Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 20

- More about bindings and scopes
- Implementation of scopes
- Closures

We have seen…

- **Binding**: association name <-> object
- Binding times
- **Object allocation policies** (static, stack, heap)
- **Scope** of a binding: textual region of the program in which the binding is active
- **Static** versus **dynamic** scoping

More about scopes, and passing subroutines as parameters

- Nested blocks and declaration order
- Modules and scopes
- Implementing Scopes
- Aliases and overloading
- Subroutines as parameter or result
- Reference (non-local) environment
- Shallow vs. deep binding
- Closures
- Returning subroutines: unlimited extent
- Object closures

Nested Blocks

C

```
{ int t = a;
  a = b;
  b = t;
}
```

Ada

```
declare t:integer
begin
  t := a;
  a := b;
  b := t;
end;
```

C++
Java
C#

```
{ int a,b;
  ...
  int t;
  t=a;
  a=b;
  b=t;
  ...
}
```

- In several languages local variables are declared in a block or compound statement
  - At the beginning of the block (Pascal, ADA, …)
  - Anywhere (C/C++, Java, …)
- Local variables declared in nested blocks in a single function are all stored in the subroutine frame for that function (most programming languages, e.g. C/C++, Ada, Java)

Declaration order and use of bindings

- Scope of a binding
  1) In the whole block where it is defined
  2) From the declaration to the end of the block
- Use of binding
  a) Only after declaration
  b) In the scope of declaration
- Many languages use **2)-a)**.
- Some combinations produce strange effects: **Pascal** uses **1) – a)**.

```
const N = 10;
...
procedure foo;
const
    M = N;          (* static semantic error! *)
   var
       A : array [1..M] of integer;
       N : real;          (* hiding declaration *)
```

Reported errors: "N used before declaration"
"N is not a constant"

Declarations and definitions

- "Use after declaration" would forbid mutually recursive definitions (procedures, data types)
- The problem is solved distinguishing ***declaration*** and ***definition*** of a name, as in **C**
- **Declaration**: introduces a name
- **Definition**: defines the binding

```
struct manager;          // Declaration only
struct employee {
    struct manager *boss;
    struct employee *next_employee;
    ...
};
struct manager {         // Definition
    struct employee *first_employee;
... };
```

Modules

- Modules are the main feature of a programming language that supports the construction of large applications
  - Support *information hiding* through *encapsulation*: explicit import and export lists
  - Reduce risks of *name conflicts;* support *integrity of data abstraction*
- Teams of programmers can work on separate modules in a project
- No language support for modules in C and Pascal
  - Modula-2 ***modules***, Ada ***packages***, C++ ***namespaces***
  - Java ***packages***

Module Scope

- Scoping: modules encapsulate variables, data types, and subroutines in a package
  - Objects inside are visible to each other
  - Objects inside are not visible outside unless *exported*
  - Objects outside are not visible inside unless *imported* [closed vs. open modules]
- A module interface specifies exported variables, data types and subroutines
- The module implementation is compiled separately and implementation details are hidden from the user of the module

Module Types, towards Classes

- Modules as abstraction mechanism: collection of data with operations defined on them (sort of *abstract data type)*
- Various mechanism to get module *instances*:
  - Modules as manager: instance as additional arguments to subroutines (**Modula-2**)
  - Modules as types (**Simula**, **ML**)
- Object-Oriented: Modules (classes) + inheritance
- Many OO languages support a notion of Module (packages) independent from classes

Implementing Scopes

- The language implementation must keep trace of current bindings with suitable data structures:
  - Static scoping: *symbol table* at compile time
  - Dynamic scoping: *association lists* or *central reference table* at runtime
- **Symbol table** main operations: *insert*, *lookup*
  - because of nested scopes, must handle several bindings for the same name
  - new scopes (not LIFO) are created for records and classes
  - the symbol table might be needed at runtime for symbolic debugging
  - bindings are never deleted
  - Other operations: *enter_scope, leave_scope*

LeBlanc & Cook **Symbol Table**

- Each scope has a serial number
  - Predefined names: 0 (*pervasive*)
  - Global names: 1, and so on
- Names are inserted in a ***hash table***, indexed by the name
  - Entries contain symbol name, category, scope number, (pointer to) type, …
- **Scope Stack**: contains numbers of the currently visible scopes
  - Entries contain scope number and additional info (closed?, …). They are pushed and popped by the semantic analyzer when entering/ leaving a scope
- Look-up of a *name*: scan the entries for *name* in the hash table, and look at the scope number *n*
  - If n <> 0 (*not pervasive*), scan the Scope Stack to check if scope *n* is visible
  - Stops at first *closed* scope. Imported/Export entries are pointer.

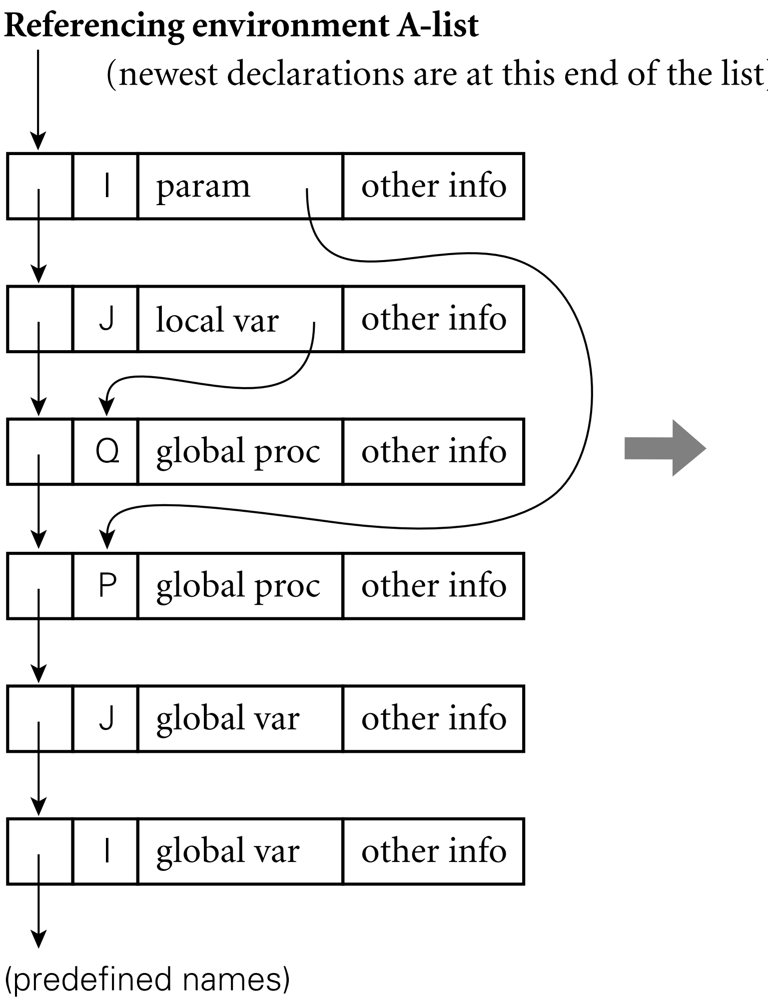
LeBlanc & Cook lookup function

```
procedure lookup(name)
    pervasive := best := null
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e       -- closer instance
                        exit inner loop
                    elsif best != null and then s.scope = best.scope
                        exit inner loop -- won't find better
                    if s.closed
                        exit inner loop -- can't see farther
    if best != null
        while best is an import or export entry
            best := best.real entry
        return best
    elsif pervasive != null
        return pervasive
    else
        return null -- name not found
```

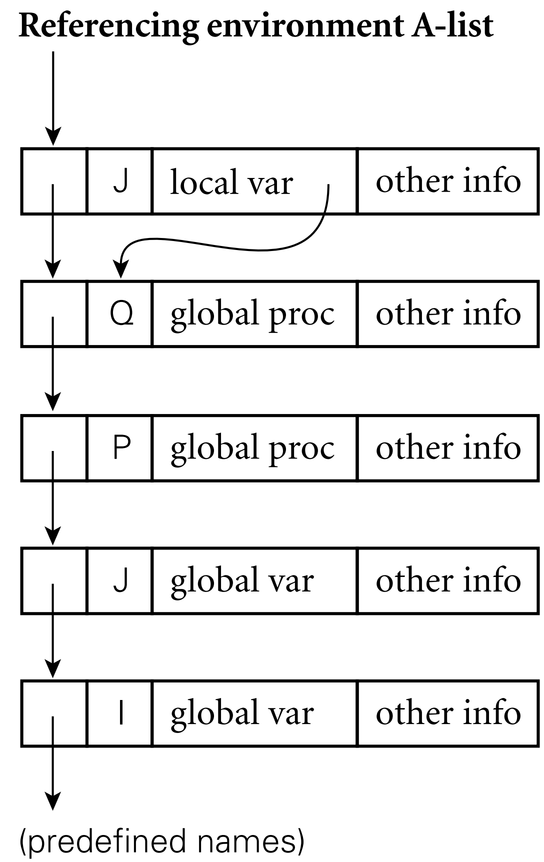
Association Lists (A-lists)

- List of bindings maintained at runtime with **dynamic scoping**
- Bindings are pushed on *enter_scope* and popped on *exit_scope*
- Look up: walks down the stack till the first entry for the given name
- Entries in the list include information about types
- Used in many implementations of LISP: sometimes the A-list is accessible from the program
- Look up is inefficient

A-lists: an example




```
I, J : integer

procedure P (I : integer)
        . . .

procedure Q
        J : integer
        . . .
        P (J)
        . . .

-- main program
. . .
Q
```



A-list after entering P in the exection of Q

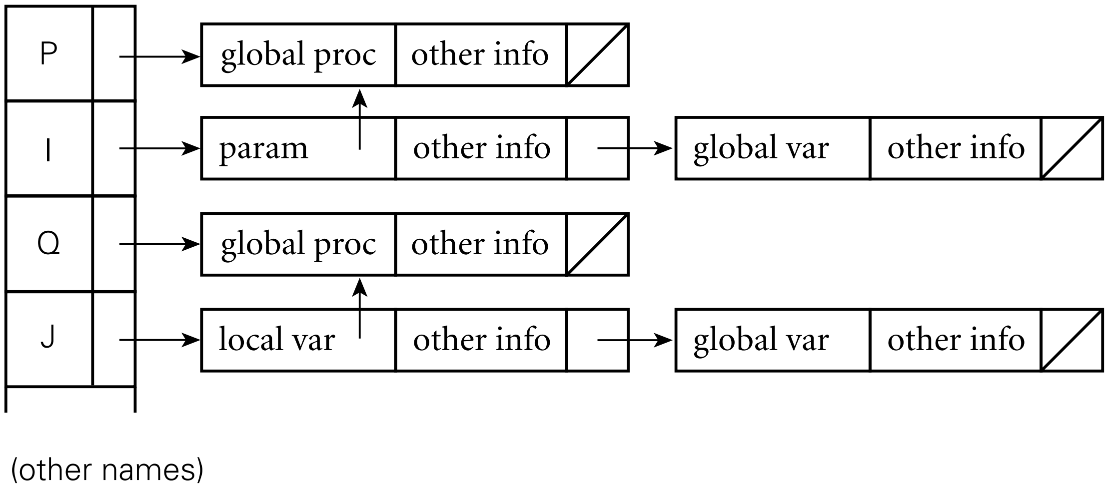
A-list after exiting P

Central reference tables

- Similar to LeBlanc&Cook hash table, but stack of scopes not needed
- Each name has a slot with a stack of entries: the current one on the top
- On *enter_scope* the new bindings are pushed
- On *exit_scope* the scope bindings are popped
- More housekeeping work necessary, but faster access

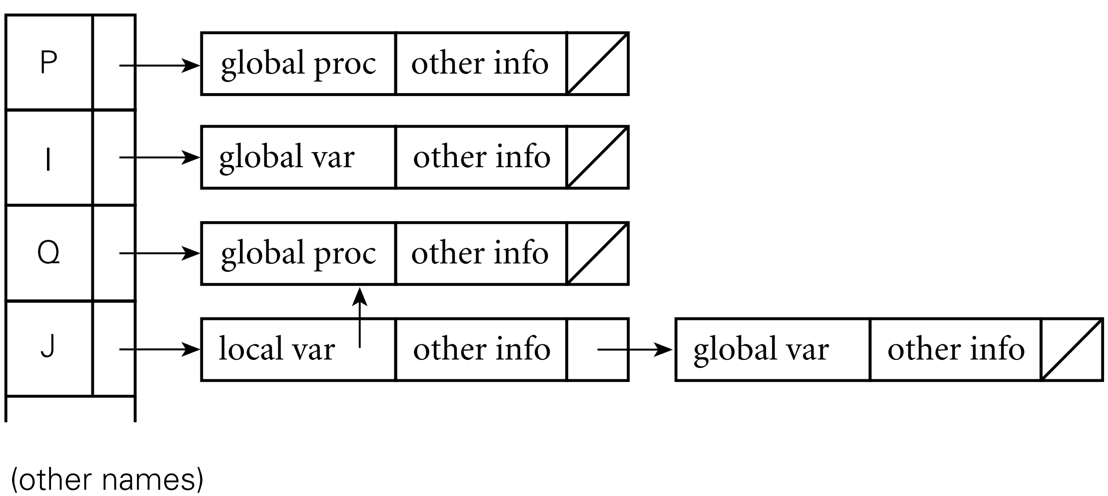
Central reference table

(each table entry points to the newest declaration of the given name)




```
I, J : integer

procedure P (I : integer)
        . . .

procedure Q
        J : integer
        . . .
        P (J)
        . . .

-- main program
. . .
Q
```

Central reference table



Not 1-to-1 bindings: **Aliases**

Aliases: two or more names denote the same object

Arise in several situations:

- Pointer-based data structures

Java:
```
Node n = new Node("hello", null);
Node n1 = n;
```

- **common** blocks (**Fortran**), variant records/unions (**Pacal**, **C**)

- Passing (by name or by reference) variables accessed non-locally

```
double sum, sum_of_squares;
...
void accumulate(double& x)
{
    sum += x;
    sum_of_squares += x * x;
}
...
accumulate(sum);
```

Problems with **aliases**

- Make programs more confusing
- May disallow some compiler's optimizations

```
int a, b, *p, *q;
    ...
a = *p;  /* read from the variable referred to by p*/

*q = 3; /* assign to the variable referred to by q */

b = *p; /* read from the variable referred to by p */
```

Not 1-to-1 bindings: **Overloading**

- A name that can refer to more than one object is said to be *overloaded*
  - Example: + (addition) is used for integer and floating-point addition in most programming languages
- Overloading is typically resolved at compile time
- Semantic rules of a programming language require that the context of an overloaded name should contain sufficient information to deduce the intended binding
- Semantic analyzer of compiler uses type checking to resolve bindings
- Ada, C++,Java, … function overloading enables programmer to define alternative implementations depending on argument types (*signature*)
- Ada, C++, and Fortran 90 allow built-in operators to be overloaded with user-defined functions
  - enhances expressiveness
  - may mislead programmers that are unfamiliar with the code

First, Second, and Third-Class Subroutines

- *First-class object*: an object entity that can be passed as a parameter, returned from a subroutine, and assigned to a variable
  - Primitive types such as integers in most programming languages
- *Second-class object*: an object that can be passed as a parameter, but not returned from a subroutine or assigned to a variable
  - Fixed-size arrays in C/C++
- *Third-class object*: an object that cannot be passed as a parameter, cannot be returned from a subroutine, and cannot be assigned to a variable
  - Labels of goto-statements and subroutines in Ada 83

- Functions in Lisp, ML, and Haskell are unrestricted first-class objects
- With certain restrictions, subroutines are first-class objects in Modula-2 and 3, Ada 95, (C and C++ use function pointers)

Scoping issues for first/second class subroutines

- Critical aspects of scoping when
  - Subroutines are passed as parameters
  - Subroutines are returned as result of a function
- Resolving names declared locally or globally is obvious
  - Global objects are allocated statically (or on the stack, in a fixed position)
    - Their addresses are known at compile time
  - Local objects are allocated in the activation record of the subroutine
    - Their addresses are computed as *base of activation record + statically known offset*

“Referencing” (“Non-local”) Environments

- If a subroutine is passed as an argument to another subroutine, when are the static/dynamic scoping rules applied?
  1) When the reference to the subroutine is first created (i.e. when it is passed as an argument)
  2) Or when the argument subroutine is finally called
- That is, what is the *referencing environment* of a subroutine passed as an argument?
  - Eventually the subroutine passed as an argument is called and may access non-local variables which by definition are in the referencing environment of usable bindings
- The choice is fundamental in languages with dynamic scope: **deep binding (1)** vs **shallow binding (2)**
- The choice is limited in languages with static scope

Effect of Deep Binding in Dynamically-Scoped Languages

Program execution:

```
main(p)
  bound:integer
  bound := 35
  show(p,older)
    bound:integer
    bound := 20
    older(p)
      return p.age>bound
    if return value is true
      write(p)
```

Deep binding

Program prints persons older than 35

- The following program demonstrates the difference between deep and shallow binding:

```
function older(p:person):boolean
  return p.age > bound
procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)
procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Effect of Shallow Binding in Dynamically-Scoped Languages

- The following program demonstrates the difference between deep and shallow binding:

```
function older(p:person):boolean
  return p.age > bound
procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)
procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Program execution:

```
main(p)
   bound:integer
   bound := 35
   show(p,older)
      bound:integer
      bound := 20
      older(p)
         return p.age>bound
      if return value is true
         write(p)
```

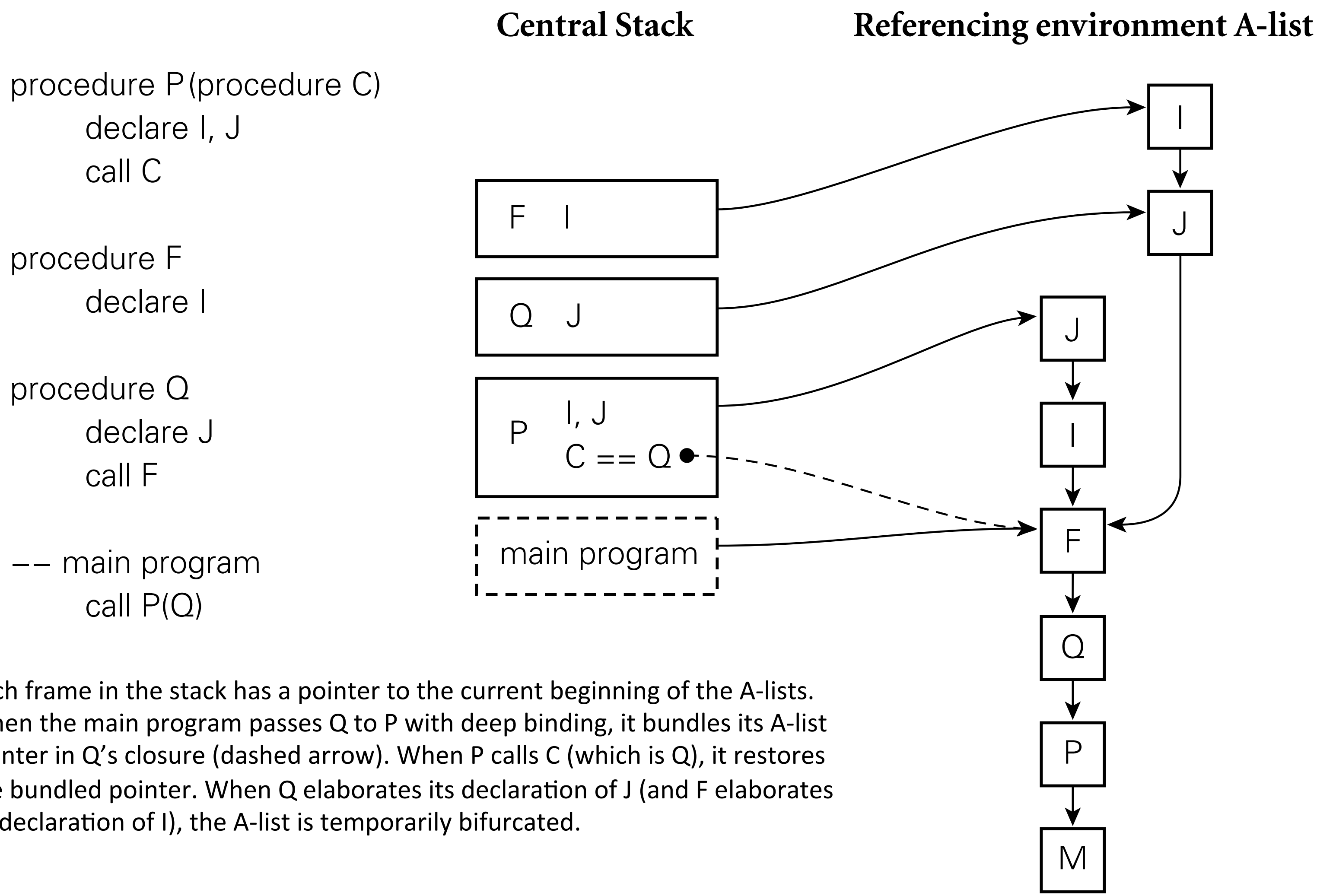
Shallow binding

Program prints persons older than 20

Implementing Deep Bindings with Subroutine Closures

- Implementation of *shallow binding* obvious: look for the last activated binding for the name in the stack
- For *deep binding,* the referencing environment is bundled with the subroutine as a *closure* and passed as an argument
- A subroutine closure contains
  - A pointer to the subroutine code
  - The current set of name-to-object bindings
- Possible implementations:
  - With Central Reference Tables, the whole current set of bindings may have to be copied
  - With A-lists, the head of the list is copied

Clusures in Dynamic Scoping implemented with A-lists



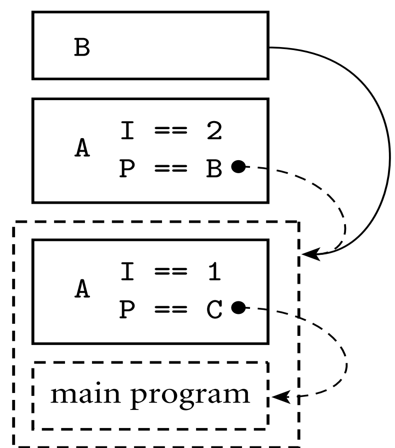
Each frame in the stack has a pointer to the current beginning of the A-lists. When the main program passes Q to P with deep binding, it bundles its A-list pointer in Q's closure (dashed arrow). When P calls C (which is Q), it restores the bundled pointer. When Q elaborates its declaration of J (and F elaborates its declaration of I), the A-list is temporarily bifurcated.

Deep/Shallow binding with **static** scoping

- Not obvious that it makes a difference. Recall:
- **Deep binding**: the scoping rule is applied when the subroutine is passed as an argument
- **Shallow binding**: the scoping rule is applied when the argument subroutine is called
- In both cases non-local references are resolved looking at the static structure of the program, so refer to the same binding declaration
- **But in a recursive function the same declaration can be executed several times: the two binding policies may produce different results**
- No language uses shallow binding with static scope
- Implementation of deep binding easy: just keep the static pointer of the subroutine in the moment it is passed as parameter, and use it when it is called

Deep binding with **static scoping**: an example in Pascal

```
program binding_example(input, output);

procedure A(I : integer; procedure P);

    procedure B;
    begin
        writeln(I);
    end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2, B);
end;

procedure C; begin end;

begin (* main *)
    A(1, C);
end.
```

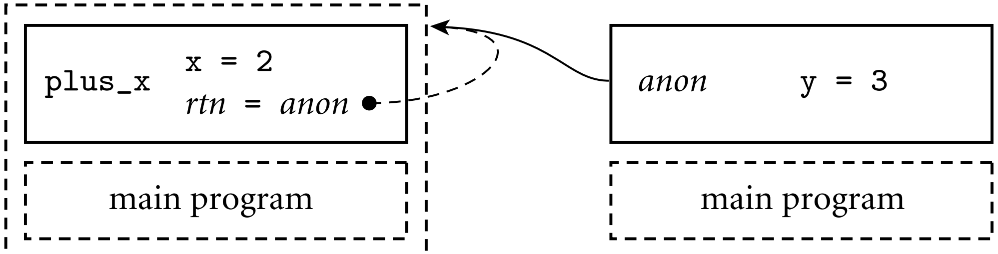


When B is called via formal parameter P, two instances of I exist. Because the closure for P was created in the initial invocation of A, B's static link (solid arrow) points to the frame of that earlier invocation. B uses that invocation's instance of I in its writeln statement, and the output is a 1. With **shallow binding** it would print 2.

Returning subroutines

- In languages with first-class subroutines, a function **f** may declare a subroutine **g**, returning it as result
- Subroutine **g** may have non-local references to local objects of **f**. Therefore:
  - **g** has to be returned as a *closure*
  - the activation record of **f** cannot be deallocated

```
(define plus-x (lambda (x)
    (lambda (y) (+ x y))))
...
(let ((f (plus-x 2)))
    (f 3))          ; returns 5
```



First-Class Subroutine Implementations

- In functional languages, local objects have *unlimited extent*: their lifetime continue indefinitely
  - Local objects are allocated on the heap
  - *Garbage collection* will eventually remove unused objects
- In imperative languages, local objects have *limited extent* with stack allocation
- To avoid the problem of dangling references, alternative mechanisms are used:
  - C, C++, and Java: no nested subroutine scopes
  - Modula-2: only outermost routines are first-class
  - Ada 95 "containment rule": can return an inner subroutine under certain conditions

Object closures

- Closures (i.e. subroutine + non-local enviroment) are needed only when subroutines can be nested
- Object-oriented languages without nested subroutines can use objects to implement a form of closure
  - a method plays the role of the subroutine
  - instance variables provide the non-local environment
- Objects playing the role of a function + non-local enviroment are called **object closures** or **function objects**
- Ad-hoc syntax in some languages
  - In C++ an object of a class that overrides **operator()** can be called with functional syntax

Object closures in Java and C++

```
interface IntFunc {                               //Java
       public int call(int i);
   }
   class PlusX implements IntFunc {
       final int x;
       PlusX(int n) { x = n; }
       public int call(int i) { return i + x; }
   }
   ...
   IntFunc f = new PlusX(2);
   System.out.println(f.call(3));      // prints 5
```

```
class int_func {                              // C++
   public:
       virtual int operator()(int i) = 0;
   };
   class plus_x : public int_func {
       const int x;
   public:
       plus_x(int n) : x(n) { }
       virtual int operator()(int i) { return i + x; }
   };
   ...
   plus_x f(2);
   cout << f(3) << "\n";                      // prints 5
```