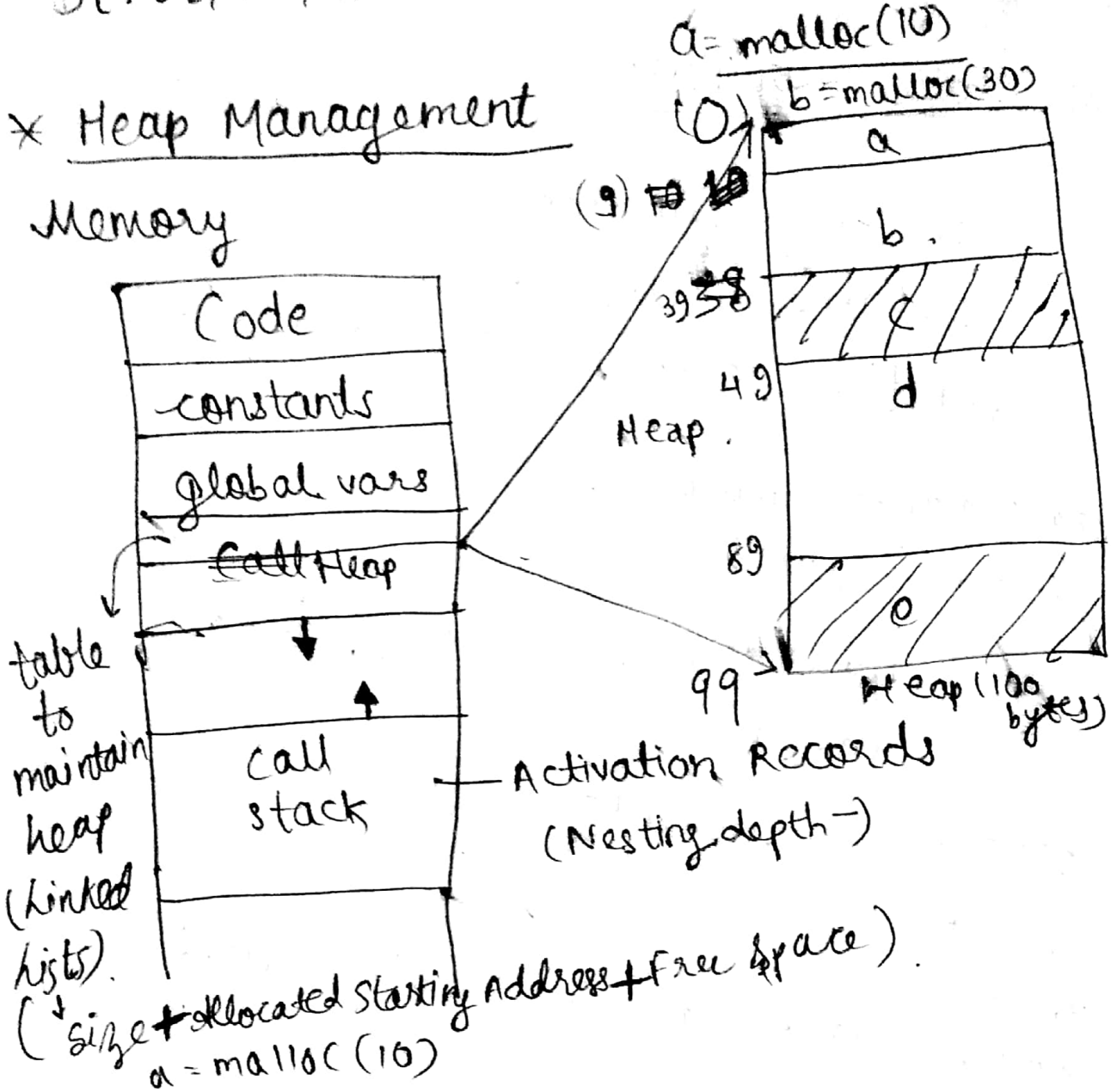
Dt: 09/02/18

* Heap Management



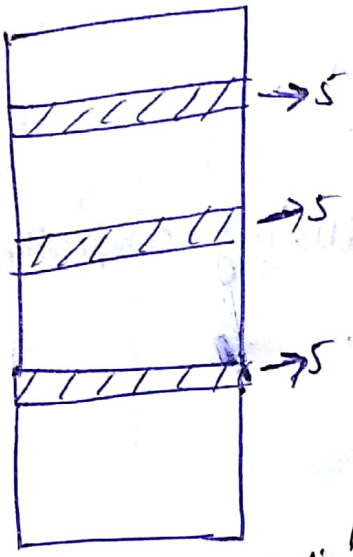

```
a = malloc(10)
b = malloc(30)
c = malloc(10)
d = malloc(40)
e = malloc(10).
free(c).
free(e)
```

First fit algorithm : First free area(hole)

Best fit algorithm : Size of memory requested (or closest to size of memory).

Worst fit algorithm: Return free area of maximum size.

→ Disadvantage: External fragementation.



(Much space but not contiguous)

Internal fragementation: ↗ depends on threshold.

16 bytes allocated to any requested.

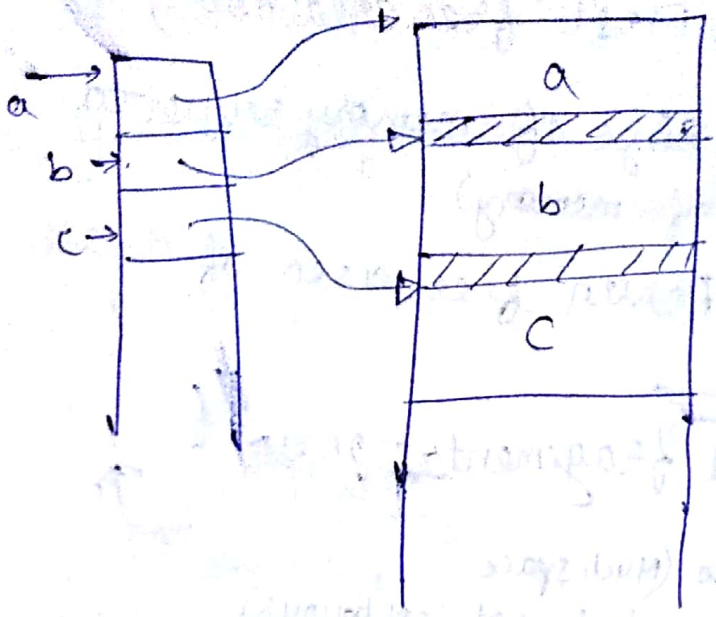
If 10 bytes asked, we waste 6 bytes because keeping overhead of 6 bytes is more trouble.

(Granularity - min. size allocated by heap)

Soln for external fragementation:

Compaction: Move the allocated spaces ahead & create larger holes.

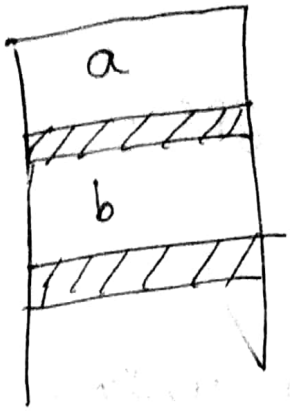
Updation of program variables is the Que.



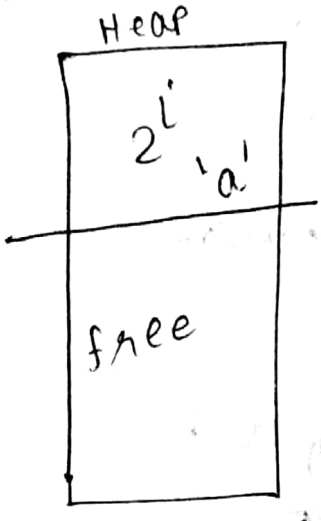
Improvement.

Change program variables during compaction (No double dereferencing).

Dt: 12/02/18



* Buddy System



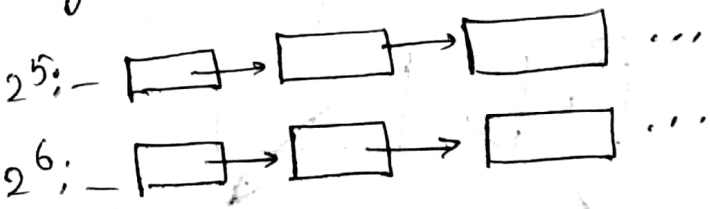
a -> x bytes

$2^i \geq x$

$2^{i-1} < x$

Find such an i.

Free lists of different sizes
[say threshold 2^5]



size: 2^n bytes

Cost: Too big internal fragementation.

* Fibonacci Heap

The sizes we look for is Fibonacci Numbes (closet to the request).

1, 1, 2, 3, 5, 8, 13, ...

* Type Descriptor

- Keeps track of pointers in a type.

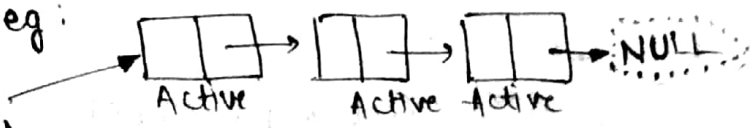
* Mark & Sweep Garbage Collection :

>> Trace the heap; mark everything as being inactive.

>> starting from pointers outside the heap mark the pointed heap element as active.

>> Collect out the inactive elements.

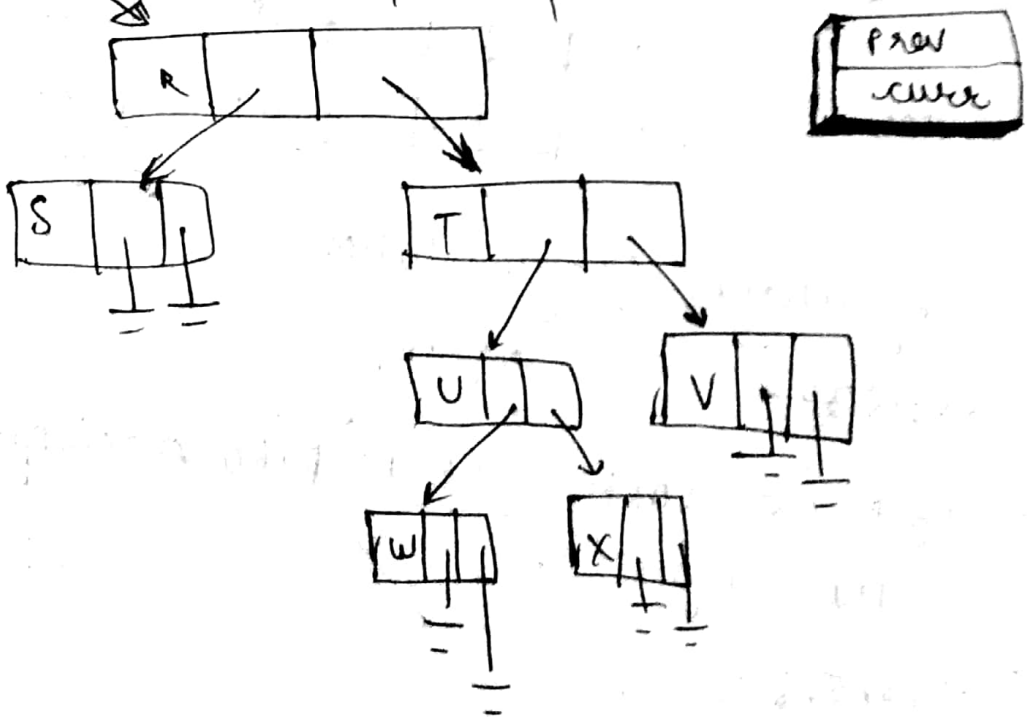
eg:

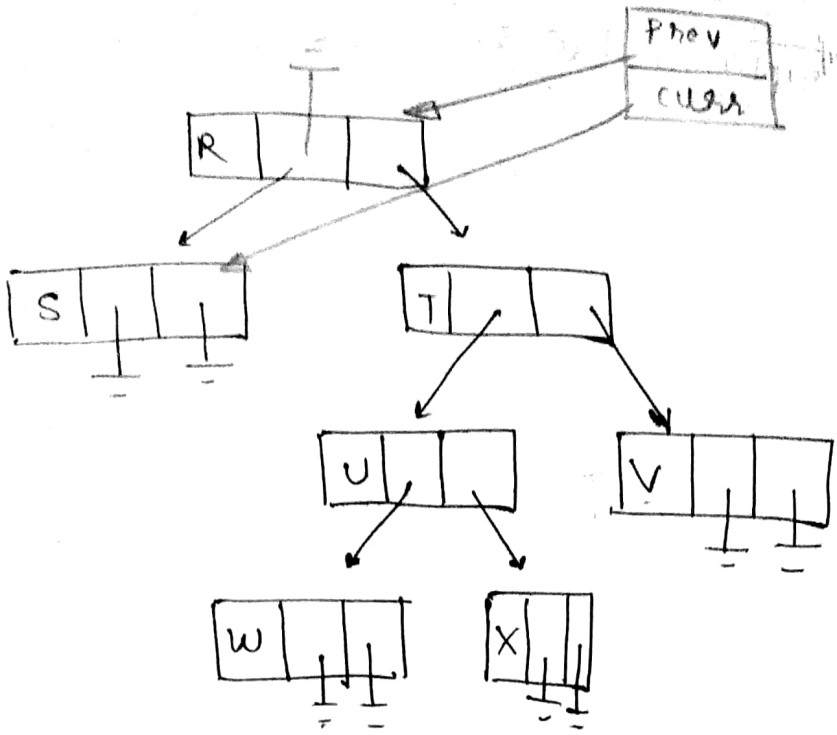


(Uses Recursion)

h
(outside the heap)

To Avoid Recursion (use of stack)



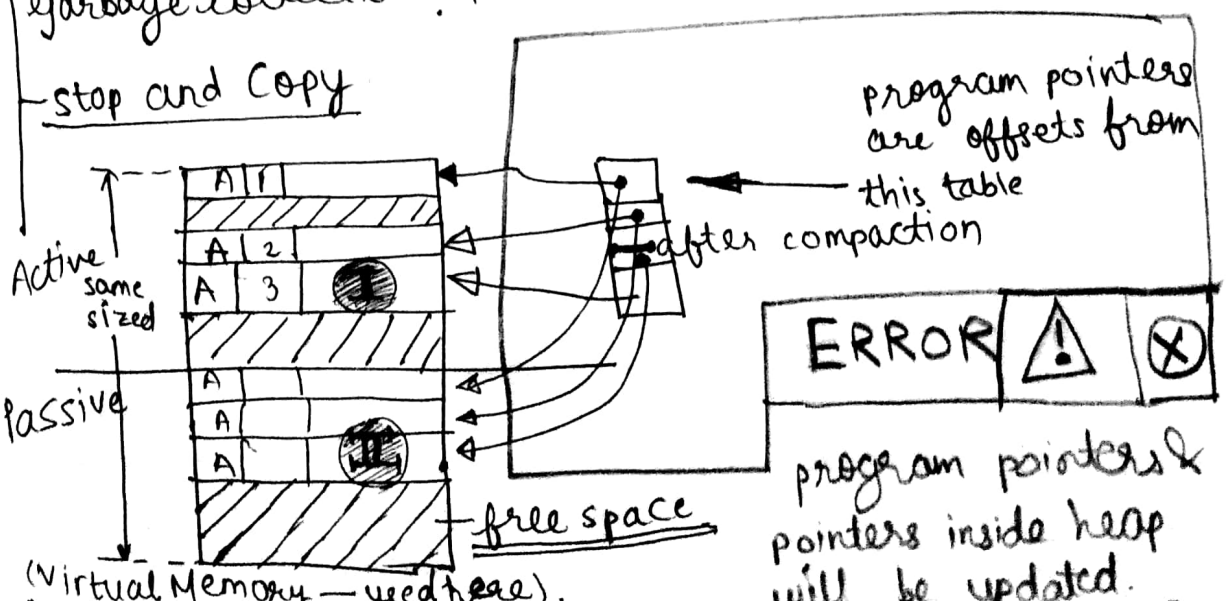


{gc - Garbage collector for Java explicitly}

* Any n-ary tree can be converted to binary tree. *

* How can we do compaction after garbage collection? *

- stop and copy



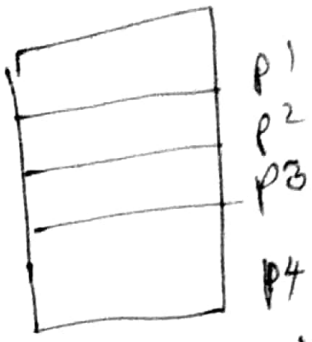
*Assume malloc uses 'I' only. Can I do it better?
When I is full, collect garbage and copy the active elements to "II".

Dt: 19 Feb 2018 →

* Type Descriptor Table :

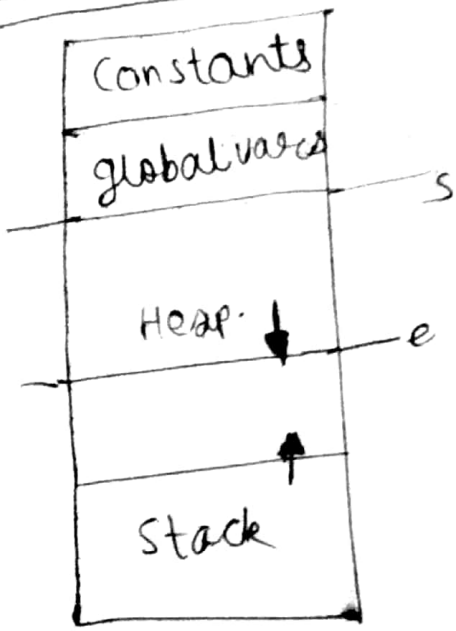
*Generational collection:

Divide the heap into multiple small parts



Perform promotion.

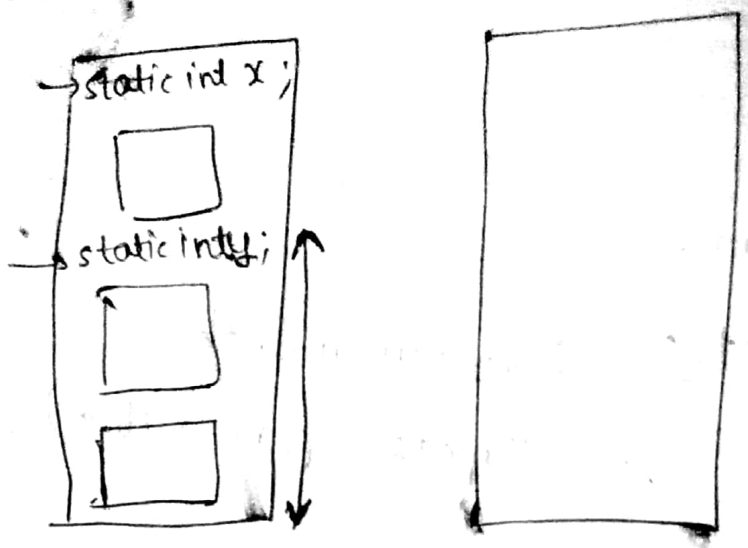
*Conservative Collection: (do away with Typedescriptor table).



Some of inactive elements are not freed/up (collected).
Compaction not possible.

Dt: 23 Feb 2018

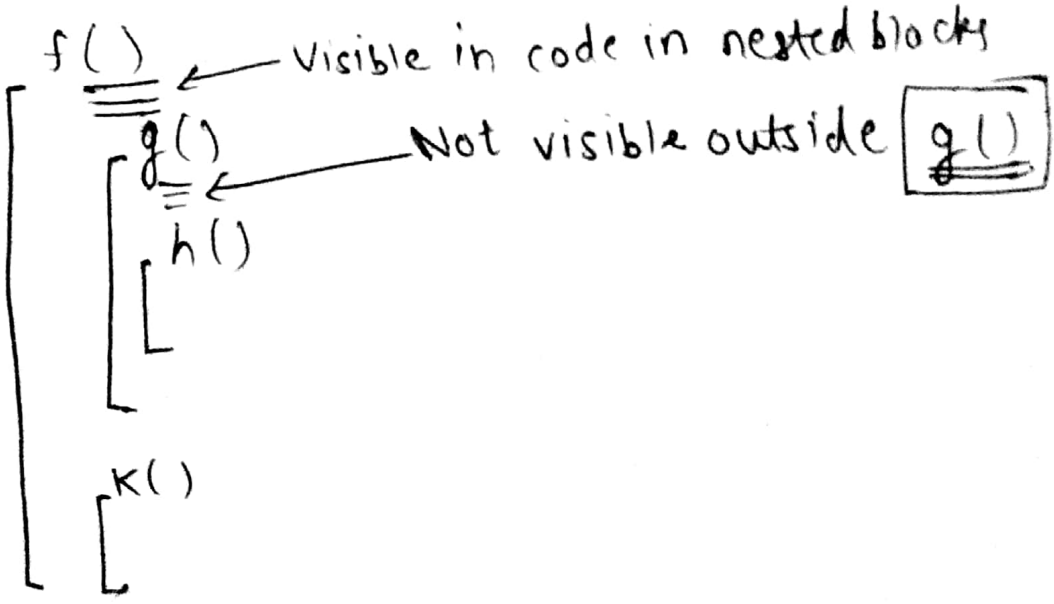
- Names & Scopes

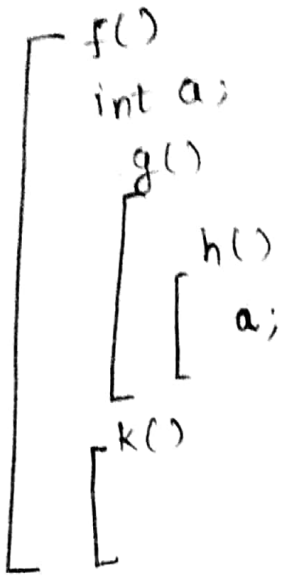


Static scoping languages: Every reference to a variable can be mapped to a particular definition;

Block structured languages

inside function begins becomes local to outside function





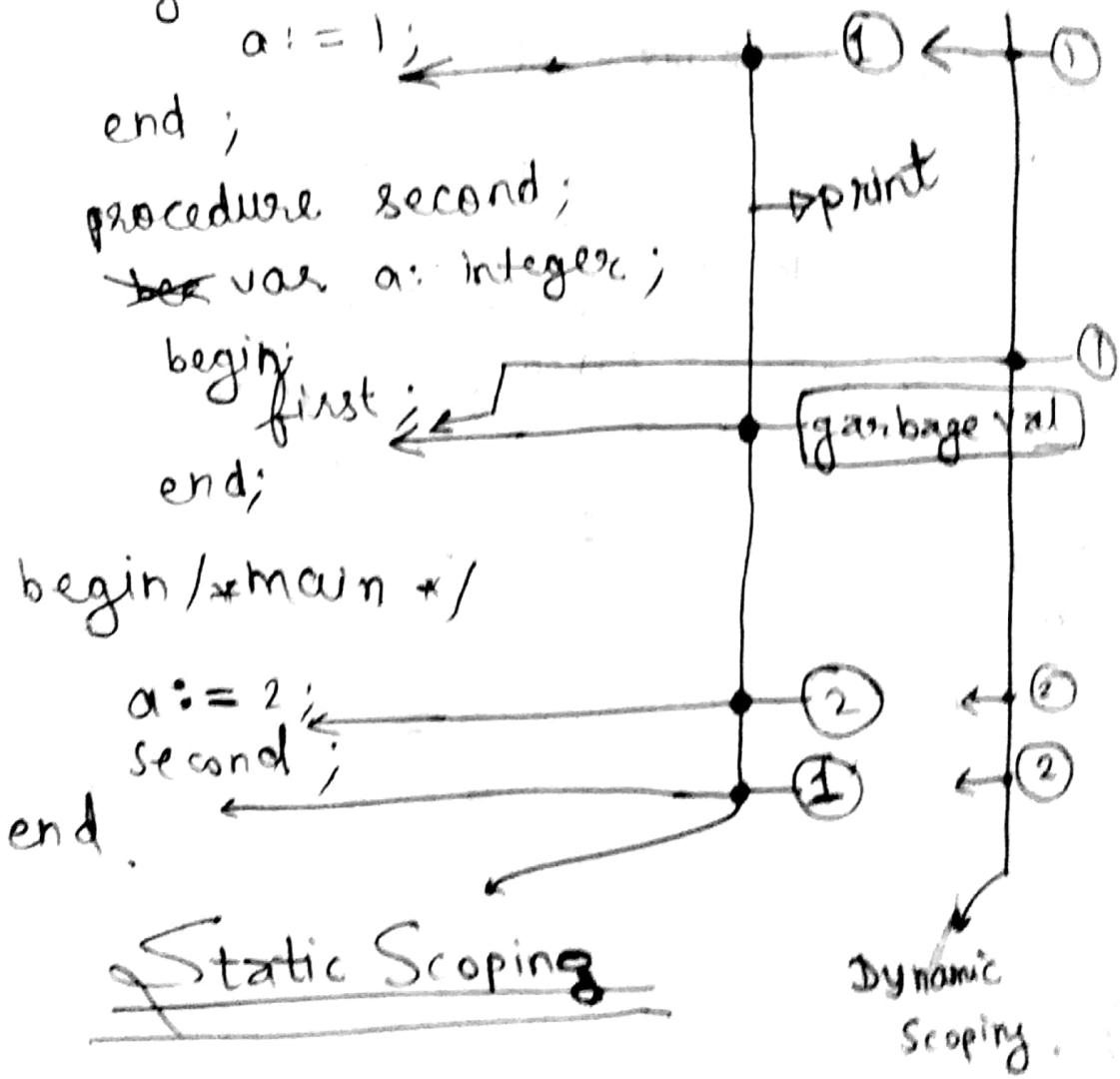
Dynamic Scoping Languages:

```
f(){
   a;//access to a
}
g(){
  int a;
   f();
}
h(){
 float a;
  f();
}
```

searches for definition in f(); if (not found) then looks where it was called from.

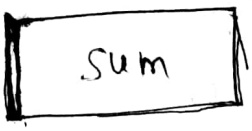
o/p depends on what kind of scoping a lang. supports.

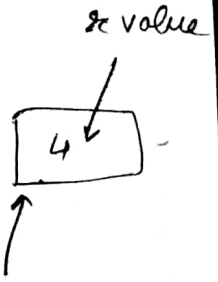
```
program abc;
  var a: integer;
    procedure first;
    begin
      a := 1;
    end;
    procedure second;
      var a: integer;
      begin
        first;
      end;
begin /*main */
  a := 2;
  second;
end.
```



```
double sum, sum_of_squares;
void accumulate (double &x ) { // CPP
    sum += x;                                  sum = (8).
    sum_of_squares += x*x;                     (64).
}
sum = 4; sum_of_squares = 0;
accumulate (sum);
```

'x' & 'sum' are aliases.



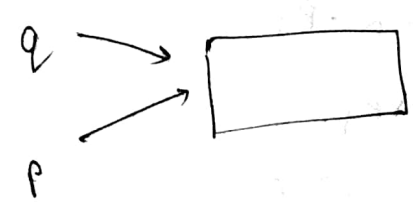


Dt: 26Feb2018

*Aliases: Two or more names referring to the same object.

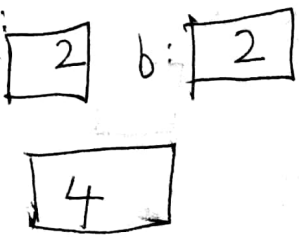
```
int a, b;
int *p, *q; *p = 10;
a = *p;
*q = 3;
b = *p;
```

if p & q refer to same object



* Var-Value Model of Variables

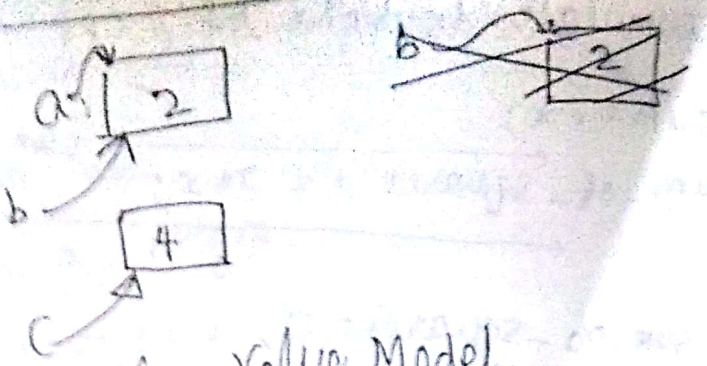
```
a = 2
b = 2
c = a+b
```



Value Model

Execution efficiency ↑

In Var-Value Model,



Var-Value Model

Sequence Controls:

-Expressions: d = a + b - c

1) Three-address code:

$R_1 \leftarrow a$

$R_2 \leftarrow b$

~~$R_3 \leftarrow c$~~

$R_1 \leftarrow R_1 + R_2$

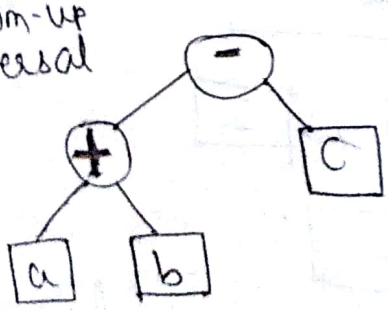
$R_2 \leftarrow c$.

$R_1 \leftarrow R_1 - R_2$.

$d \leftarrow R_1$

2) Expression Tree

bottom-up traversal



* a - f(b) - c * d.

1) Postfix form / Prefix

↓ Operator before operands

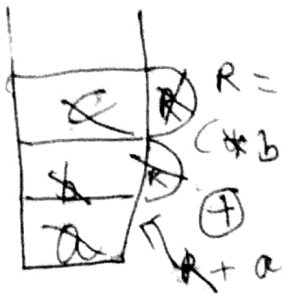
Operators after operands

a b c * +

a b x c +

a + Res

Final Result



* Order of Evaluation

a - f(b) - c * d.

↓

function may have side effects.

why order of evaluation left to compiler designer?

---

a = B[i]

C = a*2 + d*3

I II

Pipeline stalling

I

II → wait till I finishes

Dt: 01/03/18

* Boolean Expression

```
if ((a>b) and (b>c))
then
        x = x+1

else
        x = 0.
end.
```

short-circuiting of Boolean Expression

```
p = head;
while((p!=NULL) && (p->val != v))
{
        p = p->next;

}
```

```
if (((a>b) and (c>d)) or (e!=f))
then  S1;
else  S2;
```

Syntax directed

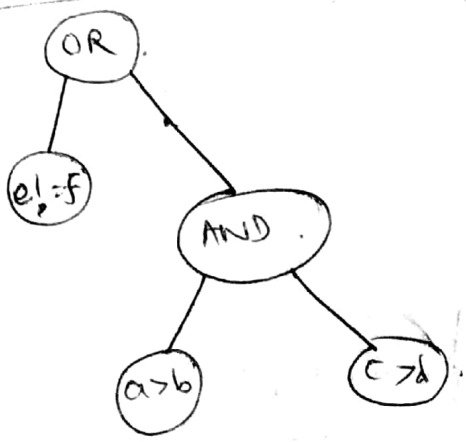
$R_1 \leftarrow a$

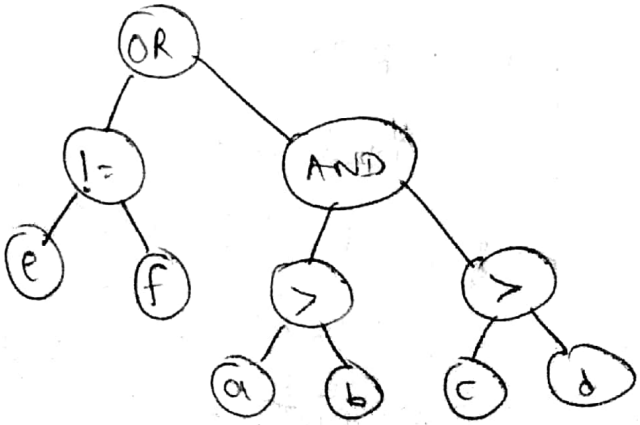
```
CMP R1, R2
JLE L1      ; A <= B Jump
L1: CMP R3, R4
    JLE L2
L2: CMP R5, R6.
    JE  L3
    S1
    JMP L4
```

pack-patching
filling label addresses.

S_2.
JMP L4.

w/o





- CMP R1, R2
  JLE L1
  CMP R3, R4
  JGT L2
  L1: CMP R5, R6
  JE L3
  L2: S1
  JMP L4.
  L3: S2.
  L4:

With short Circuiting

CMP R1, R2
JGT L1
JLE L2
L3: CMP R3, R4
JGT L4
JLE L5

R7 ← R7 & R8
CMP R5, R6
JNE L6
JE L7

Without short circuiting.

L1: R7 ← 1
JMP L3
L2: R7 ← 0
JMP L3
L4: R8 ← 1
JMP L9
L5: R8 ← 0
JMP L9.

L6: R8 ← 1
JMP ~~L8~~
L7: R8 ← 0
~~JMP~~
L8: R7 ← R7 OR R8
JF L10 (JMP on False)
S1
JMP L11
L10: L11: S2

Q. Consider
multi-lege
Dt: 05/03
case x
1: St
2,7: S
3..5:
10:
ELSE:
END.
R1 ←
CMP
JZ
CMP
JZ
CMP
JZ
CMP
~~JGE~~
JLT
CMP
JLE
L6: CMP
~~JM~~
JMP

Consider a switch stmt & convert it to multi-legged if statements. (Homework)

Dt: 05/03/18 (05th March 2018)

```
case x of
    1 : stmt A
    2,7: stmt B
    3,5: stmt C
    10 : stmt D
    ELSE: stmt E
END.
```

Assembly

```
R1 ← x.
CMP R1,#1
JZ ~~stmt~~ L1
CMP R1,#2
JZ L2
CMP R1,#7
JZ L2
CMP ~~R1~~,#3
~~JGE~~
JLT L6
CMP R1,#5
JLE L3
L6: CMP R1,#10
~~JM~~ JZ L4
JMP L5
```

```
L1: stmt 1
    JMP L#      # = 7
L2: stmt 2
    JMP L#
L3: stmt 3
    JMP L#
L4: stmt 4
    JMP L#
L5: stmt 5
    JMP L#
L7:
```

* Jump Table :

```
T: &L1
   &L2
   &L3
   &L3
   &L5
   &L2
   &L5
   &L5
   &L4

L6: r1 ← x
    CMP r1, #1
    JLT L5
    CMP r1, #10
    JGT L5
    DEC r1.
    r2 ← T[r1].   ; Jump Table
    JMP *r2;

L1: stmt A
    JMP L7

L2: stmt B
    JMP L7

L3: stmt C
    JMP L7

L4: stmt D
    JMP L7

L5: stmt E

L7:
```

(2 comparisions only)

*loops

① for

for (i ← 1 to 10 step 2 do
    S1;

R1 ← #1 R2 ← #2
LOOP: CMP R1, #10.
    JGT END.
    S1
    ~~INR R1~~.
    ~~LOOP~~.
    ADD R1, ~~#2~~ R2, R1
    JMP LOOP.
END:

~~R3 ← 1~~
R2 ← 1 step
R1 ← 1. ~~step~~ lower limit.
R4 ← 10 upper limit

Loops

```
for i ← 1 to 10   step 1 do
    S1;
```

```
R3 ← i
R1 ← 1    lower limit
R2 ← 1    step
R4 ← 10   upper limit
L1: CMP R3, R1
    JLT L2
    CMP R3, R4
    JGT L2
    S1
    ADD R3, R2
    JMP L1
L2: i ← R3
```

Optimization

```
R3 ← i
R1 ← lower limit
R2 ← step
R4 ← upper limit
CMP R3, R1
JLT L3
JMP L2
L1: S1
    ADD R3, R2
L2: CMP R3, R4
    JLE L1
L3: i ← R3.
```