Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 13

- Scoping rules and their implementation

Summary

- Scope rules
- Static versus dynamic scoping
- Modules
- Implementation of scope
  - LeBlanc & Cook symbol tables
  - A-lists
  - Central Reference Tables

Scope of a binding

- The **scope of a binding** is the textual region of a program in which a name-to-object binding is active
- **“Scope”**: textual region of maximal size where bindings are not destroyed
  - Module, class, subroutine, block, record/object
- **Statically scoped language**: the scope of bindings is determined at compile time
  - Used by almost all but a few programming languages
  - More intuitive to user compared to dynamic scoping
- **Dynamically scoped language**: the scope of bindings is determined at run time
  - Used e.g. in Lisp (early versions), APL, Snobol, and Perl (selectively)

Static (lexical) scoping

- The bindings between names and objects can be determined by examination of the program text
- **Scope rules** of the language define the scope of bindings
  - Early Basic: all variables are global and visible everywhere
  - Fortran 77:
    - scope of local variables limited to the subroutine (unless "save"-ed, like "static" in C);
    - scope of global variable is the whole program text unless hidden
  - Algol 60, Pascal, Ada, … : allow ***nested subroutine definitions***
  - Java, … : allow ***nested classes***
    - Adopt the **closest nested scope rule**

Closest Nested Scope Rule

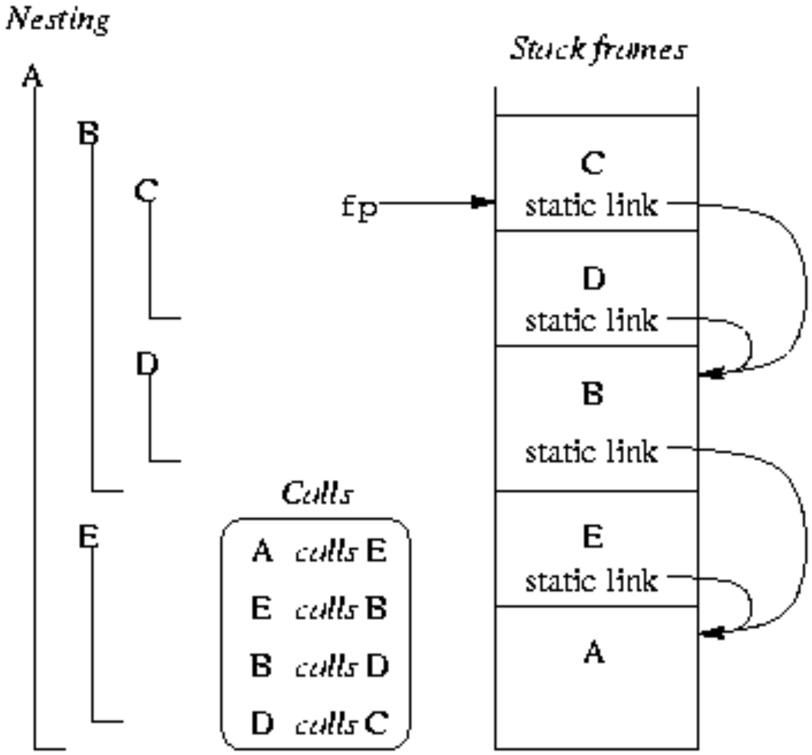
```
procedure P1(A1:T1)
var X:real;
...
  procedure P2(A2:T2);
  ...
    procedure P3(A3:T3);
    ...
    begin
    (* body of P3: P3,A3,P2,A2,X of P1,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P2: P3,P2,A2,X of P1,P1,A1 are visible *)
  end;
  procedure P4(A4:T4);
  ...
    function F1(A5:T5):T6;
    var X:integer;
    ...
    begin
    (* body of F1: X of F1,F1,A5,P4,A4,P2,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P4: F1,P4,A4,P2,X of P1,P1,A1 are visible *)
  end;
...
begin
(* body of P1: X of P1,P1,A1,P2,P4 are visible *)
end
```

- To find the object referenced by a given name:
  - Look for a declaration in the current innermost scope
  - If there is none, look for a declaration in the immediately surrounding scope, etc.
- Built-ins or predefined objects as defined in outermost scope, external to the "global" one
  - I/O routines, mathematical functions

Static Scope Implementation with Static Links

- Access to **global variable**: compiled using constant address
- Access to **local variable**: compiled using *frame pointer* (stored in a register) and statically known offset
- Access to **nonlocal variable?**
- Scope rules are designed so that we can only refer to variables that are alive: the variable must have been stored in the activation record of a subroutine
- If a variable is not in the local scope, we are sure there is an activation record for the surrounding scope somewhere below on the stack:
  - The current subroutine can only be called when it was visible
  - The current subroutine is visible only when the surrounding scope is active
- Each frame on the stack contains a *static link* pointing to the frame of the **static parent**

Example Static Links



- Subroutines C and D are declared nested in B
  - B is static parent of C and D
- B and E are nested in A
  - A is static parent of B and E
- The **fp** points to the frame at the top of the stack to access locals
- The static link in the frame points to the frame of the static parent

A Typical Calling Sequence

- The caller
  - Saves (in the dedicated area in its activation record) any **registers** whose values will be needed after the call
  - Computes values of **actual parameters** and moves them into the stack or registers
  - Computes the **static link** and passes it as an extra, hidden argument
  - Uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the **return address** on the stack or in a register
- In its prologue, the callee
  - allocates a frame by subtracting an appropriate constant from the **sp**
  - saves the old **fp** into the stack, and assigns it an appropriate new value
  - saves any **registers** that may be overwritten by the current routine (including the **static link** and **return address**, if they were passed in registers)

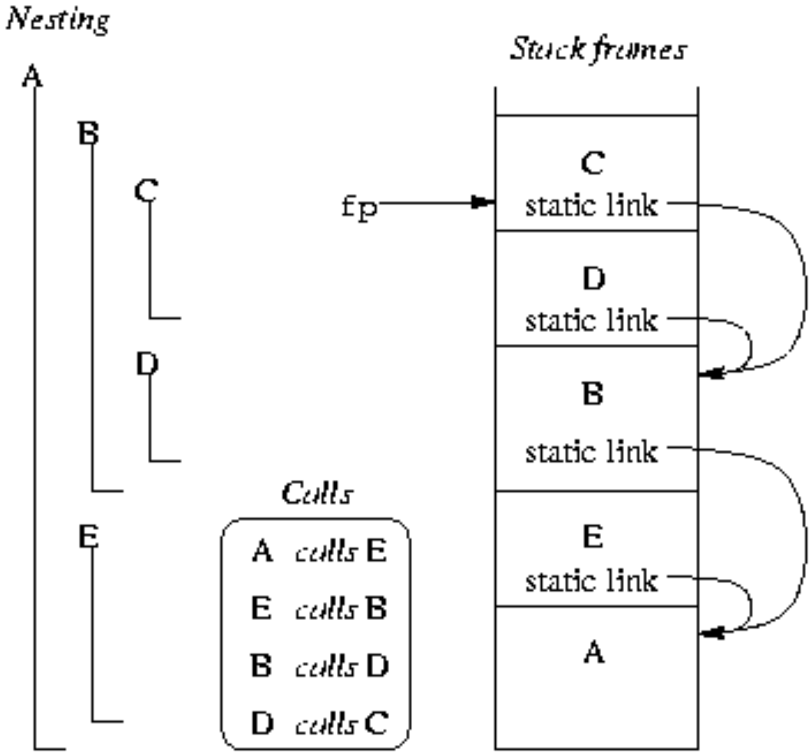
A Typical Calling Sequence (cont'd)

- After the subroutine has completed, the callee
  - Moves the return value (if any) into a register or a reserved location in the stack
  - Restores registers if needed
  - Restores the **fp** and the **sp**
  - Jumps back to the return address
- Finally, the caller
  - Moves the return value to wherever it is needed
  - Restores registers if needed

Static Chains

- How do we access non-local objects?
- The static links form a static chain, which is a linked list of static parent frames
- When a subroutine at nesting level ***j*** has a reference to an object declared in a static parent at the surrounding scope nested at level ***k***, then ***j*-*k*** static links form a static chain that is traversed to get to the frame containing the object
- The compiler generates code to make these traversals over frames to reach non-local objects

Example Static Chains



- Subroutine A is at nesting level 1 and C at nesting level 3
- When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object

Displays

- Access to an object in a scope *k* levels out requires that the static chain be dereferenced *k* times.
- An object *k* levels out will require *k* + 1 memory accesses to be loaded in a register.
- This number can be reduced to a constant by use of a **display**, a vector where the *k*-th element contains the pointer to the activation record at nesting level *k* that is currently active.
- Faster access to non-local objects, but bookeeping cost larger than that of static chain

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**. **Java** uses **1–b** for methods in a class. **Modula** uses **1–b** also for variables!
- 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;
... };
```

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, …)
- Blocks can be considered as subroutines that are called where they are defined
- 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)

Out of Scope

- Non-local objects can be *hidden* by local name-to-object bindings
- The scope is said to have a *hole* in which the non-local binding is temporarily inactive but not destroyed
- Some languages, like Ada, C++ and Java, use qualifiers or scope resolution operators to access non-local objects that are hidden
  - P1.X in Ada to access variable X of P1
  - ::X to access global variable X in C++
  - this.x or super.x in Java

Out of Scope Example

```
procedure P1;
var X:real;
  procedure P2;
  var X:integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

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 visible [*open scopes*], or are not visible inside unless *imported* [*closed scopes*], or are visible with "qualified name" [*selectively open scopes*] (eg: **B.x**)
- 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

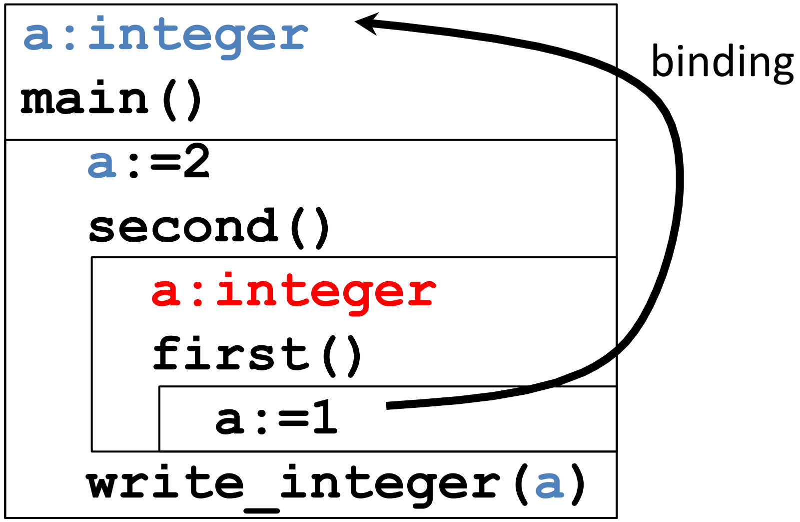
- Modules as abstraction mechanism: collection of data with operations defined on them (sort of *abstract data type)*
- Various mechanism to get module *instances*:
  - Modules as manager: instance as additional arguments to subroutines (**Modula-2**)
  - Modules as types (**Simula**, **ML**)
- Object-Oriented: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

Dynamic Scoping

- Scope rule: the “current” binding for a given name is the one encountered most recently **during execution**
- Typically adopted in (early) functional languages that are interpreted
- Perl v5 allows you to choose scope method for each variable separately
- With dynamic scope:
  - Name-to-object bindings *cannot* be determined by a compiler in general
  - Easy for interpreter to look up name-to-object binding in a stack of declarations
- Generally considered to be “a bad programming language feature”
  - Hard to keep track of active bindings when reading a program text
  - Most languages are now compiled, or a compiler/interpreter mix
- Sometimes useful:
  - Unix environment variables have dynamic scope

Effect of Static Scoping

Program execution:



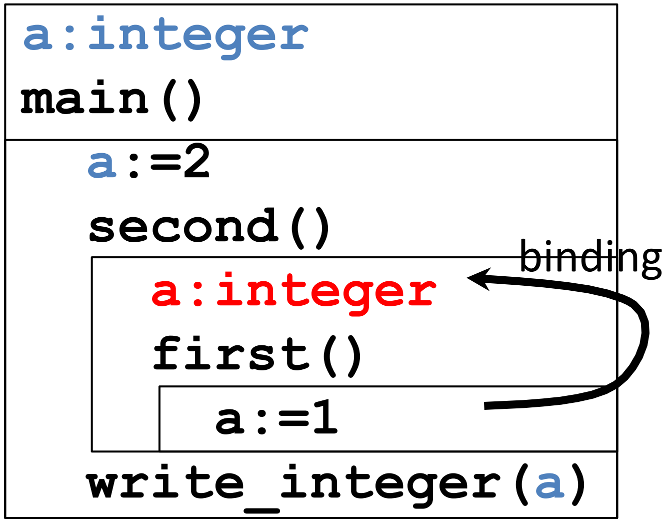
```
a:integer
main()
   a:=2
   second()
     a:integer
     first()
       a:=1
   write_integer(a)
```

Program prints “1”

- The following pseudo-code program demonstrates the effect of scoping on variable bindings:
-
```
a:integer
procedure first(){
   a:=1}
procedure second(){
   a:integer
   first()}
procedure main(){
   a:=2
   second()
   write_integer(a)}
```

Effect of Dynamic Scoping

Program execution:



```
a:integer
main()
   a:=2
   second()
      a:integer
      first()
         a:=1
   write_integer(a)
```

Program prints "2"

- The following pseudo-code program demonstrates the effect of scoping on variable bindings:
-
```
a:integer
procedure first(){
   a:=1 }
procedure second(){
   a:integer
   first()}
procedure main(){
   a:=2
   second()
   write_integer(a)}
```

Binding depends on execution

Dynamic Scoping Problems

- In this example, function **`scaled_score`** probably does not do what the programmer intended: with dynamic scoping, **`max_score`** in **`scaled_score`** is bound to **`foo`**'s local variable **`max_score`** after **`foo`** calls **`scaled_score`**, which was the most recent binding during execution:

```
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 percentage seen so far
  ...
  foreach student in class
    student.percent := scaled_score(student.points)
    if student.percent > max_score
       max_score := student.percent
}
```

Dynamic Scope Implementation with Bindings Stacks

- Each time a subroutine is called, its local variables are pushed on a stack with their name-to-object binding
- When a reference to a variable is made, the stack is searched top-down for the variable's name-to-object binding
- After the subroutine returns, the bindings of the local variables are popped
- Different implementations of a binding stack are used in programming languages with dynamic scope, each with advantages and disadvantages

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 with LIFO policy
  - new scopes (not LIFO) are created for records and classes
  - Other operations: ***enter_scope, leave_scope***
- Even with static scoping, the symbol table might be needed at runtime for ***symbolic debugging***
  - The debugger must resolve names in high-level commands by the user
  - Symbol table saved in portion of the target program code

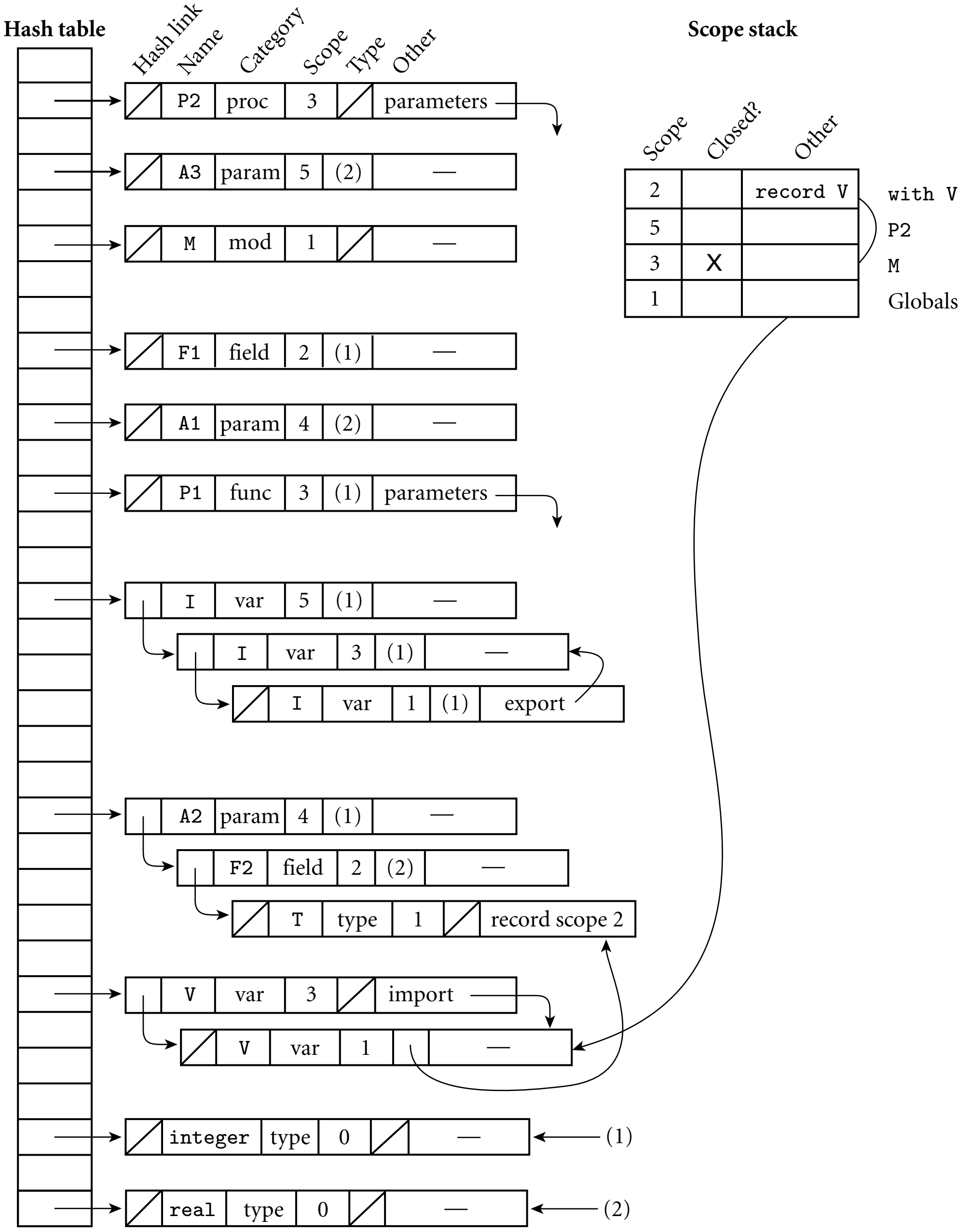
LeBlanc & Cook **Symbol Table**

Symbol table implementation for ***static scoping***, using a hash table and a stack. Managed by the semantic analyzer at compile time.

- 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 pointers.

A Modula2 program

```
type                                   [1]
    T = record
        F1 : integer;
                               [2]
        F2 : real;
    end;
var V : T;
...
module M;
    export I; import V;             [3]
    var I : integer;
    ...
    procedure P1 (A1 : real;
        A2t: integer) : real;
    begin                         [4]
        ...
    end P1;
    ...
    procedure P2 (A3 : real);
    var I : integer;              [5]
    begin
        ...
        with V do
            ...
        end;
        ...
    end P2;
    ...
end M;
```



Hash link	Name	Category	Scope	Type	Other
/	P2	proc	3	/	parameters
/	A3	param	5	(2)	—
/	M	mod	1	/	—
/	F1	field	2	(1)	—
/	A1	param	4	(2)	—
/	P1	func	3	(1)	parameters
	I	var	5	(1)	—
	I	var	3	(1)	—
/	I	var	1	(1)	export
	A2	param	4	(1)	—
	F2	field	2	(2)	—
/	T	type	1	/	record scope 2
	V	var	3	/	import
/	V	var	1		—
/	integer	type	0	/	— (1)
/	real	type	0	/	— (2)

Scope	Closed?	Other	
2		record V	with V
5			P2
3	X		M
1			Globals

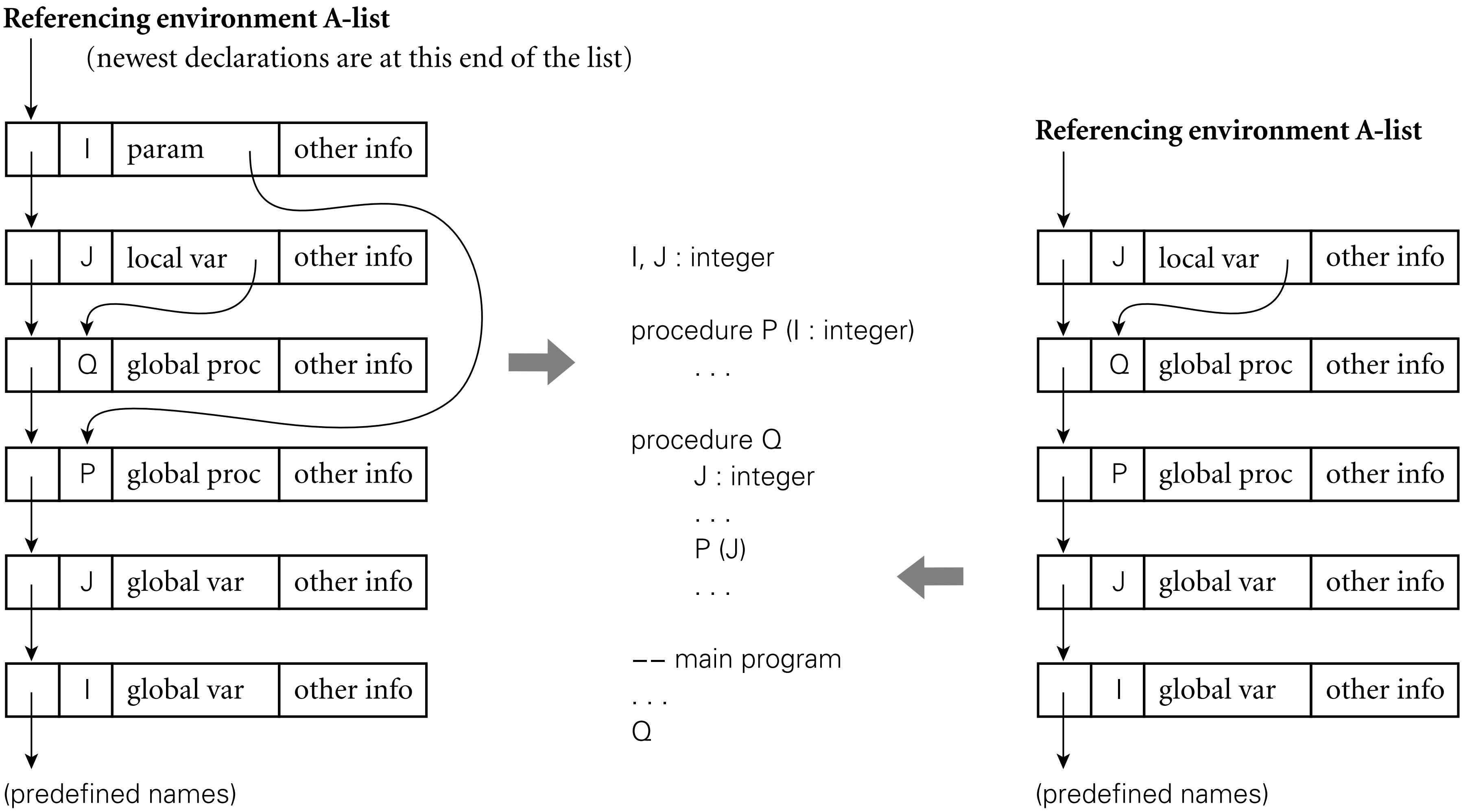
LeBlanc & Cook lookup function

```
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

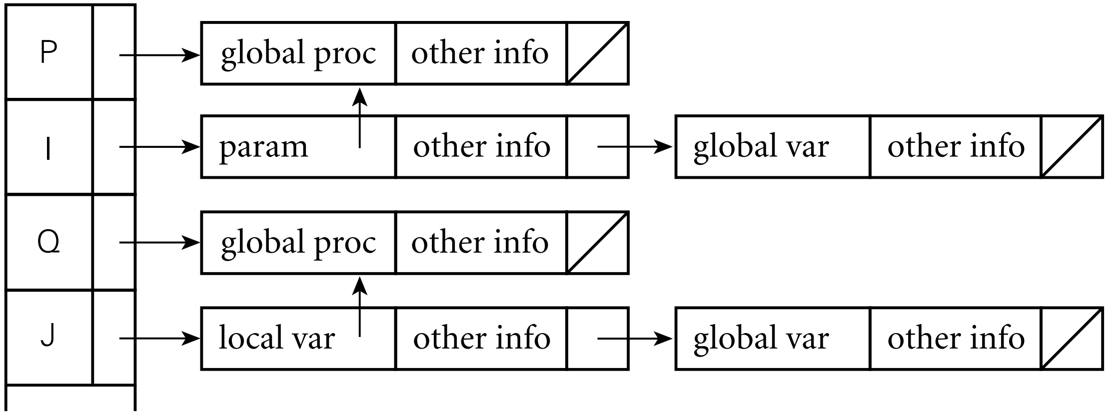
A-list after exiting P

Central reference tables

- Similar to LeBlanc&Cook hash table, but stack of scopes not needed (and at runtime!)
- 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 than with A-lists

Central reference table

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



P — global proc, other info

I — param, other info — global var, other info

Q — global proc, other info

J — local var, other info — global var, other info

(other names)

CRT after entering P in the exection of Q

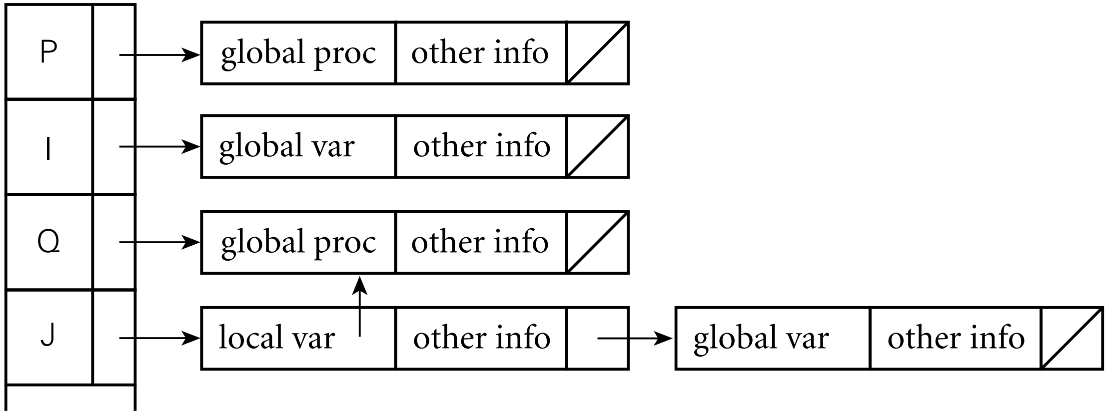
```
I, J : integer

procedure P (I : integer)
        . . .

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

–– main program
. . .
Q
```

Central reference table



P — global proc, other info

I — global var, other info

Q — global proc, other info

J — local var, other info — global var, other info

(other names)

CRT after exiting P