unit-4

* Run-Time Environments :-

* The allocation and deallocation of data objects is managed by the run-time package. The design of the runtime is influenced by the semantics of procedure.
* each execution of a procedure is referred to as ativation of the procedure.
* If the procedure is recursive, several activations are alive at the some time.

* Issues in Source language :-

* A procedure is a declaration in its simplest form associates an identifier with a statement. The identifier is the procedure name and the statement is the procedure body. procedures that returns the values are called functions in any language. A complete program will also be treated as a procedure.

* Activation Trees :-

* some of the procedure call use a tree called activation tree to predict the control entry and leaves activations.

1) Each node represents an activation of a procedure
2) The root represents the activation of the main program
3) The node for 'a' is the parent for node 'b' if and only if control flows from activation 'a to b'
4) The node for 'a' is to the left of node for 'b' if and only if the life time of 'a' occurs before the life of 'b'.

* Control stacks :-

The flow of control in a program corresponds to a depth first traversal of the activation tree that starts at the root before its children.

we can use control stack to keep track of live procedure activation to push the node for an activation begins and to pop the node from the activation ends

The scope of a declaration :-

* A declaration in a language is a syntatic construct that associates information with a name.
* The scope rules determine which name appears in the text of a program.
* An address of a name in a procedure is said to be local if it is in the scope of a declaration within a procedure otherwise the occurence is said to be non-local'

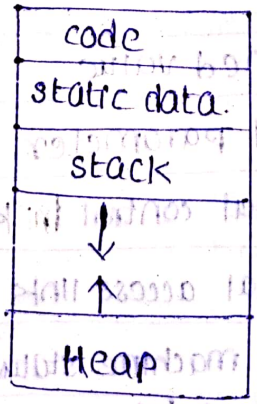
Binding :-

* It refers to the an association between the attribute and an entity. each name is declared once in a program but the same name denotes different data objects at run time.
* The term environment refers to a function that maps a name to a storage location and the term state refers to a function that maps the storage location to the value.

storage organization :-

The organization of the runtime storage in different language and might be sub-divided into

1) Generate the target (value) code
2) Data objects
3) control stack to keep track of the procedures



i.e Typical subdivision of run-time memory into code

and data areas.

* The size of the generated target code is fixed at compile time.
* The size of the data objects may also be known at compile time and can be placed in a statically determined area.
* when a call occurs the execution of an activation is interupted and the information about the status of the machine like program counter and registers is saved on to the stack.
* when control returns from the call this activations can be restarted after restoring the values of registers and program counters.
* A separate area of runtime memory called Heap holds all other information.
* The size of the stack in the heap can changeas the program executes. They can grow towards each other if needed.
* By convention stack grows down and heap grows up.
* The top of the stack is drawn towards the bottom of the page.

* Activation Record :-

* Information needed by a single execution procedure is managed using a continous block of storage called an Activation record (or) frame.
* The format of activation record is.

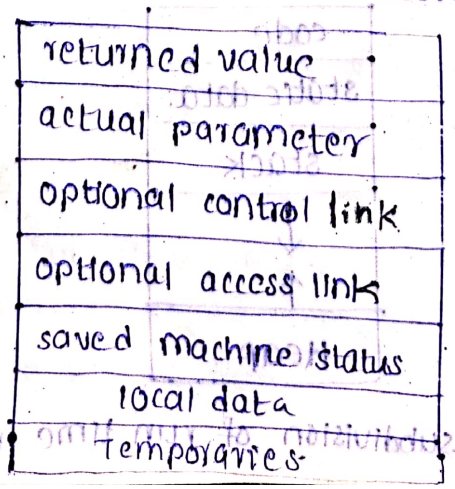


fig: format of activation record.

* Not all languages nor all compilers use all of these fields.

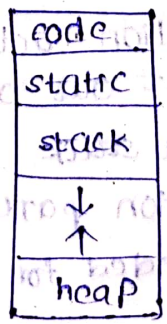
* The purpose of the fields of activation record is:

1) Temporay values which are used in the evaluation of expression.
2) The, local data holds the data that is local to an execution of procedure.
3) Saved machine status holds the state of the machine like program counter and machine registers.
4) Access links are optional to get non-local data.
5) controlling points to the activation record of the caller.
6) Actual parameters is used by the calling procedure to supply to the called procedures.
7) Return value is used by the called procedure to return a value to the calling procedure.

* Runtime storage Allocation strategies:-

A different storage allocation strategies is used in the three different areas:-

1) static allocation:- lays out storage for all type of data objects at compiler time.
2) stack allocation:- manges the runtime storage as a stack
3) Heap allocates:- some deallocates storage as needed at runtime from a data area known as head.



1) static allocation:

* In static allocation names are bound to storage (binding) as a programing compiled so there is no need for runtime support package.
* since the binding don't change at runtime, every time procedure is activated, it names are bound to the some storage allocation.

limitations of the static

1) The size of the data object in memory must be known as the compiler time.
2) recursive procedures are restructed because all activations of a procedure use the same bindings for local names
3) Datastructures can't be created dynamically

2) stack allocation:

* stack allocation based on the idea of control stack
* storage is organized as a stack and activation records are pushed and popped has activation begins and ends.
* local variables are bound to fresh storage in each activation because a new activation record is pushed on to the stack when a call is made. the value of the locals are deleted when the activation ends.

3) Heap allocation:

* It is possible with the heap allocation as:

1) The value of the local names must be retained when an activations ends
2) The activation trees is correctly depict the [between] flow of control between procedures
3) The deallocation of activation records need not occur in a last in first out order. so storage can't be organized as a stack.
4) Heap allocation parcels out pieces of continuous storage as needed for activation records, pieces

may be deallocated in any order, so the heap will consists of alternate areas that are free and are in use.

* for large blocks of the storage use the heap manager.

* procedure calls :-

* when one procedure calls another the usual method of communication between through non-local names to the parameters the called procedures.

* the common methods associated with actual and forward parameters is call by value, call by reference, copy restore, macro expansion.

1) call by value :-

it is a simplest method of parameters passing method the actual parameters are evaluated and their values are passed to the called procedure which is used in the c and c++ language. call by value can be implemented as

1) A formal parameter is treated like a local name, so the storage of the formal is activation record of the called procedure.

2) The caller evaluates the actual parameters and place the values in the storage and for the formals

2) call by reference :-

* when parameters are passed the reference (call by address / call by location) the caller passes through the called procedure a pointer to a storage address of each actual parameter.

* if an actual parameter is a name or expression having the values itself is passed.

* if the actual parameter is the expression that has no values then the expression is evaluated in new location and and the address of the location is passed.

3) copy restore :-

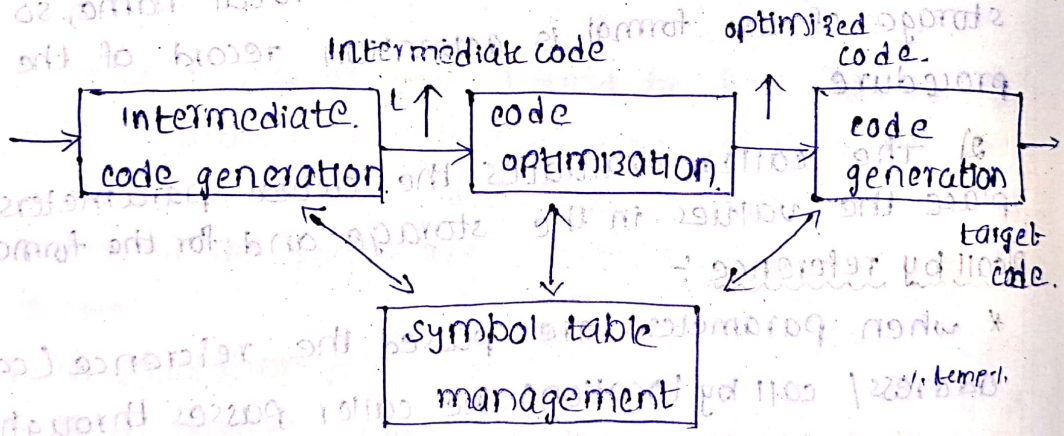
* It is a hybrid passing method between call by value & call by reference which is also called as copy restore (copy in copy out, value result)
* Before control flows to the called procedure the actual parameters are evaluated.
* when the control returns current values of the formal parameters are copied back to the values of the actual.

4) Macro Expansion :-

* which is also called as call by name. the procedure is treated as a keyword MACRO its body is substituted for all call in the caller with the actual parameters

such as substitution is called macro expansion or inline expansion. the local names of the called procedures are distinct from the names of the calling procedures and the actual parameters are surrounded by the paranthesis.

* Role of code optimization



The fifth phase of the compiler is the code optimization. It takes the input as intermediate code and produces output in the form of optimized code.

Advantages of code optimization

1) The intermediate code can be optimized to produce efficient object code.
2) The code optimization allows faster execution of the source code / program.

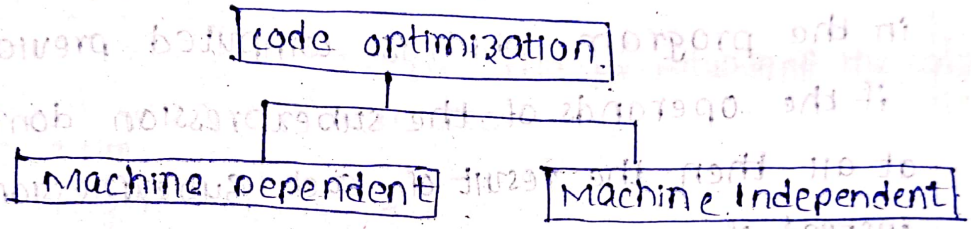
problems on the optimizing compiler:-

1) the design of the intermediate code must be simple. to optimize the code.
2) if intermediate code is complex then optimization becomes complex.
3) The optimization transformation must be applied carefully. otherwise code will become time and space inefficient

classifications of optimization:-

* The classification of optimization can be done in 2 categories

1) Machine dependent optimization.
2) Machine Independent optimization.



1) Machine Dependent:-

* It is based on charaterstics of the target code. which is acheived by allocation of sufficient number of resources to improve the execution efficiency
* it uses intermized instructions along with the data. increases the speed of the execution.

2) Machine Independent:-

* It is based on characterstics of the different programming language. It reduces the execution time.
* It uses appropriate programming structure to improve the efficiency of the target code
* A code should be analysis completely to produce a minimum amount of target code.

Principle sources of optimization.

A transformation of a program is called local if it can be performed by looking only at that statements. otherwise it is called global.

* They are number of phases in which a compiler can improve they program.

1) Common subexpression elimination.
2) Copy propagation.
3) Dead code elimination.
4) constant floding

1) Common subexpression Elimination:

the common subexpression is an expression repeated in the program which is computed previously if the operands of the subexpression don't changed at all then the result of such subexpression is used instead of recomputing at each time.

Ex: $t_1 = u * i$
$t_2 = a[t_1]$
$t_3 = u * j$
$t_4 = u * i$
$t_5 = n$

$\Longrightarrow$

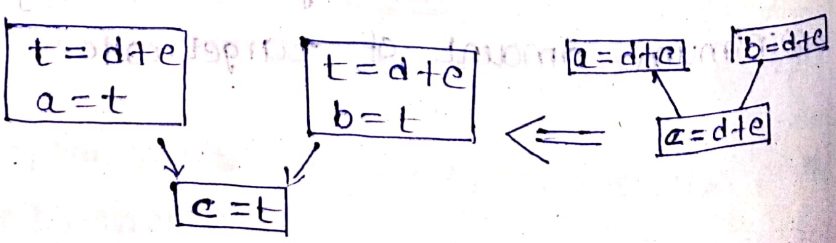
$t_1 = u * i$ [repetition statement]
$t_2 = a[t_1]$
$t_3 = u * j$
$t_4 = n$

The common expression 't4' is eliminated as its competition is already in t1 and the value of it is not changed from the subexpression.

2) copy propagation:

It means use of one variable instead of another variable.

Ex:



Here, the necessary condition is that the variable must can be assigned to another variable or constant.

3) Dead code elimination:-

* The variable is live at a point in a program. It is value can be used subsequently otherwise it is dead at that point.

Ex:-
```
i=0;
if (i=1)
{
  statement;
}
```

4) constant floding:-

* In the floding technique the computation of the constant is done at compile time instead of execution time.

Ex:- length = (22/7), *d

Here, floding is implied by perform the computation of 22/7 of compile time instead of execution time.

Strength reduction.

The strength of certain operator is higher than the others. For example the strength of * is higher than +. In this Technique higher strength operators can be replaced with lower Technique operators.

Ex:-
```
for(i=0; i<50; i++)
{
  count = i*7;
}
```
o/p - 7, 14, 21

⇒
```
temp = 7
for (i=1; i<50; i++)
{
  count = temp;
  temp = temp+7
}
```
o/p - 7, 14, 21

* **Loop optimization :-**

The code optimization can be done in the loops of a program, specifically inner loop is a place where program spends large amount of time.

They are number of different methods to carried out for the loop optimization.

1) code motion
2) Induction variable and strength reduction
3) loop invariant method.
4) loop unrolling
5) loop fusion.

1) **code motion :-**

It is a technique which moves the code outside the loop. if some expression in the loop whose result remains unchanged. even after the execution, the loop for several times. then such expression should be placed just before the loop.

Here, before the loop means at the entry of the loop

Ex:

```
while (i<=max-1)
{
    sum = sum+a[i]
}
```

```
n=max-1
while(i<=n)
{
    sum = sum+a[i]
}
```

2) **Induction variable and strength reduction :-**

A variable 'x' is called an induction variable of loop 'L', it is the value of variable changed every time it is by decremented or incremented or by constant

Ex:

```
i=i+1
t1=4*i
```

The value of i and t_1 are locked state. by the value of 'i' incremented by '1' then t_1 is incremented by 4.

∴ Hence i, t_1 are induction variables.

3) Loop invariant method :-

In this optimization technique the compution inside the loop is avoided and there by the compution over head and compiler is avoided.

Ex:

```
for i=0 to 10 do begin
    k=i+a/b
end
```

⟹

```
t=a/b
for i=0 to 10 do begin
    k=i+t
end
```

4) Loop unrolling :-

In this method the number of jumps and tests can be reduced by writing the code two times.

Ex:

```
int i=1
while (i<=100)
{
  a[i] = b[i]
  i++
}
```

⟹

```
int i=1;
while (i<=100)
{
  a[i]=b[i];
  i++
  a[i] = b[i];
  i++
}
```

5) Loop fusion :-

In loop fusion method several loops are merged to one loop.

Ex:

```
for i = 1 to n do
    for j = 1 to m do
    do { a[i,j] = 10
    }
```

⟹

```
for i = 1 to n*m do
    a[i]=10
```

* Peephole optimization :-

* It is a technique of machine dependent optimization if we apply statement by statement code generation startegy then the generated target code contains many redundant restructions so the quality of such code is very old. to optimize such target code contains transformation need to apply on the target code.

* peephole optimization is a simple and effective technique for improving the target code. this technique is applied for impove the performances of the target code by examining the short sequence of target instructions. PPO is and replacing these instructions by short sequence.

* Charaterstics of the peephole optimization :-

1) Redundant instruction elimination
2) Flow of contral optimization
3) Algebraic simplifications
4) use of machine idioms

1) Redundant instruction elimination :-

* The redundant loads and stores the can be eliminated by this transformation.

Ex :-

1) MOV R_0, a
   MOV a, R_0 ← redundancy

2) sum = 0 ← redundancy
   if (sum)
      print (sum)

3) Fun (a,b)
   {
      c = a+b
      return c ← redundancy (unreachable codes)
      print(c)
   }

repitions

2) Flow of control optimization :-

* using the peephole optimization technique unnecessary jumps can be eliminated.

```
goto test          |  goto done
...                |  ...
test: goto done    |  test: goto done
...                |  ...
done               |  done
```

unnecessary jumps need to be eliminated.

3) Algebraic simplifications :-

peephole optimization is an effective technique for Algebraic sepecifications.

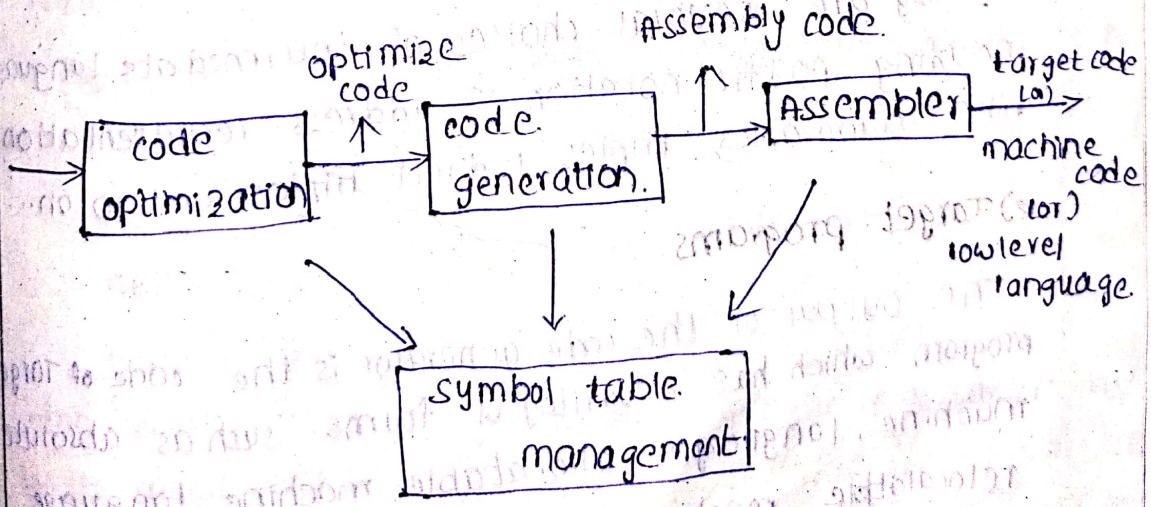
Ex:- $x = x + 0$

$x = x * 1$

4) use of machine idioms :-

the target instructions have equivalent machine instructions for performing some operations we can replace. [Hence we can] replace the target inspection file by machine equivalent instructions. In order to improve the efficiency.

* Role of code generation :-



It is the sixth phase of the compiler. it takes / notices the output in the form of target code (or) machine code (or). lowlevel language. and input is code optimization.

whenever the code generation phase procedures Assemble code it use another type of the translator which called assembler which converts assembly code into target code.

The low level language is in the form of binary coded language which is not understand to the human but it is understand to the machines.

Issues in the design of the

1) input to the code generator
2) Target programs
3) memory management
4) instruction selection.
5) ~~install~~ register allocation.

1) Input to the code generator

It consists of the intermediate representations of the source program produced by the front end together with the information in the symbol table that is used to determine the runtime addresses of the data objects denoted by the name.

They are several choices of intermediate language including postfix notation, 3-address representation like quadruples, triples, indirect triples and so on...

2) Target programs

The output of the code generator is the code of target program which has a variety of forms. such as absolute machine language, allocatable machine language relocalatble machine.

3) ~~Memory~~ ,

Absolute machine language has the advantage that can be placed in a fixed location in memory and immediately executed. Example .dcox exefile.

producing a relocatable machine language program as output allows the subprograms to be compiled separately.

producing the assembling language has output makes the process of code generation.

3) Memory management

Mapping names in the source program to the address of the data objects in runtime memory is done by the front end and the code generator. the type in a declaration determines the width the amount of storage is needed for the name. if the machine code is being generated labels in the 3-address code generated have to be converted to addresses of the Inspection.

4) Instruction selection

The nature of the insturction set of the target machine determines the difficulty of the instruction selection.

if the target machine doesn't support each datatype in a uniform manner then each exception to the general rule required the special handling

For example, for every creater 3-address code statement of the form $x=y+z$ can be translated in to the code sequence as:-

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
```

The quality of the generated target code is determine by its speed and size.

The target machine with a rich instruction set may provide several ways of implementing a given operation.

5) **Register allocation**

Instruction involve in register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. the use of the registers is subdivided into sub problems.

During register allocation we select the set of variables that will decide the register at a point in the program.

During a subsequent register assignment phase the specific register that a variable will decide residing.

The order in which the computations are performed can effect the efficiency of the target code. some computations order required fewer registers to hold intermediate results than others.

Basic Blocks :-

A basic block is a sequence block of consecutive statement in which the flow of control enters at the beginning and leaves at the end with out halt or possibility of branching

$t_1 = a * 5$
$t_2 = t_1 + 7$
$t_3 = t_2 - 5$
$t_4 = t_1 + t_3$
$t_5 = t_2 + b$

In 3-address code statement $a = b + c$ is to define 'a' and to use 'b' and 'c'

The name in the basic block is said to be live at a given point if its value is used after that point in the program.

The name in the basic block is said to be dead at a point which given if its value is not used never after that point in the program.

Algorithm for partitioning into blocks

Any given program can be partitioned into basic blocks by using the following algorithm. intermediate code. we assume that I c is already generated for given program.

step1: determine the leaders by using the following rules.

a) The first statement is a leader.

b) Any target statement of conditional (or) unconditional goto is a leader.

c) Any statement that immediately follows a goto (or) unconditional goto is a leader.

step2: The basic block is formed starting at leader statement and ending before the next leader statement appearing.

consider the following program code for computing the product of two vectors 'a' and 'b' of length '10' and partition into an basic blocks.

program code.

```
p=0
i=1
do
{
p=p+a[i] *b[i]
i=i+1
}
while(i<=10);
  a=b+c.
```

form of 3- address statement

1 p=0
2 i=1
3 $t_1 = 4*i$
4 $t_2 = a[t_1]$
5 $t_3 = 4*i$
6 $t_4 = b[t_3]$
7 $t_5 = t_2 * t_4$
8 $t_6 = p + t_5$
9 $t_7 = i+1$
10 $i = t_7$
11 if $i <= 10$ goto 3
12 $t_8 = b+c$
13) $a = t_8$

According to the algorithm statement1 is a leader by the rule step 1 – a

step-3 is also leader by the rule statement-1-b

12 is also leader by the rule step-1-c

Hence the statement 1, 2 form the Basic block.

Similarly statement 3 to 11 forms the another BasicBlock. and statement 12 and 13 forms another basic block.

p = 0 i = 1

B_1

$t_1 = 4*i$ $t_2 = a[t_1]$ $t_3 = 4*i$ $t_4 = b[t_3]$ $t_5 = t_2 * t_4$ $t_6 = p + t_5$ $t_7 = i+1$ $i = t_7$ if $i <= 10$ goto 3

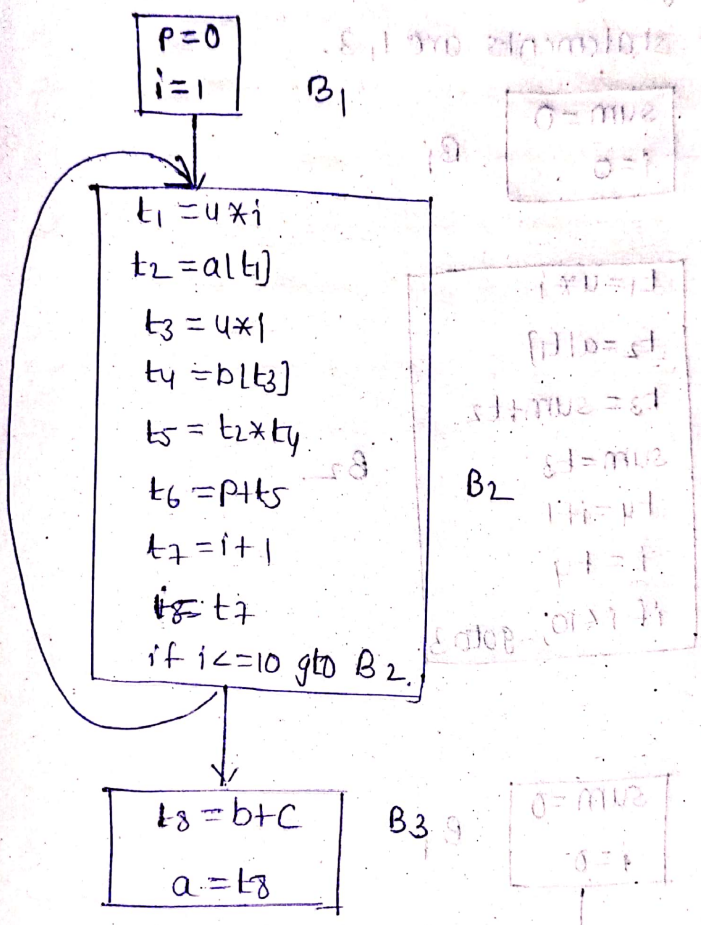
B_2

$t_8 = b + c$
$a = t_8$ B_3

Flow graph

A graph representation of the three address statements is called flow graph in which nodes represents basic blocks and which edges represent the flow of information.

From the above example we can construct the flow graph to connect the basic blocks in form of directed edges.

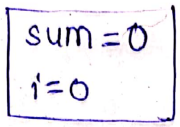


consider the following program code partition into basic blocks and draw the flow graphs.

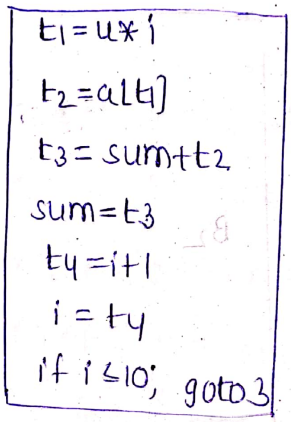
```
sum=0
for i=0; i≤10; i++;
  sum=sum+a[i]
```

```
1 sum = 0
2 for i=0
3 t1 = 4*i
4 t2 = a[t1]
5 t3 = sum + t2
6 sum = t3
7 t4 = i+1
8 i = t4
9 if i<=10; goto 3
```

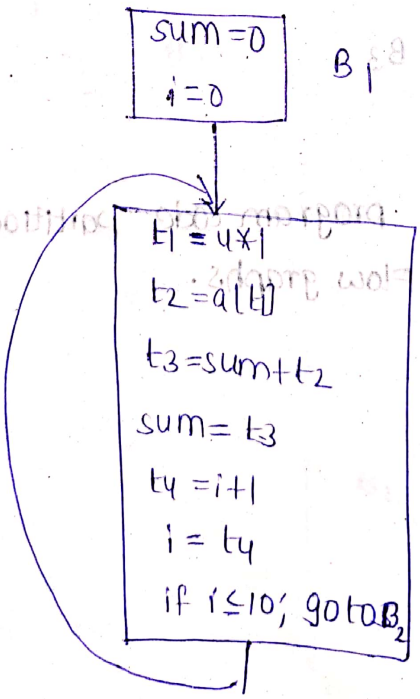
The leader statements are 1, 3.



B_1



B_2



B_1

B_2