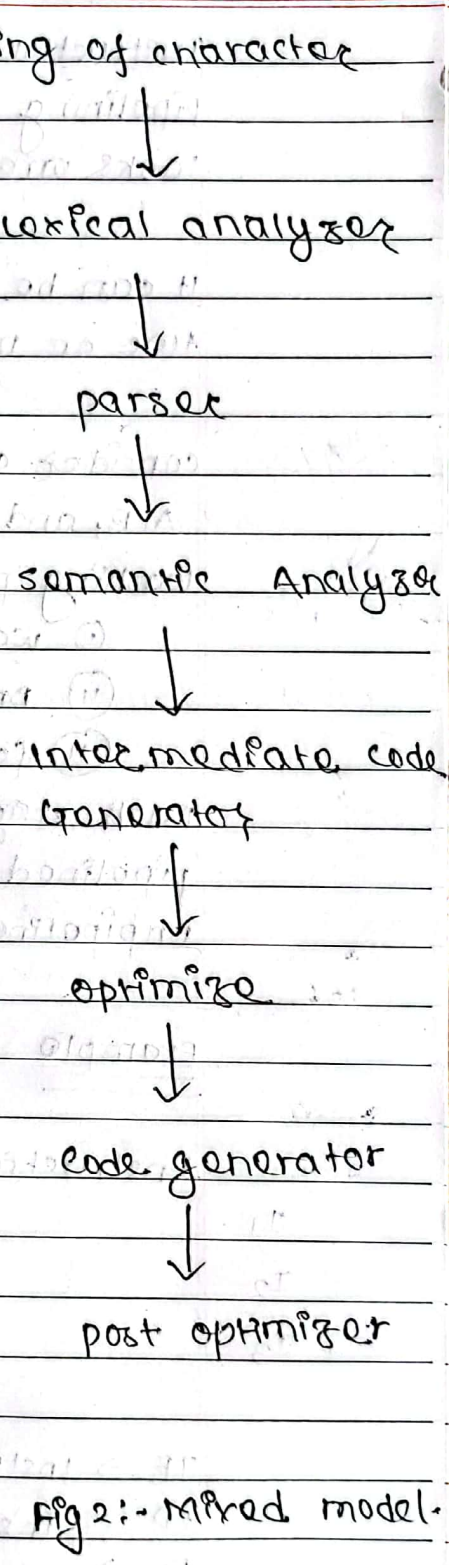


Fig 2:- Mixed model.

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

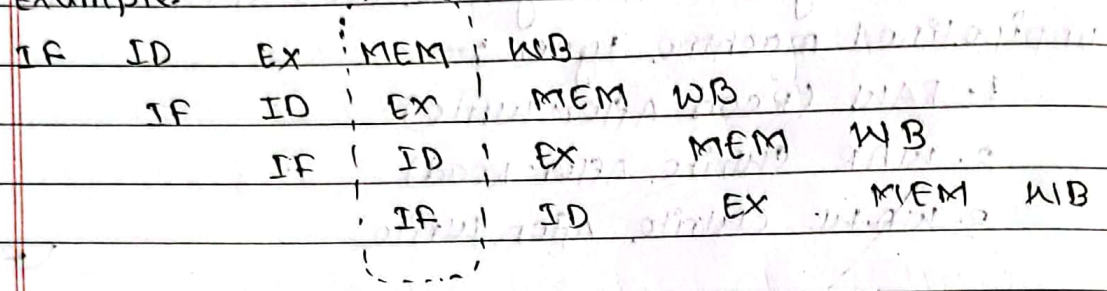
Phases	①	②	③	④	⑤
I1 :	IF	ID	EX	MEM	WB
I2 :		IF	ID	EX	MEM WB

(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

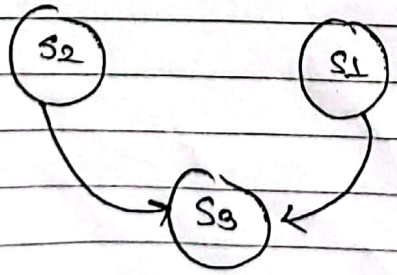


Fig:- Dependency in paths.

Lab no:-2

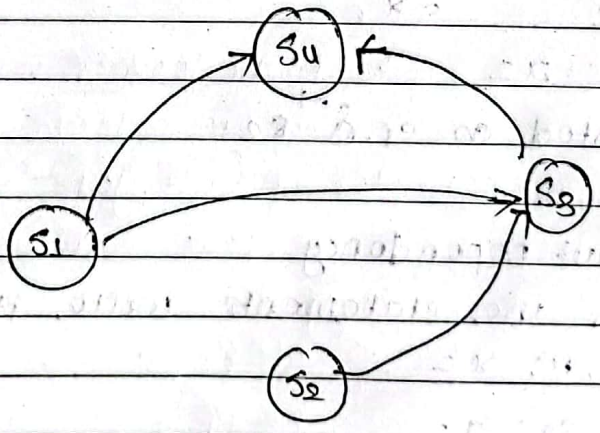
Control Hazard simulator.

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.

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 delete any statement executions, any two executions that reference a common memory element before a recording transformation will also 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 reordering transformation doesnot eliminate dependences.

Fundamental Theorem of Dependence

Statement

- A transformation is said to be valid for the program to which it applies if it preserves all dependencies in the program.

Proof

- Assume a program with loop and conditional statement free.
- Let $S_1, S_2, \ldots, S_n$ be the execution order in the original program.
- Let $\{i_1, i_2, \ldots, i_n\}$ be the permulation of the statement that represent the order of execution of statement of transformed program.
- Let it be proved by contradiction i.e the sequence of statements in the transformed program contains at least one statement that produce an incorrect result.
- Let S_k be the first statement in the new order that produce an incorrect output.
- Since all the statements that have executed before S_k produce the same value they did in the original program, there are only three ways that S_k can see an incorrect input in a specific location M

Case 1: (True Dependence)

- a statement S_m that originally stored its result in M before it was read by S_k, now stores into m after it is read by S_k ($S_m \delta S_k$)

Case 2: (Anti Dependence)

- a statement S_m that originally stored into m after it was read by S_k now writes M before S_k reads it ($S_m \delta^{-1} S_k$)

Case 3: (Output Dependence)

- two statements that both write into M before it was read by S_k ($S_m \delta^o S_k$)
- Since this exhausts the way that S_k can get the wrong value, the result is proved by contradictio

Distance Vector

- Suppose that there is a dependence from statement S_1 on iteration i of a loop nest to statement S_2 on iteration j, then the distance vector $d(i,j)$ is defined as a vector of length n such that

$$d(i,j)_K = \delta_K - i_K$$

Direction Vector

- Suppose that there is a dependence from statement S_1 on iteration i of a loop to statement S_2 on iteration j, then the direction vector $D(i,j)$ is defined as

$$D(i,j)_k = \begin{cases} "<" & \text{if } d(i,j)_k > 0 \\ "=" & \text{if } d(i,j)_k = 0 \\ ">" & \text{if } d(i,j)_k < 0 \end{cases}$$

eg.

```
for (i=1, i<=100; i++)
{
    for (j=1 ; j<=100, j++)
    {
        for (k=1, k<=200, k++)
        {
            A[i+1, j, k] = A[i, j, k+1] + B
        }
    }
}
```

- Let the iteration vector be {3, 8, 5}

S_i S_j

$S_1 \Rightarrow A[4,8,5] = A[3,8,6] + 5$

sink ← source

Distance vector

$= (4,8,5) - (3,8,6)$

$= (1, 0, -1)$

↓ ↓ ↓

< = > Direction vector

e.g.

```
for (i=1; i<=N; i++)
{
  for (j=1; j<=M; j++)
  {
    for (k=1; k<=l; k++)
    {
      A[i+1, j, k-1] = A[i,j,k] + 10
    }
  }
}
```

Find the distance and direction vector for iteration vector {1, 1, 1}

⇒ Let the direction vector be {1, 1, 1}

$S_1 \Rightarrow A[2,1,0] = A[1,1,1] + 10$

Distance vector : [2, 1, 0] − [1, 1, 1]

= (1, 0, −1)

Direction vector : (<, =, >)

Loop Carried Dependence

- dependence exists across iterations.
- i.e if the loop is removed, the dependence no longer exists.

```
DO  I = 1, N
S1 : A(I+1) = F(I)
S2 : F(I+1) = A(I)
END DO
```

Loop Independent Dependence

- dependence exist with an iteration.
- i.e if the loop is removed, the dependence still exists.

```
DO   I = 1, 10
S1 :  A(I) = ...
S2 : ... = A(I)
END DO
```

Example:

Find what type of loop dependence in following program.

1.
```
DO I = 1, 9
S1 : A(I) = ...
S2 : ... = A(10 - I)   ⇒ loop
END D DO
```

2.
```
For (i=1, i<n, i++)
{
    S1: A[i] = A[i-1] + 1;
    S2: B[i] = A[i];
}
```

$\Rightarrow S_1 \rightarrow S_1$ $\Rightarrow$ Loop carried Dependence

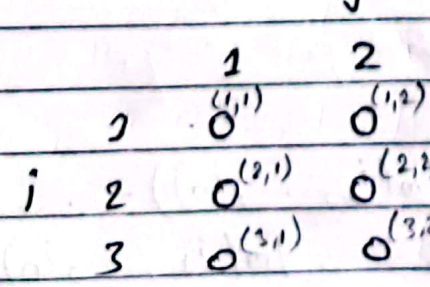
$S_2 \rightarrow S_2$ $\Rightarrow$ Loop Independent Dependence

Iteration space Traversal Graph (ISTG)

- shows graphically, the order of traversal in the iteration space.
- a node represents a point in the iteration space.
- a directed edge indicates the next point that will be encountered after the current point is traversed.

eg.

```
for (i=1; i<4; i++)
{
    for (j=1; j<4; j++)
    {
        S1: A[i][j] = A[i][j-1] + 1;
    }
}
```



Parallelization

- It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence

eg.
```
A[0] = 1
for (i=1; i<N; i++)
{
    A[i] = A[i] + A[i-1];
}
```
$A_1 = A_1 + A_0$
$A_2 = A_2 + A_1$
$A_3 = A_3 + A_2$

e.g.
```
for A[0] = 1
for(i=1; i<N; i+2)
{
    A[i] = A[i] + A[i-1]
}
```
$A_1 = A_1 + A_0$
$A_3 = A_3 + A_2$
$A_5 = A_5 + A_4$

Dependence Testing

- is the method used to determine whether dependencies exists between two subscripted reference to the same array in a loop nest

```
DO i1 = L1, U1
DO i2 = L2, U2
.
.
.
DO in = Ln, Un
```

S_1 : $A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots$

S_2 : $\ldots A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))$

```
END DO
  :
END DO
END DO
```

- dependence exist from S_1 to S_2 if $f_i(\alpha) = g_i(\beta)$ for all i, $i \leq i \leq m$ and α, β are iteration vectors.

~~GDT~~

GCD Test (Greatest common Divisor Test)

- We can use linear diophantine equation and GCD test to find the dependence in Loop.
- Diophantine equation is a polynomial equation, usually involving two or more unknowns, such that the only solutions of interest are the integer ones.

example: $Ax + By = C$ when A, B, C are integers

- If GCD (A, B) divides C, then dependency exists and in that case the loop is not parallizable.

solving diophantine equation

example: $70x + 112y = 168$

Step 1: Find GCD$(70, 112) = ? = 14$

```
70 ) 112 ( 1
   - 70
   ----
    42 ) 70 ( 1
       - 42
       ----
        28 ) 42 ( 1
           - 28
           ----
           14 ) 28 ( 2
                28
                --
```

Step 2: Find value of x and y

Reverse method

$14 = 42 - 28$

$14 = 42 - (70 - 42)$

$14 = 42 - 70 + 42$

$14 = 2(42) - 70$

$14 = 2(112 - 70) - 70$

$14 = 2(112) - 3(70)$

$14 \times 12 = 2 \times 12(112) - 3 \times 12(70)$

$168 = 24(112) - 36(70)$

$\therefore x = -36$

$y = 24$

Example:

Find dependency in following loop

```
DO i = 1, N
A(4i) = ...
... = A(2i + 2)
```

Soln

$4x = 2y + 2$

$4x - 2y = 2$

$GCD(4, -2) = 2$

- since 2 divides, there is dependency in the loop and loop cannot be executed parallely.

Example:

Find the diophantine equation to find if the following loop is parallizable or not

```
for (i=1; i<=10; i++)
{
    A[3*i + 20] = ...
    ... = A[2*i]
}
```

$3x + 20 = 2y$

$3x - 2y = -20$

$GCD(3, -2) = 1$

Example:

Find the diophantine equation to find if the following loop is parallizable or not

```
for (i=1; i<=10; i++)
{
    A[3*i + 20] = ...
    ... = A[2*i]
}
```

$3x + 20 = 2y$

$3x - 2y = -20$

$GCD(3, -2) = 1$

Exercise

Find the dependence test in following

```
DO I = 1, N
  DO J = 1, N
    S1: A(2i + 2j) = ...
    S2: ..... = A(4i - 6j + 3)
  END DO
END DO
```

1) From $S_1 \rightarrow S_2$ $\Rightarrow$ No dependency

$2a + 2b = 4a - 6b + 3$

$|2a + 2b - 4a + 6b| = 3$

$(2, 2, 4, 6) = 2$

Find dependence (intra) from $S_1 \rightarrow S_1$ and $S_2 \rightarrow S_2$

```
DO  I = 1,3
    DO  J = 1,3
        DO  k = 1,3
            S1: A(I+1, 2J+2, k+1) = B[I, 2J, k)
            S2: C(I, J, k)  = A(I, 1, 9-k)
        END DO
    END DO
END DO
```

$S_1 \rightarrow S_1 \Rightarrow$

Subscript categories for dependence testing

1. ZIV test (Zero Index Variable)
   If it contains no loop index variable

2. SIV test (Single Index Variable)
   If it contains only one index variable

3. MIV test (Multiple Index Variable)
   if it contains multiple index variables

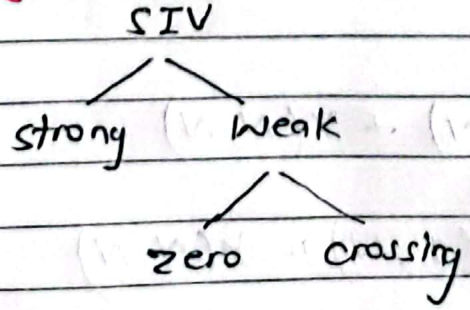
ZIV test

- ZIV subscripts contain no references to any loop variables.
  i.e they do not vary within any

eg.
```
DO I = 1, 100
   S1 : A(e1) = ...
   S2 : .... = A(e2) + B(i)
END DO
```

- where e_1 and e_2 are constants
- if $(e_1 - e_2) != 0$ then dependence doesnot exist.

SIV Testing



SIV Test

- Assume a case

S_1 : $A(\ldots, \alpha_1 i_k + c_1, \ldots) = \_\_\_\_$

S_2 : $\_\_\_\_ = A(\ldots, \alpha_2 i_k + c_2, \ldots)$

case 1 : $(\alpha_1 = \alpha_2) \Rightarrow$ strong SIV Test

case 2 : $(\alpha_1 = -\alpha_2) \Rightarrow$ weak crossing SIV Test

case 3 : $(\alpha_1 = 0$ or $\alpha_2 = 0) \Rightarrow$ weak zero SIV Test

Strong SIV Test

$$a = \frac{c_2 - c_1}{\alpha}$$

- dependence exist if $a \in z$ and $|a| \leq U - L$

eg.

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

Weak zero SIV Test

$$a = \frac{c_2 - c_1}{\alpha}$$

- if a is an integer then dependence exist

eg.

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

⇓

```
Y(1, N) = Y(1, N) + Y(N, N)
DO I = 2, N - 1
   Y(I, N) = Y(1, N) + Y(N, N)
END DO
Y(N, N) = Y(1, N) + Y(N, N)
```

Weak crossing SIV Test

$$a = \frac{c_2 - c_1}{2d}$$

- if $2a$ is integer the dependence exist

eg.

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

⇓

```
DO I = 1, (N+1)/2
   A(I) = A(N - I + 1) + C
END DO
DO I = (N+1)2, N
   A(I) = A(N - I + 1) + C
END DO
```

Unit-3

Preliminary Transformations

Loop Overheads

1. Initialize counter
2. Increment / Decrement } software
3. Test and Branch
4. FSM
5. Condition Check } Hardware
6. Extra cycles for R/W memory

Loop Induction Variable

- An induction variable is a variable whose value in each loop iteration is a linear function of the iteration index.
- the loop index is trivially an induction variable

eg:
```
J = 0
For I = 1 to N
    J = J + 1
    K = 2 * J
    M = 3 * K
    B = C + D
END FOR
```

Here, I, J, K, M are induction variable.

Induction Variable Optimization

eg.
```
for(i = 0; i < N; i++)
{
    j = 4 * i + 3;
    y = f(j);
}
```

⇓

```
for(j = 3; j < 4 * N + 3; j = j + 4)
{
    y = f(j)
}
```

Preliminary Transformation

- Most dependence tests require subscript expression to be linear of loop induction variables.
- but programs are not typically written with dependence testing in mind.

eg.
```
INC = 2
KI = 0
DO I = 1, 100
    DO J = 1, 100
        KI = KI + INC
        U(KI) = U(KI) + W(J)
    END DO
    S(I) = U(KI)
END DO
```

- Only two subscripts W(J) and S(I) are linear functions of the loop index variables.
- U(KI) cannot be tested because KI varies within the loop.
- INC is invariant within the inner loop.
- assignment of KI within J loop, increments its value by a constant amount on each loop iteration.

  KI ⇒ Auxiliary Induction Variable

- induction variable substitution transform every reference to an auxiliary induction variables into a direct function of a loop index.

Apply Induction Variable Substitution

1. Replace reference to auxiliary induction variable with function of loop index.

```
INC = 2
KI = 0
DO I = 1, 100
  DO J = 1, 100
    KI = KI + INC
    U(KI + J * INC) = U(KI + J * INC) + W(J)
  END DO
  KI = KI + 100 * INC
  S(I) = U(KI)
END DO
```

2. Remove all references to KI.

```
INC = 2
KI = 0
DO I = 1, 100
  DO J = 1, 100
    U(KI + (J - 1) * 100 + INC + J * INC) =
      U(KI + (I - 1) * 100 * INC + J * INC) + W(J)
  END DO
  S(I) = U(KI + I * (100 * INC))
END DO
KI = KI + 100 * 100 * INC
```

3. Substitute the constants

```
INC = 2
KI = 0
DO I = 1, 100
  DO J = 1, 100
    U(I*200 + J*2 - 200) =
    U(I*200 + J*2 - 200) + W(J)
  END DO
  S(I) = U(I*200)
END DO
KI = 20000
```

4. Remove all unused code

```
DO I = 1, 100
  DO J = 1, 100
    U(I*200 + J*2 - 200) =
    U(I*200 + J*2 - 200) + W(J)
  END DO
  S(I) = U(I*200)
END DO
```

Loop normalization

- to make dependence testing process as simple as possible many compilers normalize all loops to run from a lower bound of 1 to solve upper bound with stride 1 (step 1)

Algorithm

Procedure Normalized Loop (L_0)

- Let i be a unique compiler generated loop induction variable.

S_1 : replace the loop header for L_0

DO I = L, U, S

with the adjusted loop header

DO I = 1, (U - L + S) / S

S_2 : replace each reference to I with the loop by

→ i * S - S - L

end.

eg. Unnormalized

```
DO  I = 1, M
    DO J = 1, N
        A(J, I) = A(J, I-1) + 5
    END DO
END DO
```

Normalized

```
DO  I = 1, M
    DO J = 1, N - I + 1
        A(J, I-1, I) = A(J + I - 1, I - 1) + 5
    END DO
END DO
```

Data Flow Analysis

- It is an analysis that determines the information regarding the definition and use of data in a program.

eg. $S_1 : x = a + b$ ⇒ x is defined by S_1 and S_1 used a and b.

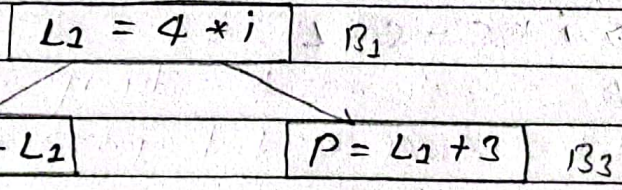
Data Flow properties:

1. Available expression
2. Reaching definition

1. Available expression

- an expression "a+b" is said to be available at a program point "x", if none of its operands gets modified their use.

eg.

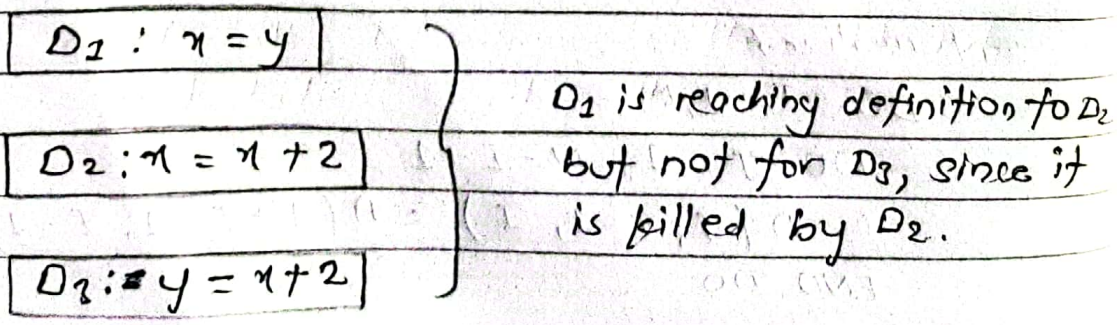


i.e 4*i (L_1) is available for block B_2 and B_3.

2. Reaching definition

a definition d is reaching to a point x if D is not killed or redefined before that point.

eg.



eg.

S_1 : $a = 3$

S_2 : $b = a + z$

S_3 : $a = x + y$

S_4 : $c = a + z$

Next - Use Information

- We need to know for each basic block, whether the value contained in the variable will be used again later.
- if a variable has no next-use, we can reuse the register allocated to the variable.

eg.

Statement	Live (L) / Dead (D)				Next Use			
	x	y	z	t_7	x	y	z	t_7
1. $x = y + z$	L	D	D	-	2	-	-	-
2. $z = x * 5$	D	-	L	-	-	-	3	-
3. $t_7 = z + 1$	-	-	L	L	-	-	4	4
4. $y = z - t_7$	-	L	L	D	-	5	5	-
5. $x = z + y$	D	D	D	-	-	-	-	-

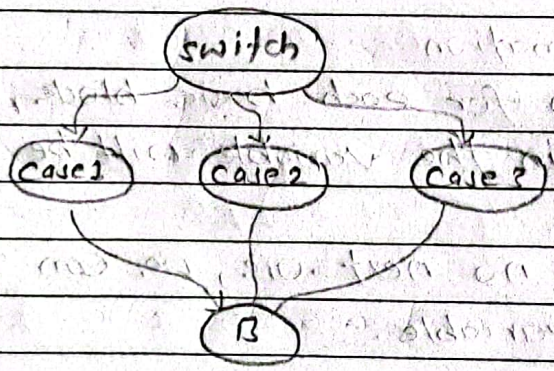
Definition-use graph

- is a graph that contains an edge from each definition point in the program to every possible use of the variable at run time.

Blocks

- basic block is a maximal group of statements such that one statement in the group is executed if and only if every statement is executed.

eg.



Definition-use graph sets

- basic block computation produces the following set.

1. Uses (b)
   - the set of all variables used within block b that have no prior definition within the block.
2. Defsout (b)
   - the set of all definitions within block b that are not killed within the block.

3. Killed (b)

- the set of all definitions that define variables killed by other definitions within b.

eg.

B_1

$a = b + c$
$d = a * 2$
$f = a + d$
$a = 2$

uses $(B_1) = \{b, c\}$
defsout $(B_1) = \{d, f\}$
killed $(B_1) = \{a\}$

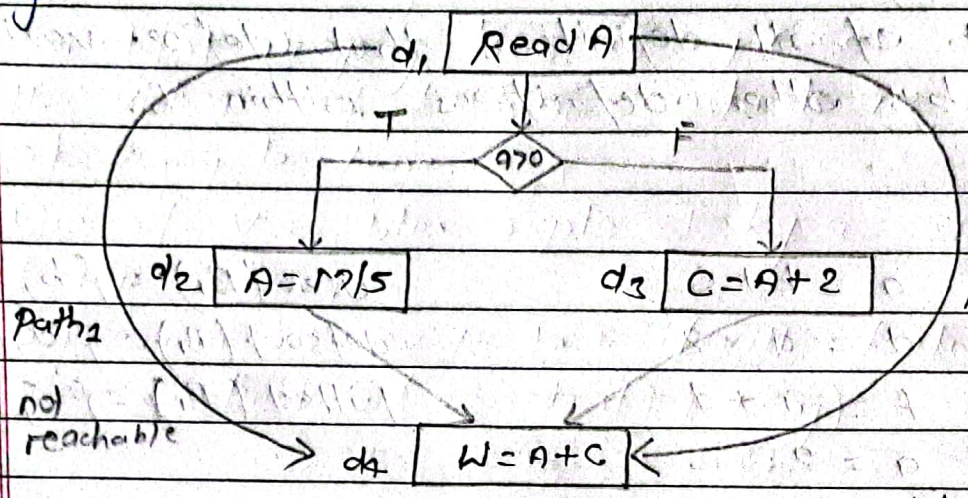
4. Reaches (b)

- the set of all definitions from all blocks that can possibly reach b.
- Let us consider a graph with one basic block b, and some number of predecessors, each of which can reach b through some form of control flow in this graph, reaches (b), along any one predecessor p is the set of all definitions that reach p and that also are not killed inside p, plus that reach the exit of p.

formally,

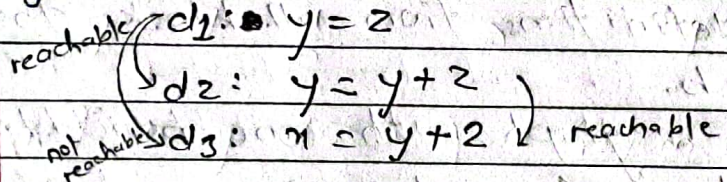
$$\text{reaches}(b) = \bigcup_{p \leftarrow p(b)} (\text{defsout}(p) \cup (\text{reaches}(p) \cap \neg \text{killed}(p)))$$

eg.



→ d_1 is reable to d_4 from path 2

eg.



Dead Code Elimination

- removes unnecessary instructions from the program.
- dead code is a section in the source code of the program which is executed but whole result is never used in any other computation

eg. $c = a * b$

$t = a$ ⇒ $c = a * b$

$d = a * b + 4$ $d = a * b + 4$

eg.

```
int test()
{
    int a, b;
    a = 1;
    b = 2;
    b = 3;
    return b;
    a = b;   → unreachable code
}
```

eg.

```
int main()
{
    int a;
    a = 1;
    a = 2;
    goto L;
    printf("%d", a);
L: return a;
}
```

//

```
int test
{
    int a, b;
    b = 3
    return b;
}
```

Dead code Elimination Algorithm

Procedure EliminateDead code (P)

worklist = {absolutely useful statements} → output, inp
control flo

While (worklist ≠ ∅) DO

repeatedly adds the statement that are necessary to compute the number of worklist

- x = an arbitrary element of worklist
- mark x as useful
- worklist = worklist − {x}
- forall (y, x) ∈ defuse-graph DO
  - if y is not marked useful then
    - worklist = worklist ∪ {y}

end

- delete every statement that is not marked useful

End procedure

Constant Propagation

- Constant propagation is a process of substituting the values of known constants in expression at compile time.
- Use of variable replaced by a constant value (propagation).
- expression is evaluated if all of its operands are constant (constant folding)

eg.

$x = 14$ $\quad y = 7 - \frac{x}{2}$ $\Rightarrow$ $x = 14$ $\quad y = 7 - \frac{14}{2}$ $\Rightarrow$ $x = 14$ $\quad y = 0$

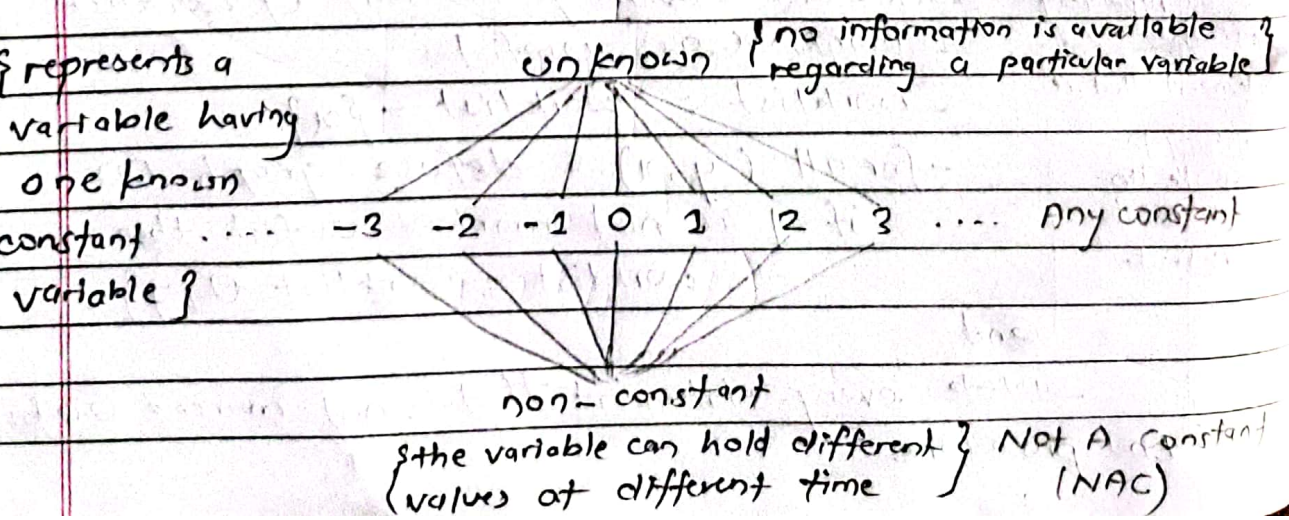
return $y * (\frac{28}{x} + 2)$ return $y * (\frac{28}{14} + 2)$ return 0;

constant propagation

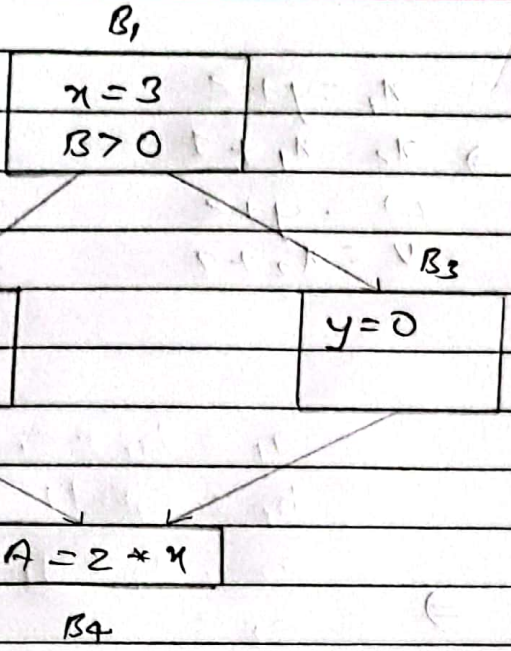
folding

Some symbols used in constant propagation.

1. $\perp$ (Bottom) $\Rightarrow$ This statement never executes
2. C (constant) $\Rightarrow$ is constant.
3. T (top) $\Rightarrow$ not a constant.



eg.



Meet Operation

1. Any constant ≤ unknown
2. N.AC ≤ Any constant
3. Unknown ∧ x = x
4. NAC ∧ x = NAC
5. NAC ∧ unknown = NAC
6. x ∧ x = x but x ∧ y = NAC
   IF x ≠ y

SSA (Single Static Assignment) form

- ~~SSA is a program is in SSA form~~
- a program is in SSA form, if each use of variable is reached by exactly one definition.
- Optimization becomes easier if each variable has only one definition.
- i.e every variable has single definition.

eg.

$x = y + z$
$x = x + 1$
$w = y + z$
$v = x + 3$

$\Rightarrow$

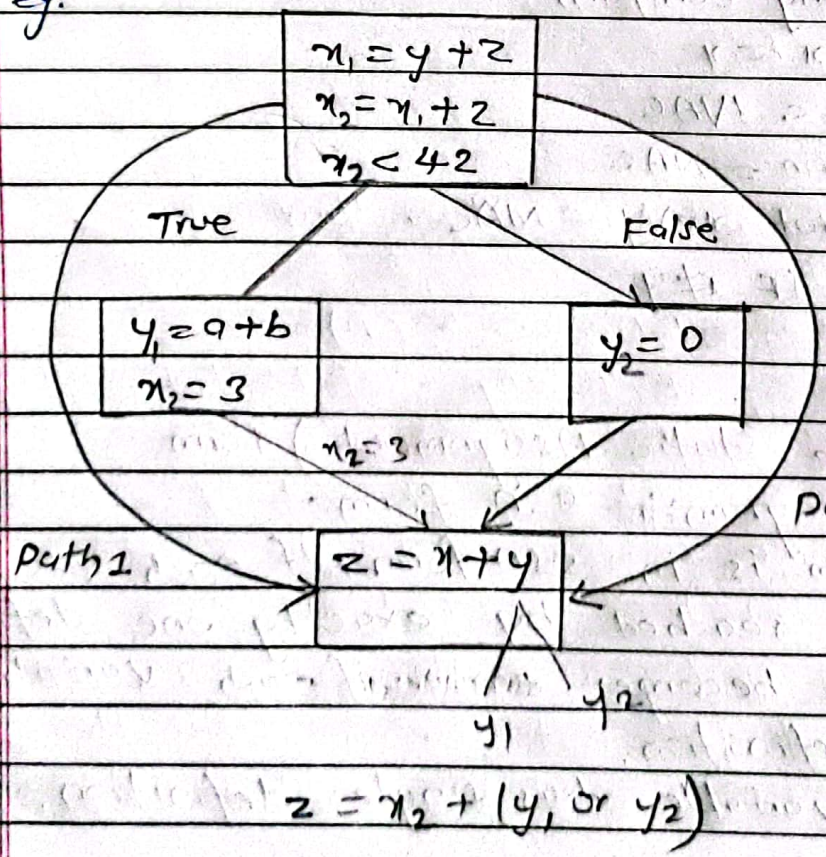
$x_1 = y + z$
$x_2 = x_1 + 1$
$w = y + z$
$v = x_2 + 3$

eg.

$a = b + c$
$b = c + d$
$a = x + y$
$b = a * b$
$x = 2 * a$
$y = x + b$

$\Rightarrow$

eg.

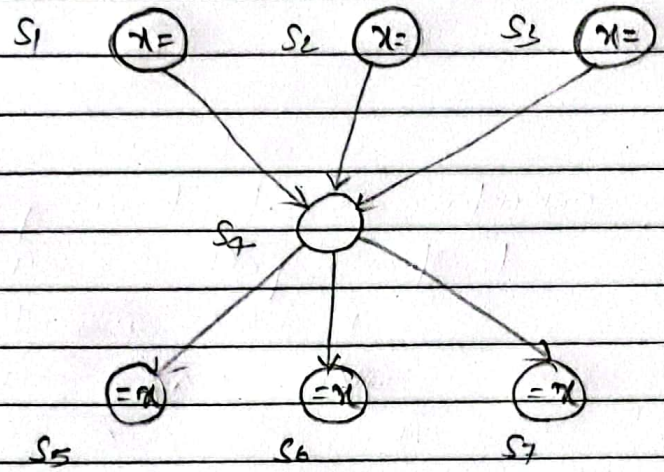


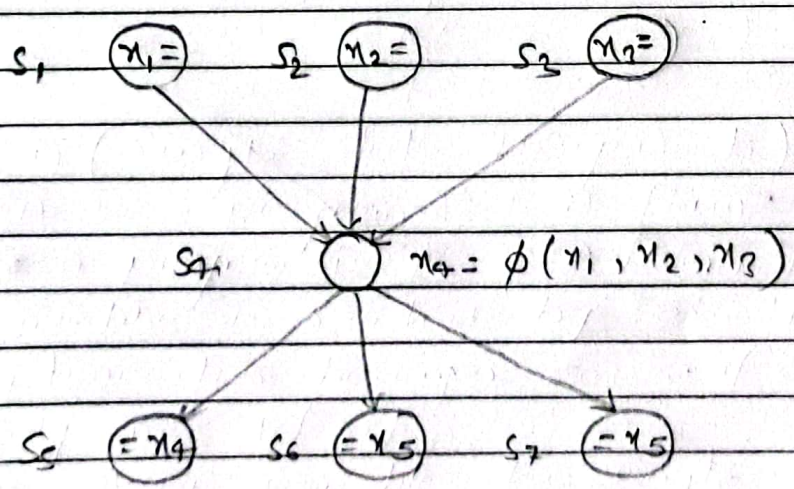
The ϕ function

- ϕ merges multiple definitions a long multiple control paths into a single definition.
- at a basic block with p predecessors, there are p arguments to the ϕ function,

$$x_{new} = \phi(x_1, x_2, \ldots, x_p)$$

- Chooses the version depending in the incomming edges.





Common subexpression elimination

- an occurance of an expression E is called a common sub-expression, if E was previously computed and the values of the variables in E have not changed since the previous computation
- is a compiler optimization technique that searches for identical expressions.
- i.e if they all evaluate to the same value then replace with a single variable holding the computation.

eg.

$a = b * c + g$
$d = b * c * k$

$\Rightarrow$

$temp = b * c$
$a = temp + g$
$d = temp * k$

eg.

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * j$
$t_8 = 4 * j$
$t_9 = a[t_8]$ $\Rightarrow$?
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * j$
$t_8 = t_7$
$t_9 = a[t_7]$
$a[t_7] = t_9$
$t_{10} = t_7$
$a[t_7] = x$
goto B_2

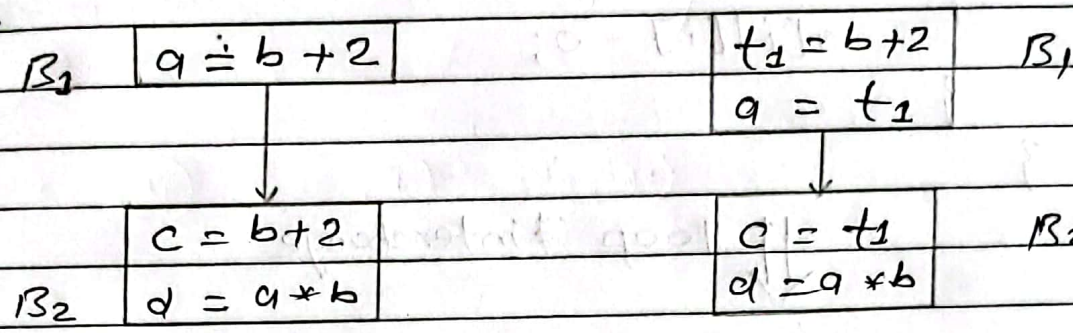
Forward substitution

- is the inverse of common sub expression elimination.
- instead replacing an expression evaluation by a copy operation, it replaces a copy by a

re-evaluation of the expression.

- it may seem that common sub expression elimination reduces the number of ALU operations.
- but still it causes a register to be occupied for a long time in order to hold an expression's value and hence reduces the number of registers available for other uses.

eg.



fig(1) fig(2)

- in fig(1), the edge from B_1 to B_2 uses two register for a and b.
- in fig(2), the edge from B_1 to B_2 uses three registers for a, b, t_1.

Unit-4

Loop Optimization

Loop Interchange

- switching the nesting order of two loops.
- i.e exchange inner loops with outer loops.

eg.

```
for (i=0; i<100; i++)
{
    for (j=0; j<100; j++)
    {
        a[j][i] = 0;
    }
}
```

⇓ loop interchange

```
for (j=0; j<100; j++)
{
    for (i=0; i<100; i++)
    {
        a[j][i] = 0;
    }
}
```

- not all loop interchange are safe.

eg. Before loop Interchange

```
DO J = I, M
    DO I = 1, N
        A(I, J+1) = A(I+1, J) + B
    END DO
END DO
```

Result Analysis

J=1, I=1 A_{12} = A_{21} + B
2 A_{22} = A_{31} + B
3 A_{32} = A_{41} + B
4 A_{42} = A_{51} + B
J=2, I=1 A_{13} = A_{22} + B
2 A_{23} = A_{32} + B
3 A_{33} = A_{42} + B
4 A_{43} = A_{52} + B

eg.

eg. After Loop Interchange

```
DO I = 1, N
   DO J = 1, M
      A(I, J+1) = A(I+1, J) + B
   END DO
END DO
```

Result Analysis

I = 1, J = 1 $A_{12} = A_{21} + B$
2 $A_{13} = A_{22} + B$
3 $A_{14} = A_{23} + B$
4 $A_{15} = A_{24} + B$

I = 2, J = 1 $A_{22} = A_{31} + B$
2 $A_{23} = A_{32} + B$
3 $A_{24} = A_{33} + B$
4 $A_{25} = A_{34} + B$

Result Analysis
I = 1, J = 1
2
3
4
I = 2, J = 1
2
3
4

Loop unrolling

- is a loop transformation technique that helps to optimize the execution time of a program.

eg.

```
sum = 0
for(i=0; i<100; i++)
{
    sum = sum + 10;
}
```

Loop unrolling ⇒

```
sum = 0
for(i=0; i<100; i+=5)
{
    sum = sum + 10;
    sum = sum + 10;
    sum = sum + 10;
    sum = sum + 10;
    sum = sum + 10;
}
```

$A_{21}+B$
$A_{31}+B$
$A_{41}+B$
$A_{51}+B$
$A_{22}+B$
$A_{32}+B$
$A_{42}+B$
$A_{52}+B$

eg.

```
for (i=0; i<100; i++)
{
    a[i] = a[i] + 5;
}
```

⇓ loop unrolling

```
for( i=0; i<100; i+2)
{
    a[i] = a[i] +5;
    a[i+1] = a[i+1] + 5;
}
```

Loop invariant

- a boolean statement that is true at the start of the loop and at the end of each iteration.

eg.

```
for (i=0; i<n; i++)
{
    x = i + a/b;
    y = y+2;
}
```

→

```
n = a/b
for (i=0; i<n; i++)
{
    x = i+n
    y = y+2;
}
```

Loop fusion

- adjacent loops are merged to one loop.

eg.

```
for(i=0; i<100; i++)
{
    x[i] = 1;
}
for(i=0; i<100; i++)
{
    y[i] = 2;
}
```

⇒

```
for(i=0; i<100; i++)
{
    x[i] = 1;
    y[i] = 2;
}
```

Fusion safety

- If a loop independent dependent between statement S_1 and S_2 in loops L_1 and L_2 respectively is fusion preventing if fusing L_1 and L_2 causes the dependence to be carried by the combined loop.
- We should not fuse loops if the fusing will violate ordering of the dependent graph.

eg.

```
DO I = 1,N
S1 : A(I) = B(I) + C
END DO
DO I = 1,N
S2 : D(I) = A(I+1) + E
END DO
```

→ fusion →

```
DO I = 1,N
S1 : A(I) = B(I) + C
S2 : D(I) = A(I+1) + E
END DO
```

Loop vectorization

- Loop vectorization transforms procedural loops by assigning a processing unit to each pair of operands.
- vectorization can significantly accelerate the large data sets.

eg.

```
for(i=0; i<100; i++)
{
    c[i] = a[i] * b[i];
}
```

vectorized

```
for(i=0; i<100; i=i+4)
{
    c[i:i+3] = a[i:i+3] * b[i:i+3]
}
```

- here c[i:i+3] represents the four array elements from c[i] to c[i+3],
  i.e c_i, c_{i+1}, c_{i+2}, c_{i+3}

- vector processor can perform four operations for a single vector instruction.

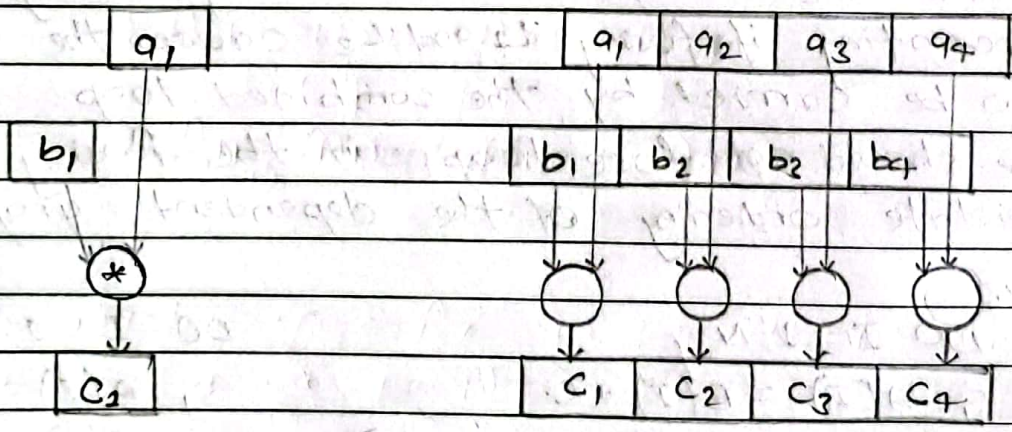


Fig: Scalar Approach

Fig: vector Approach

- loop vectorization attempts to rewrite the loop in order to execute its body using vector instructions.
- such instructions are commonly referred as SIMD (Single Instruction Multiple Data), where identical operands are performed simultaneously by the hardware.

eg.

```
for (i=1; i<=N; i++)                for(i=1; i<=N; i++)
{                                ⇒  {
  A[i] = 2 * B[i] + 1;               A[i:i+3] = 2*B[i:i+3]+(1,1,1,1)
}
```

Fig: Loop vectorization scheme with vector of length 4.

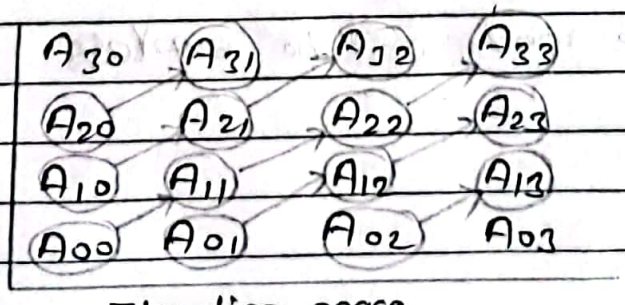
Loop skewing

- is a transformation that reshapes an iteration space to make it possible to express the existing parallelism with conventional parallel loops.
- changes the shape of the iteration space without changing the dependencies.
- looks like geometrical transformation, which can help to expose a canonical parallelism.

eg.

```
for (i = 0; i<N; i++)
{
  for(j=0; j<m; j++)
  {
      A[i][j] = A[i-1][j-1];
  }
}
```

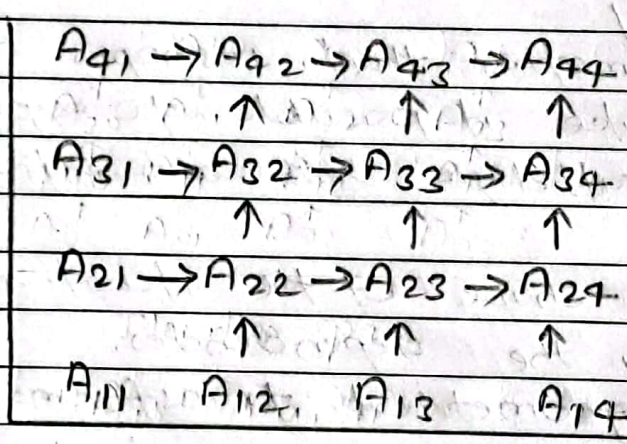
⇓



Iteration space

eg.

```
DO I = 1, N
   DO J = 1, N
      A[I,J] = A[I-1,J] + A[I,J]
   END DO
END DO
```



- the iteration space for $S_1 : \rightarrow [0,N] \times [0,M]$
- $(i,j) = (i'-1, j'-1)$
- $P(i,j) = P(i'-1, j'-1)$
- i.e $P = (i-j)$ satisfy the space partition constraint because $i'-1-(j'-1) = i'-j'$
- the maximum and minimum value that P can take is $[-M, N]$
- $i_{min} - j_{max} = 0 - M = -M$
  $i_{max} - j_{min} = N - 0 = N$
- now the naive code can be express as

```
for (p = -M, p <= N ; p++)
{
  for (i=0; i<=N ; i++)
  {
    for (j=0; j<=M ; j++)
    {
      if (p == i-j)
      {
        A[i][j] = A[i-1][j-1];
      }
    }
  }
}
```

only execute the statement S_1 if we are in correct processor.

- now tighten the bound for innermost loop (j)
- since $p = i - j$, $j = i - p$
- now we get

```
for (p = -M; p <= N; p++)
{
  for (i=0; i<=N; i++)
  {
    if (i-p >= 0 && i-p <= M)
    {
      A[i][i-p] = A[i-1][i-p-1]
    }
  }
}
```

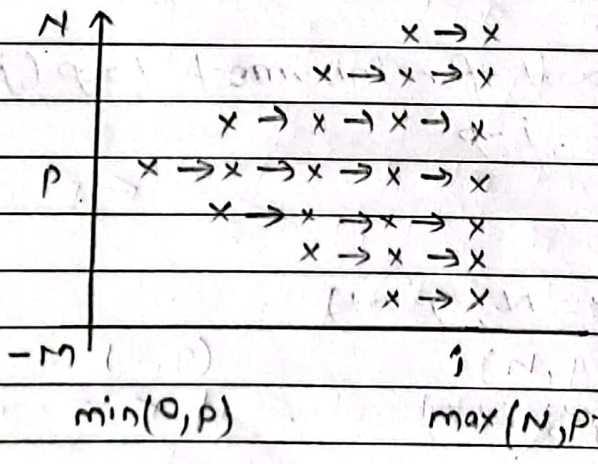
- now empty iterations is removed in j loop.
- there ~~exist~~ is still some empty, iteration in i loop.
- since $i - p \geq 0 \Rightarrow i \geq p$

  $i - p \leq M \Rightarrow i \leq M + p$
- now,

```
for (p = -M; p <= N; p++)
{
    for (i = max(0, p), i <= min(N, M+p); i++)
    {
        A[i][i-p] = A[i-1][i-p-1]
    }
}
```

(previous lower bound; previous upper bound)

- now the iteration space code becomes



now outerloop can be parallelize.

- as p is smaller, less point, as it increases the point is more & more
- some where in middle, it becomes very large, then it starts decreasing again.

① $p = N$

i < $\max(0, N) = N$

$\min(N, N+N) = N$

② $p = -M$

i < $\max(0, -m) = 0$

$\min(N, -m+n)$

$(N, 0) = 0$

```
DO I = 1, N
  S1 : T = A(I)
  S2 : A(I) = B(I)
  S3 : B(I) = T
END DO
```

Scalar Expansion

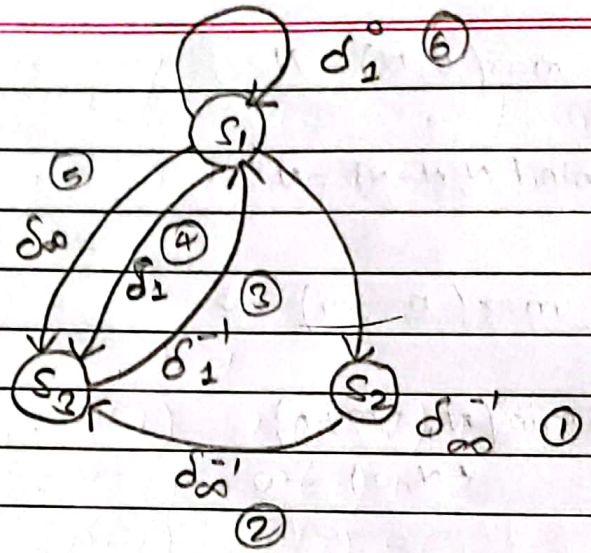
- the programs frequently used scalar temporaries in computations involving arrays.

eg.

```
DO  I = 1, N
   S1 : T = A(I)
   S2 : A(I) = B(I)
   S3 : B(I) = T
END DO
```

} swaps the contents of two arrays.

- Let us create the dependence graph for above program.



Case 1 → Loop independent dependency (Anti Dependency) due to A(I)

Case 2 → Loop independent dependency (Anti dependency) due to B(I)

Case 3 → T is used in S_3 and again defined in next iteration, so loop carried dependence (true dependency)

case 4 → S_1 of first iteration defines T and S_2 of second iteration is using it, so loop carried dependency (true dependency)

case 5 → true dependency and loop independent due to T.

case 6 → In 1st iteration T is defined and in second iteration it is again defined so loop carried dependency (output dependency)

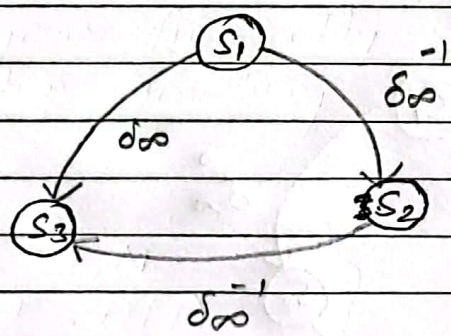
- all loop carried dependencies are due to scalar variable T.
- so use one vector variable instead of a scalar variable.
- i.e if each iteration has a separate location to use as a temporary, these dependencies disappear.

```
DO I = 1, N
  S1: T$(I) = A(I)
  S2: A(I) = B(I)
  S3: B(I) = T$(I)
END DO
T = T$(N)
```

} replacing T by T$, where T$ is compiler generated temporary variable

→ Here T is original variable of a program and may have some use after this program.

- now the dependency graph becomes



- i.e here is no cycle and no loop dependent dependency.
- so iteration can be parallelized.
- after scalar expansion the code can be vectorized as,

```
S1  T$(1:N) = A(1:N)
S2  A(1:N) = B(1:N)
S3  B(1:N) = T$(1:N)
    T = T$(N)
```

Array Renaming

- array locations are sometimes reused, creating unnecessary anti dependencies and output dependencies.

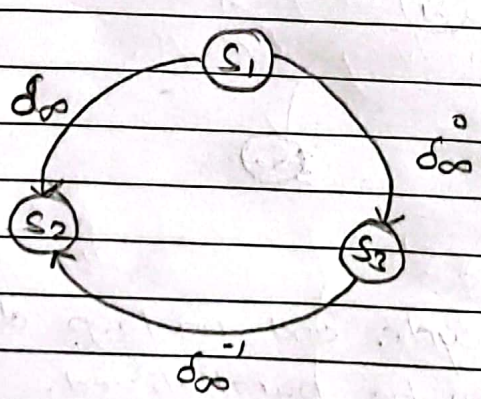
eg.

```
DO I = 1, N
  S1 : A(I) = A(I-1) + X
  S2 : Y(I) = A(I) + Z
  S3 : A(I) = B(I) + C
END DO
```

- here A(I) is used for two different things within the loop.

① the definition in S_3 and S_1
② use in S_2.

- the dependency graph looks like.

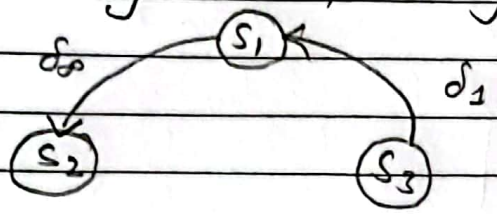


- Since A(I) is redefined in S_3, it has no relation with A(I) of S_2, so break it.

- So let A(I) is renamed by A$(I)
- now the program becomes

```
DO I = 1, N
  S1 : A$(I) = A(I-1) + X
  S2 : Y(I) = A$(I) + Z
  S2 : A(I) = B(I) + C
END DO
```

- after renaming the dependency graph looks like



- now the program can be vectorized as

```
A(1:N) = B(1:N) + C
A$(1:N) = A(0:N-1) + X
Y(I:1) = A$(1:N) + Z
```