Unit - IV

Data structure support & code optimization

A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called a symbol table.

Symbol table :- A compiler uses S.T to keep track of scope & binding information about names.
(semantics of variable)

→ The symbol table is searched everytime a name is encountered in the source text. changes to the table occur if a new name or new information about an existing name is discovered.

→ It is used to store information about the occurrence of various entities such as, objects, classes, variable names, fn etc.

→ It is used by both analysis phase & synthesis phases.

L.A S.A SE.A I.C

I.C C.O T.C.S

→ The symbol table used for following purposes:

* It is used to store the name of all entities in a structured form at one place.
* Determine whether a given name is in the table & scope of variable
* Add a new name to the table.
* Access the info. associated with a given name,
* Add new info. for a given name.
* Delete a name or group of names from the table.

→ Initially the size of the S.T is fixed when the compiler is written; then the size must be chosen large enough to handle any S.P that might be presented.

⇒ It maintains an entry for each name in the following format: < Symbolname, type, attribute >.

↓

Symbol table stores an entry in this format.

Ex:- S.T has to store info about the following variable declaration: static int salary interest; then it should store the entry such as; < interest, int, static >

Symbol name → type → attribute

Operations of S.T:- The basic operations defined on a S.T include

1. allocate - to allocate a new empty symbol table
2. free - to remove all entries and free storage of S.T
3. lookup - to search for a name & return pointer to it's entry
4. Insert - to insert a name in S.T & return a pointer to its entry
5. set-attribute - to associate an attribute with given entry
6. get-attribute - to get an attribute associated with a given entry.

L-value & R-value :- The L-value & R-Value prefixes come from left & right side assignment..

Ex:- a := I + 1

↑ ↑

L-value R-value

Compiler uses following types of information from Symbol table.:

1. Data type 2. Name, 3. Declaring procedures 4. Offset in storage
5. If structure are record then Pointer to structure Variable,
6. Parameter passing values. 7. Number & type of arguments passed to the function 8. Base address.

There are two types of name representation

1. Fixed length names 2. Variable length names

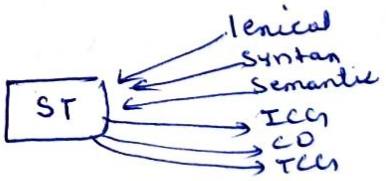
* A fixed space for each name is allocated in symbol table.

	name							attribute
c	A	L	C	U	L	A	T	E
s	U	M						
a								
a	v	g						

* Variable length Names

	Name	Attribute
starting index	length	
0	10	
10	4	
14	2	
16	2	

* Usage of S.T by various phases:-



Lexical:- scanner: creates entries for identifiers. (scans the entire given sp if any new identifier is occur that it created as entry in S.T)

Syntax:- Adds the info regarding attributes like line no, type, scope etc.

Semantic:- using available info checks semantics & updates the S.T.

ICG:- Available info helps in adding Temporary variables in formation

Code optimization:- Available info is used in machine dependent Optimization

TCG:- Generates the TC using address info of identifiers.

Symbol table Entries:-

= = = attributes

Name

Name	type	size	Dimension	line of Declaration	line of usage	Address
count	int	2	0	—	—	—
x	char	1	1	—	—	—

Initially S.T size is fixed.

Ex:- int count;

char x[] = "CVR College";

Symbol table operations:-

Insert(): It is more frequently used in analysis phase. when tokens are identified & names are stored in table.

→ It takes the symbol & its value in the form of argument.

Ex:- int x; insert(x, int). Syntax:- insert (variablename, type);

lookup():- It is used to search a name & it determine

* The existence of symbol in the table
* Declaration of the symbol before it is used
* check wheather the name is used in the scope
* Initilization of the symbol.
* checking whether the name is declared multiple items.

⇒ contains single variable non block structured language uses insert() & lookup()

Block structured lang insert() lookup() set() Reset()

variable declaration may happen multiple times

Syntax: lookup(symbol)

Symbol table organization:-

```
Var x, y : integer
Procedure P:
Var x, a : boolean;
procedure q:
Var x, y, z : real;
begin
----
end
```

z	real
y	real
x	real

symbol table for q

q	real
x	boolean
a	boolean

S.T for P

P	proc.
y	int
x	int

S.T for main

Implementation of symbol table:- The S.T can be implemented in the unordered list if the compiler is used to handle the smallest amount of data.

→ A S.T can be implemented in one of the following techniques.

1. Linear list (sorted or unsorted)
2. Hash table
3. Binary search tree (BST)

* Symbol table (S.T) are mostly implemented as hash table.

Linear list : ~~Here we use linked list~~.

The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

→ A linear list of records is the easiest way to implement S.T.

→ The new names are added to the symbol table in the order they arrive. whenever a new name is to be added first it searched linear to check if the name is already present in table or not. Time complexity - O(n), less space, additions are simple but it takes higher access time.

Name 1	info 1
Name 2	info 2
Name 3	info 3
:	:
Name n	info n

Linear list.

2. Hash table:- It is used to search the records of S.T. In hashing two tables are maintained a hash table and symbol table.

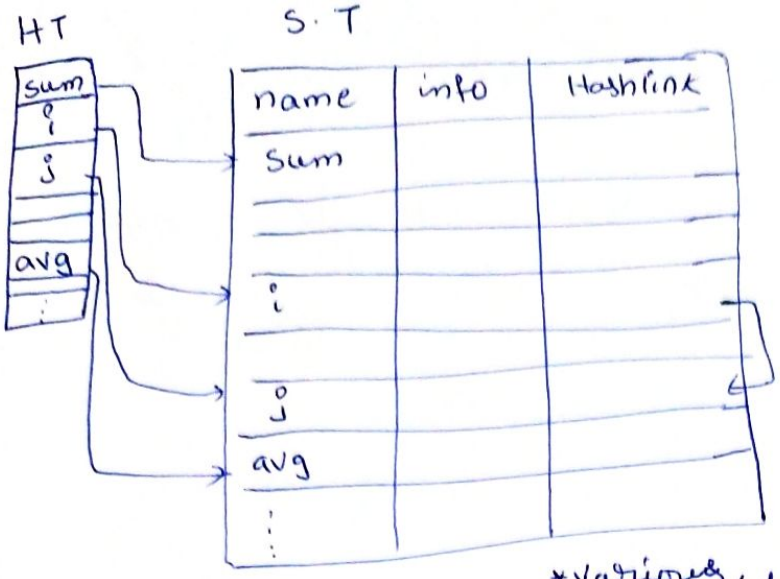
→ Hash table contains of 'k' entries from 0, 1 to k-1. These entries are basically pointers to S.T pointing to the names of S.T

→ To determine where the name is in S.T, we use a hash function 'h' such that h(name) will result any integer b/w 0 to k-1. We can search any name by Position = h(name)

7 7/10 = 7 12/10 = 2

→ using this position we can obtain the exact locations of name in S.T.

Hash tables:



Adv:- quick search

Disadv:- complicated to implement
Extra space is required, scope of variable is very difficult.

*The hash fn should result in uniform distribution of names in S.T.
* The hash fn should be s.T there will be min. no. of collisions.
*Collision is such a situation where hash fn results in same location.
*Various collision resolution techniques are 1. open adding 2. chaining 3. Rehashing

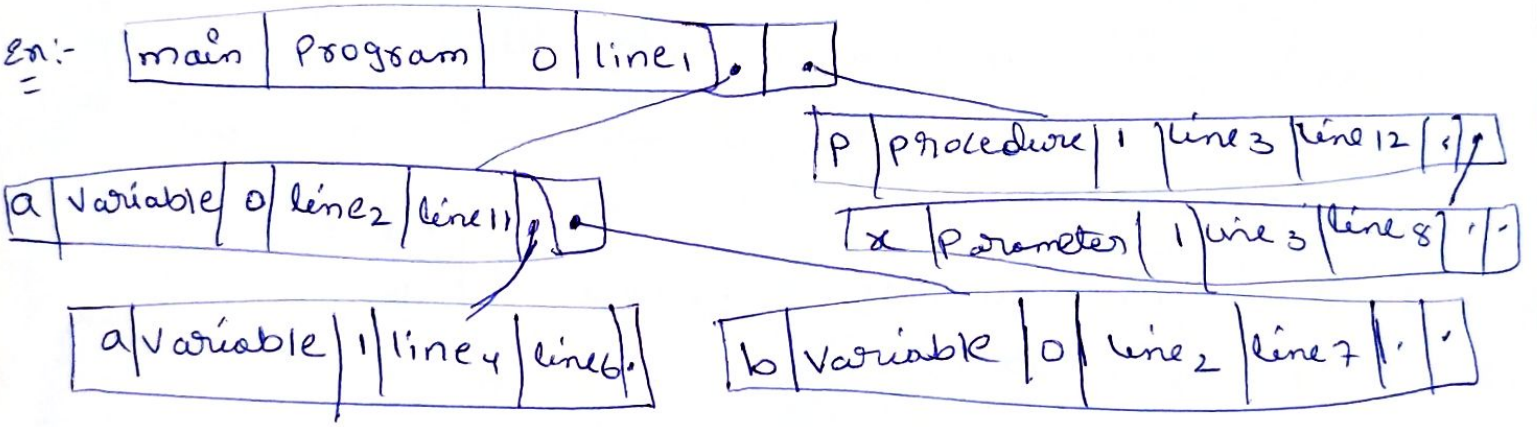
3. Binary trees:- Efficient approach for S.T organization. We add two links left & right in each record in the search tree.

→ whenever a name is to be added, first search in the tree, if it does not exists then a record for new name is created & added at proper position. (This has alphabetical accesibility).

→ Basic formate of BST is

leftnode	symbols	info	Rightnode

Ex:-



Adv: insertion of any symbol is efficient & searched efficiently.

Disadv: This structure consumes lot of space in storing

al ST.
Storage allocation strategies/storage organization:-

It is the simplest allocation scheme in which allocation of data objects is done at compile time because the size of every data item can be determined by the compiler.

The different ways to allocate memory in compiler is

1. Static storage allocation - Static storage
2. Stack storage allocation } - Dynamic storage
3. Heap storage allocation }

1. Static storage allocation (SSA):- The allocation of memory during compilation time is known as SSA. (ordinary, array variable).

→ In this once allocation of memory is done then it is not possible to change its size during runtime or execution time.

→ In static allocation, names are bound to storage locations.

→ If memory is created at compile time then the memory will be created in static area and only once.

→ Static allocation supports the dynamic data structure that means, memory is created only at compile time and deallocated after program completion.

→ The drawback with SSA is that the size & position of data object should be known at compile time & restriction of the recursion procedure.

2. Stack storage allocation:- storage is organized as stack (LIFO) Activation records are pushed & popped. Activation record contains the locals so that they are bound to fresh storage in each activation record

→ The value of locals is deleted when the activation ends.

→ It works on the basis of LIFO & this allocation supports the recursion process.

Activation record :- whenever a function or procedure call occurs then an activation record get created an activation record information is pushed on the top of the stack.

Fields of activation record :-

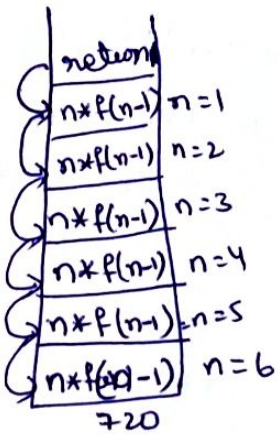
Actual parameters	→ It holds the actual parameters of calling fn
Returned values	→ To store the result of fn call.
control / dynamic calls	→ It Points to the activation record of calling fn.
Access link / Static link	→ It refers to the local data of called fn but found in another activation record
Saved machine status	→ Stores address of next instruction to be executed
local variables	→ These variables are local to a fn
Temporary variables	→ Needed during expression evaluation

Heap storage allocation :- It is the most flexible allocation scheme.

→ Allocation & deallocation of memory can be done at any time & at any place depending upon the users requirement.

→ Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back & supports the recursion process.

Ex :-
```
fact (int n)
{
  if (n<=1)
    return 1;
  else
    return n (n*fact(n-1));
}
fact(6)
```



activation ends & poped all records.

Block structure and non-block structure storage allocation:- (5)

The storage allocation can be done for 2 types of data variables

1. local data 2. Non local data (Global data).

→ local data can be accessed with the help of activation record.

→ Non-local data can be accessed using scope information.

* The block structured storage allocation can be done using static scope or lexical scope

* The non block structured can be done by dynamic scope.

→ In source program, every name possesses a region of validity called the scope of that name.

→ A block is a set of statements containing its own local data declarations. Delimiters marks the beginning & end of a block. 'C' uses the braces '{' and '}' as delimiters, while the ALGOL is to use begin and end.

→ B_1 & B_2 are two blocks and nested inside the other

```
B1
{
  B2
  {
  }
}
```

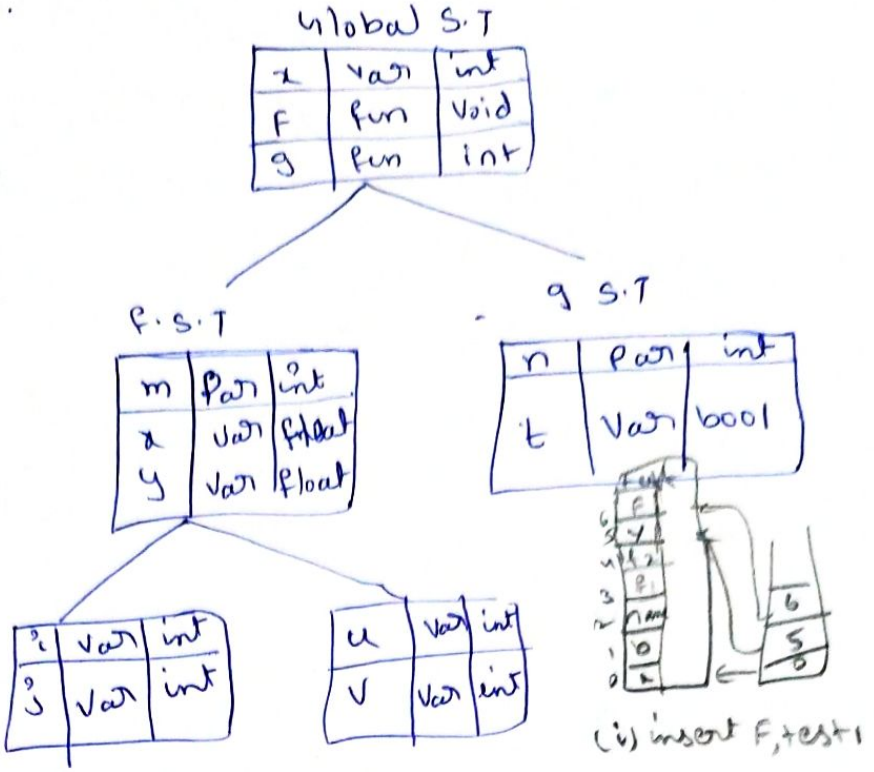
This nesting property is some times referred to as block structure. [B_1 begins, then B_2 but B_1 ends before B_2]

The Rules in a Block structured language are as follows:-

1. If a name declared within block B, then it will be valid only within B.
2. If B is nested within B_2 then the names that is valid for B_2 is also valid for B_1 unless the name's identifier is redeclared in B_1.
3. The scope rules need a more complicated organization of Symbol table than a list of associations b/w names & attributes.

When ever a new block is entered then a new table is entered onto the stack. The new table holds the name that is declared as local to this block.

Ex:-
```
int x;
void f(int m)
{ float x,y;
  { int i,j;
    int u,v;
  }
}
int g(int n)
{ bool t;
}
```



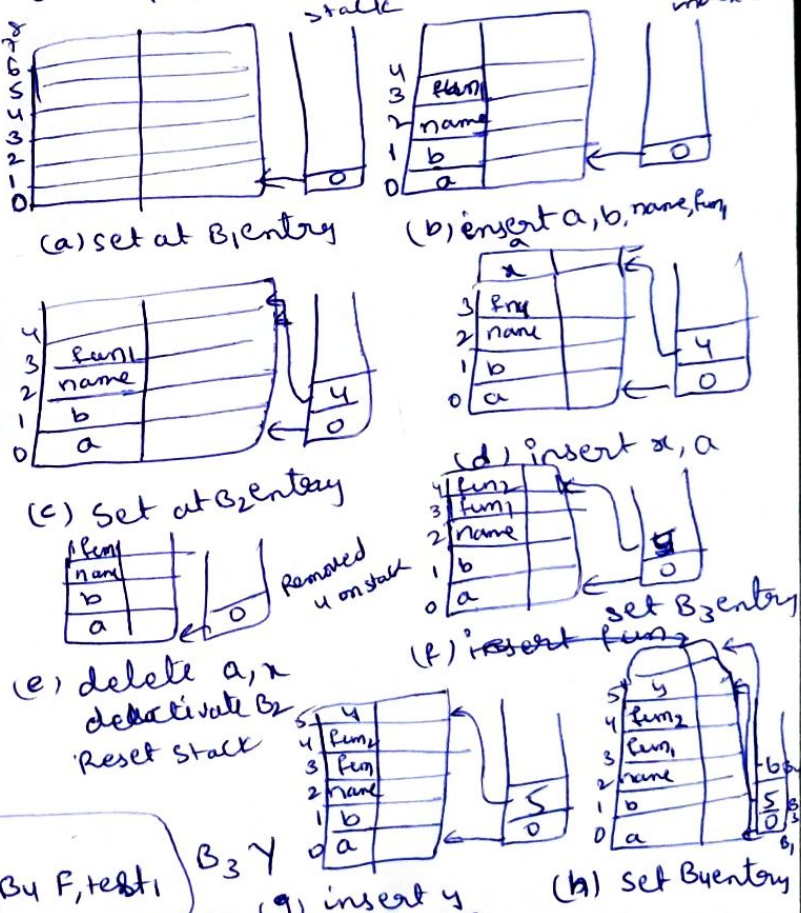
→ Block structure can be implemented using stack allocation.

Here we use insert(), lookup() & set() & reset() operations.

→ Concept of block structure was first introduced in ALGOL family as languages.

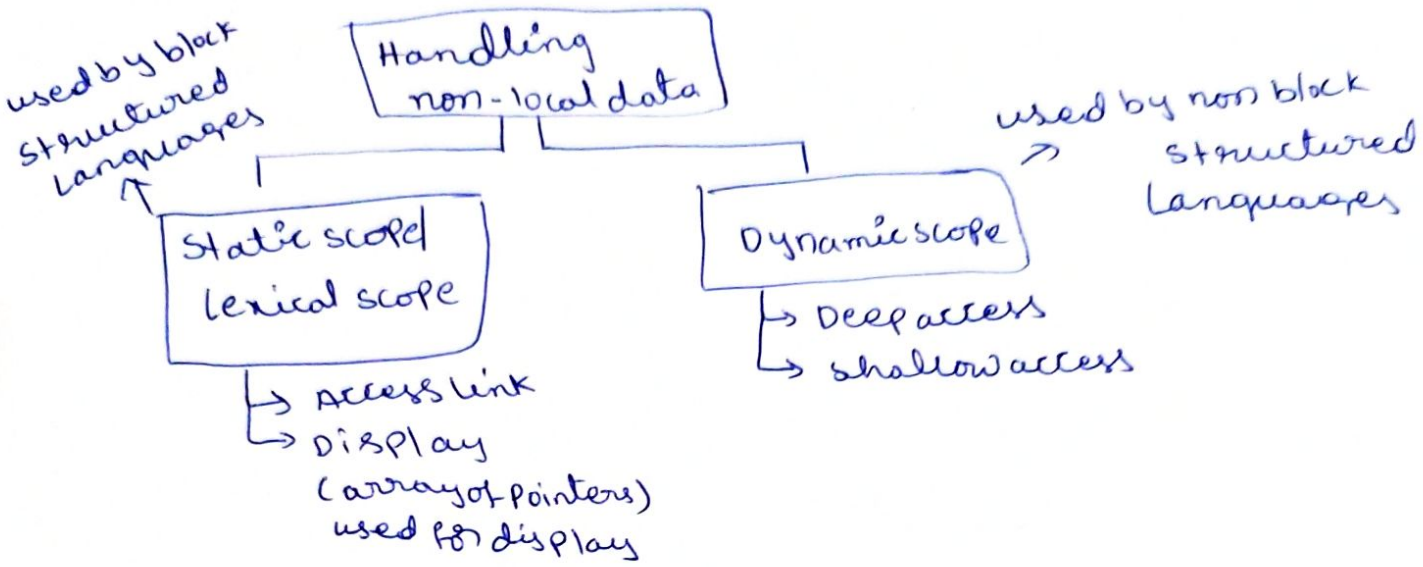
Ex:-
```
B1  main()
    {
      Real a,b;
      String name;
      =
      B2 fun1(int x)
B2    { int a;
        =
        call fun2(x+1)
        =
      } end fun1
      B3 fun2(int y);
      {
B4    { Array int f(y);
```

B1: a,b, name, fun1, fun2
B2: x,a, fun2
B3 y
B4 F, test1



Non block structure storage allocation :-

A Programming Language that does not permit the creation of blocks within a Program is a non block structured language Example of such a language is fortran.



Dynamic scope :-

Deep access : The idea is keep a stack of active variables, use control links instead of access links & when you want to find a variable then search the stack from top to bottom looking for most recent activation record that contains the space for desired variables. This method of accessing non local variables is called deep access. This method of accessing S.T is needed to be used at runtime.

Shallow access : The idea is to keep a central storage with one slot for every variable name. If the names are not created at runtime then that storage layout can be fixed at compile time otherwise when new activation of procedure occures, then that procedure changes the storage entries for its locals at entry and exit.

→ Deep access longer time to access the non locals while shallow access allows fast access.

Code optimization

It is the fifth stage/phase of a compiler. optimization is a program transformation technique used in synthesis phase, which tries to improve the code by making it consume less resources & deliver high speed.

→ code optimization produces straightforward code by using compiling algorithms can often be made to run faster or take less space or both.

→ The compiler that consists of code optimizers is called as optimizer compiler.

→ The compilers that considers highlevel languages as source and generates again high level language after code optimization. such type of compilers is called as source-source code optimization

→ The compilers that considers highlevel language as source and generates low level language, such type of compilers is known as simple optimization.

→ we widly used simple optimization process.

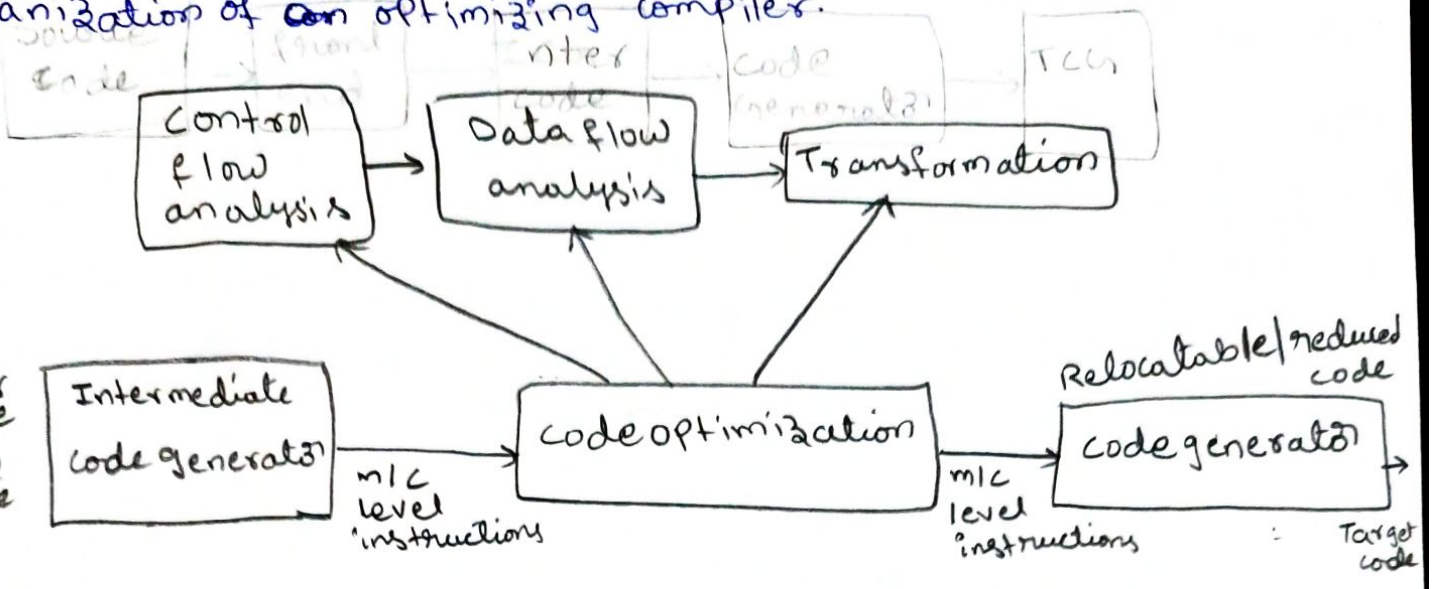
→ In this process must follow the three rules given below.

1. The optimization must be correct, it must not, in any way, change the meaning of the program.
2. Optimization should increase the speed of the program, if possible The compilation time must be kept reasonable.
3. The optimization process should not delay the overall compiling process.

→ The following are considerations or benefits of code-optimization

1. Increase in higher efficiency.
2. Reducing complexity
3. Reduce in size
4. Execution time is less
5. usage of space is less

Optimization can be done in almost all phases of compilation & Organization of an optimizing compiler.



→ Code optimization is located in b/w intermediate code generator and code generator phases. & code optimization is further divided into three phases.

1. Control flow analysis :- This sub phase will determine how control flows within the program of m/c level instructions.

2. Data flow analysis :- It corrects the data regarding the Program like no. of wanted instructions and no. of loops.

3. Transformation :- Various code optimization techniques are applied on the program in order to improve it after the completion of control flow and data flow analysis according to the situation.

→ The optimization can be categorized broadly into two types:

1. machine Independent
2. machine dependent.

1. are program transformations, that improve target code, without taking into consideration of any properties of target machine.

(or)

This code optimization phase attempts to improve the Intermediate code to get better target code as the output. The part of

intermediate code which is transformed here does not involve any CPU registers.

→ machine independent optimization is based on characteristics of the pgm languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the Execution time.

2. machine dependent optimization:

→ It is done after the target code has been generated & when the code is transformed according to the target m/c architecture. It involves CPU registers & may have absolute memory references. It put efforts to take max adv of memory hierarchy.

Scope of optimization: optimization techniques can be applied in source program based on scope of optimization. There are 2 ways to applied techniques.

1. Local optimization 2. Global optimization.

1. In this optimization consider the statement within basic blocks
2. It is applied to small block of statements.
3. Techniques used for this optimization are Variable propagation, common subexpression elimation.

2. Global optimization:-

→ when a global optimization has to be performed its scope is throughout the program or within some procedure.

2. It is applied to large program segments like fns, loops etc
3. Techniques used for this optimization are control flow analysis, Data flow analysis.

Basic blocks :- Source code generally have a no. of instructions which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks donot have any jump statements among them. i.e when the 1st instruction is executed, all the instructions in the same basic block will be executed in their seq of appearance without losing the flow control of the program.

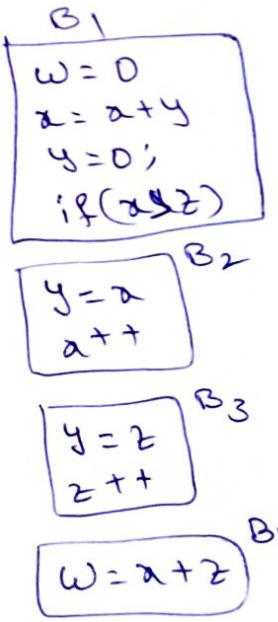
→ A program can have various constructs as basic blocks, like if-then-else, switch & loops such as Do-while, for etc.

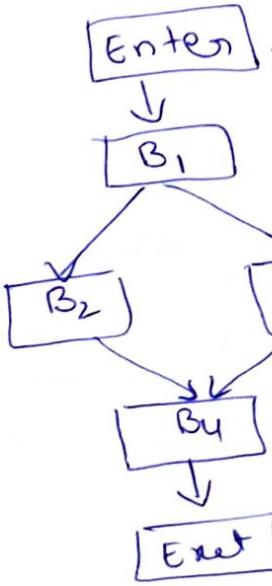
Basic block identification :- *1. It is impt concept from both CG & CD.
*2. It plays an impt role in identifying variables which are being used more than once in a single basic block.
*3. If any variable is being used more than once, the reg memory allocated to that variable is not emptied until the block finishes execution.

We may use the following algorithm to find the basic blocks in a program.

1. Search header statements of all the basic blocks from where a basic block starts.
   (a) First stmt of a prgm
   (b) Stmt's that are target of any branch.
   (c) Stmt's that follow any branch stmt.
2. Header stmt's and the stmt's following them form a basic block.
3. A basic block does not include any header stmt of any other basic block.

Ex :-
```
w = 0
x = x + y;
y = 0;
if (x > z)
{ y = x;
  x++;
}
else
{ y = z;
  z++;
}
w = x + z;
```





→ Basic blocks in a prgm can be represented by control flow graph.

→ It is a useful tool that helps in optimization

Control flow graph

The principal sources of optimization / code optimization techniques

Code optimization is done in the following different ways.

(1) Common sub-Expression elimination:

The common sub expression is an expression appearing repeatedly in the program which is computed previously, but the values of variables in expression have not changed at all then result of such subexpression is used instead of recomputing it each time.

Ex1:- a = b+c → Here c = b+c, b value is changed so, c is not replaced
b = a-d
c = b+c
d = a-d

after ⇒ optimization

a = b+c
b = a-d
c = b+c
d = b.

Ex2:

t1 = 4*i
t2 = a[t1]
t3 = 4*j
t4 = 4*i
t5 = n
t6 = b[t4]+t5

⇒

t1 = 4*i
t2 = a[t1]
t3 = 4*j
t5 = n
t6 = b[t1]+t5

; is not been changed from def.

x = a+b
a = b
z = a+b+10 ⇒ z = x+10 is not same

(2) Compile time evaluation:-

→ The evaluation is done at compile time instead of run time. It can be implemented in two ways.

(a) constant folding: Evaluate the exp and submit the result.

Ex1:- area = (22/7) * r * r Here 22/7 is calculated at compile time & result 3.14 is replaced so, area = 3.14 * r * r.

x = 12.4, y = x/2.3
evaluate x/2.3 as 12.4/2.3 at compile

(b) constant/variable propagation:- Here constant replaces a variable.

(or) variable propagation means use of one variable instead of another.

Ex1:- x = pi;
area = x * x * x;
↓ variable propagation
area = pi * x * x.

Ex2: pi = 3.14, r = 5
A = pi * r * r.
A = 3.14 * 5 * 5.

→ The diff b/w constant propagation & folding:
Variable is substituted with its assigned const.
↳ The variables whose values can be computed at compile time.

(3) Code motion / Code movement :-

There are 2 basic goals of code movement.

(a) To reduce the size of the code i.e, to obtain the space complexity.

(b) To reduce the frequency of execution of code i.e, to obtain the time complexity.

→ It is a technique which moves the code outside the loop if it won't have any difference if it executed inside or outside the loop.

Ex:- for(i=0;i<100;i++)
{ x=y+2;
a[i]=6*i;
}

⇒

x=y+2;
for(i=0;i<100;i++)
{ a[i]=6*i;
}

Ex2: for(i=0;i<=10;i++)
{ x=4*5;
k=(4*5)+50;
}

⇓

y=4*5;
for(i=0;i<=10;i++)
{ x=y;
k=y+50;
}

(4) Dead code elimination :-

→ A variable is said to be live in a prgm if the value contained into it is used subsequently.

→ The variable is said to be dead at a point in a program if the value contained into it is never been used.

⇒ The code containing such a variable supposed to be a dead code and optimization can be performed by eliminating such a dead code.

→ It includes eliminate those statements which are never executed or if executed output is never used.

Ex1:- i=0;
if(i==1) ⇒ i=0;
{ a=z+i;
}

Ex2: i=5; → dead assignment.
x=i+10;

Ex2:- int add(int x,int y)
{ int z;
z=x+y;
return(z);
printf("%d",z); → dead code stmt.
}

(5) strength reduction :- The strength of certain operators is higher than others for ex:- strength of '*' is higher than '+'.

→ In strength reduction technique the higher strength operators can be replaced by lower strength operators.

Ex:-
```
for(i=1;i<=50;i++)
{
  count = i*7;
}
```
⇒
```
temp=7;
for(i=1;i<=50;i++)
{
  count = temp;
  temp = temp+7;
}
```

* Thus we get the value of count as 7, 14, 21... upto less than 50.

Induction variables & reduction in strength :-

The induction variable is integer scalar identifier used in the form of $V = V \pm$ constant. Here V is a induction variable.

Loop optimization :- The code optimization can be significantly done in loops of the program.

→ Specially inner loop is a place where program spends large amount of time.

Hence if no. of inst's are less in inner loop then the running time of the program will get decreased to a larger extent.

So, loop optimization is a technique in which code optimization performed on inner loops.

→ There are several loop optimization techniques.

1. Loop invariant computation :- one statements in the loop whose results of the computations do not change over iterations.

Ex:-
```
a=10
b=20;
c=30;
for(i=0;i<5;i++)
{
  c=a+b; -①
  d=a-b; -②
  e=a*b; -③   } loop in variant computations
  s=s+i; -④
}
```

Here ①, ② ③ are eliminate in loop and define outside of the loop.

④ is depends on loop Variable 'i'

④ is in loop.

2. loop fusion (merge) :- In loop fusion method several loops are merged to one loop to reduce loop overhead & improve performance

Eg:-
```
for i=1 to n do
  for j=1 to m do
    a[i][j]=10
```
can be written as →
```
for i=1 to n*m do
  a[i]=10
```

Ex2:-
```
for(i=0;i<10;i++)
  a[i]=a[i]+10;
for(i=0;i<10;i++)
  b[i]=b[i]+10;
```
→
```
for(i=1;i<10;i++)
{ a[i]=a[i]+10;
  b[i]=b[i]+10;
}
```

3. Loop unrolling :- In this method the no. of jumps and tests can be reduced by writting the code 2 times.

→ loop overhead can be reduced by reducing the no. of iterations and replacing the body of loop.

Ex:-
```
for(i=0;i<100;i++)
{
add();
}
```
⇒
```
for(i=0;i<50;i++)
{ add();
  add(); } ② 50x2=100 times
}
```