- Parameter passing mechanisms

  Formal arguments and actual arguments must be associated during run-time to do parameter passing.

  Declaration of formal arguments allow space allocation of definite size in the callee's AR

  1. call-by-value: Caller evaluates and puts values of arguments in callee's AR

  2. call-by-reference: caller puts addresses of actual arguments in

callee's AR (r-value of formal parameter is l-value of actual parameter)

what if actual argument is an expresssion with no l-value?

3. copy-restore (value result, mixture of 1 & 2): caller evaluates actual parameters and passes r-values to callee. Caller saves their l-values. Upon return, it restores the r-value of formal parameters to l-values of actuals.

4. call-by-name: literally substitute actuals for the formals. Local names in callee that clash with names in actuals are renamed.

l-value of actuals are passed in 2 and 3; r-value in 1, and the text itself

in 4.

ex: a hypothetical language whose parameter passing is left unspecified

```
proc swap(x,y)
    temp:=x;
    x:= y;
    y:= temp;
endproc;
```

assume we make the call `swap(i,a[i])`

in call-by-value, no change in actuals

in call-by-reference, l-values of `i, a[i]` are passed on. `x,y` indirectly refers to the same l-value.

in copy-restore, r-values of `i, a[i]` are passed on, but on return, `x,y`'s r-values are put in l-values of `i, a[i]`

in call-by-name, i is substituted for x, and a[i] for y:

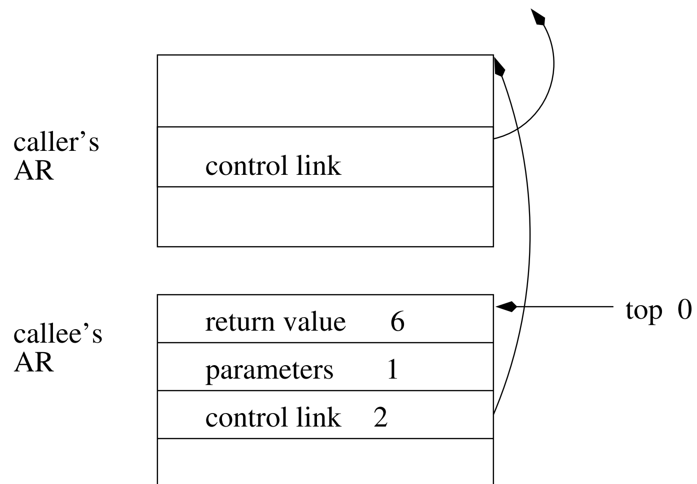
```
temp := i;

i:= a[i];

a[i]:= temp
```

if I_0 is initial value of `i`, `a[a[`I_0`]]` becomes I_0, not `a[`I_0`]`.

Argument passing shows that there are some things that the caller must pass on to the callee before the call, and return values of the callee must be accessible to the caller after the call is over. How is this done?



CALLER
————————

0. push control stack (enw entry for
 callee's AR)

1. evaluate actual parameters
 and put them in callee's AR
 (they are a fixed distance away
 from caller's AR)

2. set the control link of the CALLEE
 to point to the CALLER (it is also
 fixed distance away from caller's AR)

3. save next intsruction in CALLEE's
 machine status (also fixed distance away)

9. caller retrieves return value (although
 stack is popped, it is still fixed distance away
 from the caller's AR)

CALLEE
————————

4. save machine status
5. initialize and execute
6.. put return value in place
7. restore machine status and
 and reset instruction pointer
 to caller's next stmt (was saved
 as part of machine status in 3)
8. pop control stack

- Why do we need Access links in addition to control links?

  In *dynamic scoping*, the calling sequence determines the environment. Since calling sequence is shown by control links in ARs, there is no need for extra links.

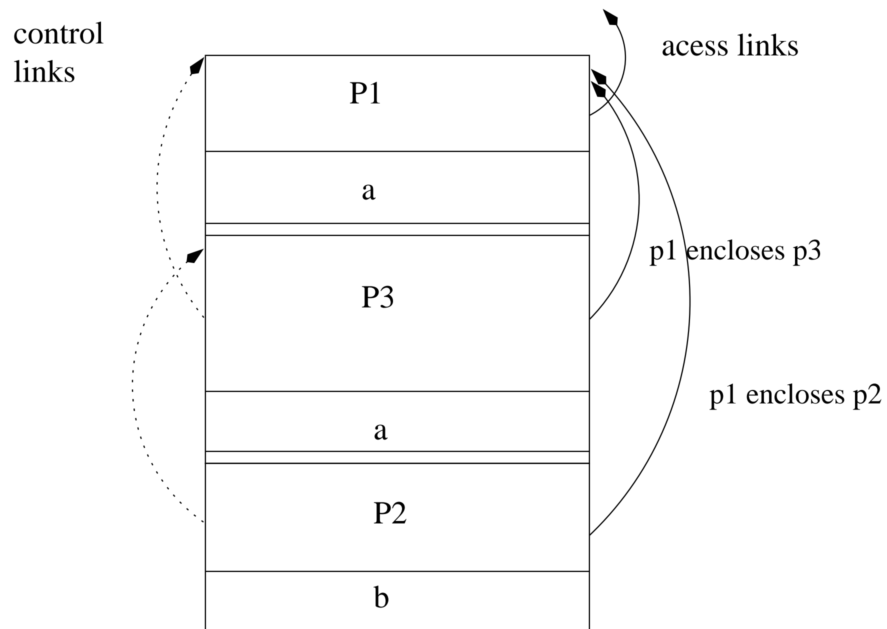
  In *lexical scoping*, textual enclosures in the program determine the environment. But this enclosure is not necessarily the same as calling sequence (hence we need extra information)

```
proc p1 ()

      var a;
```

```
...
proc p2 ()
     var b;
     begin
          x := a +b ;
     end
proc p3()
     var a;
     begin
          p2();
     end
```

- Access link: if q is nested immediately in p (textual enclosure), q's access link points to the *most recent activation* of p (and there must be one, why?)



Looking for globals in lexical scope: if not in current AR, follow the access link

- Accessing the access links: if a proc at nesting depth n_p refers to a nonlocal $\mathtt{a}$ at n_a:

  $n_a \leq n_p$ due to lexical scoping. Follow $n_p - n_a$ number of access links to reach the AR of $\mathtt{a}$'s definition block. This value can be computed at compile time (for each nonlocal in every block)

  The tuple ($n_p - n_a$,a's offset) defines the address of any nonlocal

- Setting up the access links: assume p calls x:

  if $n_p < n_x$, x is nested in p. Access link of x points to the access link in AR of the caller (second from top in stack)

  if $n_p \geq n_x$, x is not declared within p (due to property of lexical scope). the procedures at nesting depths 1,2, $n_x - 1$ must be the same (again due to lexical scope). $n_p - n_x$ number of links reach the level of x again, hence $n_p - n_x + 1$ links reach the block that most closely encloses p and x. This can be computed at compile time as well.

- If nesting is deep and calling sequences are not hierarchical, this method causes a lot of hopping along access links. One alternative is to use *display* mechanism.

- displays make use of the fact that, in lexical scope, two procedures at same nesting depth cannot refer to each other's bindings as nonlocal reference. So, it is possible to use only one storage for any nesting depth n.

`d[n]` is an array of AR pointers, it's value is set during calls and returns. `d[i]` points to the AR of the most current activation of active procedure at level i.

When a new AR for a proc at level i is set up

1. save the value of `d[i]` in the new AR (later to be restored to this value upon return)

2. set `d[i]` to point to the new AR

- Why does this scheme work? let's say p calls q:

  - if $n_p < n_q$ then $n_q = n_p + 1$ because otherwise q will not be visible to p. This means that there is no need to change the first n_p entries in the display.

  - if $n_p \geq n_q$ then `d[i]` entries for $i = 1, 2, \ldots, n_q - 1$ are the same. Save `d[`n_q`]` and reset.

- Lexical scoping makes most of the procedure interactions predictable at compile time. Access to nonlocals can be fixed at compile time too. Dynamic scoping puts the burden on run-time (though nonlocal management is trivial thru ARs). Dynamic scoping seems better suited to

domains of computations in which a procedure is expected to behave differently depending on who calls it (as in some AI applications).