→ Machine Model

<op> L_1, L_2 (2 addr model)

our convention : $L_2 \leftarrow L_2$ <op> L_1

<op> L_1, L_2
- L_1 : mem/reg
- L_2 : reg (dstn must be reg) / mem (some allow)

- getreg (addr) : allocates reg for addr.
- codegen (3AC) : generates list of machine instrs for every 3AC instr we provide
- register descriptor : map of registers to variables
  $r_i \longrightarrow$ <addr>

uses

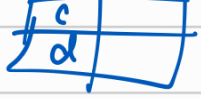
→  → we need to know mapping of registers

code

- address descriptor : var → { r_i, addr }
  can also be more than one reg having same var. Addr of a var is fixed throughout the prog.

assume memory allocator: given a variable allocates its to memory location, pehle se done.

a	addr
b	addr



3AC $a \leftarrow a+b$

2AC $a \leftarrow +b$

Add L_1, L_2 → must be in reg.

$L_2 \leftarrow L_1 + L_2$

compiler side

if addr_desc[a].first ≠ NULL
  r_a = addr_desc[a].fi
else r_a = getreg(a)
  mov a, r_5.

} Let $r_a = r_5$

if addr_desc[b].fi ≠ NULL
  r_b = addr_desc[b].fi. Let $r_b = r_7$
  add r_7, r_5
else
  Ad_b = addr_desc[b].sec Let Ad_b = 1000
  add (1000), r_5.

→ $a \leftarrow b+c$
  r_5 r_7

$r_5 \rightarrow a$

$a \rightarrow \{r_5, \langle addr \rangle\}$

mov b, r_5
add r_7, r_5 $a = b+c$

---

r_5 = free

a = Mem
b = Mem
c = Mem

mov b, r_5
$r_5 = b$

a = Mem
b = Mm, r_5
c = Mm

add <c>, r_5
$r_5 = a$

a = r_5 (only in reg)
b = Mem
c = Mem

we need to do spill / mov then a = mem, r_5

$r_5 \rightarrow a$	$r_5 \rightarrow$ free
$a \rightarrow r_5$	$a \rightarrow$ mem.

a spilled

Lec 04/02

Compiler

Binary code = foo code (ours) + Runtime code (compiler puts)

```
void foo(int x);        → formal parameter
main() {
    a = 5;
    foo(a);             → actual parameter
}
```

Activation Tree:
tells how things are getting called.
which func calls which func.

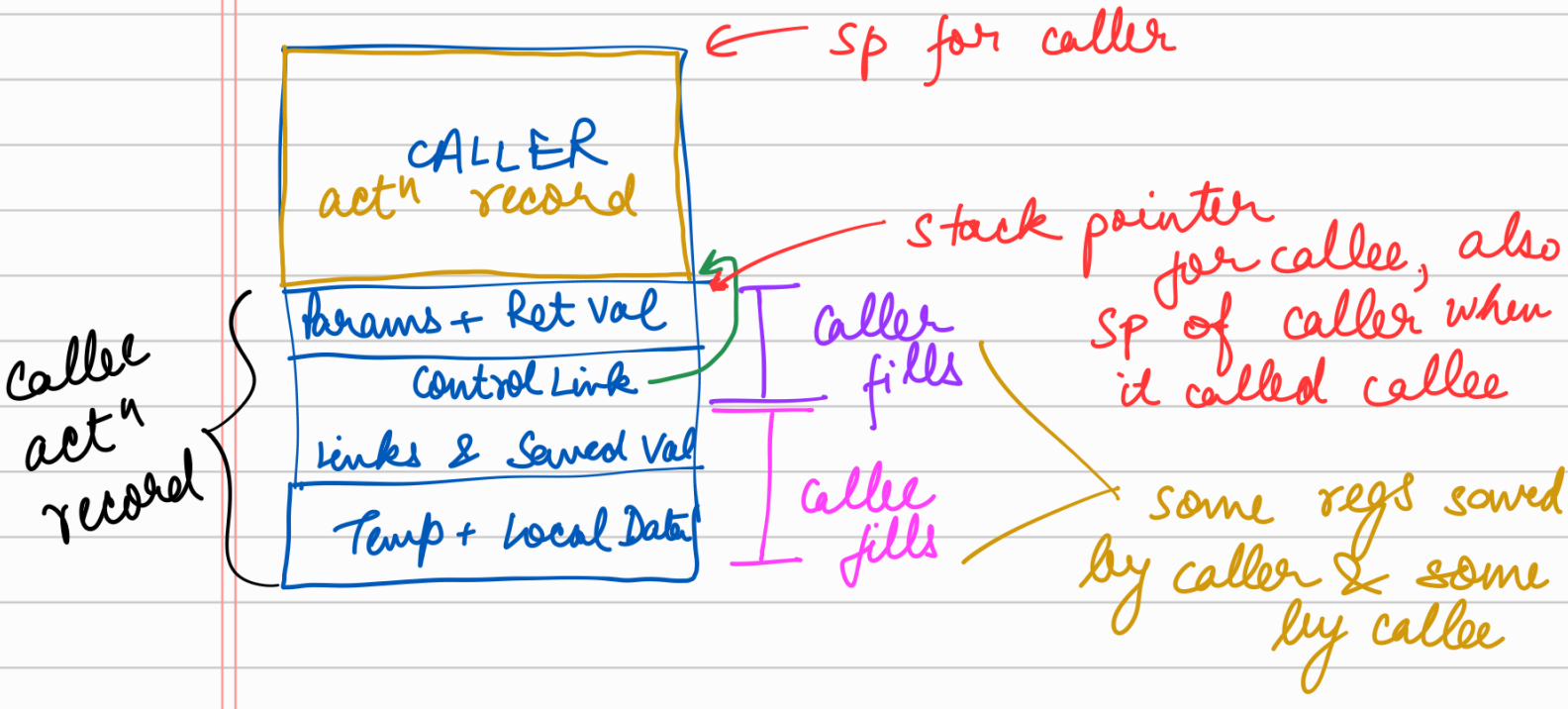
→ Similar to a dfs on this tree. So we use stack to do func calls.

→ **Activation Record**

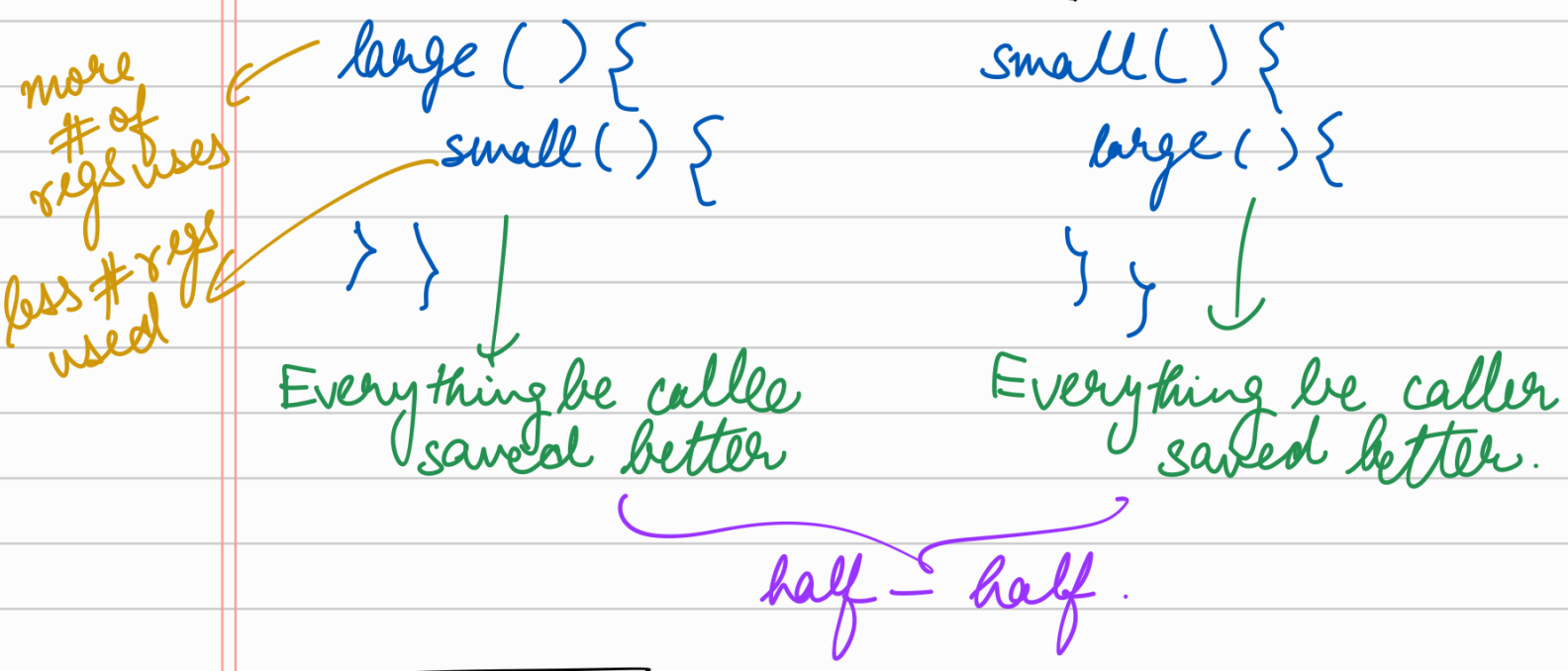
(per call

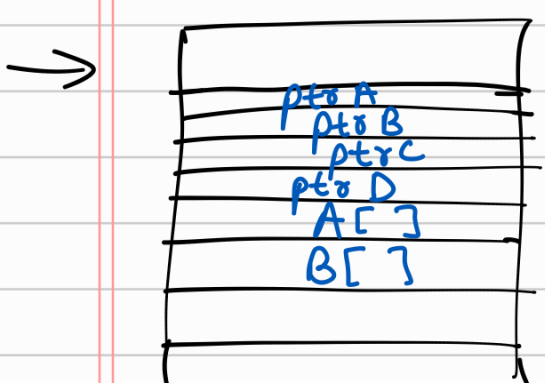
Linked list of records, per activation (of any funcⁿ)

Control links: pointer to the base of caller actⁿ record. Because the record size is variable (Temporaries, local data size variable.



Caller v/s Callee saved






```
int a;
int b[100];
int c;
int d[20]
```

for caching of c.

Lec 10/02/25

Compile Time Code v/s Run Time Code.

user code + data

push activation record, manage memory etc

Heap region – more overhead for management
free v/s occupied

Control Link : back pointers in a tree.

Stack Allocation –
life time of a program contains all programs called after it. (Ordering)

Dangling Reference – accessing something which is not in the scope of the prog at that point.

Malloc / Free : library calls manage memory during runtime. may use OS help.

Compiler may inject code into binary at runtime. (Brown col. code in example)

p = q } inc refct of q
dec refct of p

Unreachable garbage

based ga
collection

unreachable. But will never be dealloc. because everyone's refct >0

Liveliness based garbage collection

p is allocated & live, but second one is allocated but never used. Compiler can't figure out and dealloc second one, although this should be done for better performance.

Marking Based

Graph reachibility on Heap so whatever is not reachable can be deallocated.

Lec 11/02/25

Implicit Dealloc^n: C#, Java. No apis for free or dealloc.

Explicit Dealloc^n: C, C++, free, munmap

State of a prog: the values of all variables at that point of time.

name $\xrightarrow{\text{mapping}}$ value

Implicit decl^n v/s Explicit decl^n of variables

- Implicit decl^n → infer at runtime
- Explicit decl^n → told by programmer

Pascal
- procedure : void ret type
- functions : non void ret type

Variable Scope Management

i) Make their names diff and ensure correct names written during using.

ii) use a stack to maintain scope.

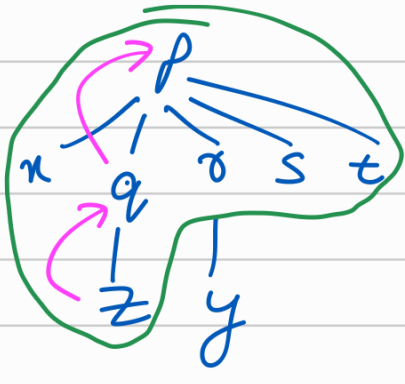
Access Links

in a procedure's activation record, access link points to AR of parent procedure (not who called it, but the procedure inside which it was declared → enclosing procedure

→ points to record in lexical scope.

Lec 17/02

scope resolution finds the nearest upper enclosure which has that variable



runtime act^n record of z has runtime setup access link which points to q. Same func diff act^n records may have diff access links.

establish only that part of graph which is req to resolve anything — siblings + ancestors

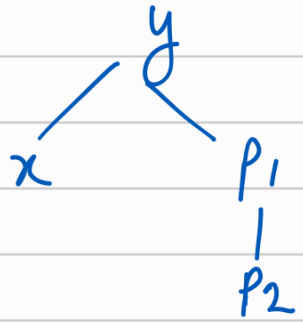
p calls x :

① $n_p < n_x$

(act^n record : access link of x points to p)

② $n_p \geq n_x$

p_2 calls x



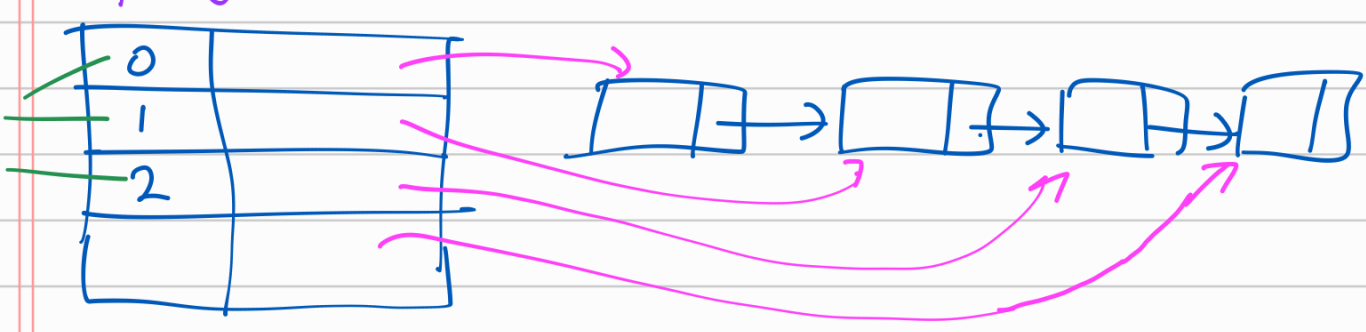
(act^n record of x: access link → y, control link → p2)

Access Link Array (Display)

if we can make an array of head ptrs because for each variable we would need # hops traversal for linked list.

Display Table

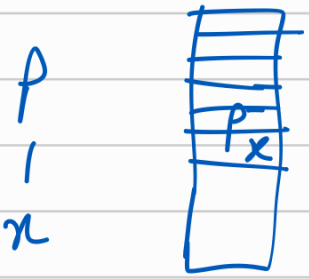
lexical chain

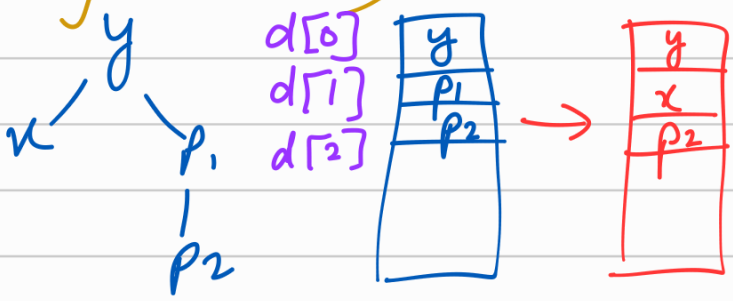


whenever a call is made, we add an entry into the table & act^n record destroyed then we remove its entry.

A procedure can only see variables of its and all parents in its lexical chain

(caching addresses)





x has more depth.
So correct to insert a new entry into the table
(index inc automatically)

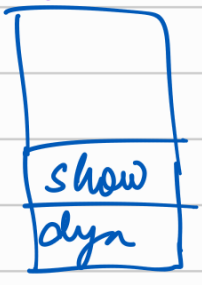
p_2 called x
first save act^n rec of p_1 before overwriting then on return from x restore it back.

⇓

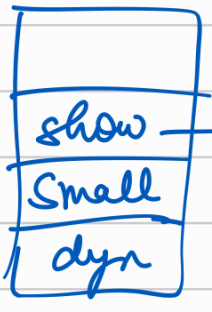
Whenever u are over-writing store it, because x may call someone, then p_2 will be overwritten but when p_1 was overwritten by x we did not know if below was useful or garbage

Dynamic Scoping

Static



(r = 0.25)



→ can see only its & dynamic
r = 0.25

dynamic

(r = 0.25)

traverses down and first act^n record which contains x

$\gamma = 0.125$

(couldn't reach dyn. found at small)

Make them global:
whenever you encounter change in value you save & then update the global variable. restore on return.

- C does not have call by reference.
It only has call by value.
Passing ptrs is also call by value.

reference — same memory addr of both

copy restore - diff value in actual params (old val) copied into local vars and before return, actual params values get updated

call by name -

```
foo(int y) {
    ≡≡≡
}
```

```
int x;
foo(x);
```

replaced by code of foo all written here, (y) of foo replaced by x.

if x was a local var of foo, then gibberish