



Names, Scopes and Bindings

Chapter 3

Name, Scope, and Binding

- *Ease of programming* – main driving force behind the design of modern languages
- Core issues in language design:
 - names – abstraction
 - control flow
 - types, composite types
 - subroutines – control abstraction
 - classes – data abstraction
- High level programming – more abstract
 - Farther from hardware
- *Abstraction* – complexity becomes manageable
 - This is true in general

Name, Scope, and Binding

- *Name*: a character string representing something else
 - Abstraction
 - Easy for humans to understand
 - Much better than addresses
- *Binding*: association of two things
 - Example: between a name and the thing it names
- *Scope* of a binding: the part of the program (textually) in which the binding is active
- *Binding Time*: the point at which a binding is created

Binding

- Static vs. Dynamic
 - Static: bound before run time
 - Dynamic: bound at run time
- Trade-off:
 - *Early* binding times: greater *efficiency*
 - *Late* binding times: greater *flexibility*
- Compiled vs. Interpreted languages
 - *Compiled* languages tend to have *early* binding times
 - *Interpreted* languages tend to have *late* binding times

Language	Binding Time	Advantage
Compiled	Early (static)	Efficiency
Interpreted	Late (dynamic)	Flexibility

Lifetime and Storage Management

- *Lifetime* of name-to-binding:
 - from creation to destruction
 - Object's lifetime \geq binding's lifetime
 - Example: C++ variable passed by reference (`&`)
 - Object's lifetime < binding's lifetime – *dangling reference*
 - Example: C++ object
 - created with `new`
 - passed by reference to subroutine with `&`
 - deallocated with `delete`
- *Scope of a binding*:
 - the textual region of the program in which the binding is *active*

Lifetime and Storage Management

- *Storage Allocation* mechanisms:

- Static

- absolute address, retained throughout the program

- Stack

- last-in, first-out order; for subroutines calls and returns

- Heap

- allocated and deallocated at arbitrary times

first
second
third

Lifetime and Storage Management

- *Static allocation:*
 - global variables
 - code instructions
 - explicit constants (including strings, sets, etc.)
 - `A = B / 14.7`
 - `printf("hello, world\n")`
 - small constants may be stored in the instructions
 - C++ **static** variables (or Algol own)
- Statically allocated objects that do not change value are allocated in read-only memory
 - constants, instructions

Lifetime and Storage Management

static allocation —> global

stack-based allocation —> local

■ *Stack-based allocation:*

- parameters, local variables, temporaries
- allocate space for recursive routines
- reuse space

■ Frame (activation record) for each subroutine call:

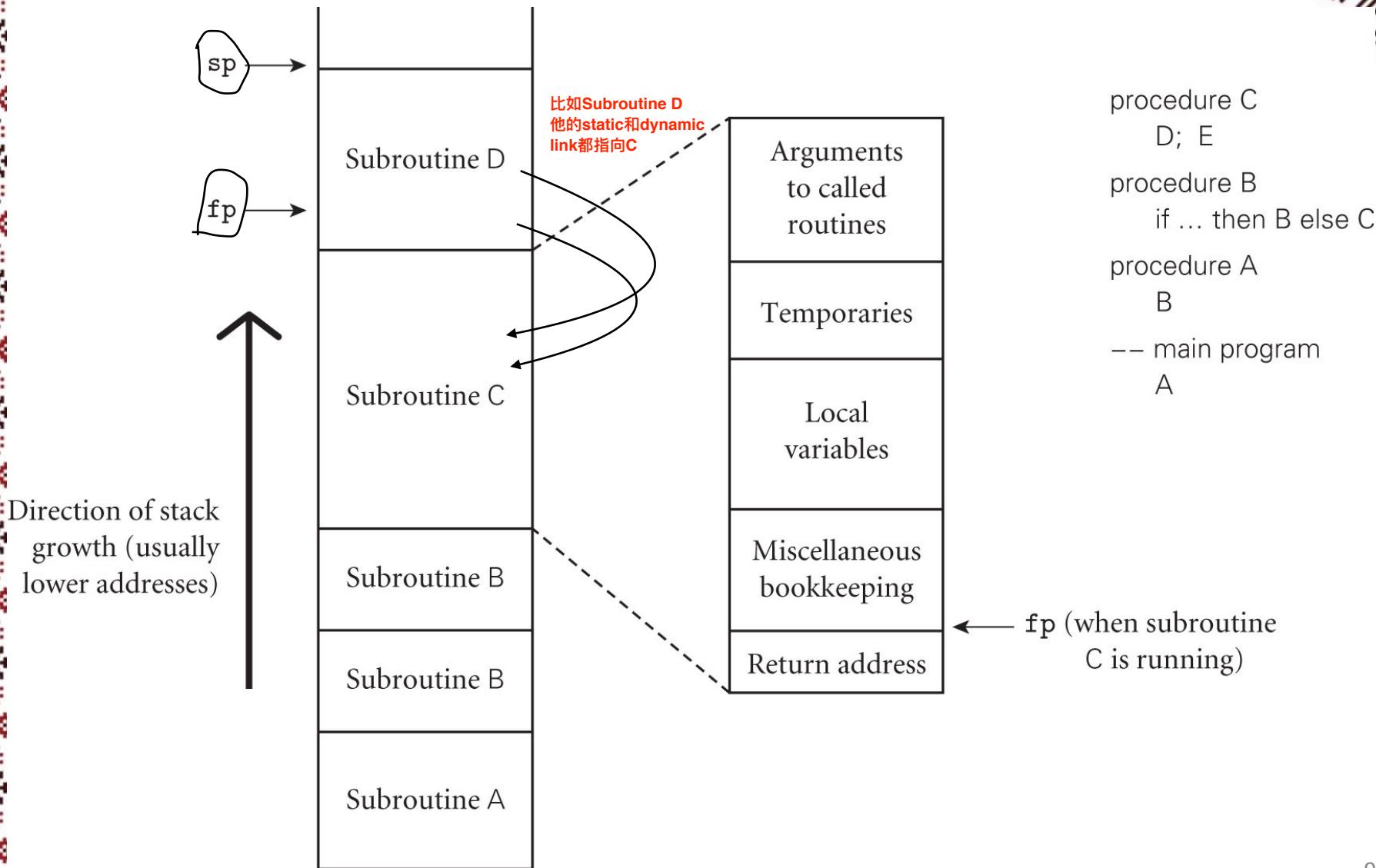
- position in stack: *frame pointer*
- arguments and returns
- local variables, temporaries:
 - *fixed offset* from the frame pointer at compile time
- return address
- *dynamic link*: reference to (stack frame of) caller
- *static link*: reference to (stack frame of) routine inside which it was declared

static link意思是是在哪这个function被declare的

static link用来寻找global variable的

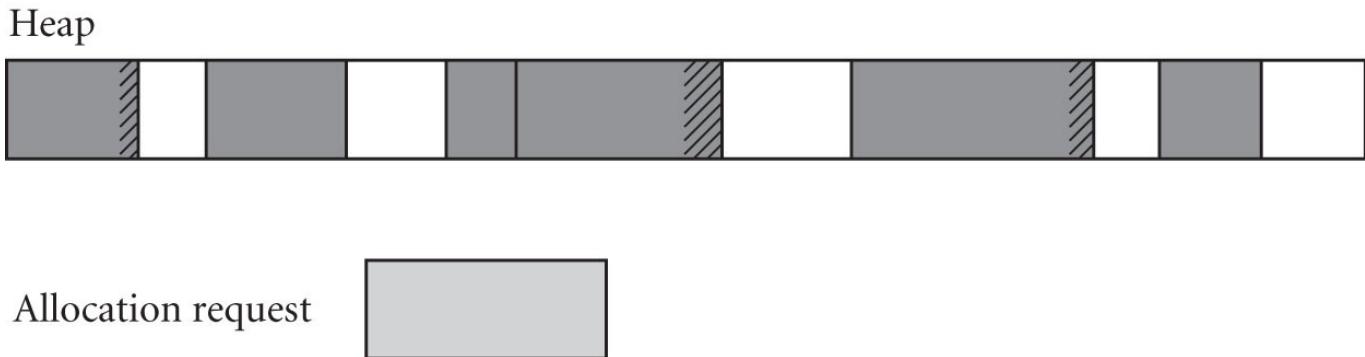
dynamic link to
return to previous
function
who called it

Lifetime and Storage Management



Lifetime and Storage Management

- *Heap allocation*
- (different from “heap” data structure for priority queues)
- dynamic allocation: lists, sets, strings (size can change)
- single linked list of free blocks
- fragmentation: internal, external



Lifetime and Storage Management

- Heap allocation
- allocation algorithms
 - first fit, best fit – $O(n)$ time
 - pool allocation – $O(1)$ time
 - separate free list of blocks for different sizes
 - *buddy system*: blocks of size 2^k
 - *Fibonacci heap*: blocks of size Fibonacci numbers
- defragmentation

Lifetime and Storage Management

- Heap maintenance
- Explicit deallocation
 - C, C++
 - simple to implement
 - efficient
 - object deallocated too soon – *dangling reference*
 - object not deallocated at the end of lifetime – *memory leak*
 - deallocation errors are very difficult to find
- Implicit deallocation: *garbage collection*
 - functional, scripting languages
 - C#, Java, Python
 - avoid memory leaks (difficult to find otherwise)
 - recent algorithms more efficient
 - the trend is towards automatic collection

Scope Rules

- *Scope of a binding:*
 - textual region of the program in which binding is active
- Subroutine entry – usually creates a new scope:
 - create bindings for new local variables
 - deactivate bindings for redeclared global variables
 - make references to variables
- Subroutine exit:
 - destroy bindings for local variables
 - reactivate bindings for deactivated global variables
- *Scope:* maximal program section in which no bindings change
 - *block:* module, class, subroutine
 - C: { ... }
 - *Elaboration time:* when control first enters a scope

Scope Rules

- *Referencing environment*
 - the set of active bindings; determined by:
 - *Scope rules* (static or dynamic)
 - *Binding rules* (deep or shallow)
- *Static Scoping (Lexical Scoping)*
 - almost all languages employ static scoping
 - determined by examining the text of the program
 - at compile time
 - *closest nested rule*
 - identifiers known in the scope where they are declared and in each enclosed scope, unless re-declared
 - examine local scope and statically enclosing scopes until a binding is found

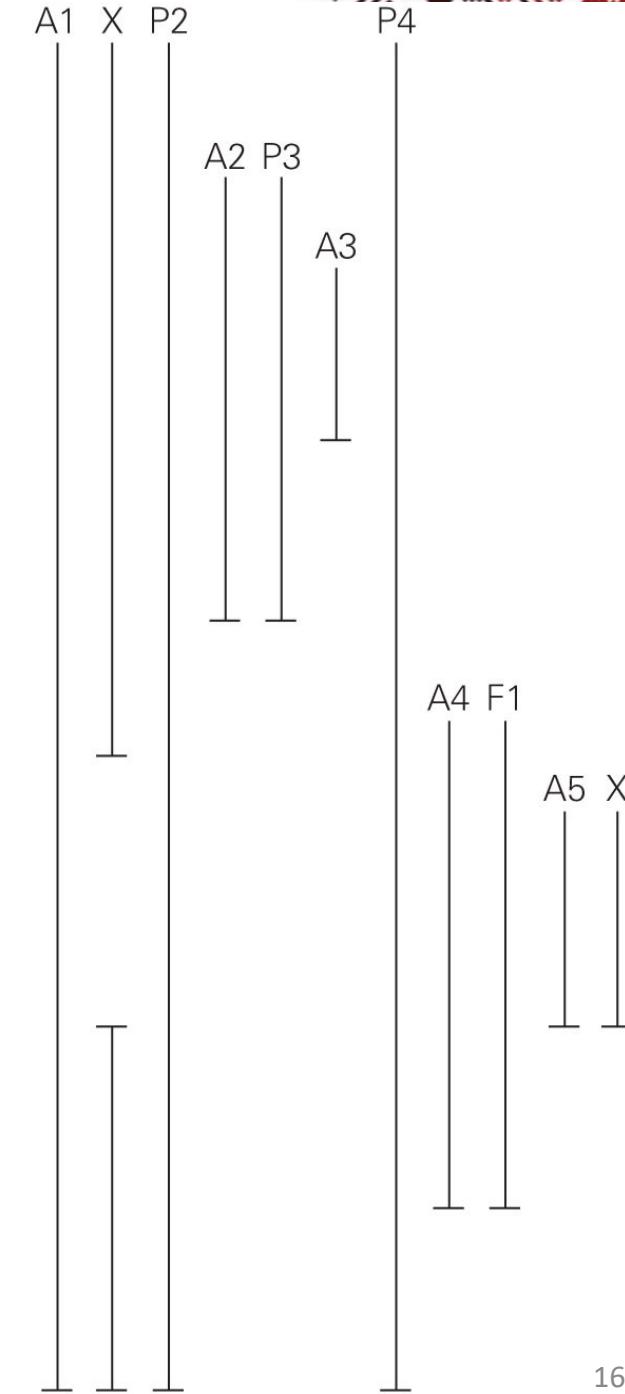
Scope Rules

- Subroutines
 - bindings created are destroyed at subroutine exit
 - exception: `static` (C), `own` (Algol)
 - nested subroutines: *closest nested scope*
 - Python, Scheme, Ada, Common Lisp
 - not in: C, C++, Java
 - access to non-locals: *scope resolution operator*
 - C++ (global): `::x`
 - Ada: `MyProc.x`
 - built-in objects
 - outermost scope
 - outside global

Scope Rules

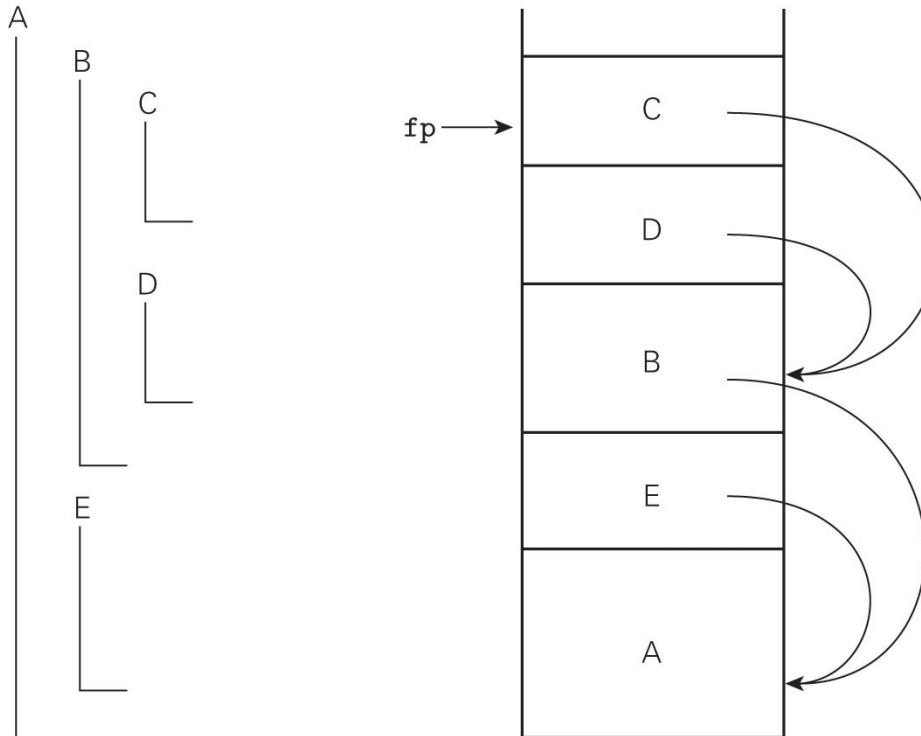
- Example:
Nested subroutines

```
procedure P1(A1)
    var X          -- local to P1
    ...
    procedure P2(A2)
        ...
        procedure P3(A3)
            ...
            begin
                ...
                end          -- body of P3
            ...
            begin
                ...
                end          -- body of P2
            ...
            procedure P4(A4)
                ...
                function F1(A5)
                    var X  -- local to F1
                    ...
                    begin
                        ...
                        end          -- body of F1
                    ...
                    begin
                        ...
                        end          -- body of P4
                    ...
                    begin
                        ...
                        end          -- body of P1
                    ...
                end
            end
        end
    end
end
```



Scope Rules

- Access to non-locals: *static links*
 - each frame points to the frame of the routine inside which it was declared
 - access a variable in a scope k levels out by following k static links and then using the known offset within the frame



Scope Rules

- Declaration order
 - object x declared inside block B
 - the scope of x may be:
 - the entire block B or
 - only the part of B after x 's declaration

Scope Rules

- Declaration order

- Example: C++

```
int n = 1;  
void f(void){  
    int m = n; // global n  
    int n = 2; // local n  
}
```

- Example: Python – no declarations

```
n = 1  
def f():  
    m = n # error  
                # add "global n" to use the global n  
    n = 2
```

Scope Rules

- Declaration order

- Example: Scheme

let is binding 这个是binding, let A be 1 within body

(let ((A 1))

在括号里的是 body B is not in the scope of (A 2), so it is 1

(let ((A 2)
 (B A))

 ; return 1

return B

(let ((A 1))

(letrec ((A 2)

 (B A))

 ; return 2

when you use
Let recursive
B is in scope of
(A 2), so B is 2

Scope Rules

- *Dynamic Scoping*
- binding depends on flow at run time
 - use the most recent, active binding made at run time
- Easy to implement – just a stack with names
- Harder to understand
 - not used any more
 - why learn? – history

Scope Rules

- Example: Dynamic Scoping

```
n: integer          – global
procedure first()
    n := 1
procedure second()
    n : integer – local
    first()
n := 2
if read_integer() > 0
    second()
else
    first()
write_integer(n)
```

static scoping就是N都是指global的N

- Static scoping: prints 1

Dynamic scoping是指上一个N是什么 把这个理解成stack 每次初始化一个N都会加在最上面 而上一个N是Top Most N

- Dynamic scoping: prints 2 for positive input, 1 for negative

Scope Rules

- Example: Dynamic scoping problem
 - `scaled_score` uses the wrong `max_score`

```
max_score : integer      — maximum possible score
function scaled_score(raw_score : integer) : real
    return raw_score / max_score * 100
...
procedure foo( )
    max_score : real := 0 -- highest % seen so far
    ...
    foreach student in class
        student.percent := scaled_score(student.points)
        if student.percent > max_score
            max_score := student.percent
```

Binding of Referencing Environments

- Referencing environment: the set of active bindings
 - static: lexical nesting
 - dynamic: order of declarations at run time
- Reference to subroutine: when are the scope rules applied?
 - Shallow binding: when routine is called
 - default in dynamic scoping
 - Deep binding: when reference is created
 - default in static scoping
- Example (next slides)

Binding of Referencing Environments

```
type person = record
```

```
    ...
```

```
        age : integer
```

```
    ...
```

```
threshold : integer
```

```
people : database
```

```
function older_than_threshold(p : person) : boolean
```

```
    return p.age ≥ threshold
```

```
procedure print_person(p : person)
```

```
    -- Call appropriate I/O routines to print record on standard output.
```

```
    -- Make use of nonlocal variable line_length to format data in columns.
```

```
    ...
```

- `print_routine`
 - shallow binding
 - to pick `line_length`
- `older_than_threshold`
 - deep binding 因为用threshold这个global variable所以是 static scoping
所以这是deep binding
 - otherwise, if `print_selected_records` has a variable `threshold`, it will hide the one in the main program

Binding of Referencing Environments

```
procedure print_selected_records(db : database;
                                predicate, print_routine : procedure)
                                line_length : integer

                                if device_type(stdout) = terminal
Use local variable  
所以是shallow binding line_length := 80
                                else      -- Standard output is a file or printer.
                                line_length := 132
                                foreach record r in db
                                    -- Iterating over these may actually be
                                    -- a lot more complicated than a 'for' loop.
                                    if predicate(r)
                                        print_routine(r)

-- main program
...
threshold := 35
print_selected_records(people, older_than_threshold, print_person)
```

Binding of Referencing Environments

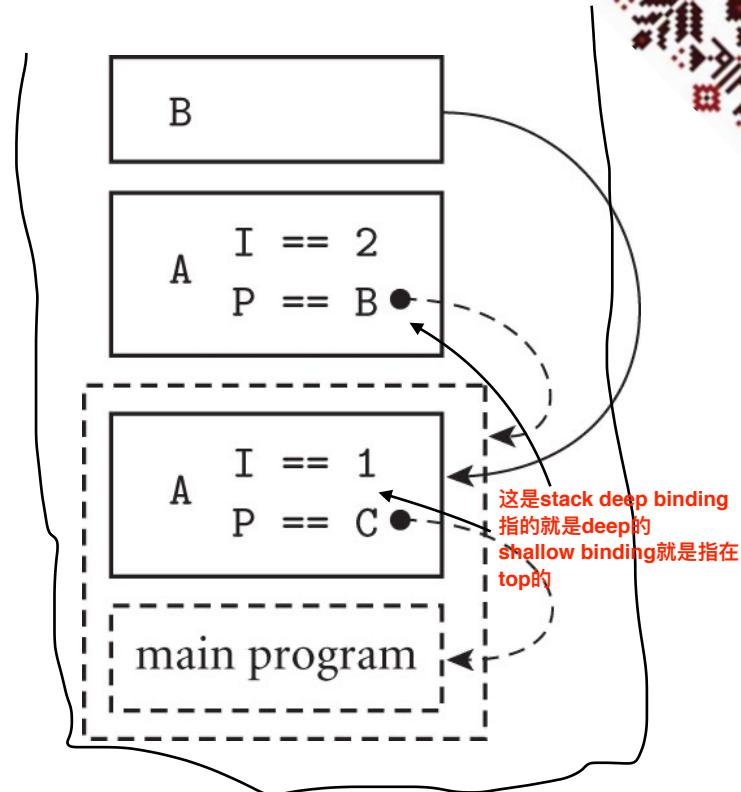
- Deep binding implementation: *subroutine closure*
 - explicit representation of a referencing environment
(the one in which the subroutine would execute if called now)
 - reference to subroutine
- Why binding time matters with static scoping?
 - the running program may have two instances of an object
 - only for objects that are neither local nor global
 - Examples when it does not matter:
 - subroutines cannot be nested: C
 - only outermost subroutines can be passed as parameters:
Modula-2
 - subroutines cannot be passed as parameters: PL/I, Ada 83

Binding of Referencing Environments

- Example: Deep binding in Python

```
def A(I, P):
    def B():
        print(I)
    # body of A:
    if I > 1:
        P()
    else:
        A(2, B)
def C():
    pass # do nothing
A(1, C) # main program; output 1
```

因为I和P不存在global define过的东西，所以当main program call的时候 reference就bind I和1, C和P在reference time binding就是deep binding。
同样道理 local variable之所以是 shallow binding 是因为local variable 被bind 的时机是在 subroutine被call了之后，所以他是 shallow binding



- referencing environment captured in closures: dashed boxes, arrows
- when **B** is called via **P**, two instances of **I** exist
- the closure for **P** was created in the initial invocation of **A**
- **B**'s static link (solid arrow) points to the frame of the earlier invocation

Binding of Referencing Environments

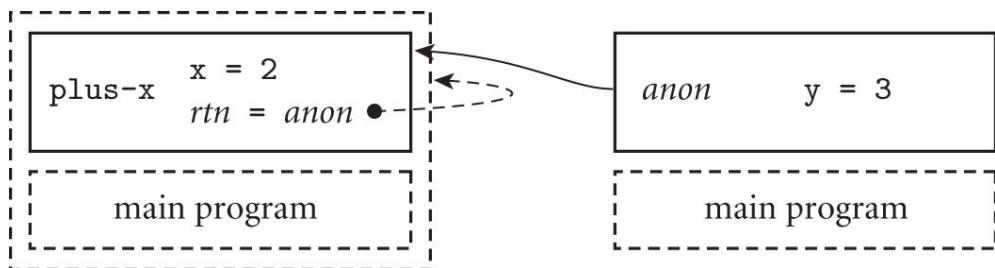
- *First-class* values
 - can be passed as a parameter
 - can be returned from subroutine
 - can be assigned into a variable
- *Second-class* values
 - can only be passed as a parameter
- *Third-class*: none
 - Other authors may have different definitions: no second-class; first-class may require anonymous function definition (lambda expressions)
- Subroutines:
 - first-class: functional and scripting languages, C#
 - C, C++: pointers to functions are first-class
 - second-class: most imperative languages
 - third class: Ada83

Binding of Referencing Environments

- First-class subroutines: additional complexity
 - a reference to a subroutine may outlive the execution of the scope in which that subroutine was declared
 - Example: Scheme

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

这行function被
return了



- `plus-x` returns an unnamed function (3rd line), which uses the parameter `x` of `plus-x`
- when `f` is called in 5th line, its referencing environment includes the `x` in `plus-x`, even though `plus-x` has already returned
- `x` must be still available – *unlimited extent* – allocate on heap (C#)

Binding of Referencing Environments

- *Lambda expressions*

- come from lambda calculus: anonymous functions
- Example: Scheme

```
((lambda (i j) (> i j) i j) 5 8) ;return 8
```

- Example: C#: **delegate** or =>

```
(int i, int j) => i > j ? i : j
```

Binding of Referencing Environments

- First-class subroutines
 - are increasingly popular; made their way into C++, Java
 - Problem: C++, Java do not support unlimited extent
- Example: C++

```
for_each(v.begin(), v.end(),
    [](int e){ if (e < 50) cout << e << " "; }
);
```

Binding of Referencing Environments

- Lambda functions in Python

- Example

```
ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
```

```
# Lexicographic sort
```

```
print(sorted(ids))
=> ['id1', 'id100', 'id2', 'id22', 'id3', 'id30']
```

```
# Integer sort
```

```
sorted_ids = sorted(ids, key=lambda x: int(x[2:]))
print(sorted_ids)
=> ['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
```

Binding of Referencing Environments

- Lambda functions in Python
 - Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)  
print(mydoubler(11))  
=> 22
```

```
mytrippler = myfunc(3)  
print(mytrippler(11))  
=> 33
```