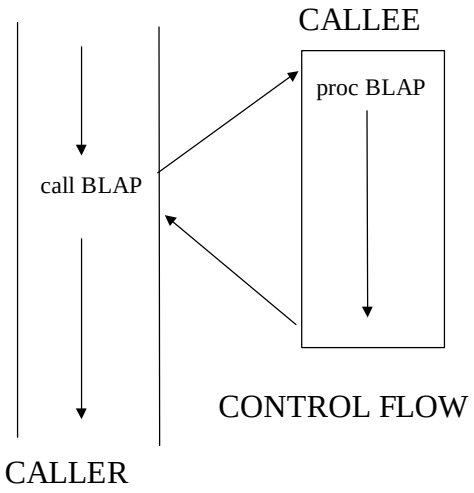
Subprograms

- **Characteristics:**
  - **single entry point**
  - **calling unit suspends**
  - **control returns to caller**



Subprograms

Terminology Problem

in Calling Unit	in Called Unit
actual parameter	dummy parameter
actual parameter	formal parameter
argument	parameter

- **arguments & parameters**
  - **p(5.1, a + b) proc p(x: real, y: int)**
  - **Correspondence-- matching method**
  - **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 arguments and parameters:**

    argument -> parameter "in mode" (calling -> called)

    parameter -> argument "out mode" (called -> calling)

    both "inout mode"
  - **Transfer by copy or by access path (pointer)?**
  - **When are arguments evaluated?**

Methods

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

    => no access to outer environment
  - **evaluation: argument evaluated at time of call**

    e.g., f(2 + 3) ≡ f(5) y := 5

    F(y) ≡ F(5)
  - **Note: protects arguments, *may be inefficient (copy)***
- **Pass-by-result**
  - **mode: *out***
  - **transfer: by copy**
  - **evaluation: address to copy back to evaluated at time of call**
  - **What happens here? What is final value of x on return?**

    p(x,x) Collision: **procedure** P (a, b);

    a := 5; b := 7;

Methods (continued)

- **Pass by value/result**
  - **mode: *inout***
  - **transfer: by copy (in and out)**
  - **evaluation: at time of call**
    - value
    - return address
- **Pass by reference**
  - **mode: *inout***
  - **transfer: by shared access path**
    - addr ( )
  - **evaluation: address of arguments evaluated at time of call**
  - **Note: pass-by-value/result ¹ 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 alias 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)*

* Actually NOT available after the return

Another Method

- **Pass-by-name**
  - ➡ **Symbolic* substitution of actual for each occurrence of formal**
  - ➡ **mode: *inout***
  - ➡ **transfer: ?**
  - ➡ **evaluation: when formal is *demanded* in *body* of procedure - *not at time of call!***

    => eval 0 or more times

- **Advantage: Delays evaluation of arguments**

```
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)
```

* NOT Textual; textual substitution is the method in MACROs

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      ====>  x + (x + 1) =>p2.x + (p1.x + 1)
end;
```

    NOT same x

    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.
```

Pass-by-value will **not** work here

Pass-by-reference **will** work

Pass-by-name works only **sometimes**

```
swap (a[i],a[jj])          swap(i,a[i])

    -- temp := a[i]            -- temp := i;
    -- a[i] := a[jj];          -- i := a[i];
    -- a[jj] := temp;          -- a[i] := temp;

      YES                      NO
```


SWAP: Call-by-reference

```
procedure swap(x,y);
var temp:int;
begin
        temp := x;
        x := y;
        y := temp;
end
-----------------------------------------------
Call:   swap ( i , a[i] )  where  i = 2  and  a[2] = 5

        =====>  swap ( ^i , ^a[2] , ^swap-code )

                temp := ^i          temp  <-  2
                ^i  :=  ^a[2]       i  <-  5
                ^a[2]  :=  temp     a[2]  <- 2

                VALUES APPROPRIATELY SWAPPED
```

SWAP: Call-by-name

```
procedure swap(x,y);
var temp:int;
begin
        temp := x;
        x := y;
        y := temp;
end
-------------------------------------------------
Call:   swap ( i , a[i] )  where  i = 2  and  a[2] = 5

        =====>  swap ( ^i , ^a[2] , ^env , ^swap-code )

        temp := eval ( i , env )                    temp  <-  2
        addr ( i , env )  :=  eval ( a[i] , env )   i  <-  5
        addr ( a[i] , env )  :=  temp               a[5]  <- 2
```

NOTE: VALUE OF a[2] IS **NOT** CHANGED!

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();      Note: p3 is not normally visible to p1
   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").**

DEEP OR SHALLOW BINDING

- **Deep binding refers to static (lexical, textual, compile time) binding of non-local identifiers**
- **Shallow binding refers to dynamic (calling sequence, calling chain, run time) binding**
- **NOTE TO REMEMBER FOR LATER**
  - **when we get to Activation Record Stacks, shallow binding will be seen as binding that is "most recent" in the stack (i.e. following the dynamic link)**
  - **deep binding follows the static links that generally point to deeper activation records in the stack**