Subprograms

- Characteristics:
  - → single entry point
  - → calling unit suspends
  - → control returns to caller
- Formal parameters & actual parameters
  - → proc p(x: real, y: int)   p(5.1, a + b)
- Correspondence
  - → positional
  - → keyword p(y => a + b, x => 5.1)
- Default values
  - → proc p(x: real := 1.0, y: int)
  - → p(y => a + b)

Parameter Passing

- **Issues**
  - → **Data flow between formal and actual parameters:**

actual -> formal	"in mode"	(calling -> called)
formal -> actual	"out mode"	(called -> calling)
both	"inout mode"	

  - → **Transfer by copy or by access path (pointer)?**
  - → **When are actuals evaluated?**

Methods

- Pass-by-value
  - → mode: *in*
  - → transfer: by copy

    => no access to outer environment
  - → evaluation: actual evaluated at time of call

    e.g., $f(2 + 3) \equiv f(5)$
  - → Note: protects actuals, *may be inefficient (copy)*

- Pass-by-result
  - → mode: *out*
  - → transfer: by copy
  - → evaluation: address to copy back to evaluated at time of call
  - → Note: what happens here?

    p(x,x)

Methods (continued)

- Pass by value/result
  - → mode: *inout*
  - → transfer: by copy (in and out)
  - → evaluation: at time of call
- Pass by reference
  - → mode: *inout*
  - → transfer: by shared access path
  - → evaluation: address of actuals evaluated at time of call
  - → Note: pass-by-value/result ≠ pass-by-reference

Example -- value/result vs. reference

```
program foo;
var x : int;
    procedure p(y : int);
    begin
        y := y + 1;
        y := y * x;
    end;
begin
    x := 2;
    p(x);
    print(x)
end.
```

Here, y is an <u>alias</u> for x

	value/result		*reference*	
	x	**y**	**x**	**y**
(entry to p)	2	2	2	2
(after y := y + 1)	2	3	3	3
(at p's return)	6	6	9	9

Another Method

- **Pass-by-name**
  - → **Textual substitution of actual for each occurrence of formal**
  - → **mode:** ***inout***
  - → **transfer: ?**
  - → **evaluation: when formal is *demanded* in *body* of procedure**

    => eval 0 or more times

- **Advantage: Delays evaluation of actuals**

```
function f(p: bool, c: real, a: real):real
begin
   if p then c else a
end


f(x = 0, 1.0, 1.0/x)
```

Pass-by-Name (continued)

- **Disadvantages**
  - → **inefficient**

    re-evaluation of actuals

    need *thunk* = (code,env)

    special implementation issue

```
procedure p1;
var x:int;
begin
    x := 2;
    p2(x+1);
end;

procedure p2(y:int);
var x:int;
begin
    x := 5;
    glob := x + y
end;
```

NOT same x

x + x + 1 i x + 2 + 1

Now: glob = 11 or 8?

Pass-by-Name (continued)

→ **May be hard to understand**

```
procedure swap(x,y);
var temp:int;
begin
    temp := x;
    x := y;
    y := temp;
end.

swap (a[i],a[j])
        -- temp := a[i]
        -- a[i] := a[j];                YES
        -- a[j] := temp;

swap(i,a[i])
        -- temp := i;
        -- i := a[i];
        -- a[i] := temp;
```

Consider case when i = 2, a[2] = 5!

Procedure and Function Arguments

```
program p123();
procedure p1(p);         -- p is a procedure parameter
var x: int;
begin {p1}
   x := 1;
   p();                  -- the passed procedure is invoked
end; {p1}

procedure p2();
var x : int;
   procedure p3();
   begin {p3}
      print (x)          -- must be the x in p2, not the x in p1!
   end; {p3}
begin {p2}
   x := 2;
   p1(p3)
end {p2}

begin {main}   p2();   end. {main}
```

- **Note: This requires *thunks* for static scoping ("deep binding").**

Implementing Subprograms

- Call
  - → provide storage for parameters
  - → provide storage for locals
  - → save execution status of caller
  - → provide access to non-locals
- Return
  - → copy back parameter values (if necessary)
  - → may deallocate local storage
  - → restore access to non-locals

    registers

    variables with same name as locals
  - → return control to caller

Preliminaries

- Need memory for code & data
  - → code: static (*instruction space*)
  - → data: changes with each activation => activation record (*data space*)

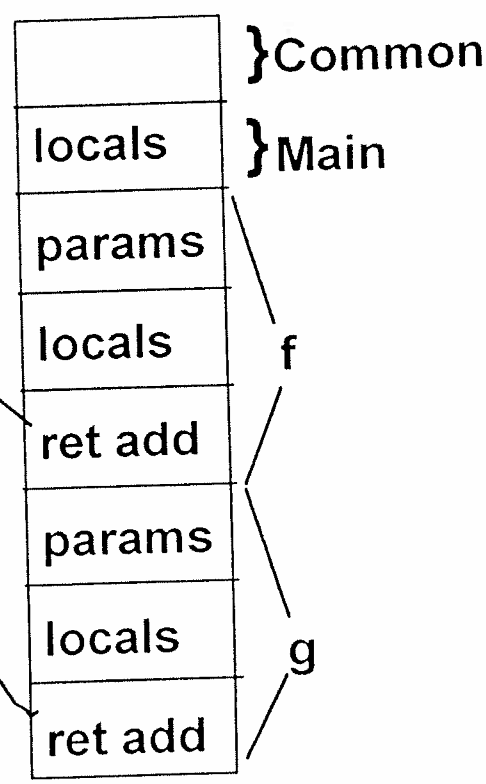
- Three ways to allocate storage for data:
  - → static
  - → stack-based
  - → dynamic

Static Allocation (FORTRAN)

- **Simplifying factors**
  - → no recursion
  - → size of variables known statically
  - → access non-local variables with COMMON statement

- **Result**
  - → can allocate all memory statically
  - → example:

```
main
  ...
    f (...)
  ...
proc f (...)
  ...
    g (...)
  ...
proc g( )
  ...
    g (...)   -- !!!
```



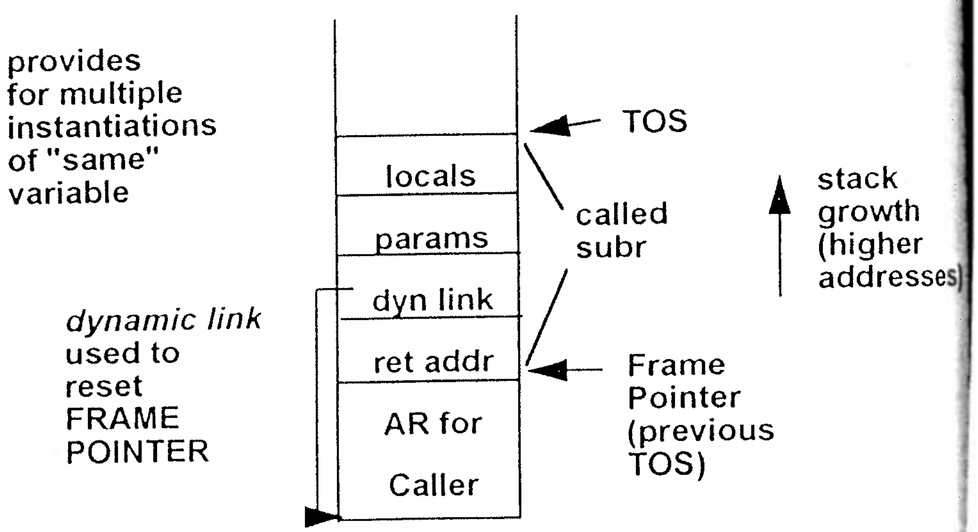
  - → What about recursion???

Handling Recursion

- procedure activations are LIFO => use a stack
- for now, two simplifying assumptions:
  - → no non-local references
  - → size of all variables known statically

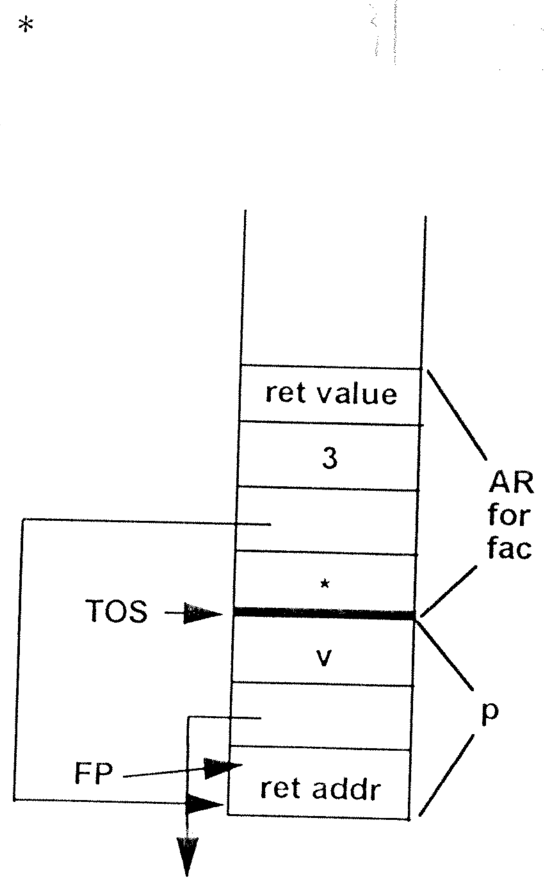
Activation Records on a Stack

- Call
  - → set return addr, dynamic link
  - → push actual parameter values
  - → allocate space for locals
  - → FP := TOS, reset TOS
- Return:
  - → instruction pointer := mem[FP] (return address)
  - → TOS := FP (clear this activation record)
  - → FP := mem[FP + 1] (dynamic link)



Example: factorial

```
program p;
var v : int;
    function fac(n : int) :int;
    begin
        if n ≤ 1 then
            fac := 1
        else
            fac := n * fac(n - 1);
    end;
begin
    v := fac(3);
    print(v)
end.
```



Lifting Restrictions

- **Allow size of variables to be determined dynamic**
  - → **semi-dynamic -- once size is fixed it remains**

    e.g., arrays
  - → **dynamic**

    e.g., varibales
- **Allow non-local references**

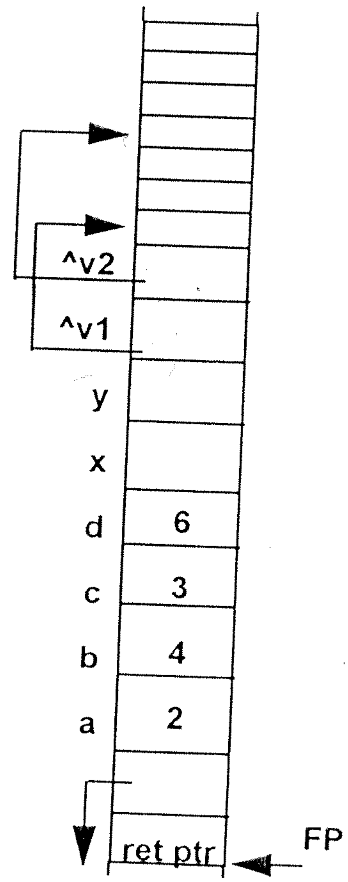
Semi-Dynamic Variables

- **Size fixed at *unit invocation time***

```
proc p(a, b, c, d: int);
var x,y: int;
    v1[a..b], v2[c..d]: array of int;
...
p(2, 4, 3, 6)
```

- **Solution**

Allocate space for pointer to each semi-dynamic variable first, then space for actual arrays; offset of pointer known statically.



Dynamic Variables

- **Example: flex array (Algol 68) => size of array may change arbitrarily at runtime**
  - → **cannot be stored on stack**

    proc p;

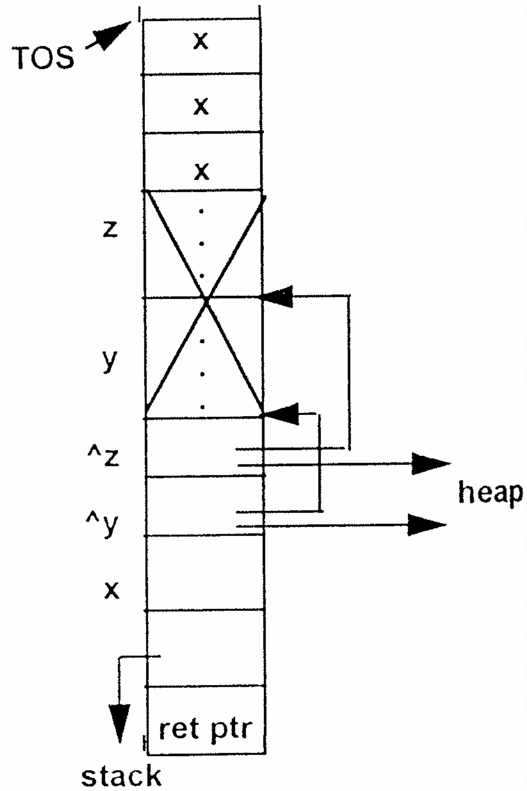
    var x : int;

    y,z : flex array;
  - → **need a heap**

    less effcient

    more flexible
  - → **store pointers to heap in stack**

At any point in time you might need more space but next stack space might be in use.



Non-local References

- **Need**
  - → **which Activation Record (AR)**
  - → **variable's local offset within AR**

    Note: this is *all* we need for a *local* reference; we already have the pointer to the AR

- **Two cases**
  - → **static scoping -- reference is determined by static layout of program**
  - → **dynamic scoping -- reference is determined by call chain**

Static Scoping

- Need access to the static environment => *static link*.

For a given procedure or function f, f's static link (usually) points to the most recent AR for the procedure or function that statically encloses f.

Exception: where f is passed as a parameter

Static Links

- **Using static links:**
  - → **The AR containing the definition of a variable used in f is always a fixed distance d from f's AR along the static chain. (set of static links)**
  - → **If f uses a non-local x that is defined in g,**

    d = level(f) - level(g) # links that you traverse
  - → **Know levels and that x was defined in g <u>statically</u>!**
- **Setting static links:**
  - → **If a calls b, then b's static link should be set to the AR (level(a) - level(b) + 1) links along th static chain starting at a.**
  - → **cases:**

    level(a) = level(b) - 1 (b in a)

    level(a) = level(b)

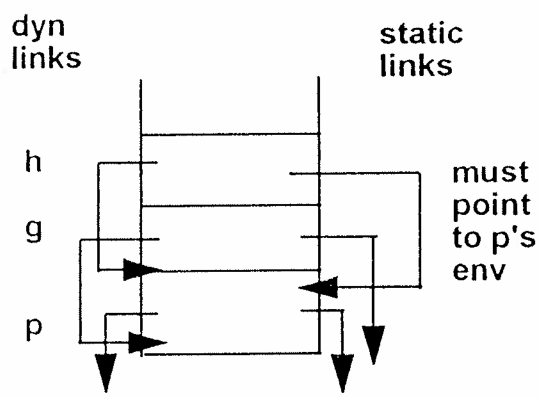
    level(a) > level(b) (b outside a)
  - → **it works!**

```
proc a
   proc b
   proc c
   proc d
   proc e
```

Procedure Parameters

```
proc p;
var x;
    proc h;
    ...x...
    end {h}
...g(h)...
end {p}

proc g(f : procedure);
var x;
...f...
end {g}
```



→ **g can't set up h's static link; it must be passed in from p**

this is the *env* part of a thunk = (code, env)!

Procedure Parameter Confusion

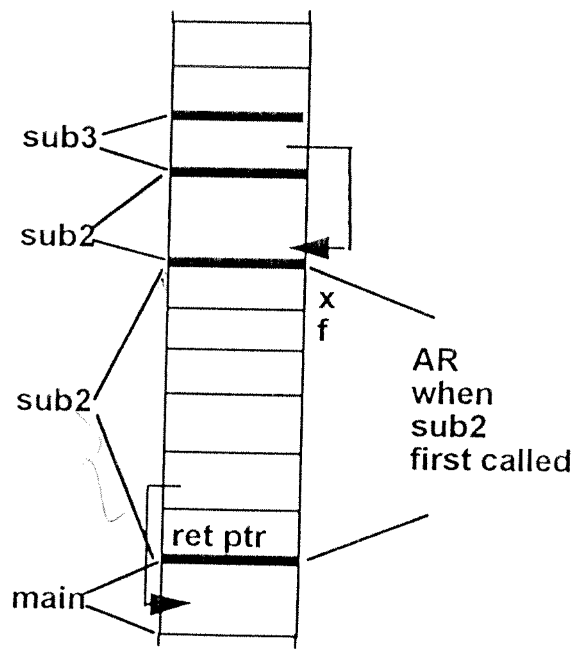
```
program main;

  procedure sub1;
  ...
  end {sub1}

  procedure sub2 (proc f)
  var x:
    procedure sub3;
      ...
      x := 0;
      ...
    end {sub3}
  ...
  f;
  sub2(sub3);
  end {sub2}
...
sub2(sub1);
end {main}
```

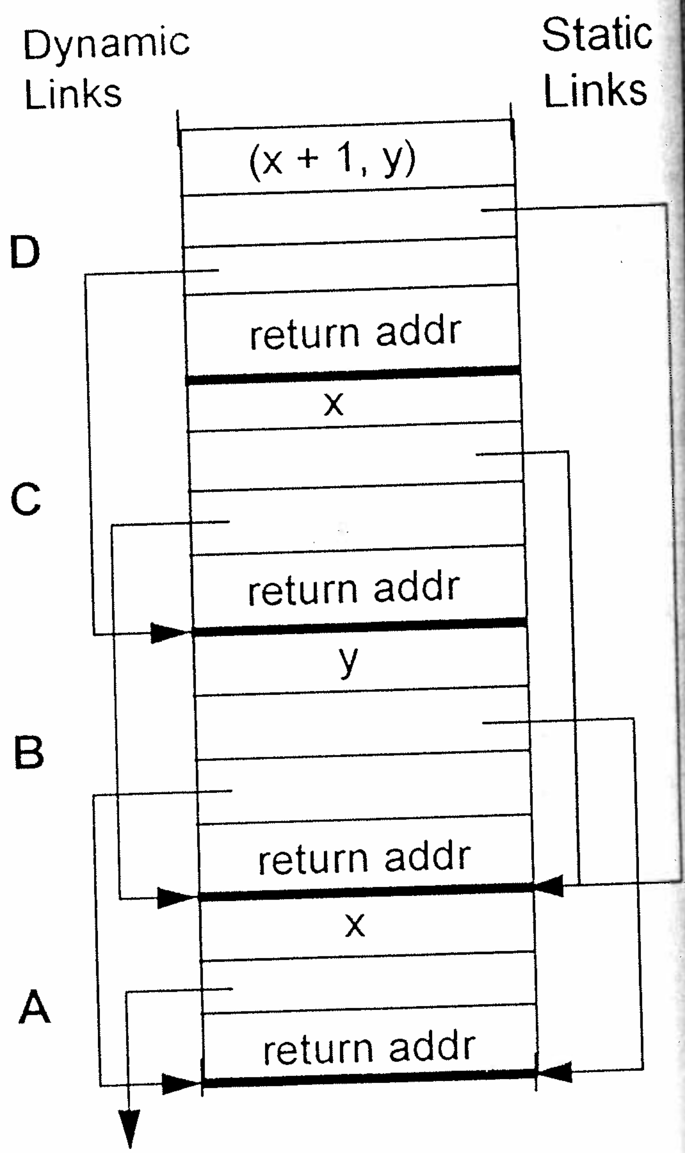
main -> sub2 -> sub2 -> f

referencing environmnet is not most recent invocation of sub2



Example: A -> B -> C -> D

```
proc A;
var x;
    proc B;
    var y;
        proc C;
        var x;
        ... x + y..
        D;
        end {C}
        proc D(z);
        ...x + y...
        end {D}
    C;
    ...
    end{B}
B;
...
end{A}
```



→ **When are dynamic & static links the same?**

Displays

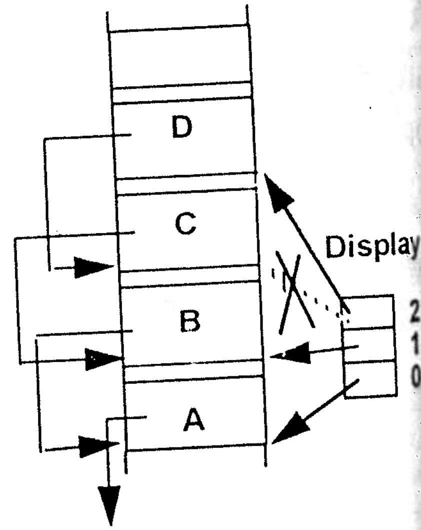
- A *display* contains pointers to the currently accessible activation records at each static level.

- A display is usually implemented as an array, with size equal to the maximum nesting depth of the program.

A -> B -> C -> D

```
proc A;
var x;
  (1) proc B;
      var y;
    (2) proc C;
        var x;
        ... x + y..
        D;
        end {C}
    (2) proc D(z);
        ...x + y...
        end {D}
      C;
      ...
      end{B}
  B;
  ...
  end{A}
```

D's ref env ≡
D at level 2
B at level 1
A at level 0



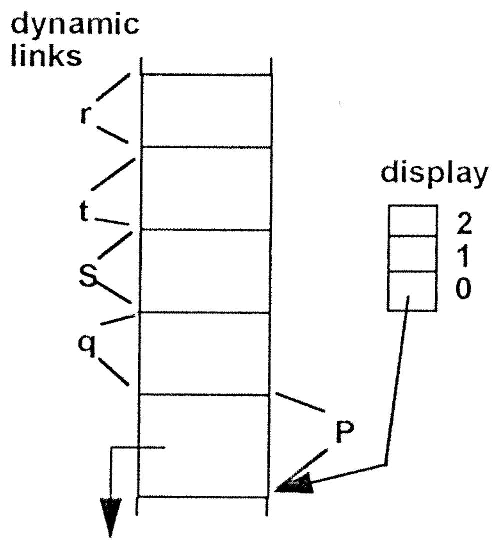
when D exits
DSP[2] is reset
to value stored
in D's AR
i.e. ^ to C

Procedure Parameters with Display

```
Proc P;
var x;
(1) proc q;
    var y;
  (2) proc r;
        ...
        x + y
      end {r}
    ...
    S(r)
    end{q}
...
q;
end {P}

(0) Proc S (proc f);
var x,y;
    proc t(proc g);
        . . .g. . .
    end {t}
. . .t(f). . .
end {S}
```

P->q->S->t->g/r



for uniformity you might want to store *entire* display *all the time*

Displays vs. Static Links

- References to non-locals:
  - → static links: follow ($level_{def}$ - $level_{use}$) static links
  - → display: follow one pointer
- Procedure call: (f -> g)
  - → static links: follow ($level_f$ - $level_g$ + 1) static links
  - → display: save all or part of display on stack, update with new values
- Procedure return:
  - → static links: none
  - → display: restore to saved value

Dynamic Scoping

- **Deep access (reference to x)**
  - → **Follow dynamic links until an AR containing x is found.**
  - → NOTES:

    # of dynamic links to be followed cannot be determined statically

    Names of variables must be stored in ARs

- **Shallow Access (reference to x)**
  - → **Maintain central table with entry for each variable in program.**

    Current values of variables live in table, not on stack.
  - → **Procedure call:**

    for each local var x save current value of x (in table) on stack.
  - → **Procedure return:**

    Restore saved values of variables to table.
  - → **All variable access refers to table.**