Dt: 09/02/18

* Heap Management

Memory

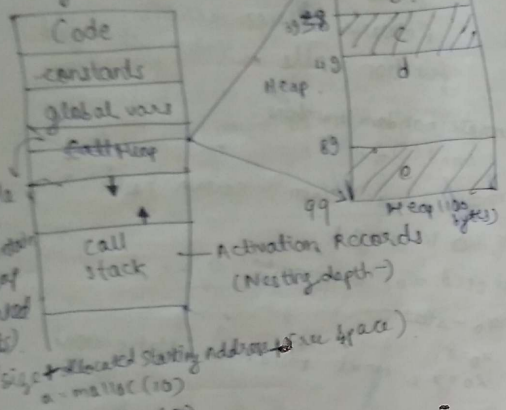


table to maintain heap (linked lists)

(Size, allocated, Starting address, free space)

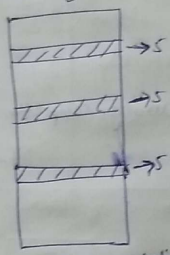
a = malloc(10)
b = malloc(30)
c = malloc(10)
d = malloc(40)
e = malloc(10).
free(c).
free(e)

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 fragmentation.



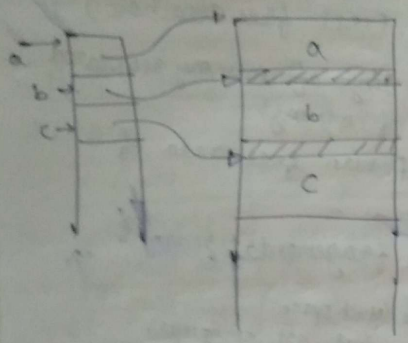
** (Much space but not contiguous)

Internal fragmentation: 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 fragmentation:

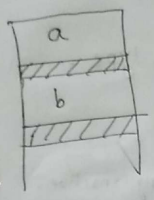
Compaction. Move the allocated spaces ahead & create larger holes.
Updation of program variables is the issue.



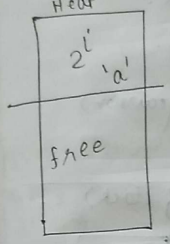
Improvement.

Change program variables during compaction (No double dereferencing).

Dt: 12/02/18



* Buddy System



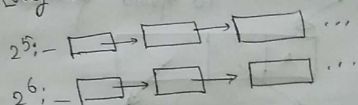
a - x bytes

$2^i \geq x$

$2^{i-1} < x$

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 Numbers (closest to the request).

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

* Type Descriptor

- Keeps track of pointers in a type.

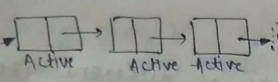
* Mark & Sweep Garbage Collection :

>> 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:

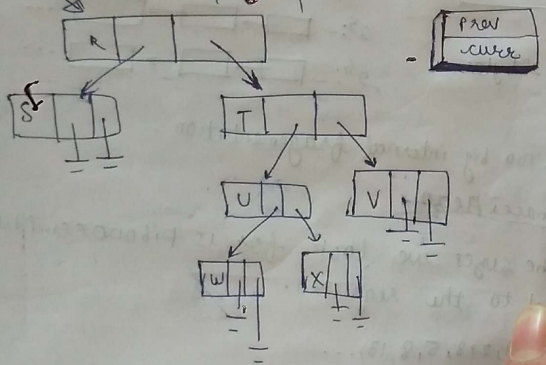


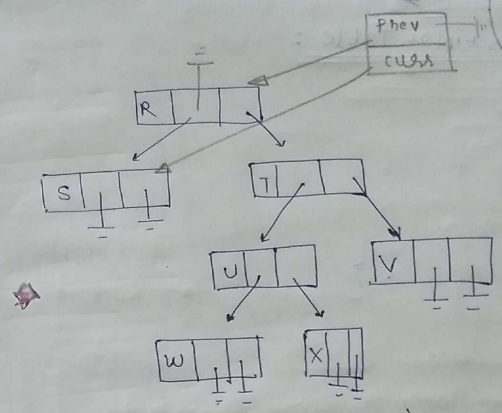
(Uses Recursion)

h
(outside the heap)

To Avoid Recursion (use of stack)

root



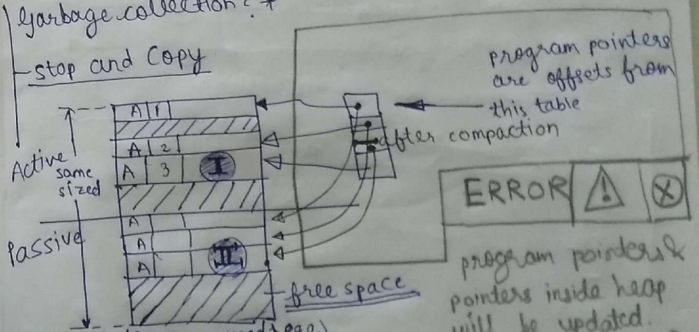


{gc - Garbage collector for Java explicitly.}

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

* How can we do compaction after Garbage collection? *

- stop and Copy



(Virtual Memory — used here).

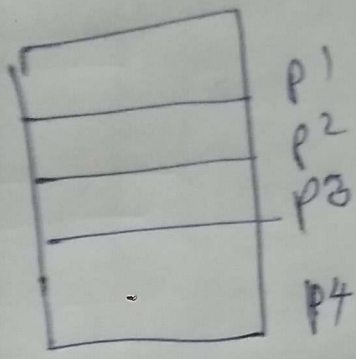
~~Assume~~ malloc uses I only. When I is full, collect garbage and copy the active elements to "II".

program pointers & pointers inside heap will be updated.

Can I do it better?

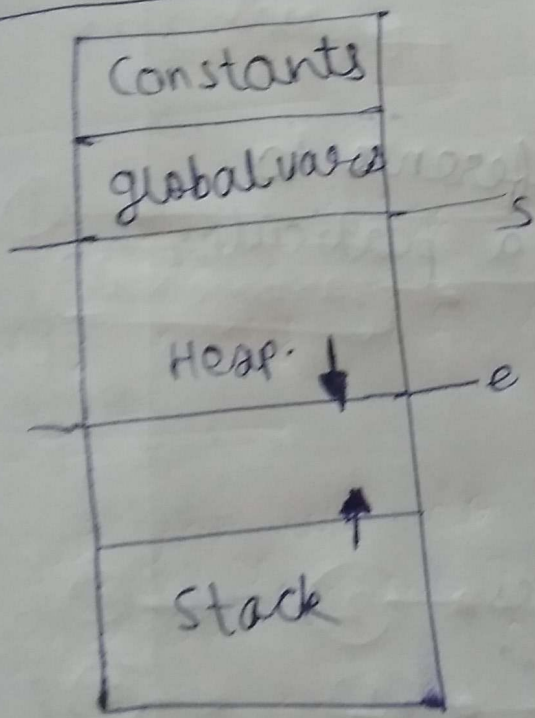
*Generational collection:

Divide the heap into multiple small parts.



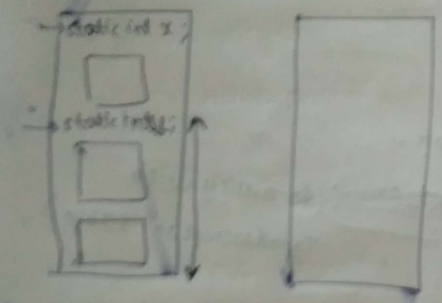
Perform promotion.

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



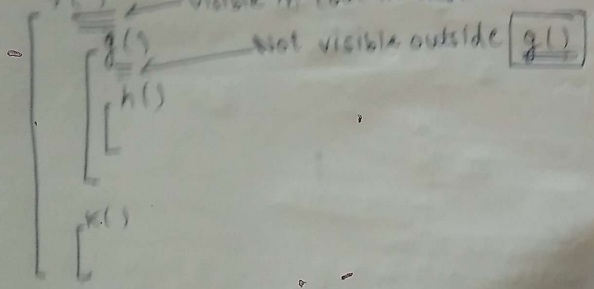
Some of inactive elements are not freed/up (collected).
Compaction not possible.

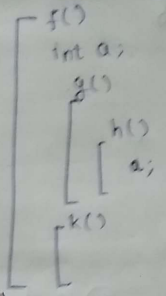
• Names & Scopes



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





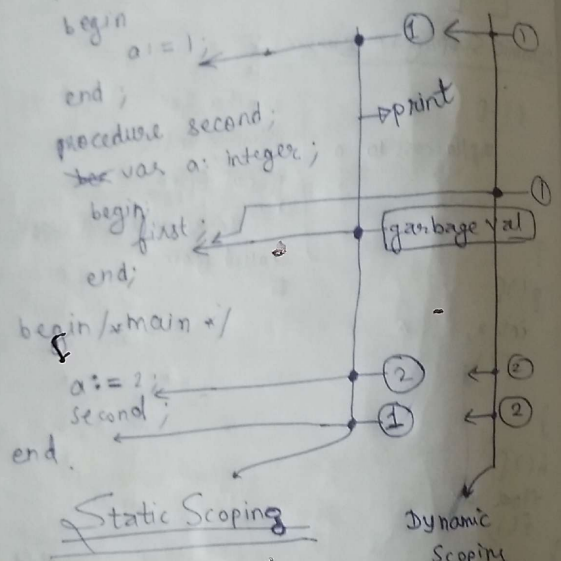
Dynamic Scoping Languages:

```
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.
```



Static Scoping

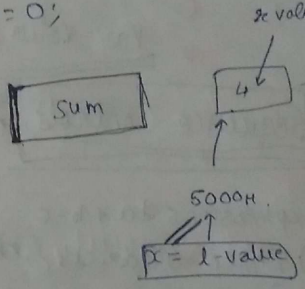
Dynamic Scoping.

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

sum = 8.

64.

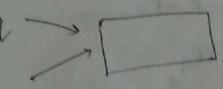
'x' & 'sum' are aliases.



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
```

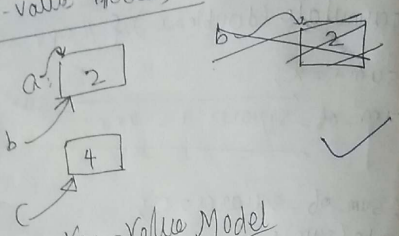
a: 2 b: 2

c: 4

Value Model

Execution Efficiency ↑

In var-value Model,



Var-Value Model

Sequence Controls:

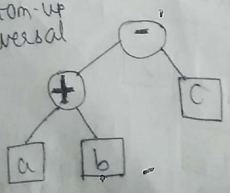
Expressions: a + b - c

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

3) Post-fix form / Prefix

Operators after operands

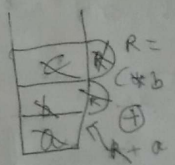
a b c * +

a b * c +

a + Res

Final Result

Operator before operands



* 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 (lp!=NULL) && (p→val != v))
{
    p = p→next;
}

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

Syntax directed

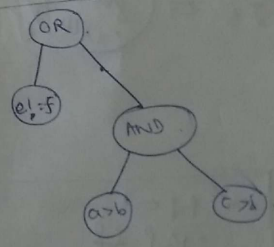
R1 ← a

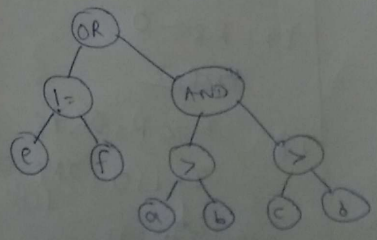
CMP R1, R2
JLE L1 ; A<=B Jump
L1: CMP R3, R4
JLE L2
L2: CMP R5, R6.
JE L3
S2
JMP L4

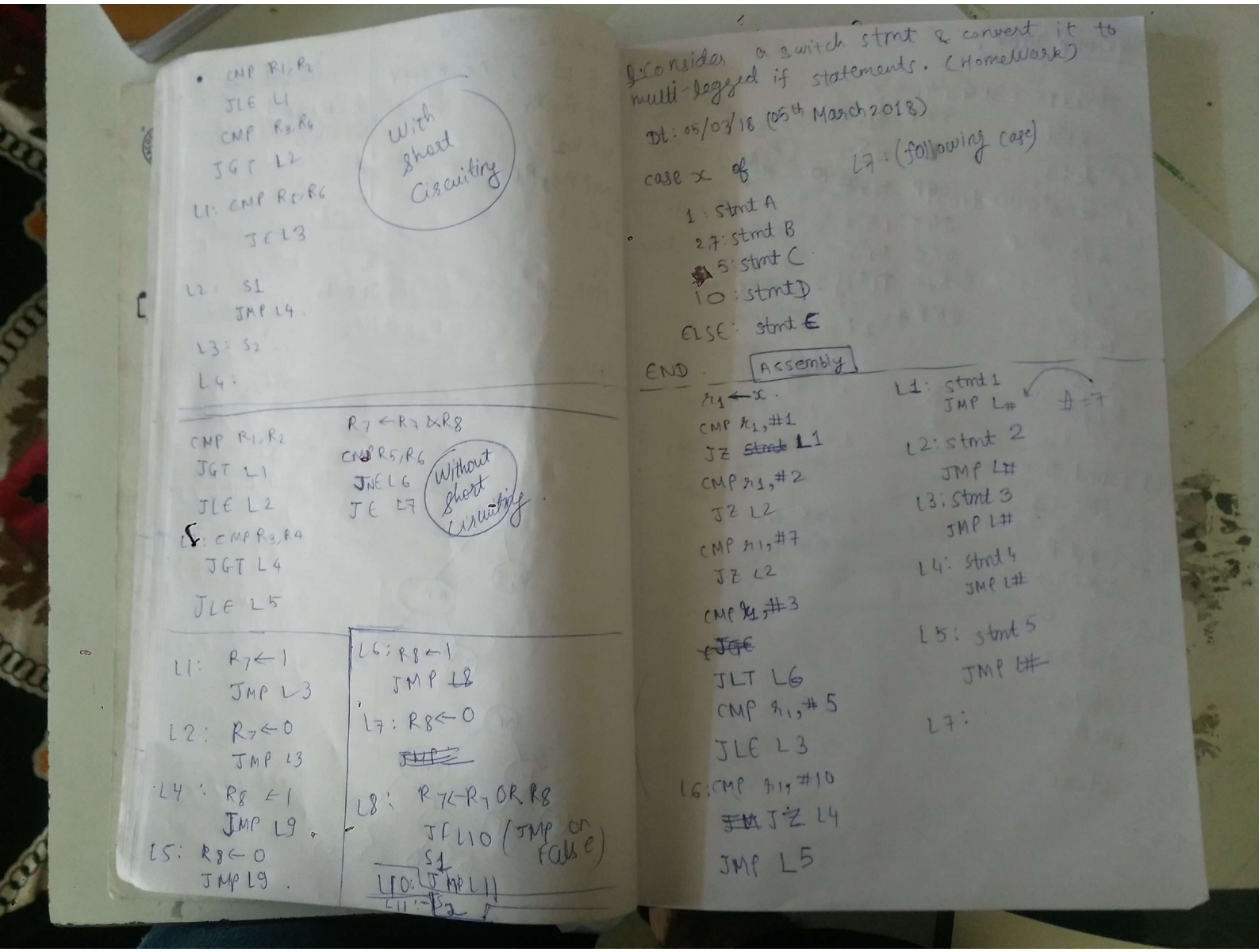
w/o

back-patching
filling label addresses.

S2.
JMP L4.







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
L1: 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
L8: R7 ← R7 OR R8
JF L10 (JMP on false)
S1
L10: JMP L11
L11: S2

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

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

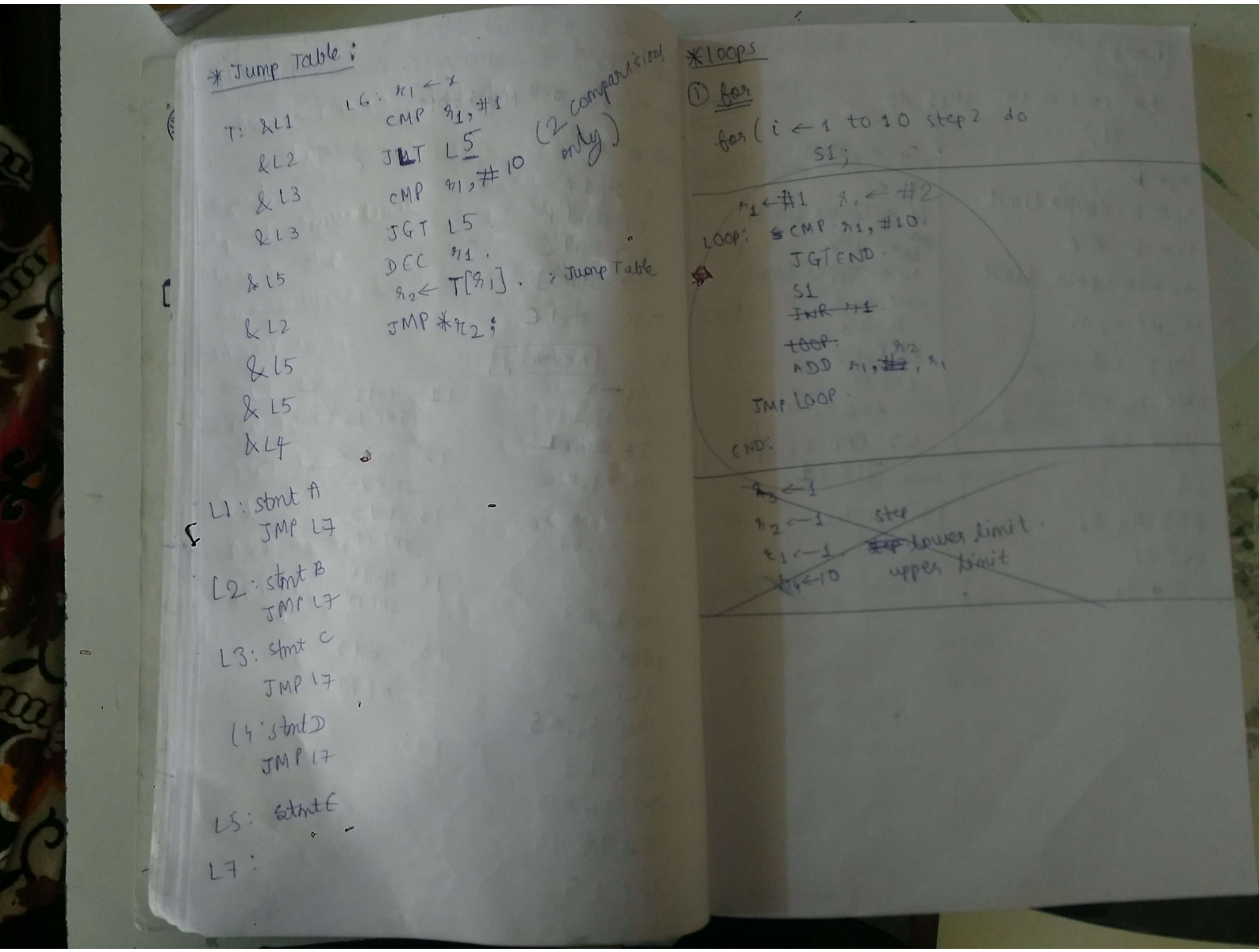
case x of
1: Stmt A
2,7: stmt B
5: stmt C
10: stmt D
ELSE: stmt E
END

L7: (following case)

Assembly

R1 ← x
CMP R1, #1
JZ ~~Stmt~~ L1
CMP R1, #2
JZ L2
CMP R1, #7
JZ L2
CMP R1, #3
~~JZ~~
JLT L6
CMP R1, #5
JLE L3
L6: CMP R1, #10
~~JNE~~ 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

LG: 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 comparisons 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, r2, r1
      JMP LOOP
END:
```

r3 ← 1
r2 ← 1 step
r1 ← 1 lower limit
r4 ← 10 upper limit

Loops

for i ← 1 to 10 step 1 do
    S1;

$R_3 \leftarrow i$
$R_1 \leftarrow 1$ lower limit
$R_2 \leftarrow 1$ step
$R_4 \leftarrow 10$ upper limit
L1: CMP R_3, R_1
JLT L2
CMP R_3, R_4
JGT L2
S1
ADD R_3, R_2
JMP L1
L2: $i \leftarrow R_3$

Optimization

$R_3 \leftarrow i$
$R_1 \leftarrow$ lower limit
$R_2 \leftarrow$ step
$R_4 \leftarrow$ upper limit
CMP R_3, R_1
JLT L3
JMP L2
L1: S1
    ADD R_3, R_2
L2: CMP R_3, R_4
    JLE L1
L3: $i \leftarrow R_3$

Dt: 08/03/18.

While loop

while ((a<b) && (c<d)) do
    S;

R1 ← a R3 ← c
R2 ← b R4 ← d.

LO: CMP R1, R2
    JGE L1
    CMP R3, R4
    JGE L1
    S
    JMP LO.
L1:

LO: CMP R1, R2
    JGE L1
    CMP R3, R4
    JGE L1
LOOP: S
    CMP R1, R2
    JGE L1
    CMP R3, R4
    JLE LOOP
L1:

Optimization

```
    JMP L0
L2: S
L0: CMP R1, R2
    JGE L1
    CMP R3, R4
    JLT L2
L1: ——
    ——
    ——
```

```
S1
while S2 do
begin
    S4
    S3
end
```

```
for(Object i: Array)
{
    sout(Array[i]);
}
```

```
#define sout(x,y)
        printf(x,y);
#define println()
        printf("\n");
```

```
for(S1; S2; S3)
{
    S4;
}
```

* SUBPROGRAMS

Signature →

~~f~~ f_1: int x int → float

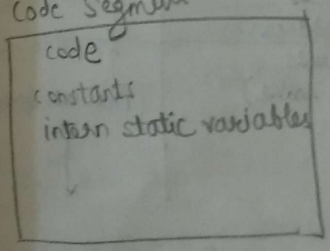
takes 2 integers — returns float.

float f1(int a, int b){

```
S;
f1(x,y);
S2;
}
```

code segment - Code, intern static variables

└ code segment/function.

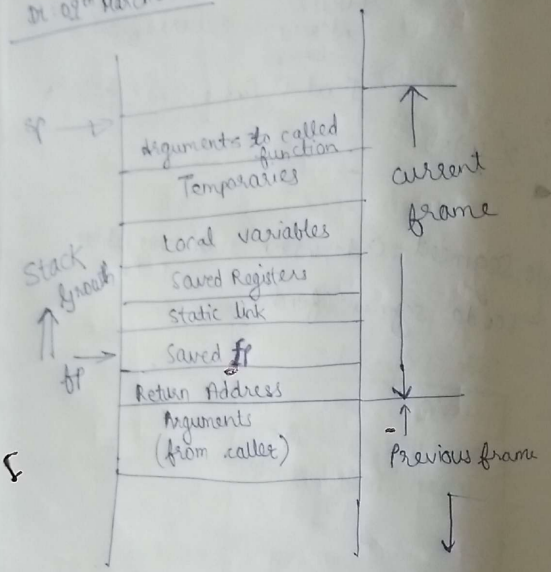


1) parameters
2) local variables
3) return address
4) return value

Activation Record

Calling Sequence

Dt 09th March 2018




```
f(int b)
{  int a, c;        → fp[-dc]
   a = b + c;
   return;
}
```

fp[db]

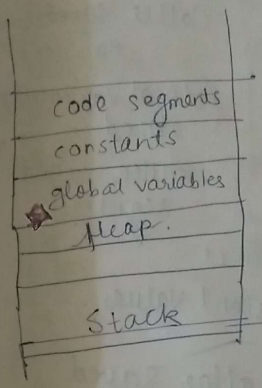
R1 ← fp[db] (MOV R1, fp[db])

R2 ← fp[-dc] (MOV R2, fp[-dc])

ADD R1, R2

MOV fp[-da], R1.

Stack



→ activation record of main function.

Caller.

1) Establishment of parameters
2) Save the return address on the stack
3) Store the frame pointer (old) value.

R1 R5 R7 R8 — callee

R5 R8 R10 — caller

4) Store Registers
5) Update the value of fp to the new value.
6) Update the program counters.

CPL=CO; Registers

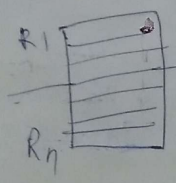
Caller - Saved Registers

Callee - Saved Registers

(local variables stored in callee-saved registers)

p: // caller
begin f()
end

Caller Saved Registers stores some Transient values.



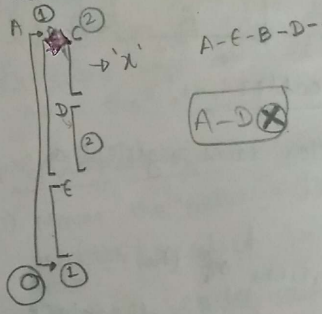
Caller saved Registers

Callee saved Registers

Dt: 12 March 2018

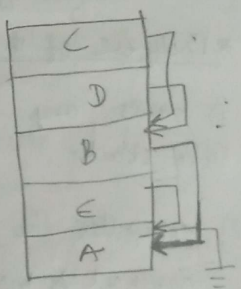
* Implementation of static scoping

Using 'static link', which is passed by the caller as an implicit additional parameter.



A-E-B-D-C

A-D ⊗



A-E-B-D-C.
0 1 1

If $level_{callee} = level_{caller} + 1$

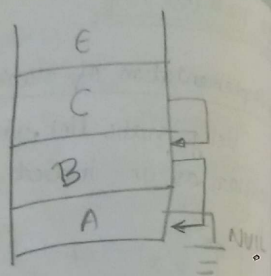
then static Link callee = fp caller

else Dereference ($l_{callee} - l_{caller} + 1$) times the static link of the caller

Use this value as static link of callee.

A-B-C-E
0 1 2 1

A-B-C-E
0 1 2 1



*Prologue of the caller

1) saves any caller-saves registers on the stack.
2) computes the values of the argument & moves them on the stack.
3) compute the static link & pass it as an extra hidden parameter.
4) Use a special subroutine 'call' instruction to jump to the callee & simultaneously store the return address.

*Prologue of the callee

1) Allocates a frame on the stack by subtracting a value equal to the size of AR caller from the s.p.
2) saves the old fp on the stack & assign a new value to the fp.
3) saves any callee-saves registers on the stack.

*Epilogue of the Callee

1) Moves the return value to register / specified location on the stack.
2) Restore callee-saved registers.
3) Restores fp & sp.
4) Jump back to the return address.

```
f(){  g(){
        }
g();
}
```

Epilogue of the Caller

1) Move the return value to the correct place
2) Restore the caller-saved registers.

Dt: 15/9/18. CPL

1) Leaf routine
   a) Not save ret. address if h/w stores in a register
   b) Static link computation & its passing not required.
   c) avoid saving caller saved registers

#define DIVIDES(a,n) (!(n % a))

DIVIDES(y+z, x) y+z % x
(PROBLEM)

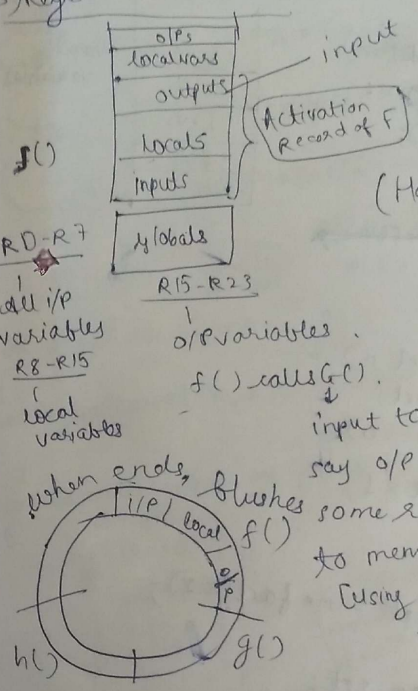
#define SWAP(a,b) {int t=a; a=b; b=t;}

SWAP(x,t);
↑
Problem

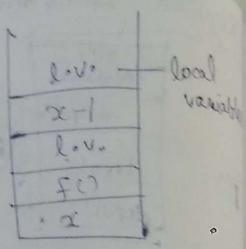
Evaluation before expansion in inline func^n (of arguments)

- inline func^n - expand func^n in place w/o any calling mechanism.
OR intelligent compilation

3) Register windows



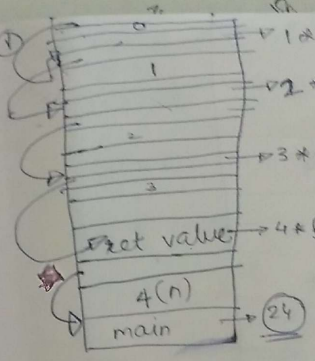
* f(x){
if(base-condition)
return v;
else f(x-1);
}



Dt: 16/07/18.

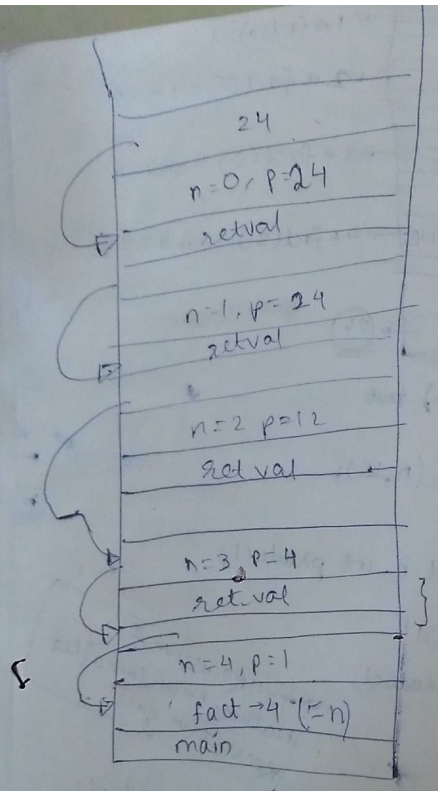
```
int fact(int n)
{
    int ret;
    if(n==0)
    { ret = 1;
    }
    else
    { ret = n * fact(n-1);
    }
    return ret;
}
```

fact(4).



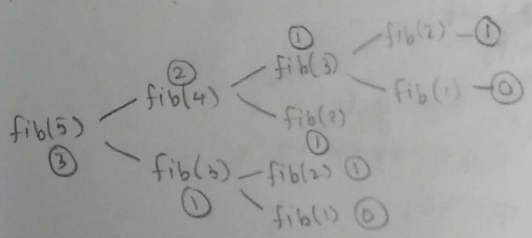

```
int fact(int n) int
{
int fa = factorial(n, 1);
}
int factorial(int n, int product)
{
    int ret;
    if(n==0)
    {
      ret = product;
    }
    else
    {
      ret = factorial(n-1, n * product);
    }
    return ret;
}
```

Tail Recursive – Nothing to be done after Recursive call




```
int fib(int n)
{   int ret;
    if(n==0)
    {
      ret=0;
    }
    else if(n==1)
    {
      ret=1;
    }
    else
    {
      ret=fib(n-1)+f(n-2);
    }
    return ret;
}
fib(4)
```

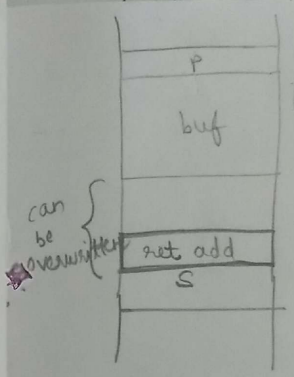
0 1 1 2 2 ...




```
int f[100] = {-1,-1,....}
int fib(int n)
{   int ret;
    if(n==0){ f[0]=0;
              ret=0;
    }else if(n==1){ f[1]=1;
              ret=1;
    }else{
```

```
    if(f[n]!=-1)
    {
      ret = f[n];
    }
    else
    {
      ret = f(n-1)+f(n-2);
      f[n] = ret;
    }
  }
  return ret;
}

int get_asct_num(FILE* s){
    char buf[100],*p;
    p = buf;
    do
    {
      *p = getc(s);
    }while(*p++ != '\n');
    *p = '\0';
    return (atoi(buf));
}
```



H/W range check for arrays