* Intermediate Code Generator

* Symbol table

- Symbol table is series of rows each row containing list of attributes value that are associated with a particular variable.

- Typical View of a symbol table (1.1)
  - 7 attributes

(1.1)

	Variable Name	Address	Type	Dimension	line Declared	line referred	Pointer
1.	COMPANY#	0	2	1	2	9,14,25	7
2.	X3	4	1	0	3	12,14	0
3.	FORM 1	8	3	2	4	36,37,38	6
4.	B	48	1	0	5	10,11,13,23	1
5.	ANS	52	1	0	5	11,23,25	4
6.	M	56	6	0	6	17,21	2
7.	FIRST	54	1	0	7	28,29,30,38	3

up down (z axis)
forword - backword (y-axis)
begin(0,0) left write(x-axis)

Drawbacks

① A major Problem in Symbol table organization can be the variability in the length off identifiers (variable) Names.

To solve this Problem we have many solution but main approches are there.

1. which facilitates quick table access and another.
2. which suppose the efficient storage of variable name.

Sol^n ① we will give max size (eg. 16) to all variables in list.

② String Descriptor (in details):-

Variable name		other Attributes					
Position	length	Address	Type	Dimension	line Declared	line referred	Pointer
1	8	0	2	1	2	9,14,25	7
9	2	4	1	0	3	12,14	0
11	5	8	3	2	4	36,37,38	6
16	1	48	1	0	5	10,11,13,23	1
17	3	52	1	0	5	11,23,25	4
20	1	56	6	0	6	17,21	2
21	5	54	1	0	7	28,29,30,38	3

String area:-

C	O	M	P	A	N	Y	#	X	3	...

* Type:
1 - real no.
2 - int no.
3 - character
6 - Proc

* Dimension:
0 - Simple form
1 - vector form
2 - Metricies

→ line Declared :- In which line it is declared and if declared again in below Program then generates error so the sol^n is cross-reference.

Name	Type	Dimension	Declared Attr.	Reference
ANS	real	0	5	11,23,25
B	real	0	5	10,11,13,23
COMPANY#	int	1	2	9,14,25
FIRST	real	0	7	28,29,39,38
FORM1	char	2	4	36,37,38
M	Proc	0	6	17,21
X3	real	0	3	12,14

* Implementation of Symbol table:-

- A symbol table can be Implemented in one of the following Techniques:
① linear list (Sorted or unsorted)
② Hash Table
③ Binary Search Tree

* Operation of a symbol Table:-
① insert()
↳ insert(a, int) (a: symbol, int: type)
② look up()
↳ lookup(a)
↳ retrival
③ delete() - delete(Symbol)
④ Scope Menegement

Ex int value; → (Global value)

f^n ① void one()
{
int a;
int b;
f^n/loop nesting
{
int c;
int d;
}
int e;
int f;
int g;
}

void two()
{
int x;
int y;
{
int p;
int q;
}
int r;
}

Name	Variable	Type
value	var	int
one	procedure	void
two	procedure	void

a	var	int
b	var	int
e	var	int

x	var	int
y	var	int
z	var	int

c	var	int
d	var	int

f	var	int
g	var	int

p	var	int
q	var	int

Q* Symbol table organization for block structured languages.

```
BBLOCK:
    REAL X,Y, STRING NAME;
    M1 : PBLOCK (INTEGER IND);
         INTEGER X;
         CALL M2 (IND+1);
    END M1;

    M2 : PBLOCK(INTEGER J);
         BBLOCK:
         ARRAY INTEGER F(J); LOGICAL TEST;
         END;
    END M2;
    CALL M1(X/Y);
END;
```

→ A tree showing the offsets of the set & reset operation

before(:) it's declare in set
before END all are in Reset

Ans

Operation	Symbol table Contents (variable name only) Active	InActive
Set BLK 1	Empty	Empty
Set BLK 2	M1, NAME, Y, X	Empty
Reset BLK 2	X, IND, M1, NAME, Y, X	Empty
Set BLK 3	M2 M1, NAME, Y, X	X, IND
Set BLK 4	J, M2, M1, NAME, Y, X	X, IND
Reset BLK 4	TEST1, F, J, M2, M1, NAME, Y, X	X, IND
Reset BLK 3	TEST1, F, J, M2, M1, NAME, Y, X	TEST1, F, X, IND
Reset BLK 1	M2, M1, NAME, Y, X	J, TEST1, F, X, IND
end of Compilation	Empty	M2, M1, NAME, Y, X, J, TEST1, F, X, IND

Ex-2

```
1  BBLOCK;
     REAL 2; INTEGER 4;
2    SUB-1: PBLOCK (INTEGER J);
       :
3    BBLOCK;
       ARRAY STRING S(J+2); LOGICAL FLAG;
       :                    INTEGER Y;
       :
     END;
4  SUB-2: PBLOCK(REAL W);
       REAL J: LOGICAL TEST1, TEST2, TEST3;
       :
     END SUB-2;
   END SUB-1;
```

PTJ -> Pointer

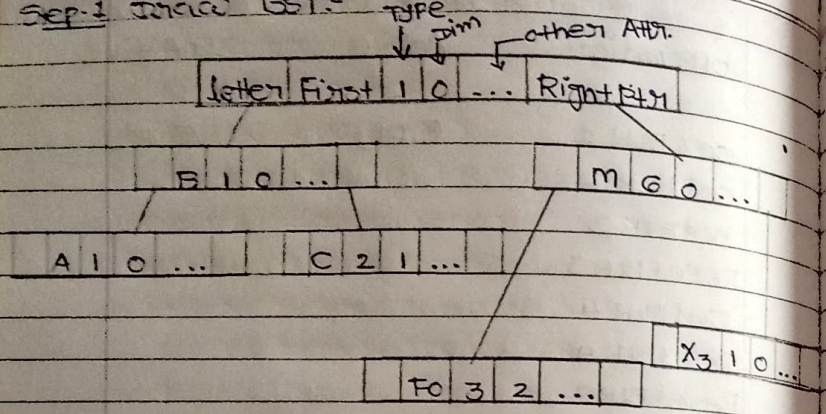
Symbol table Contents (variable name only)

operation	Active	InActive
set BLK 1	Empty	Empty
set BLK 2		
set BLK 3		
Reset BLK 3		
Set BLK 4		
Reset BLK 4		
Reset BLK 2		
Reset BLK 1		
End of Compilation		

BST

No.	Name Field	Type	Dimension	other Attr.	left PTJ. Attr	Right PTJ. Attr
1	FIRST	1	0		2	5
2	B	1	0		3	4
3	ANS	1	0		0	0
4	COMPANY#	2	1		0	0
5	M	6	0		6	7
6	FORM	3	2		0	0
7	X3	1	0		0	0

Step-1 Index BST:-

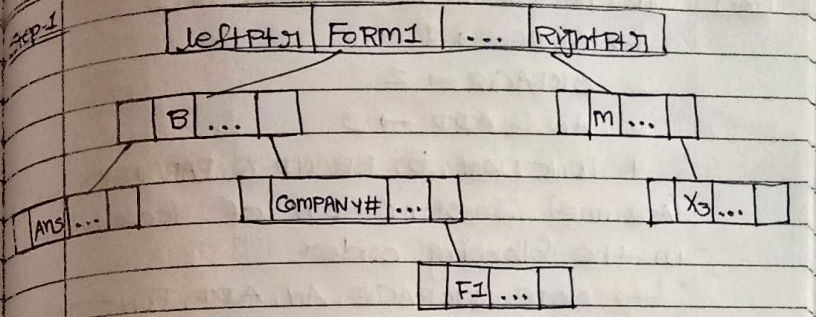


Ex-2 FORM1, B, COMPANY#, ANS, M, X3, FIRST

Step-2

	Name Field	other Attr.	left Ptr	Right Ptr
1	FORM1		2	5
2	B	...	4	3
3	COMPANY#		0	7
4	ANS		0	0
5	M		0	6
6	X3		0	0
7	FIRST		0	0

Step-1

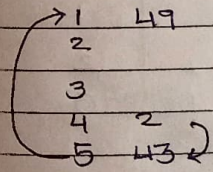


* Symbol table organization for Non-block structured languages:
  - ↳ unordered
  - ordered (binary Search, linear, hash)

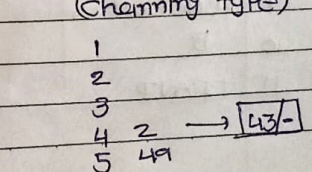
LST :-	n
BST :-	log n
AVL :-	not in syllabus

* Hash Table
(Collision are occur then the 2 solⁿ are there)
① open Addressing
② chaining

① open Addressing



② Seprate chaining (Chaining type)



Ex-1 The name NODE map in to 1

↳ NODE → 1

STORAGE → 2

AN & ADD → 3

FUNCTION, B, BRAND & PARAMETER → 9

(Assume) Instruction of variables in the following order.

↳ NODE, STORAGE, AN, ADD, FUNCTION, B, BRAND, PARAMETER

Collision resolution using open addressing:

M = 11

	NAME	other Attr.	No. of Probs	
1	NODE		1	
2	STORAGE		1	
3	AN		1	Collision are occur then Probs + 1 get
4	ADD		2	
5	PARAMETERS		8	(3+1+1+1+2)
6	Empty			
7	Empty			
8	Empty			
9	FUNCTION		1	
10	B		2	
11	BRAND		3	

Ex-2 $h(x) = (x \bmod m) + 1$

9, 1, 6, 11, 3, 8, 0, 5, 10, 2, 7, 12, 4

M = 13

	Name	other attributes	no of probs
1	0		1
2	1		1
3	2		1
4	3		1
5	4		1
6	5		1
7	6		1
8	7		1
9	8		1
10	9		1
11	10		1
12	11		1
13	12		1

* Seprete Channing :-

linked list representation:-

Position Name other attr.

9 function → B → Brand

Ex-1

Position	VariName	other Attri	link
1	NODE		
2	STORAGE		
3	ADD		1
4	Empty		
5	Empty		
6	Empty		
7	Empty		
8	Empty		
9	B		3
10	Empty		
11	Empty		

(alphabetically wise)

	VariName	other Attri	link
1	AN		
2	FUN		
3	BRAND		
4	PARAMETER		

overflow area

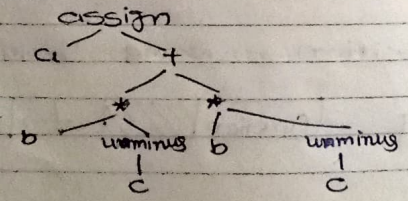
Prime Area

* Intermediate Code Generator

Position of intermediate code Generation

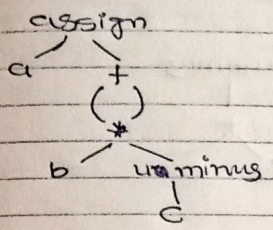
a := b * -c + b * -c

① Syntax tree



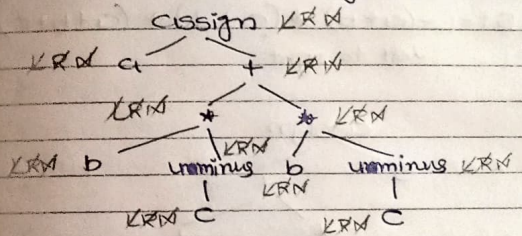
Syntax tree
DAG (Common write one time)
Postfix

② DAG



③ Postfix notation for Syntax tree

L → left
R → Right
N → node

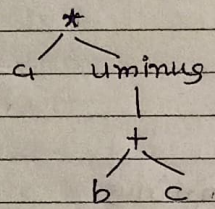


abc uminus * bc uminus * + assign

* Arithmetic operation

S := a * -(b+c) Convert into Syntax tree, Postfix notation

① Syntax tree:

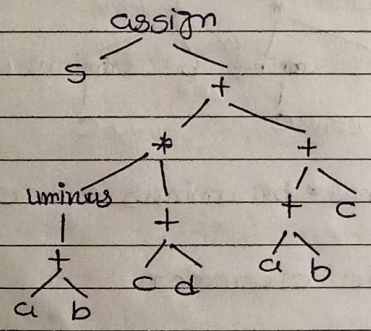


② Postfix notation

abc + uminus *

Q-2 S := -(a+b) * (c+d) + (a+b+c)

left to right



Sabt uminus cd + * ab + c + + assign

3 Address Code

x := y op z → Syntax Unary operation

(x := op y) assignment Instruction

x := y Copy statement

unconditional jump / conditional jump

Go to L

↓

statement if x relop y → go to L

Unary operation

x := y + z

t_1 := y + z

x := t_1

Q-1 3 Address Code :-

Syntax tree	DAG
t_1 := -c	t_1 := -c
t_2 := b * t_1	t_2 := b * t_1
t_3 := -c	t_3 := + t_2
t_4 := b * t_3	a := t_3
t_5 := t_2 + t_4	
a := t_5	

Ex-2

$t_1 := b + c$

$t_2 := -t_1$

$t_3 := a * t_2$

$a := t_3$

Ex-3

$t_1 := a + b$

$t_2 := -t_1$

$t_3 := c + d$

$t_4 := t_2 * t_3$

$t_5 := a + b$

$t_6 := t_5 + c$

$t_7 := t_4 + t_6$

$a := t_7$

Ex if A<B then 1 else 0 (looping Syntax tree ✗ Flow graph ✓)

Three:- (Condition true → goto (4))

① if (A<B) goto (4)

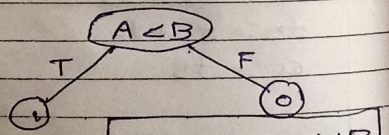
② $T_1 := 0$ ← false Condition

③ goto (5)

④ $T_1 := 1$

⑤ End / (empty) / blank

Flow graph:-



Postfix:- 10A<B

Ex if A<B & C<D then T=1 else T=0

3 Address Code:

① if (A<B) goto (4)

② $T_1 := 0$

③ goto (7)

④ if (C<D) goto (6)

⑤ goto (2)

⑥ T = 1

⑦ empty

Ex C=0

do {

if (a<b) then

x++;

else

x--;

c++;

}

while (c<5)

Three Address Code:-

① C=0

② if (a<b) goto (6)

```
(3)  t1 = x - 1
(4)  x = t1
(5)  goto (8)
(6)  t2 = x + 1
(7)  x = t2
(8)  t3 = c + 1
(9)  c = t3
(10) if (c < 5) goto (2)
(11)
```

Ex While (A < C and B > D)
do
if A = 1 then
C = C + 1
Else
while A <= D
do A = A + B

```
(1)  if (A < C) goto (3)
(2)  goto (14)
(3)  if (B > D) goto (5)
(4)  goto (14)
(5)  if A = 1 goto (11)
(6)  if A <= D goto (8)
(7)  goto (1)
(8)  t2 = A + B
(9)  A = t2
(10) goto (6)
(11) t2 = c + 1
(12) c = t2
(13) goto (1)
(14)
```

Ex Switch (ch)
{
case 1 : c = a + b;
break;
Case 2 : c = a - b;
break;
}

```
if ch = 1 goto L1
if ch = 2 goto L2
L1 : T1 = a + b
     c = T1
     goto Last
L2 :

LAST :
```

* Implementation of 3 Address Code

① Quadruple
② Triple
③ Indirect Triple

Ex-① $a := -(a*b) + -(a*b)$

write a 3 Address Code:

$T_1 := a * b$
$T_2 := \text{uminus } T_1$
$T_3 := a * b$
$T_4 := \text{uminus } T_3$
$T_5 := T_2 + T_4$
$a := T_5$

① Quadruple

	OP	arg 1	arg 2	result
(0)	*	a	b	T_1
(1)	uminus	T_1		T_2
(2)	*	a	b	T_3
(3)	uminus	T_3		T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5		a

$S := -(a+b)*(c+d) + (a+b+c)$

Ex-② $a := b * -c + b * -c$

$T_1 := \text{uminus } c$
$T_2 := b * T_1$
$T_3 := \text{uminus } c$
$T_4 := b * T_2$
$T_5 := T_2 + T_4$
~~$T_6 :=$~~
$a := T_5$

Quadruple

	OP	arg 1	arg 2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_2	T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5		a

Ex-③ $S := -(a+b)*(c+d) + (a+b+c)$

$T_1 := a + b$
$T_2 := \text{uminus } T_1$
$T_3 := c + d$
$T_4 := T_2 * T_3$

$T_5 := a + b$
$T_6 := T_5 + c$
$T_7 := T_4 + T_6$
$S := T_7$

	OP	arg1	arg2	result
(0)	+	a	b	T_1
(1)	uminus	T_1		T_2
(2)	+	c	d	T_3
(3)	*	T_2	T_3	T_4
(4)	+	a	b	T_5
(5)	+	T_5	c	T_6
(6)	+	T_4	T_6	T_7
(7)	:=	T_7		S

② Triple — no result — a in 3rd arg.

Ex-1

	OP	arg1	arg2
(0)	*	a	b
(1)	Uminus	(0)	
(2)	*	a	b
(3)	Uminus	(2)	
(4)	+	(1)	(3)
(5)	:=/ assign	a	(4)

a := (4)

Ex-2

	OP	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=/ assign	a	(4)

Ex-3

	OP	arg1	arg2
(0)	+	a	b
(1)	uminus	(0)	
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	a	b
(5)	+	(4)	c
(6)	+	(3)	(5)
(7)	:=	S	(6)

③ Indirect Triple

Ex-1 ①

	Statements
(0)	→ (14)
(1)	→ (15)
(2)	→ (16)
(3)	→ (17)
(4)	→ (18)
(5)	→ (19)

②	OP	arg1	arg2
(14)	*	a	b
(15)	Uminus	(14)	
(16)	*	a	b
(17)	Uminus	(16)	
(18)	+	(15)	(17)
19)	:=/assign	a	(18)

Ex x[i] := y

Ex x := y[i]

✡ * Code optimization Techniques

① Compile time Evaluation
- ① Constant folding
- ② Constant Propagation

② Common Sub Expression Elimination

③ Variable Propagation

④ Code movement (loop invariant) (code motion)

⑤ Strength Reduction

⑥ Dead Code Elimination

① Constant folding

Ex area = (22/7) * r * r;

area = (3.14) * r * r;

② Constant Propagation

Ex Pi = 3.14; r = 5;

area = Pi * r * r;

area = 3.14 * 5 * 5;

② Common Sub Expression Elimination (CSE)

Ex
a = b+c
b = a-d
c = b+c
d = a-d

a, c not a CSE because value of b is changed but b,d is a CSE

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

CSE Technique

Ex

$t_1 := 4 * i$	$t_1 = 4 * i$
$t_2 := a[t_1]$	$t_2 = a[t_1]$
$t_3 := 4 * j$	$t_3 = 4 * j$
$t_4 := 4 * i$	~~$t_4 = t_1$~~
$t_5 := n$	$t_5 = n$
$t_6 := b[t_4] + t_5$	$t_6 = b[t_1] + t_5$

(Constant)

If n value is given

$t_1 = 4 * i$

$t_2 = a[t_1]$

$t_3 = 4 * j$

$t_4 = b[t_3] + n$

Constant Propagation

If not given n value → Variable Propagation

③ Variable Propagation

use of one variable instead of another

Ex

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

area = Pi * r * r;
```

④ Code movement

Two goals ① reduce the size of the code

② reduce the frequency of execution of code.

Ex

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

only need i so not need x

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

Ex

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

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

⑤ Strength Reduction Change higher expensive operator to lower

Ex

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

```
temp = 0                    Count = 0
for(i=1; i<=50; i++)
{
    temp = temp + 7;        Count + 7,
    Count = temp;
}
```

(c) Dead Code Elimination

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

i=0; but i=0 so not go into for loop

Ex
```
int add (int a, int b)
{
  int x,y,z;
  return(a+b);
  PF ("Hello..");
  PF ("Hi..");
}
```
Dead Code Elimination

Ex
```
for (i=0; i<n; i++)
{
  y = y+8*7+2-1;
}
```
Constant folding

* Loop optimization

Techniques of loop optimization

1. Code motion
2. Strength reduction
3. loop unrolling
4. loop Fusion

3. loop unrolling
In this method no of jumps & test can be reduced by writing the code 2 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++;
}
```

4. loop Fusion
In this method several nested loops are merged to one loop.

Ex

```
for i=1 to n do
    for j=1 to m do
        A[i,j] = 10

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

Runtime Environment

1. Source language issue
   1. activation tree
   2. Control stack
2. Storage organization
   1. Sub division of run time storage
   2. Activation Record
3. Storage allocation strategies
   1. stack allocation
   2. static allocation
   3. heap allocation
4. Access to non local names
   Find Activation Records
   1. Access link
   2. displays

Q

```
main()
{
    int n;
    readarray();
    quicksort(1, n);
}
```

```
quicksort (int m, int n)        i = 4
{
    int i = Partition(m,n);
    quicksort (m, i-1);
    quicksort (i+1, n);
}
```

* Activation Tree (No distinct)

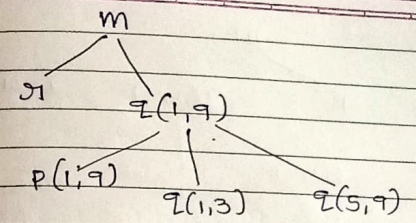
Step-1 Activation of Procedures.

Execution begins...
Enter readarray
leave readarray
Enter quicksort(1,9) (n=9)
Enter Partition(1,9)
leave Partition(1,9)
Enter quicksort(1,3) (i=4)
leave quicksort(1,3)
Enter quicksort(5,9)
leave quicksort(5,9)
leave quicksort(1,9)
Execution Ends / terminated

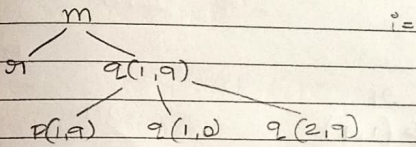
Step-2 Activation Tree

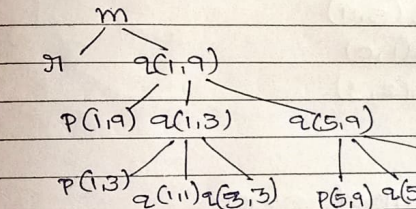
main → m (quicksort) q( ) → q
r() → r P(_) → P (Partition)

detection range.
1 to 4
1 to 9
1 to 12
6 marks

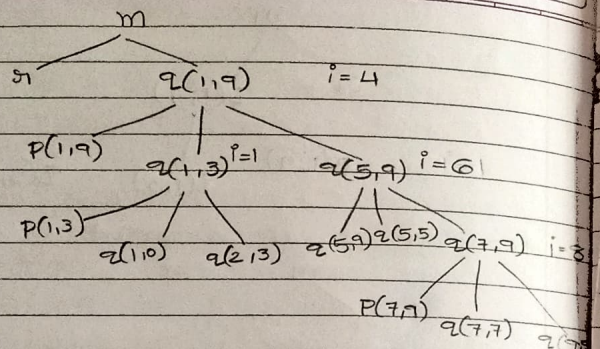


Partition at 1st Element



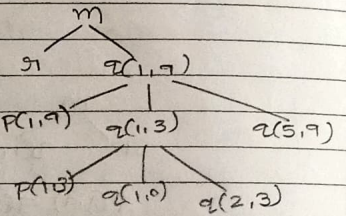


i value given 4th, 1st, 6th, 8th even If it is out of range, we must Perform the operation



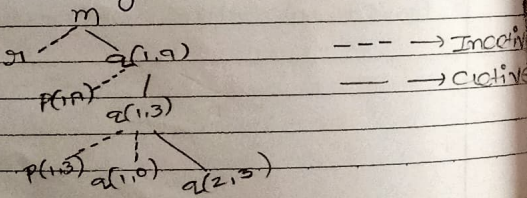
Ex

r
p(1,9)
q(1,3)
q(5,9)
q(1,0)
q(2,3)



* Control stack

Step-3 - When control enters the Activation represented by q(2,3).



④* Access to non local ~~needs~~ names

```
Program sort(input,output);
    Var a: array[0..10] of integer;
        x: integer;
    Procedure readarray;
        Var i: integer;
        begin... a End {readarray};
    Procedure Exchange(i,j: integer);
        begin
        x = a[i], a[i] = a[j], a[j] = x
        End {Exchange};
    Procedure quicksort(m,n: integer);
        Var k,v: integer;
        function Partition(y,z: integer): integer;
        Var i,j: integer;
        begin ... a. .
                ... v. ..
                ... Exchange(i,i);
        End {Partition};
        begin... End {quick Sort};
        begin... End {Sort};
```

- lexical scope with nested Procedures

Types

① block
② without nested Procedures
③ with nested Procedures

Step-1 lexical scope with nested Procedu

Sort
readarray
Exchange
QuickSort
Partition

Step-2 Nesting Depth

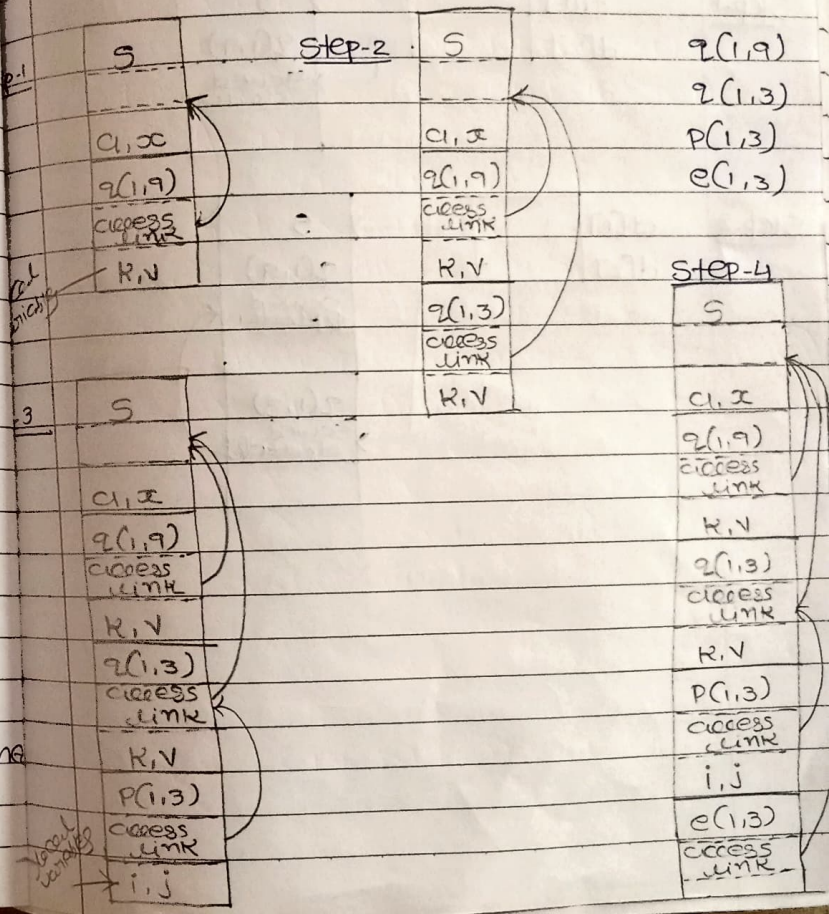
Start with 1

Sort ——— ①
readarray ——— ②
Exchange ——— ②
QuickSort ——— ②
Partition ——— ③

Step-3 ① Access link

- A direct Implementation of lexical scope for nested Procedures is obtaine by adding a Pointer called as access link to each activation record.

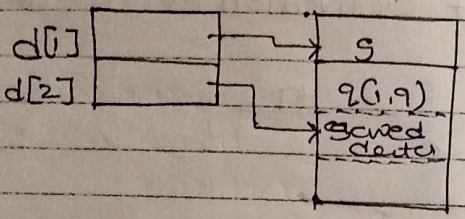
If Procedure P is nested immediately with queue in the source text then the access link in an activation record for P points to the access link in the record for the most recent activation of queue.



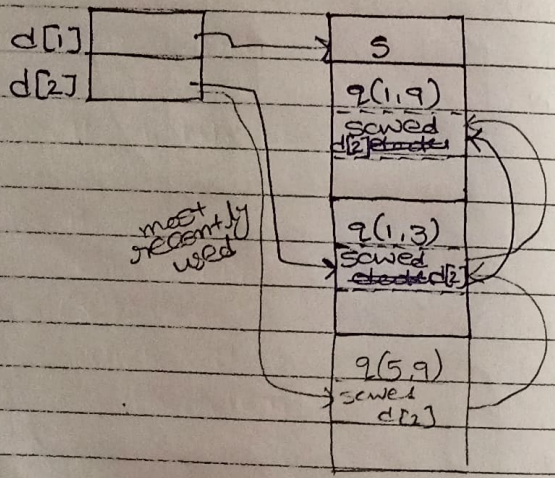
② Displays

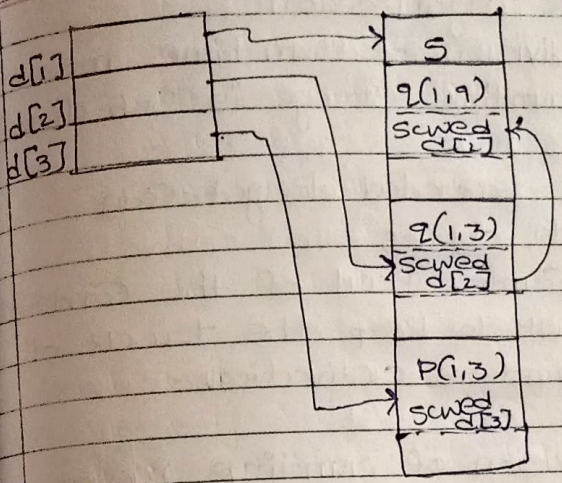
Faster access to non locals than access links can be obtain using an array d of pointers to activation records called ~~for the~~ display.

Step-1

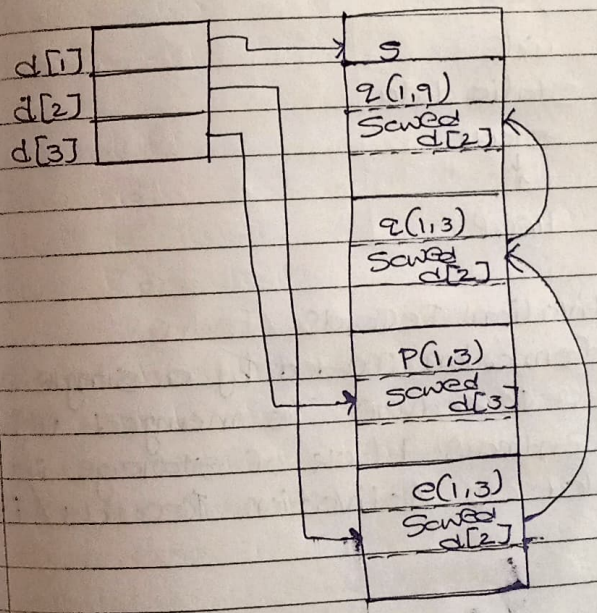


Step-2





4



* Storage organization
- Sub division of runtime memory
The runtime storage is sub divided into 3 parts:
① The generated target code
② Data objects
③ A Counter Part of the Control Stack to keep the track of Procedure activation.

2 marks

Sub division of runtime storage memory

draw figure

Code
Static data
stack
↓
↑
Heap

* Activation Records / Frames
- Information needed by a single exec of a procedure is manged using a continous block of storage is called as Activation Records / Fram

return value
actual Parameters
optional control link
optional access link
saved machine status
local data
temporaries

• temporaries
temporary value such as those arising in the evaluation of the expression are stored in the temporary fields.

• local data
- The field for local data holds data that is local to execution of the procedure

• saved machine status
- The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the value of the program counter & machine register that

have to be restore : When control returns from the procedure.

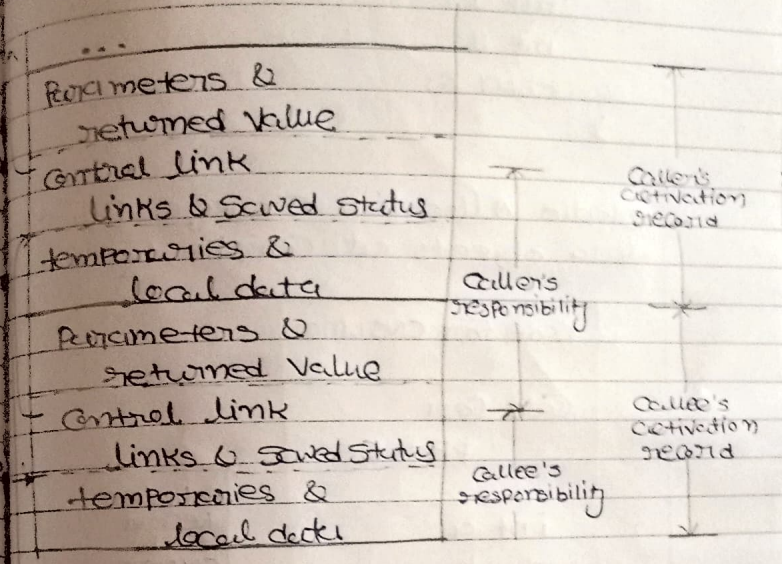
* optional access link
- It is used to refer to non local data held in other activation record.

* optional control link
- It points to the activation record of the calling procedure.

* Actual Parameters
- The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

* return value
- The field for the return value is used by the called procedure to return a value to the calling procedure.

division of task between caller & callee.

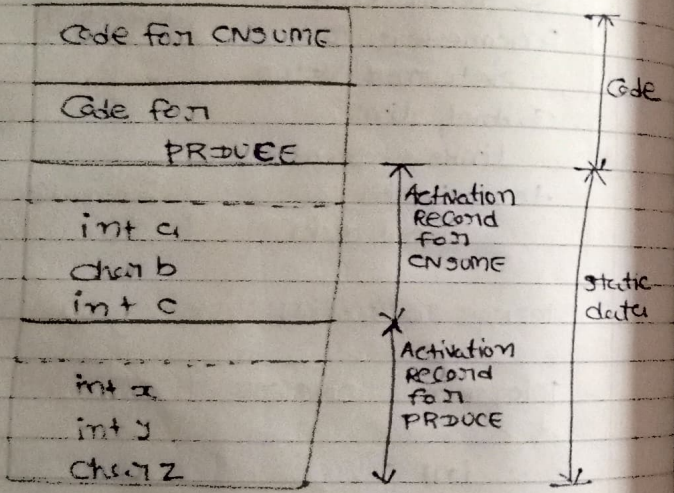


Storage allocation strategies

```
Program CNSUME
{
    int a;
    char b;
    int c;
}
```

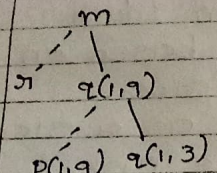
```
Program PRODUCE
{
    int x;
    int y;
    char z;
}
```

- Static Allocation Storage for all data objects at Compile time



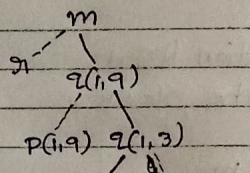
* Static Stack allocation
Stack allocation is based on the idea of the control stack. Storage is organized as a stack & activation records are PUSH & POP as activation with in & activation end respectively.

Position in Activation Tree	Activation Records on the Stack	Remarks
m	m n: integer	Frame for m
m r	m n: integer r	r is activated
m r q(1,9)	m n: integer q(1,9) i: integer	r has been POPPED & q(1,9) PUSHED



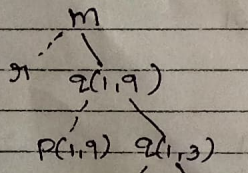
m
n: integer
q(1,9)
i: integer
q(1,3)
i: integer

Control has just returned to q(1,3)



m
n: integer
q(1,9)
i: integer
q(1,3)
i: integer
P(1,3)

Control has just returned to P(1,3)



m
n: integer
q(1,9)
i: integer
q(1,3)
i: integer
q(1,0)
i: integer

Control has just returned to q(1,0)

upward → heap
downward → stack

Dracobook

Whenever storage is deallocated the problem of dangling reference arises. A Dangling reference occurs when there is a reference to storage that has been deallocated

Heap allocation

Position in Activation Tree	Activation Records on the Heap	Remarks