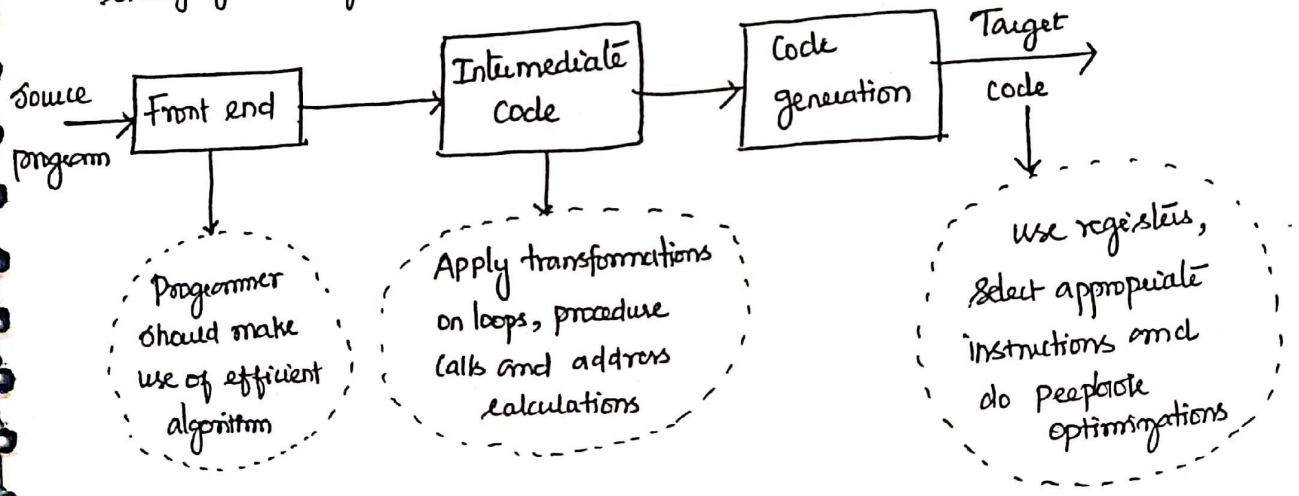
UNIT-V CODE OPTIMIZATION

- Introduction - principal sources of Optimization.
- Optimization of basic blocks
- Loop optimization.
- Introduction to Global Data flow analysis
- Runtime Environments - Source language issues
- Storage organization
- Storage Allocation strategies - Access to non-local names
- parameter passing

Introduction - Code Optimization

The code optimization is required to produce an efficient target code. There are two important issues that need to be considered while applying the techniques for code optimization.

1. the semantic equivalence of the source program must not be changed.
2. the improvement over the program efficiency must be achieved without changing the algorithm of the program.



Classification of Optimization:

1. Machine dependent => It is based on characteristics of target machine. for the instruction set used and addressing modes used for the instructions to produce the efficient target code.
2. Machine independent => Based on programming language characteristics such as programming structure, efficient algorithm etc..

Principal sources of Optimization:

⇒ The optimization can be done locally or globally. If the transformation is applied on the same basic block that kind of transformation is done locally otherwise transformation is done globally.

⇒ While applying the optimizing transformations the semantics of the source program should not be changed.

Function preserving transformations

There are a number of ways in which a compiler can improve the program without the changing the function it computes.

1. compile time evaluation — constant Folding, constant propagation.
2. common sub expression Elimination.
3. Copy propagation.
4. Code movement
5. Strength reduction.
6. Dead code Elimination.
7. Loop optimization

compile time evaluation:

compile time evaluation means shifting of computations from run time to compilation time.

(i) Folding ⇒ computation of constant is done at compile time instead of execution time.

Example: length = (22/7) * d

Here performing the computation of 22/7 at compile time.

(ii) constant propagation ⇒ Value of variable is replaced and computation of an expression is done at the compilation time.

Example:
```
Pi = 3.14;
r = 5;
Area = pi * r * r;
```

Here at the compilation time the value of Pi is replaced by 3.14 and r by 5 Then computation of 3.14 * r * r is done during compilation.

Loop invariant computation.

⇒ The loop invariant computation can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop. This method is also called code motion.

Example:

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

⇒

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

Strength reduction:

⇒ The strength of certain operator is higher than others. For instance strength of * is higher than +. In strength reduction technique the higher strength operators can be replaced by lower strength operators.

Example:

```
for(i=1; i<=50; i++)
{
  ....
  count=i*7;
  ....
}
```

⇒

```
temp=7;
for(i=1; i<=50; i++)
{
  ....
  count=temp;
  temp=temp+7;
}
```

Dead code elimination:

⇒ A variable is said to be live in a program 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.

Example:

```
i=j
....
x=i+10;
....
```

⇒ Here optimization can be performed by eliminating the assignment statement i=j.

```
i=0;
if(i=1)
{
  a=x+5;
}
```

⇒ Here if statement is dead code as this condition will never get satisfied, hence if statement can be eliminated and optimization can be done.

Common subexpression elimination:

⇒ The common subexpression is an expression appearing repeatedly in the program which is computed previously.

⇒ Then if the operands of this sub expression do not get changed at all then result of such sub expression is used instead of recomputing it each time.

Example:

```
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
```

The common sub expression $t4 = 4*i$ is eliminated as its computation is already in $t1$ and value of i is not been changed from definition to use.

Copy propagation:

Variable propagation means use of one variable instead of another.

Example:

```
x = Pi
....
area = x*r*r;
```

⇒

```
area = Pi*r*r
```

Here, variable x is eliminated. Here the necessary condition is that a variable must be assigned to another variable or some constant.

Code movement:

There are two basic goals of movement of code,

1. To reduce the size of the code, to obtain the space complexity.
2. To reduce the frequency of execution of code, to obtain the time complexity

Example:

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

⇒

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

Here, the code for $y*5$ will be generated only once.

Loop optimization:

Loop optimization is a technique in which code optimization performed on inner loops. the loop optimization is carried out by following methods.

1. Code motion
2. loop invariant method
3. Induction Variable and Strength reduction.
4. Loop unrolling
5. loop fusion.

Code motion:

⇒ Code motion is a technique which moves the code outside the loop.

⇒ If there lies some expression in the loop whose results remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop.

Example:

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

⇒

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

Induction Variables and reduction in Strength:

A Variable X is called an induction Variable of loop L. if the value of Variable gets changed everytime. It is either decremented or incremented by some constant.

Example,

```
B1
i = i+1
t1 = 4*j
t2 = a[t1]
if t2<10 goto B1
```

⇒ The value of i and t1 are in locked state. (ie) When value of i gets incremented by 1 then t1 gets incremented by 4. Hence i and t1 are induction Variables.

Reduction in Strength:

Higher strength operators can be replaced by lower strength operators.

Example,

```
for(i=1; i<=50; i++)
{
  ....
  count = i*7;
  ....
}
```

⇒

```
temp=7;
for(i=1; i<=50; i++)
{ ...
  count=temp;
  temp=temp+7;
}
```

Loop invariant method:

The computation inside the loop is avoided and thereby the computation overhead on compiler is avoided.

Example,

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

Loop unrolling:

The number of jumps and tests can be reduced by writing the code two times.

Example,

```
int i=1;                        int i=1;
while (i<=100)          ⇒       while (i<=50)
{ a[i]=b[i];                    {
  i++;                            a[i]=b[i];
}                                 i++;
                                  a[i]=b[i];
                                }  i++;
```

Loop fusion or loop jamming ⇒ Several loops are merged to one loop.

Example,

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

Problem: optimize the following code using various optimization technique.

```
i=1, S=0;
for (i=1; i<=3; i++)
  for( j=1; j<=3; j++)
    c[i][j]= c[i][j]+ a[i][j] + b[i][j];
```

Solution:

Consider two dimensional array representation in three address code

a[i][j] = 4*(20i+j) + (base -84)

Three address code:

```
    i=1
    s=0
    i=1
L2: j=1
L1: t1=4*i
    t1=t1+j
    t2=t1*4
    t3=a[t2]
    t4=i*20
    t4=t4+j
    t5=t4*4
    t6=b[t5]
    t7=t3+t6
    t8=i*20
    t9=t8+j
    t10=t8*4
    t11=c[t9]
    t12=t7+t11
    c[i][j]=t12
    j=j+1
    if j<=3 goto L1
    i=i+1
    if i<=3 goto L2
```

Now using common subexpression elimination,

```
    s=0
    i=1
L2: j=1
L1: t1=4*i
    t1=t1+j
    t2=t1*4
    t3=a[t2]
    t4=b[t2]
    t5=t3+t4
    t6=c[t2]
    t7=t5+t6
    c[i][j]=t7
    j=j+1
    if j<=3 goto L1
    i=i+1
    if j<=3 goto L2
```

Optimization of Basic block:

Two types of optimizations that can be done on basic blocks.

1. Structure preserving transformations
2. Use of algebric identities.

Structure preserving transformations:

⇒ Structure preserving transformations can be applied by applying some Principle techniques such as common sub-expression elimination, variable and constant propagation, code movement, dead code elimination.

⇒ The structure preserving transformation is a DAG based transformation. common subexpression can be easily detected by observing the DAG for corresponding basic block.

Consider,

$m = n * P$

$n = m + q$

$P = n * P$

$q = m + q$

if we assume the value of

$n = 1$

$P = 2$ and $q = 3$

then the expression becomes,

$m = n * P = 1 * 2$

$\boxed{m = 2}$

$n = m + q$

$n = 2 + 3$

$\boxed{n = 5}$

$P = n * P$

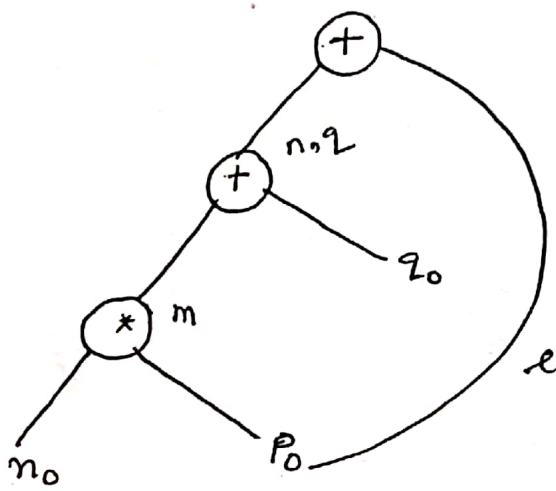
$= 5 * 2$

$\boxed{P = 10}$

$q = m + q$

$= 2 + 3$

$\boxed{q = 5}$



DAG for identifying common subexpression

⇒ Two common sub expressions such as $n * P$ and $m + q$. But for the common sub expression $n * P$ the value of n gets changed when it reappears.

Hence $n * p$ the value of $n * p$ is not a common subexpression.

⇒ But the expression $m + q$ gives the same result in repetitive appearance and values of m and q are consistent each time.

Therefore, $m + q$ is supposed to be the common subexpression. Ultimately n and q are the same.

2. Use of algebric identities :

Algebric identities are used in peephole optimization techniques.

Example
$a+0=a$
$a*1=a$
$a/1=a$
} => using strength reduction technique => instead of using $2*a$, we can use $a+a$.
Instead of using $a/2$, we can use $a*0.5$

∴ use of lower strength operator instead of higher strength operator makes the code efficient.

constant folding => Instead of using $a=2*5.4$, we can use $a=10.8$. This saves the effort of compiler in doing computations.

common subexpression elimination => use of associativity and commutativity.

Example: $x=y*z$, $t=z*r*y$ } => $x=y*z$, $t=x*r$

Global Data Flow Analysis

=> Global optimization is applied over a broad scope such as procedure or function body.

=> Global optimization : A program is represented in the form of program flow graph. the program flow graph is a graphical representation in which each node represents the basic block and edges represent the flow of control from one block to another.

=> There are two types of analysis performed for global optimizations

1. control flow analysis
2. Data flow analysis.

control and data flow analysis:

=> control flow analysis determines the information regarding arrangement of graph nodes, presence of loops, nesting of loops and nodes visited before execution of a specific code.

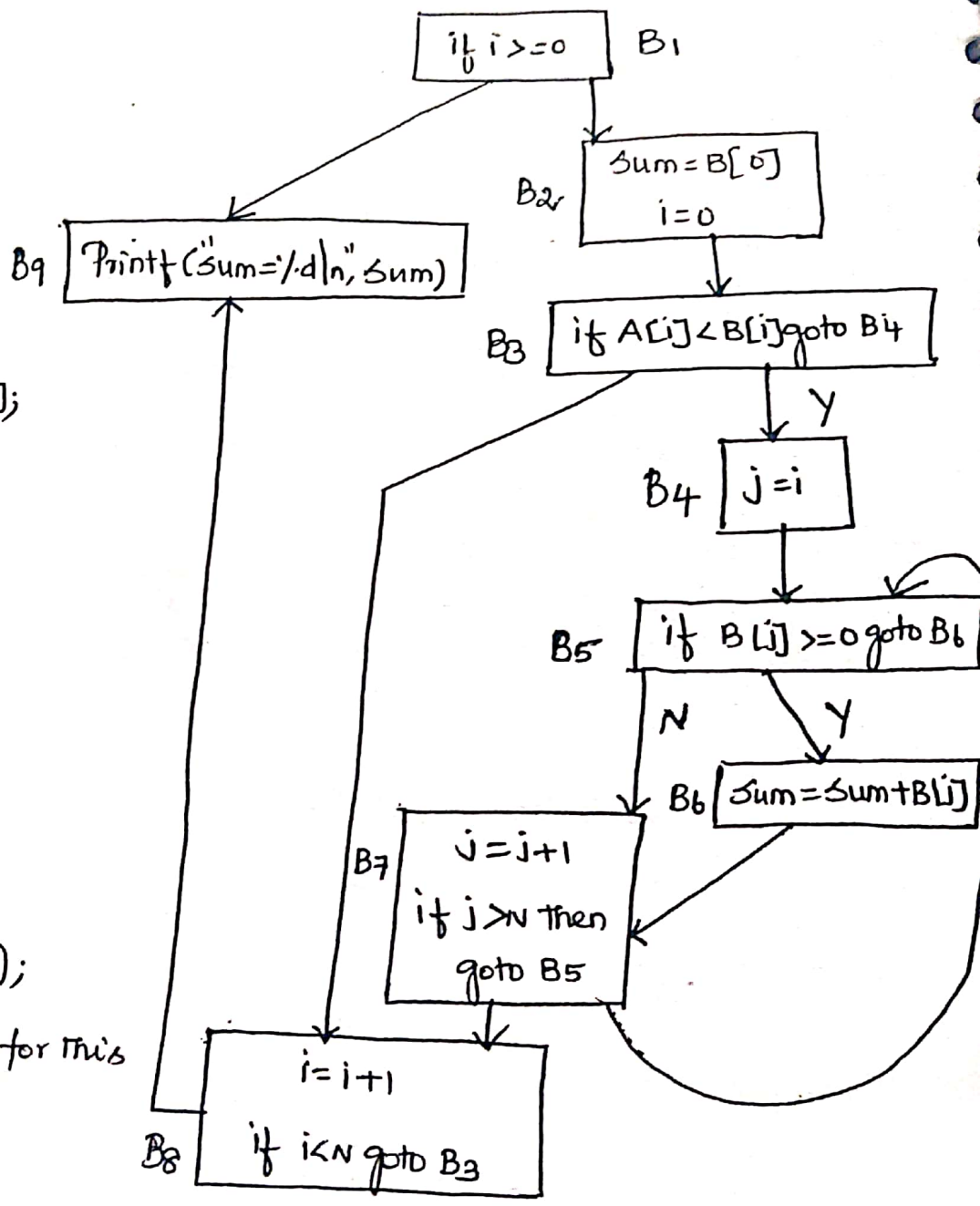
⇒ In data flow analysis the analysis is made on the data flow. Data flow analysis determines the information regarding the definition and use of the data in the program.

⇒ The data flow analysis is basically a process in which the values are computed using data flow properties.

Example: Consider following progeam,

```
if (i>=0)
{
  sum = B[0];
    i = 0;
L1: if (A[i] < B[i])
  {
    j = i;
L2: if (B[i]>=0)
  {
    sum = sum + B[j];
  }
  j = j+1;
 if(j<N) goto L2;
}
  i = i+1;
}
  i = i+1;
 if(i<N) goto L1;
}
printf("sum = %d\n", sum);
```

Draw a control flowgraph for this code.



Run Time Environment

Source language Issues:-

⇒ There are various language features that affect the organisation of memory. The source language issues are,

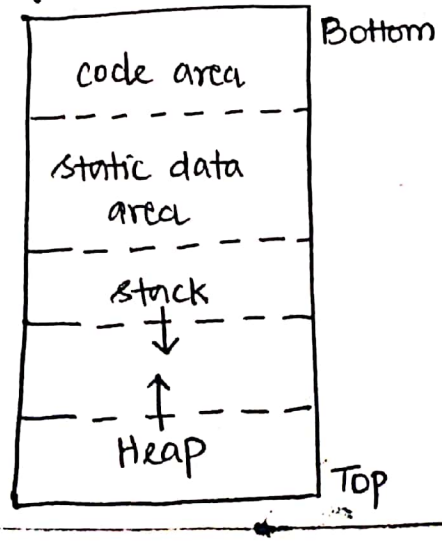
- Does the source language allow recursion?
  - ⇒ While handling the recursive calls there may be several instance of recursive procedures that are active simultaneously.
  - ⇒ Memory allocation must be needed to store each instance with its copy of local variables and parameter passed to that executive procedure
  - ⇒ But the number of active instances is determined by runtime.
- How the parameter are passed to the procedure?
  - ⇒ There are two methods of parameter passing: call by value and call by reference the allocation strategies for each of these methods are different.
  - ⇒ Some language support passing of procedure itself as parameter or a return value of the procedure itself could be procedures.
- Does the procedure refer nonlocal names? How?
  - ⇒ Any procedure has a access to its local names. But a language should support the method of accessing non local names by procedures.
- Does the language support the memory allocation and deallocation dynamically?
  - ⇒ The dynamic allocation and deallocation of memory brings the effective utilization of memory.

Storage organization:-

⇒ The compiler demands for a block of memory to operating system. The compile utilizes this block of memory for running the compiled program. This block of memory is called run time storage

⇒ Run-time storage is subdivided to hold code and data such as,

i) The generated target code
ii) Data Objects
iii) Information which keeps track of procedure activations.



⇒ The size of generated code is fixed. Hence the target code occupies the static determined area of the memory. compiler places the target code at the lower end of the memory.

⇒ The amount of memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory.

⇒ compiler prefers to place the data objects in the statically determined area because these data objects then can be compiled into target code.

⇒ The counterpart of control stack is used to manage the active procedure. Managing of active procedures means that when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack.

⇒ When the control return from the call this suspended activation is resumed after storing the values of relevant registers. Also the program counter is set to the point immediately after the call.

⇒ This information is stored in the stack area of runtime storage.

⇒ The heap area is the area of runtime storage in which the other information is stored. For example, memory for some data items is allocated under the program control.

⇒ Memory required for these data items is obtained from this heap area. Memory for some activation is also allocated from heap area.

Storage Allocation Strategies

There are three different storage allocation strategies based on the division of runtime storage.

1. Static allocation ⇒ The static allocation is for all the data object at compile time.
2. Stack allocation ⇒ stack is used to manage the runtime storage.
3. Heap allocation ⇒ Heap is used to manage dynamic memory allocation.

Static Allocation:

⇒ Static allocation is done for all data objects at compile time.

⇒ Data structure cannot be created dynamically because in static allocation compiler can determine the amount of storage required by each data object.

⇒ Memory allocation: The names of data objects are bound to storage at compile time.

⇒ Merits and limitations: This allocation strategy is simple to implement but supports static allocation only. Similarly recursive procedures are not supported by static allocation strategy.

Stack Allocation:

⇒ In stack allocation, stack is used to manage runtime storage.

⇒ Data structures and data objects can be created dynamically.

⇒ Memory allocation: Using last In FIRST OUT (LIFO) activation records and data objects are pushed on to the stack. The memory addressing can be done using index and registers.

⇒ Merits and limitations: It supports dynamic memory allocation but it is slower Than static allocation strategy.

Supports recursive procedures but references to non local variables after activation record cannot be retained.

Heap allocation:

⇒ In heap allocation, heap is used to manage dynamic memory allocation.

⇒ Data structures and data objects can be created dynamically.

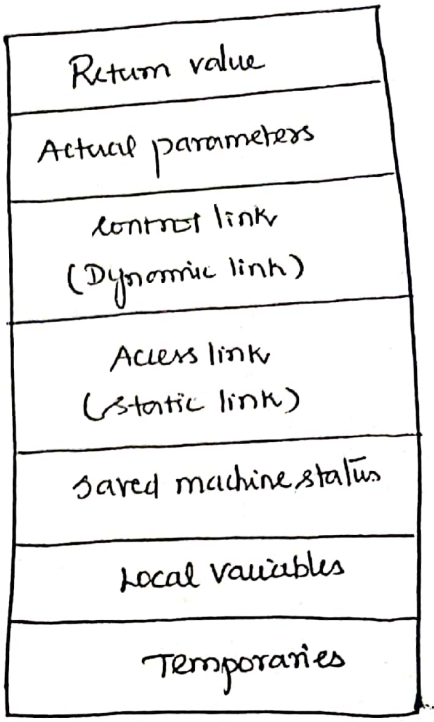
⇒ Memory allocation: A contiguous block of memory from heap is allocated for activation record or data object. A linked list is maintained for free blocks.

⇒ Merits and limitations: Efficient memory management is done using linked list. The deallocated space can be reused. But since memory block is allocated using best fit, holes may be introduced in the memory.

Activation Record

⇒ The activation record is a block of memory used for managing information needed by a single execution of a procedure.

⇒ Various fields of activation records are,

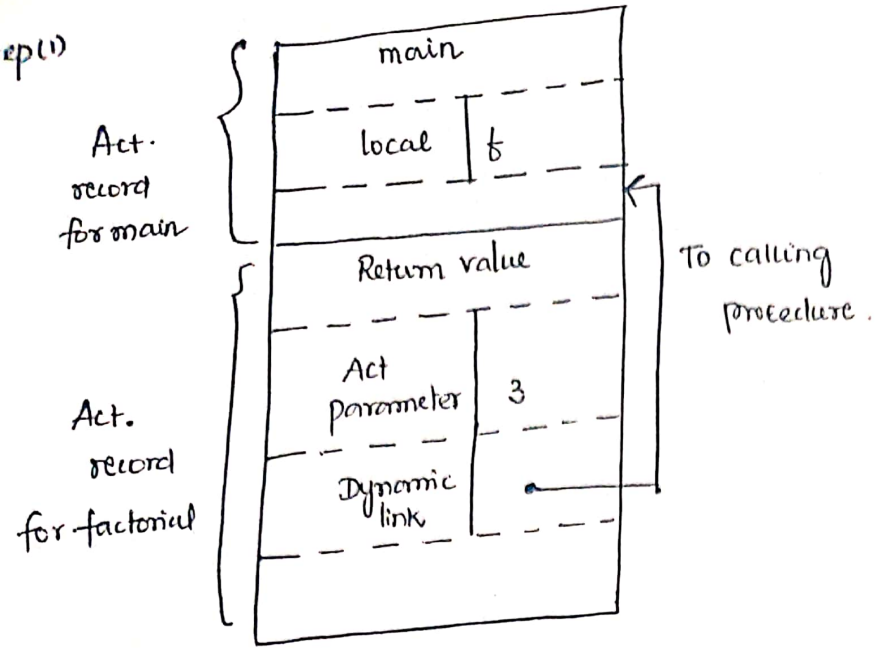


1. Temporary values ⇒ The temporary variables are needed during the evaluation of expressions. Such variables are stored in memory field of activation record.
2. Local variables ⇒ The local data is a data that is local to the execution of procedure is stored in this field of activation record.
3. Saved machine registers ⇒ This fields hold the information regarding the status of machine just before the procedure is called.
   This field contains the machine registers and program counter.
4. Control link ⇒ This field is optional. It points to the activation record of the calling procedure. This link is called dynamic link.
5. Access link ⇒ This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.
6. Actual parameters ⇒ This fields holds the information about the actual parameters. These actual parameters are passed to the called procedure.
7. Return values ⇒ This field is used to store the result of a function call.

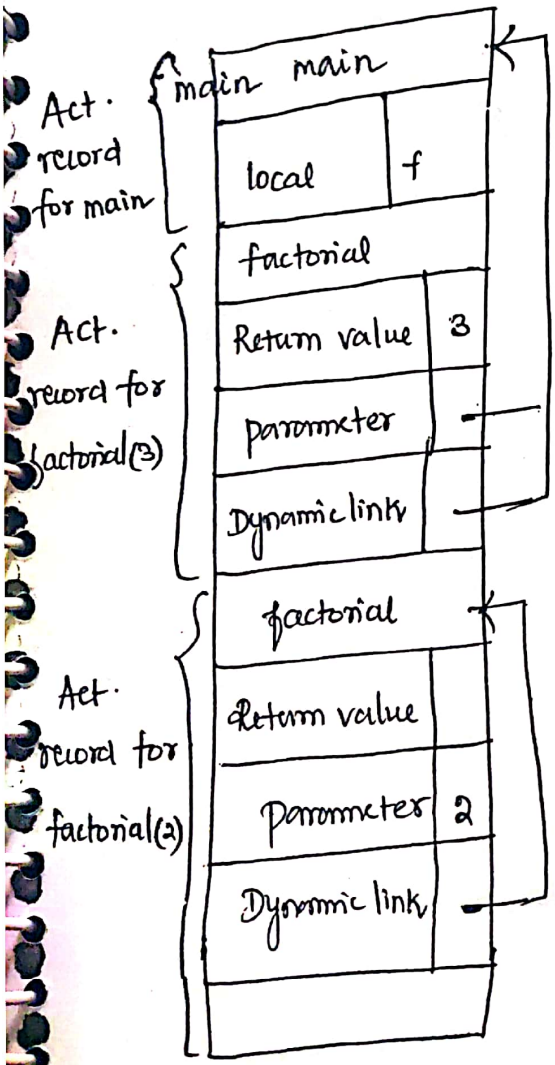
Example,

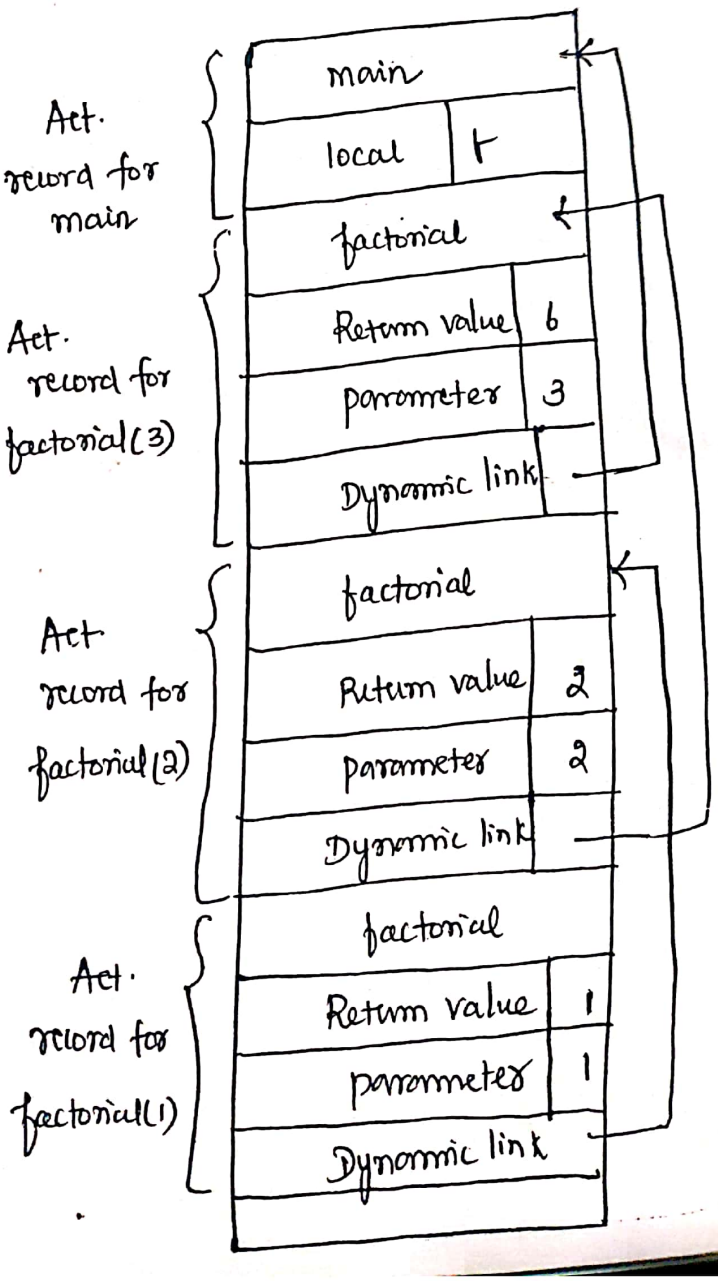
```
main()
{
  int f;
  f = factorial(3);
}
int factorial(int n)
{ if(n==1)
    return 1;
  else
    return (n* factorial(n-1));
}
```

Step(1)



Step(2)





Parameter passing

There are two types of parameters.

i) Formal parameters

ii) Actual parameters.

Most common methods are,

1. Call by value => Actual parameters are evaluated and their r-values are passed to called procedure.

   C, C++ use actual parameter passing method.

2. Call by reference => Also called as call by address or call by location. The L-val the address of actual parameter is passed to the called procedure activati record.

   C++, PASCAL's var parameters.

3. Copy restore => Hybrid between Call by value and call by reference. This method is known as copy-in-copy-out or values result.

   => The calling procedure calculate the value of actual parameter and it is then copied to activation record for the called procedure.

   => During execution of called procedure, the actual parameters value is not affected.

   => If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

   'Ada' this parameter passing method is passed.

4. Call by name => procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formals.

   => The actual parameters can be bounded by paranthesis to preserve their integrity.

   => The local names of called procedure and names of calling procedure are distinct.

   'ALGOL' uses call by name method.