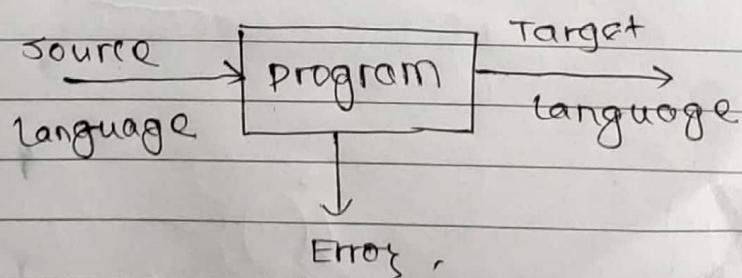
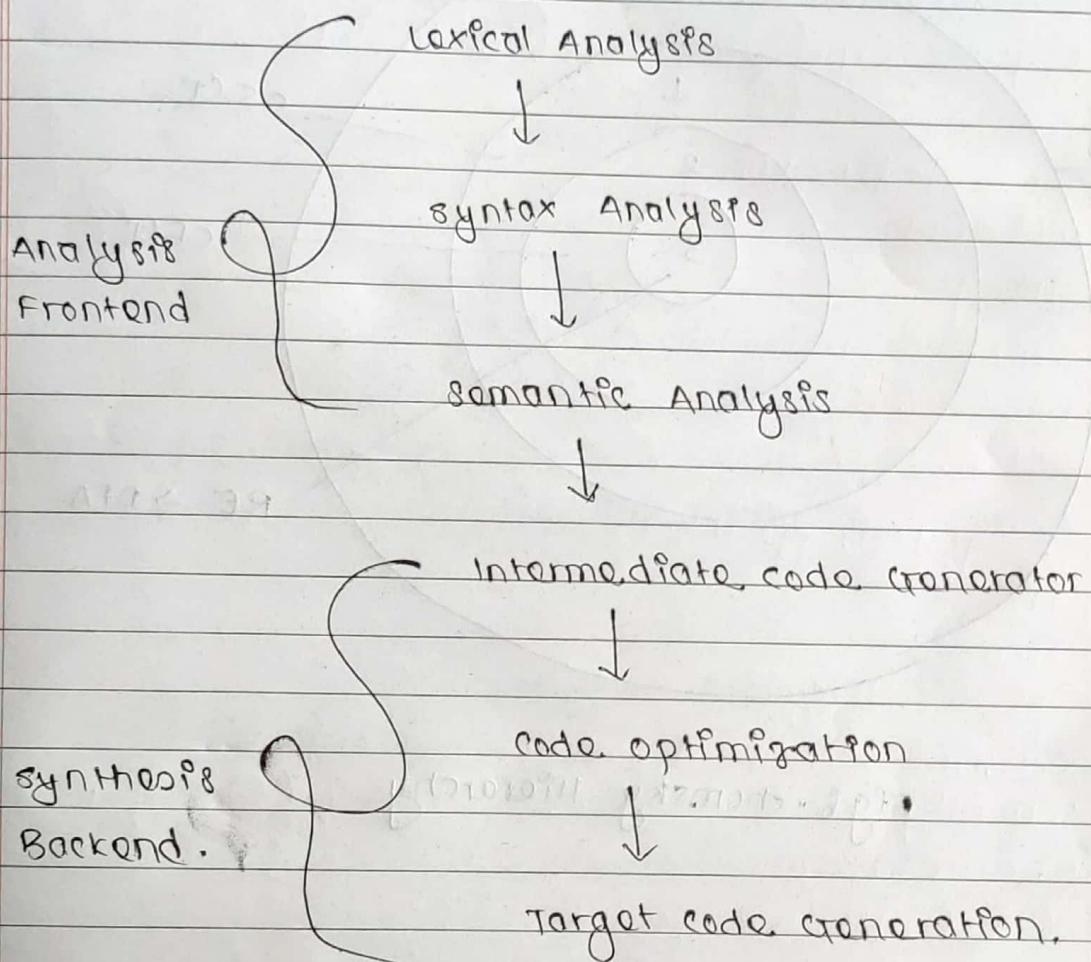


compiler:-



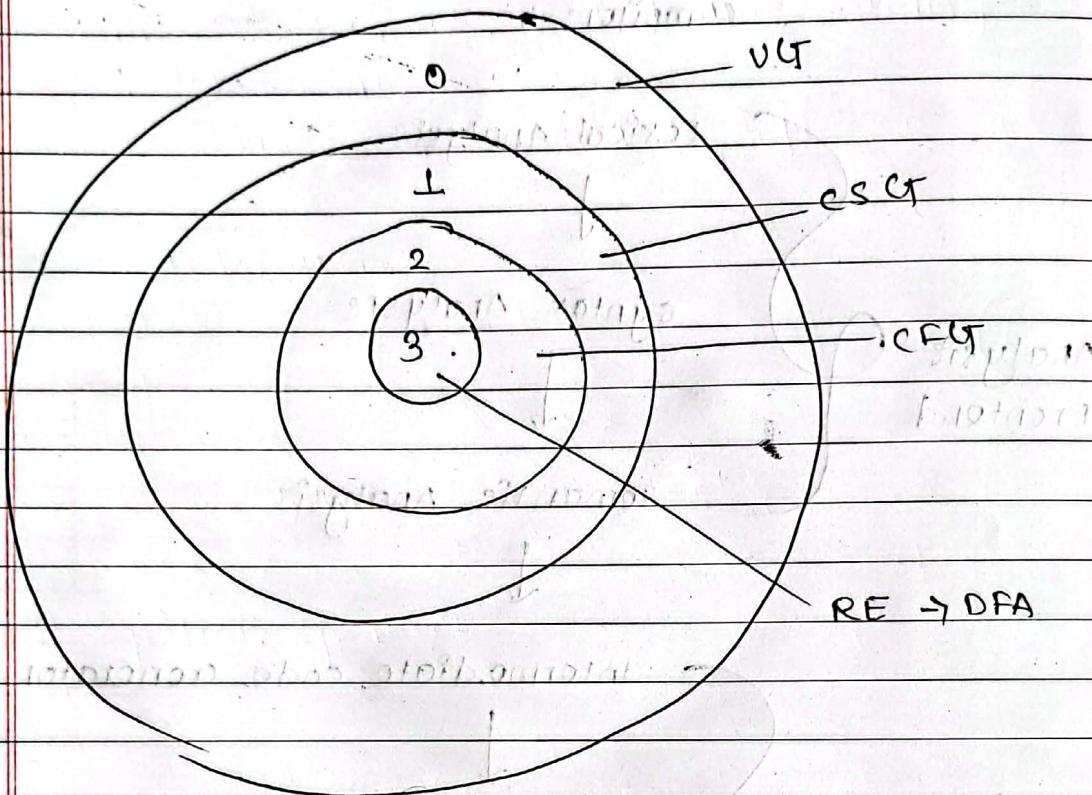
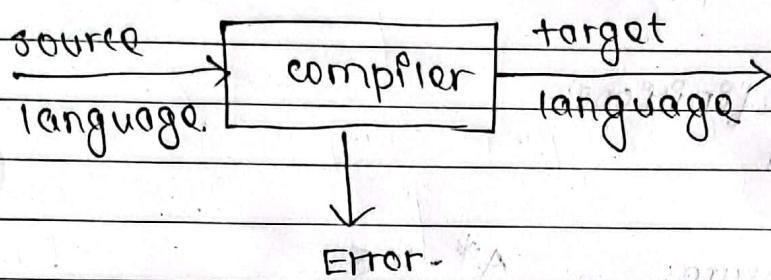


Fig :- chomsky hierarchy

0 : VGT
1 : CS CT
2 : CFG
3 : RE \rightarrow DFA

Unit-1 Review of compiler structure.

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



phases of compiler.

lexical Analysis

Syntax Analysis

semantic Analysis

Intermediate code Generation

code optimization

target code optimization.

Lexical Analysis.

Scans the input and tokenizes into a block of stream, token, pattern and lexons.

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 $DFA = (\Sigma, Q, q_0, f, \delta)$.

where,

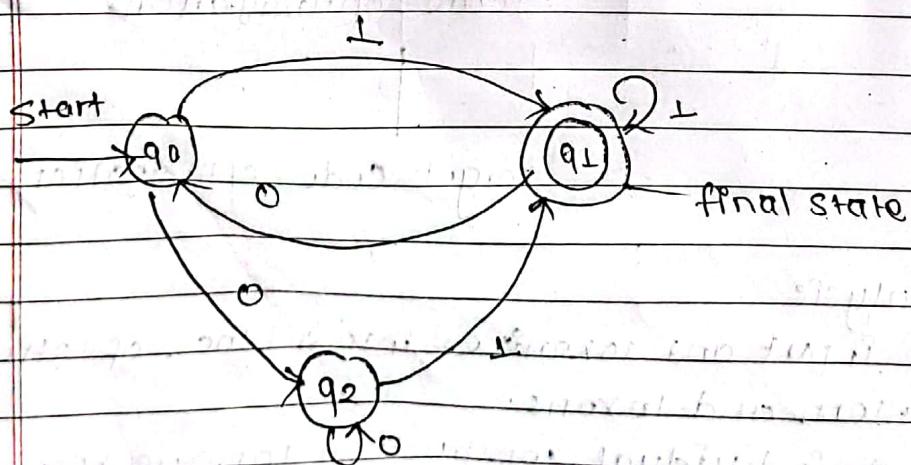
$Q \rightarrow$ finite set of states

$q_0 \rightarrow$ starting state

$f \rightarrow$ finite set of final states

$\delta \rightarrow$ transition function, $\delta: Q \times \Sigma \rightarrow Q$

$\Sigma \rightarrow$ finite set of input alphabet.



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

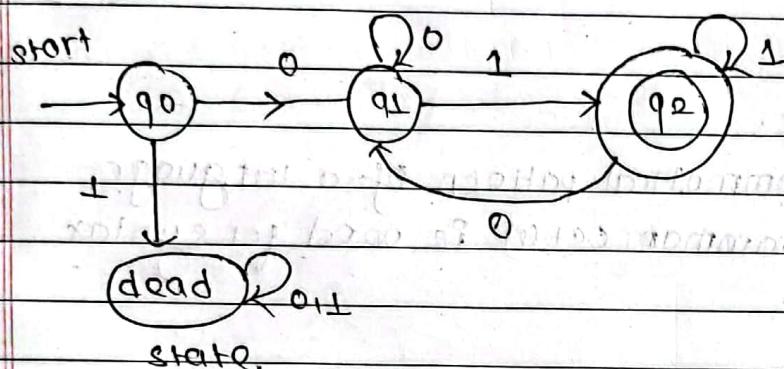
$$q_0 = \{q_0\}$$

$$F = \{q_1\}$$

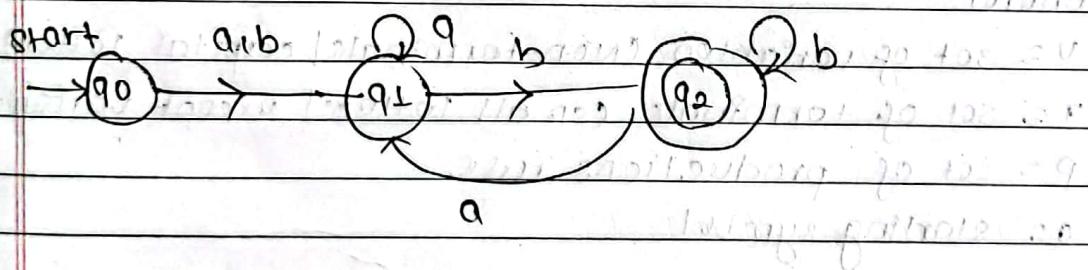
$$S = S(q_0) = q_0$$

$$S(q_0) = q_2$$

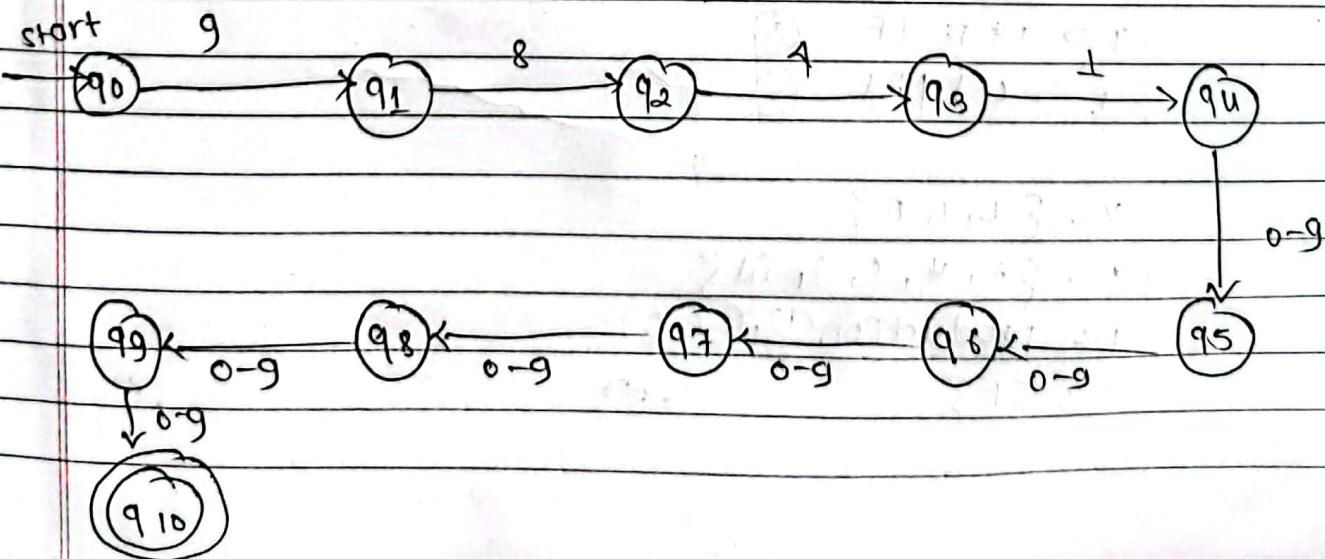
$$0(0+1)^* 1$$

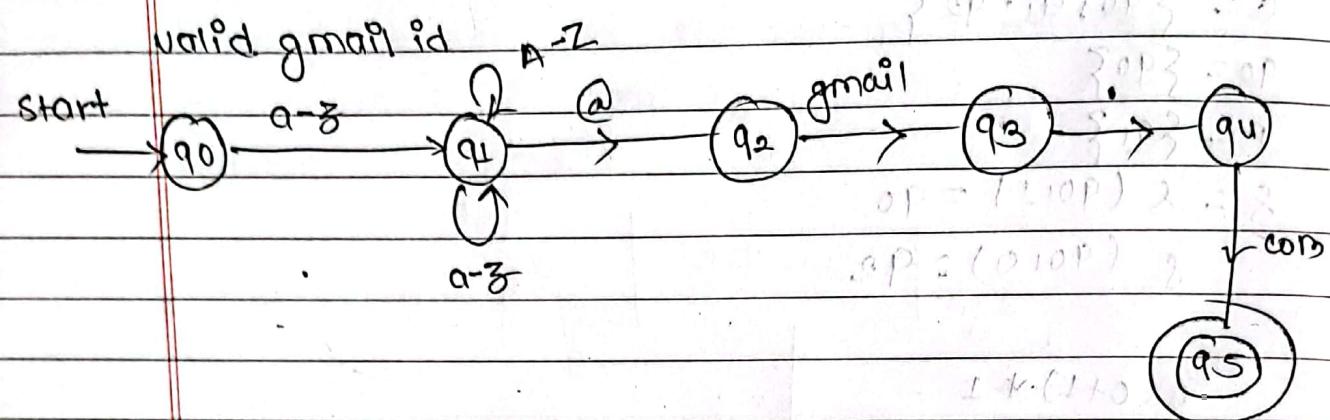


$$(a+b)(a+b)^*b$$



prepaid NTC mobile number





Syntax Analysis

- * 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 production rule

S = starting symbol

Example:

$$E \rightarrow E + T \mid T \quad (1)$$

$$T \rightarrow T * F \mid F \quad (2)$$

$$F \rightarrow (E) \mid id \quad (3)$$

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

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

$$P = \text{production } (1), (2), (3)$$

$$S = \{ E \}$$

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, $w = \text{id} + \text{id} * \text{id}$

$$E \rightarrow ETT \quad \therefore E \rightarrow ETT$$

$$E \rightarrow T+T \quad \therefore E \rightarrow T$$

$$E \rightarrow F+F \quad E \rightarrow F$$

$$E \rightarrow Pd+T*F \quad E \rightarrow T*F$$

$$E \rightarrow \text{id} + F*F \quad T \rightarrow F$$

$$E \rightarrow \text{id} + \text{id} * \text{id} \quad T \rightarrow \text{id}$$

$$w = Pd * (Pd + Pd)$$

$$E \rightarrow E+T$$

$$E \rightarrow E+F$$

$$E \rightarrow E+(E)$$

$$E \rightarrow E+(T)$$

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

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

$$E \rightarrow T+(Pd * \text{id})$$

$$E \rightarrow F+(Pd * \text{id})$$

$$E \rightarrow \text{id} + (Pd * Pd)$$

$$E \rightarrow E+T$$

$$E \rightarrow E+F$$

~~$$E \rightarrow E+(E) T*F+T$$~~

~~$$E \rightarrow F*(E)+T$$~~

E

$$W = PdA \cdot Cid + PdJ$$

$$E \rightarrow E + T$$

$$E \rightarrow$$

Syntax Directed Translation

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

Production

$$E \rightarrow E + T$$

semantic action

$$E \cdot \text{value} = E \cdot \text{val} || T \cdot \text{val} || +$$

$$E \rightarrow E - T$$

$$E \cdot \text{value} = E \cdot \text{val} || T \cdot \text{val} || -$$

$$E \rightarrow T$$

$$E \cdot \text{val} = T \cdot \text{val}$$

$$T \rightarrow 0$$

$$T \cdot \text{val} \leftarrow "0"$$

$$T \rightarrow 1$$

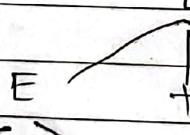
$$T \cdot \text{val} \leftarrow "1"$$

$$\vdots$$

Example: $9 - 5 + 2$

$E \{ E \cdot \text{val} = 9 - 5 + 2 \} \{ \text{infix to postfix conversion} \}$

$$E \cdot \text{val} = 9 - 5$$



$$+ \{ T \cdot \text{val} = 2 \}$$

$$T \cdot \text{val} = E \cdot \text{value} = 9$$

$$T \{ T \cdot \text{val} = 5 \}$$

$$5$$

$$T \cdot \text{val} = 9$$

$$T$$

$$5$$

$$9$$

Semantic Analysis

Concerns with the meaning of a language construct.

Example:- type checking

Implicit / explicit type conversion.

Example:-

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

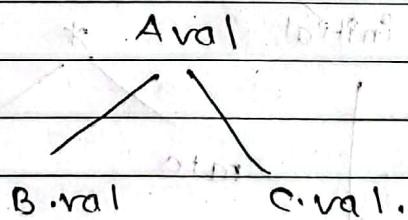
2. If (root & L = A+B) { if root == true & L.type = A.L.type
 Q18Q C-D) else S.type = C.type }

Synthesized and Inherited Attribute.

An attribute which depends upon child is synthesized attribute.

An attribute which 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:-

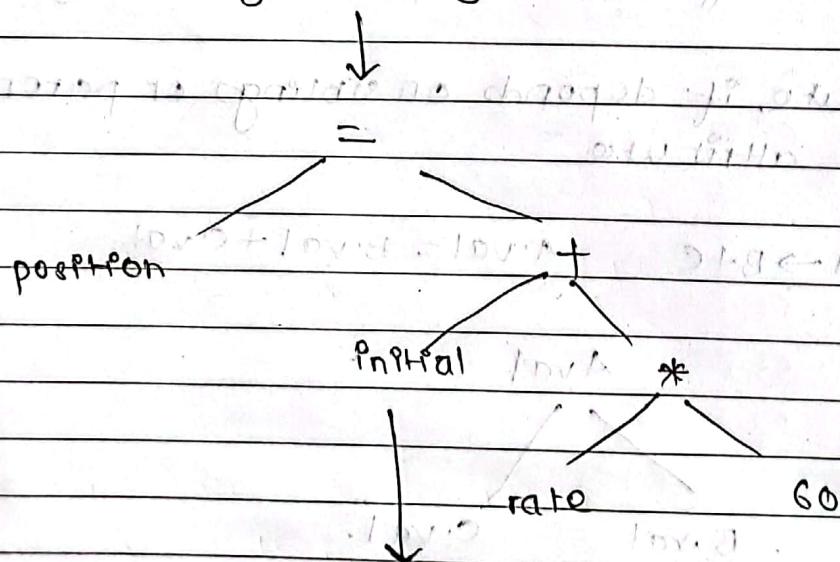
position = initial + rate * 60

Lexical Analysis

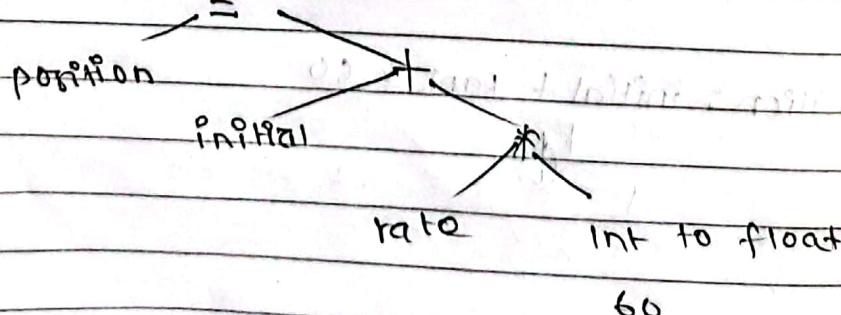
position	initial	+	rate	*	60
id ₁	id ₂	id ₃			integer number

assignment addition multiplication
operator operator operator

Syntax Analysis



Semantic Analysis



Intermediate code generation (three address code)

$t_1 = \text{In to read } (60)$

$t_2 = \text{rate} * t_1$

$t_3 = \text{initial}$

$t_4 = t_3 + t_2$

position = t_4

Code optimization.

$t_1 = \text{In to Real } (60)$

$t_2 = \text{rate} * t_1$

position = initial + t_2

Target code generation.

LOAD F R1, #600

LOAD F R2, rate

MUL F R1, R2

LOAD F R3, initial

ADD F R1, R3

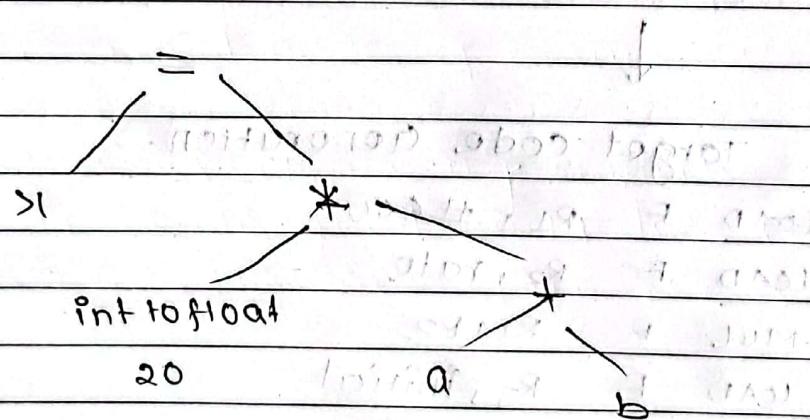
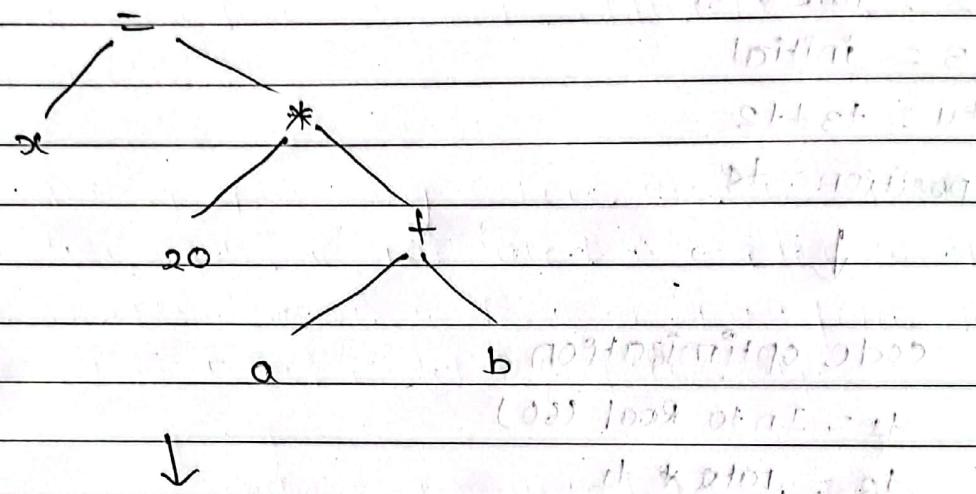
LOAD F position, R1

$$x = 20 * (a+b)$$

ASS. OP

$$x = 20 * (a + b)$$

id₁ Integer id₂ id₃



Intermediate code generation.

$$t_1 = \text{Int to real } 20$$

$$t_2 = (a+b) \quad a+b$$

$$t_3 = t_1 * t_2$$

$$n = t_3$$

$$t_1 = \text{Int to float}$$

$$t_1 = a+b$$

$$t_2 = a+b$$

$$t_2 = \text{Int to float}$$

$$t_3 = a$$

$$t_3 = a$$

$$t_4 = t_3 + t_1$$

$$a = t_4$$

Importance of pipelining
using a non-optimizing will usually result in a non-efficient code when compared to more sophisticated compilers.

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 cycles)

ADD a, b, c

ADD c, r1, d

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

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

The optimized code is more efficient and has 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

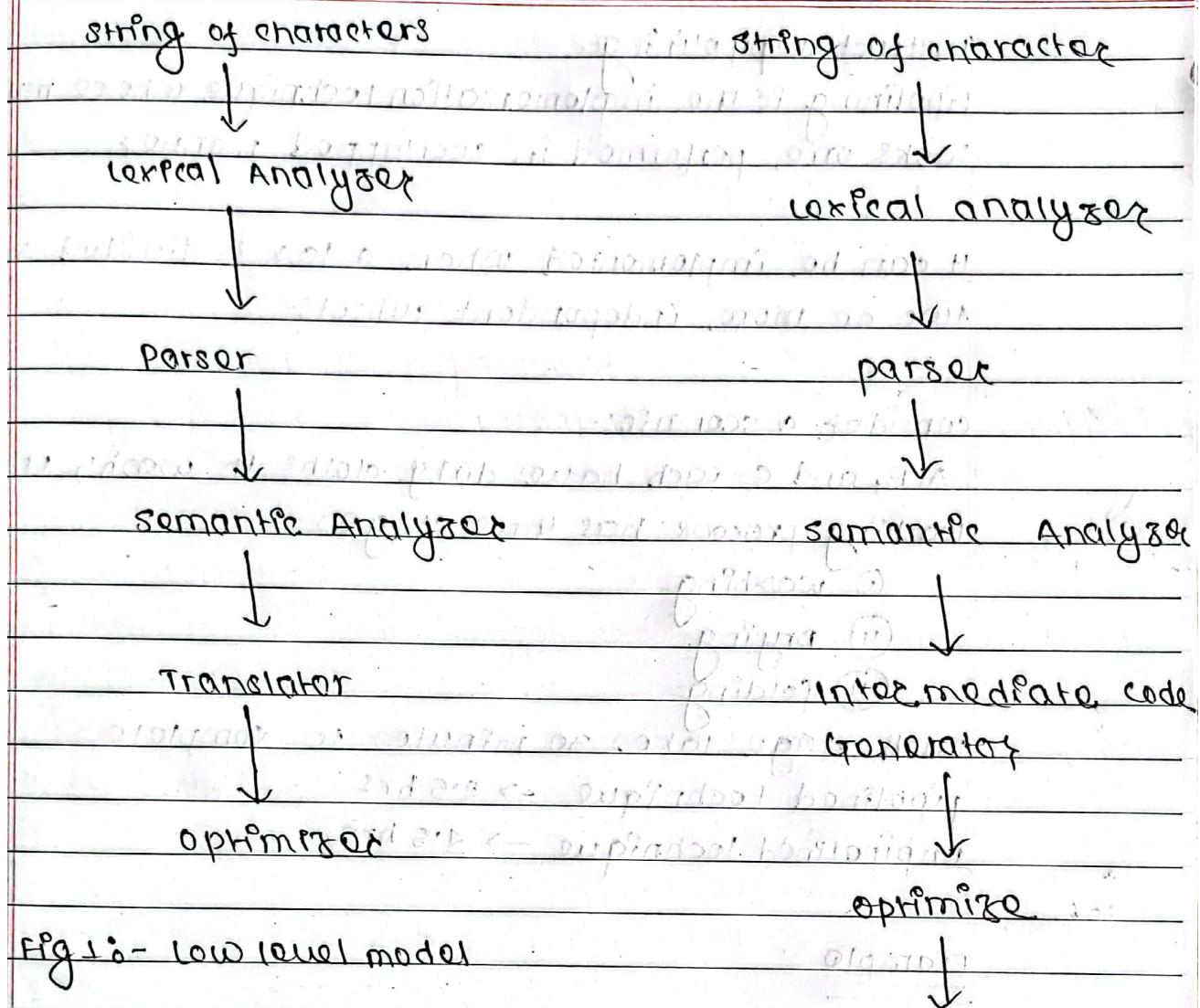


Fig 1:- Low level model

code generator

post optimizer

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 cloths to wash; the washing process has three stages.

- ① washing
- ② drying
- ③ folding

Each stage takes 30 minutes to complete.

Pipelined technique \rightarrow 2.5 hrs

Unpipelined technique \rightarrow 4.5 hrs

Example

	Instructions					cycles		
	①	②	③	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 \rightarrow Instruction fetch

ID \rightarrow Instruction decode

EX \rightarrow Execution

MEM \rightarrow Memory access

WB \rightarrow 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-process of a compiler.

#define PI 3.14

void main()

{

int PI, a,b

float area= PI * r * r;

printf ("PI = %f", PI);

}

Fig:- Input file

void main()

{
int PI,a,b;

float area= 3.14 * r * r

printf ("PI = %f", 3.14)

}

Fig:- output file.

Data Hazard

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

1. RAW (Read After Write)

2. WAR (Write After Read)

3. WAW (Write After Write)

(write back the
data in pipeline)

RAW

I1 : $R2 = R2 + R3$

I2 : $R5 = R2 + R4$

phases ① ② ③ ④ ⑤ ⑥

I1: IF ID EX MEM WB

I2: IF ID EX MEM WR

↑

I2 (fetch data in
3rd phase)

~~WAW~~

(data from I1 is available in I2)

WAR (Write After Read)

Write back the data in pipeline

data from I1 is available in I2

data from I2 is available in I3

Write back the data in pipeline

data from I2 is available in I3

data from I3 is available in I4

data from I4 is available in I5

Write back the data in pipeline

data from I5 is available in I6

data from I6 is available in I7

structural hazard

- different instructions accessing same resource

example:

```

IF ID EX MEM WB
    
```

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

refers to the machine that is designed to improve the performance of the execution of scalar instructions

Example:-

IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB

Assignment 1 :-

VLSI

Processor parallelism

Due date

20/7/9-08-15

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:-

$$s_1 : \pi = 3.14$$

$$s_2 : R = 5.0$$

$$s_3 : \text{Area} = \pi * R * R$$

- 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_3 because $\{s_1, s_2, s_3\}$ or $\{s_2, s_1, s_3\}$ both gives same result

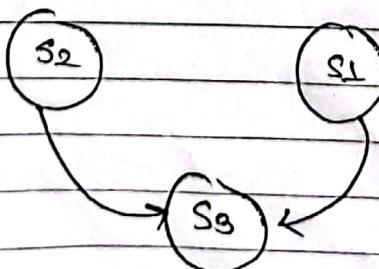


Fig:- Dependence in paths.

Lab no:-2

control hazard simulator.

I_L → 100

I₂ → 102

100

I₃ → 104

PC

I₄ → 108

I₅ → 108

Instruction : Add 100 + 102 = 202

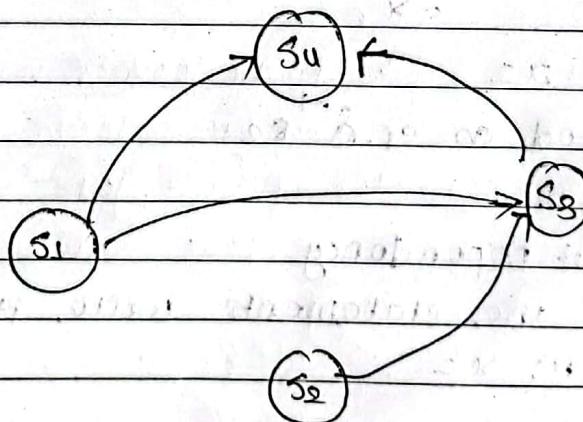
$$S_1 : a = 100$$

$$S_2 : b = 102$$

$$S_3 : c = a * b$$

$$S_4 : d = a + c$$

control proper!



Control Dependence:

Dependence due to control flow

1. If ($T = 0.0$) $s_1 \rightarrow s_2$

$s_2: A = A/T$

$s_2: \text{CONTINUE}$

Here s_2 cannot be executed before s_1 . Execution s_2 before s_1 could cause control hazards.

Load store classification

1. True dependence

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

$s_1: x =$

$s_2: x$

denoted by $s_1 \delta^s s_2$

2. Anti-dependence

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

$s_1: x =$

$s_2: x =$

denoted by $s_1 \delta^{-1} s_2$

3. Output dependency

Both the statements write pt into same location

$s_1: x =$

$s_2: x =$

denoted by $s_1 \delta^0 s_2$

Example:

S₁: $x = 2$

S₂: $y = 3$

S₃: $x = 2$

S₄: $w = x * y$

Find dependency type between S₁, S₂, and S₃.

for ($i=1$; $i \leq n$; $i++$)

 for ($j=1$; $j \leq n$; $j++$)

S₁: $x[i,j] = A[i,j] + Y[i,j]$

S₂: $Y[i,j] = C[i,j] + X[i,j]$

S₃: $X[i,j] = Y[i,j] + D[i,j]$

S₁, S₂ \Rightarrow True dependence.

S₂, S₃ \Rightarrow Antidirectional dependency

S₁, S₃ \Rightarrow 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

$$A_1 \leftarrow A_1 + B_1$$

$$A_2 \leftarrow A_2 + B_2$$

$$A_3 \leftarrow A_3 + B_3$$

Fig: (1)

DO I=1, N

 S₁: $A(I+2) = A(I) + B(I)$

END DO

$$A_3 = A_1 + B_1$$

$$A_4 = A_2 + B_2$$

$$A_5 = A_3 + B_3$$

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 arbitrary 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

$DO \ I = 0, 10, 2$

$s_1 <some\ statement>$

$END\ DO$

For $I = 4,$

$, u = 0 + 2$

S

$= 3$

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

Example :-

```

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

```

The iteration vector $s_1 [(2,1)]$ denotes the instance at s_1 executed during the second iteration of I and first iteration of J .

Iteration space.

The set of all 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^p and j^q for the nest, such that,

- $i^p < j^q$ or $i^p = j^q$ and there is a path from s_1 to s_2 in the body of the loop.

- s_1 accesses memory location M on iteration i^p and s_2 accesses M on iteration j^q

- one of these access 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 described 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 at the code, without adding or deleting any executions of any statements since a reordering transformation does not delete any statement executes any two executions that reference a common memory element before a reordering transformation will reference the same memory element after that transformation. Hence, if there is a dependence between the statements before the transformations, there will also be one afterward.

i.e. a reordering 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.

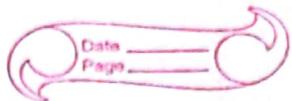
Reordering Transformations

- 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 reordering transformation will also reference the same memory element after that transformation.
- Hence, if there is a dependence between the statements before the transformations, there will also be one afterward.
- i.e a reordering transformation does not 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, \dots, s_n be the executing order in the original program.
- Let $\{i'_1, i'_2, \dots, i'_n\}$ be the permutation 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 conforms 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 ($s_m s^o s_k$)
- since this exhausts the way that s_k can get the wrong value, the result is proved by contradiction

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

$$d(i, j)_k = s_k - i_k$$

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

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

e.g.

for ($i=1, i \leq 100; i++$)

{

 for ($j=1 ; j \leq 100, j++$)

{

 for ($k=1, k \leq 200, k++$)

{

$$A[i+1, j, k] = A[i, j, k+1] + B$$

}

}

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

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

sink source

Distance vector

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

$$= (1, 0, -1)$$

$$\downarrow \quad \downarrow \quad \downarrow \\ < = > \quad \text{Direction vector}$$

e.g.

for ($i=1, i \leq N, i++$)

 |
 | for ($j=1, j \leq M, j++$)

 |
 | for ($k=1, k \leq l, k++$)

$$A[i+1, j, k-1] = A[i, j, k] + 10$$

g
g

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

\Rightarrow Let the direction vector be $\{1, 1, 1\}$

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

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 in 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
 $S_1 : A(I) = \dots$
 $S_2 : \dots = A(10 - I)$ \Rightarrow Loop
END DO

2. For ($i=1, i < n, i++$)

{

$$S_1: A[i] = A[i-1] + 1;$$

$$S_2: B[i] = A[i];$$

}

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

$S_1 \rightarrow S_2 \Rightarrow$ 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.

e.g.

for ($i=1; i < 4; i++$)

{

for ($j=1; j < 4; j++$)

{

$$S_1: A[i][j] = A[i][j-1] + 1;$$

}

j

	1	2	3
1	$O^{(1,1)}$	$O^{(1,2)}$	$O^{(1,3)}$
2	$O^{(2,1)}$	$O^{(2,2)}$	$O^{(2,3)}$
3	$O^{(3,1)}$	$O^{(3,2)}$	$O^{(3,3)}$

Parallelization

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

e.g. $A[0] = 1$

for ($i=1$; $i < N$; $i++$)

$$A_1 = A_0 + A_0$$

$$A_2 = A_2 + A_1$$

$$A[i] = A[i] + A[i-1]; \quad A_3 = A_3 + A_2$$

}

e.g. for $A[0] = 1$ no off-loop parallelism possible -

for ($i=1$; $i < N$; $i+2$)

$$A[i] = A_1 + A_0$$

$$A_3 = A_3 + A_2$$

$$A[i] = A[i] + A[i-1] \quad A_5 = A_5 + A_4$$

Dependence Testing

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

DO $i_1 = L_1, U_1$

DO $i_2 = L_2, U_2$

:

DO $i_n = L_n, U_n$

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

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

END DO

END DO

END DO

- dependence exist from s_1 to s_2 if
 $f_i(\alpha) = g_i(\beta)$ for all i , $1 \leq i \leq m$ and
 α, β are iteration vectors.

~~GD~~

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 $\text{GCD}(A, B)$ divides C , then dependency exists and in that case the loop is not parallelizable.

Solving diophantine equation

example: $70x + 112y = 160$

Step 1: Find $\text{GCD}(70, 112) = ? = 14$

$$\begin{array}{r}
 70) 112 (1 \\
 - 70 \\
 \hline
 42) 70 (1 \\
 - 42 \\
 \hline
 28) 42 (1 \\
 - 28 \\
 \hline
 14) 28 (2 \\
 - 28 \\
 \hline
 0
 \end{array}$$

Step 2: Find value of m and y

Reverse method

$$14 = 42 - 28$$

$$14 = 42 - (70 - 42)$$

$$14 = 42 - 70 + 42$$

$$14 = 2(42) - 70$$

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

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

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

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

(After some trial and error fashion) first add

and find $x = -36$ (long division will give us this)

so $y = 24$ (add 96 to both sides first)

so x always dependency on y (rest of steps will be similar)

Example: Is parallelism mean no dependency?

Find dependency in following loop which also

DO $i = 1, N$

$A(4i) = \text{something}$ (say A has 4 columns)

$= 4(2i+2)$ objects ($4, i$) and $2i+2$

so?

$$4x = 2y + 2$$

$$4x - 2y = 2$$

$$\text{GCD}(4, -2) = 2$$

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

Example:

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

for ($i=1; i \leq 10; i++$)

{

$$A[3 \times i + 20] = \dots$$

$$\dots = A[2 + i]$$

}

$$3x + 20 = 2y$$

$$3x - 2y = -20$$

$$\text{GCD}(3, -2) = 1$$

Example:

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

for ($i=1; i \leq 10; i++$)

{

$$A[3 \times i + 20] = \dots$$

$$\dots = A[2 \times i]$$

}

$$3x + 20 = 2y$$

$$3x - 2y = -20$$

$$\text{GCD}(3, -2) = 1$$

Exercise.

Find the dependence test in following

DO $I = 1, N$

DO $J = 1, N$

$$S_1: A[2i + 2j] = \dots$$

$$S_2: \dots = A(4i - 6j + 3)$$

ENO DO

END DO

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

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

$$(2a + 2b - 4a + 6b) = 3 \cdot$$

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

Find dependence (intra) from $s_1 \rightarrow s_1$ and $s_2 \rightarrow s_2$

DO $I = 1, 3$

DO $J = 1, 3$

DO $k = 1, 3$

$$s_1 : A(I+J, 2J+2, k+1) = B(I, 2J, k)$$

$$s_2 : C(I, J, k) = A(I, 2, 9-k)$$

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

e.g. DO I = 1, 100

$s_1: A(e_1) = \dots$

$s_2: \dots = A(e_2) + B(i)$

END DO

- where e_1 and e_2 are constants

- if $(e_1 - e_2) \neq 0$ then dependence does not exist.

SIV Testing

SIV

strong weak

zero crossing

SIV Test

- Assume a case

$$S_1: A(\dots, \alpha_1 i_k + c_1, \dots) = \underline{\quad}$$

$$S_2: \underline{\quad} = A(\dots, \alpha_2 i_k + c_2, \dots)$$

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 $a \in \mathbb{Z}$ and $|a| \leq N-L$

eg.

DO $I = 1, N$

$$S_1: A(I+2*N) = A(I+N)$$

END DO

Weak Zero SIV Test

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

- if a is an integer then dependence exist

eg.

DO $I = 1, N$

$$S_1: Y(I, N) = Y(1, N) + Y(N, N)$$

END DO

$$\downarrow \\ Y(1, N) = Y(1, N) + Y(N, N)$$

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

$$q = \frac{C_2 - C_1}{2\alpha}$$

- if $2q$ is integer the dependence exist

e.g.

DO $I = 1, N$

$$S_1 : A(I) = A(N - I + 1) + C$$

END DO

\downarrow

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