- Dynamic aspects of language processing need to be dealt with during execution: local names, implicit flow of control, storage organization, etc.

  1. storage organization (run-time stack management for flow of control)

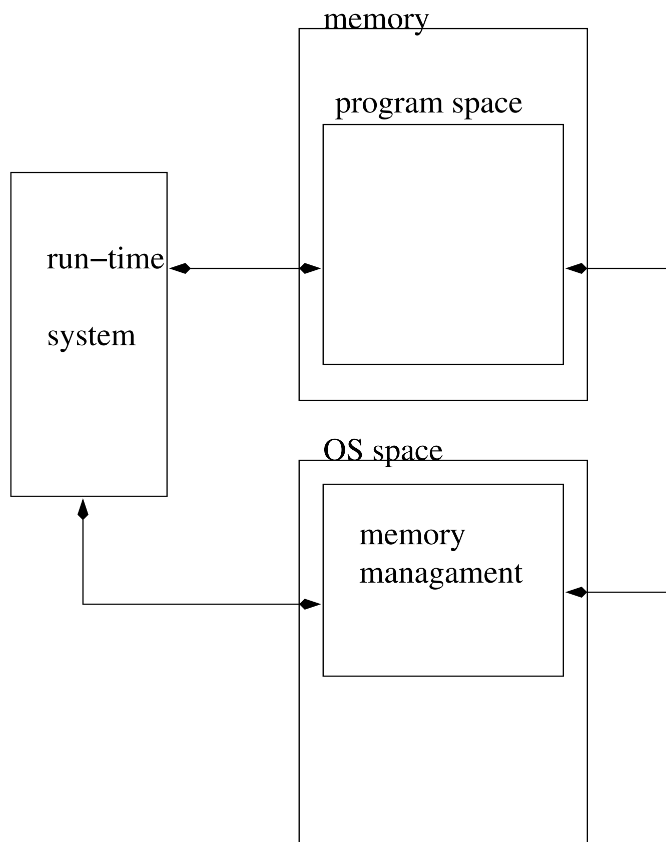
  2. storage allocation (static or dynamic layout of program/data space)

  3. Run-time counterparts of SL constructs (nesting, block structure)

  4. Access to names (local/nonlocal, lexical/dynamic scoping)

5. Parameter passing

6. symbol-table access and management



functional division of labor
for run−time, executable code and OS

- Translating IC declarations to storage layout within a block:

  Relative addressing within a block

  - base address of the block

  - offset from base

  - amount of space (can be deduced from type of the symbol)

  ex:
```
struct symtab {
  char *name;
  struct typenode *type;
  int blockno;
```

```
int addr;};
```

```
func p ();

var a:  int; b:real; c:int
```

assuming one word for integers and two words for reals:

p

0	a
1	b
3	c

syntax-directed way of calculating storage:

```
P -> proc ID { blockno++; offset=0} (Params) Decl Body { ...}

Decl -> Decl ; D | D { ...  }

D -> ID : T { enter(ID.val,T.type,blockno,offset)

           offset=offset+T.width}

T -> int { T.type=INT; T.width=2}

T -> array [num] of T { T.type=array(num.val,T1.type)

           T.width=num.val * T1.width}
```

Records can be treated like procedures as far as storage layout is concerned.

- Run-time management of storage
  - lifetime of variables
  - parameter passing
  - function call/return
  - dynamic storage

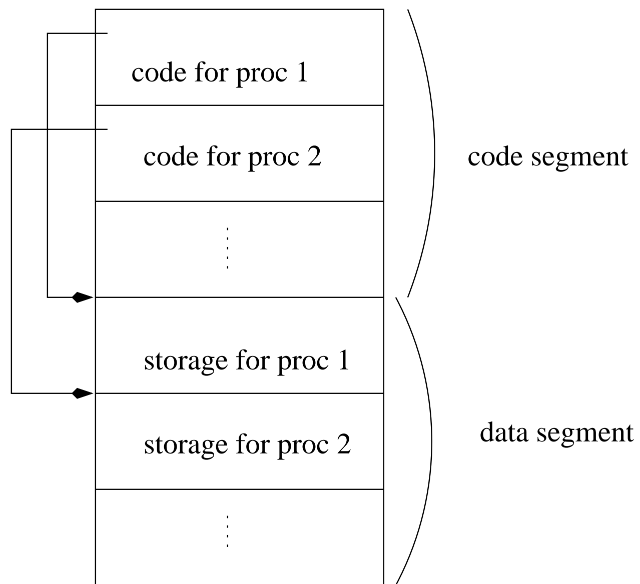
- Allocation strategies:

  1. Static. All storage laid out at compile time (no stack/heap). eg. Fortran

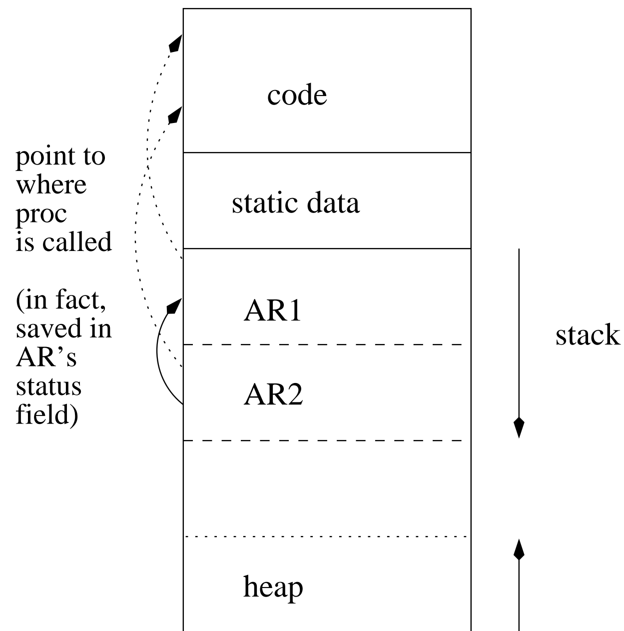
  2. Dynamic. Some storage laid out at run-time (at least, the procedure activations).

  In static allocation, there is no room for dynamic aspects.

  Recursion is not normally treated as dynamic aspects of PLs, but it *cannot* be handled either:



FORTRAN's storage allocation
(predefined storage for every proc)

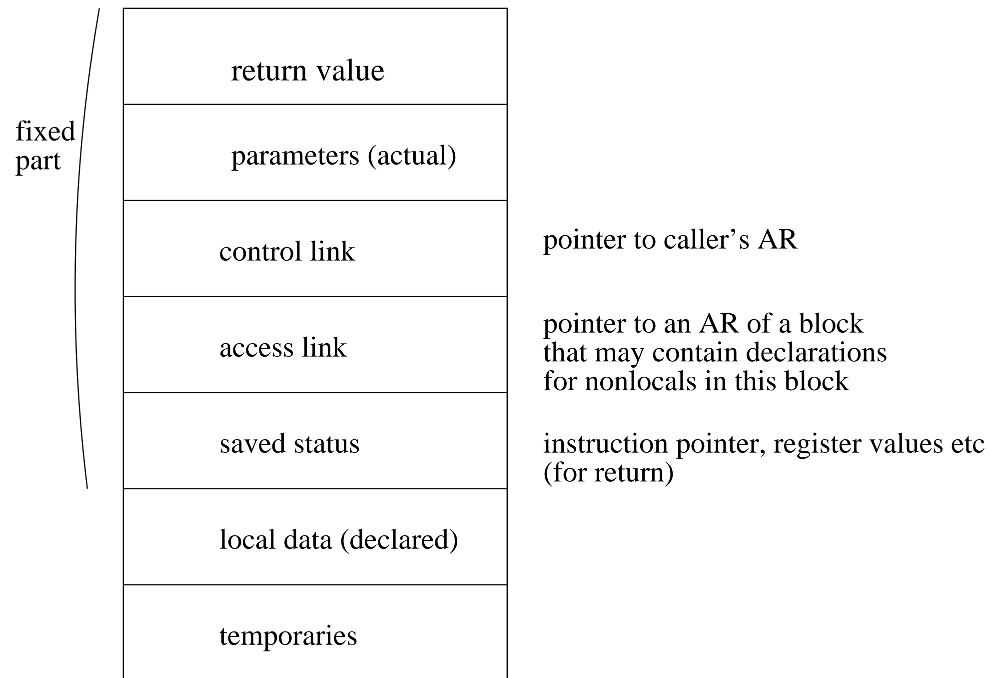


- The structure of AR

fixed part: things that can be located by the caller without having to know the internal structure of the callee.

block-dependent part: procedural-specific stuff (local vars etc.)

- Why does the caller need the fixed part? Some things must be put in there *before* control is transferred to the callee, and some things must be accessible *after* the callee returns.



the structure of a single activation record

- RECURSION and RUN-TIME STACK

Most target languages lack facilities for recursion (in fact, most ILs don't have it either)

A recursive call must be translated to nonrecursive calls either in SL or in IL. Nonrecursive version is mapped to TL just like other SC or IC

ex: doing it in pseudo-SL (pseudo because it makes reference to run-time stack, which is not accessible to SL)

an algorithm for recursion removal:

1. push values of locals onto the stack

2. push address of next instruction on to the stack

3. set the values of parameters, and go to the beginning

4. when procedure completes, it pops return address and values of locals, resets the locals and goes to return address.

For *end (tail) recursion*, the stack is not necessary since there is no need to keep variables, and where you return is end of procedure.

```
procedure traverse(t:link)
    begin
       if t <> z then begin
          visit(t);
          traverse(t↑.l);
          traverse(t↑.r)
       end
    end
```

remove tail recursion:

```
procedure traverse(t:link)
```

```
label 0,1;
begin
    0:if t = z then goto 1;
    visit(t);
    traverse(t↑.l);
    t:=t↑.r; goto 0
1:end
```

remove nontail recursion (need stack for t and address):

```
procedure traverse(t:link)
     label 0,1,2,3;
     begin
         0:if t = z then goto 1;
         visit(t);
         push(t); push(3:);
         t:=t↑.l; goto 0;
         3:  t:=t↑.r; goto 0
         1:  if stackempty then goto 2;
         lbl:= pop(); t:= pop(); goto lbl;
```

```
2:end
```

In IC, push/pop will be instructions to run-time stack manager.