Organization of Programming Languages CS320/520N

Lecture 05

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Names, Bindings, and Scopes

- A **name** is a symbolic identifier used to refer to an object:
  - Variables, constants, operations, types, …
  - Names are essential for abstraction:
    - *process abstraction*, e.g. subroutines.
    - *data abstraction*, e.g. classes.
- A **binding** is an association, such as:
  - between a name and an object (e.g. variable).
  - between a question and an answer (e.g. what sorting algorithm?).
- The **scope** of a name binding is that region of the program in which the binding is active:
  - *scoping rules* define this region.

Binding Time

- **Binding time** is the time at which a binding takes place:
  - Language design time – bind operator symbols to operations.
  - Language implementation time – bind floating point type to a representation.
  - Program writing time – choose algorithms, data structures, names.
  - Compile time – bind a variable to a type in C or Java.
  - Link time – bind a name in one module to an object in another module.
  - Load time – bind a C or C++ `static` variable to a memory cell.
  - Runtime – bind a nonstatic local variable to a memory cell.
- Example: count = count + 1;

Bindings: Static vs. Dynamic

- There are 2 types of bindings: **static** and **dynamic**.
- A binding is **static** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs during execution or can change during the execution of the program.

The Type

- The type of a variable determines:
  - The set of values that the variable can store.
  - The set of operations that are defined on these values.

- For example, the **int** primitive type in Java:
  - specifies the range [–2147483648, 2147483648].
  - operations such as addition, substraction, multiplication, division and modulo.

Static Type Binding

- **Explicit declaration** – a program statement used for declaring the types of variables
  - Most programming languages (C/C++, Java, …)
  - In C/C++:
    - declarations only specify types and other attributes;
    - definitions specify attributes and cause storage allocation.
- **Implicit declaration** – a default mechanism for specifying types of variables
  - In PERL, prefixes define types: any name beginning with $ is a scalar (numeric or string), @ is an array, % is a hash structure.
  - In ML, types are implicitly associated using type inference.

Dynamic Type Binding

- The variable is associated with a type every time it is assigned a value through an assignment statement.
- Examples (Javascript):

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

- Advantage: flexibility (generic program units).
- Disadvantages:
  - Usually purely interpreted ⇒ slow execution.
  - Costly implementation of dynamic type checking.

Binding Times

- **Early binding times** are associated with greater efficiency:
  - Compiled languages tend to have early binding times.
    - generate memory layout for global variables, efficient code to access them.
    - static type checking.

- **Later binding times** are associated with greater flexibility:
  - Interpreted languages tend to have later binding times.
    - allow a variable name to refer to objects of multiple types:
      - generic subroutines.
    - dynamic type checking.

Key Events & Lifetimes

- When discussing bindings, important to distinguish names from objects they refer to.

- Key events:
  - creation of objects.
  - creation of bindings.
  - references to objects (which use bindings).
  - deactivation and reactivation of bindings (temporary unusable).
  - destruction of bindings.
  - destruction of objects.

Key Events & Lifetimes

- The **lifetime of a** name-object **binding** is the period of time between the creation and the destruction of this binding.
- The **lifetime of an object** is the time between the creation and destruction of an object.
  - Name-object binding lifetime and object lifetime do not necessarily coincide.
    - Bindings may outlive objects:
      - dangling references.
    - Objects may outlive bindings:
      - memory leaks.
      - reference parameters.

Object Lifetime and Storage Management

- Object lifetimes corespond to 3 principal storage allocation mechanisms:
  - **Static** allocation:
    - code, global variables, static or own variables, explicit constants.
  - **Stack**-based allocation:
    - parameters, local variables, temporary values.
  - **Heap**-based allocation.
- Depending on their lifetime, 4 categories of variables:
  1. **Static**.
  2. **Stack**-Dynamic.
  3. Explicit **Heap**-Dynamic.
  4. Implicit **Heap**-Dynamic.

Static Variables

- A **static** variable is bound to a memory cell before execution begins and remains bound to the same memory cell throughout execution.
  - Example: C and C++ `static` variables, global variables.
    Fortran local variables (before Fortran 90).

```
int  myFunction() {
        static int count = 0;
        …
        count++;
        return count;
}
```

Static Variables

- Advantages:
  - efficiency: direct addressing, no run-time overhead for allocation & deallocation.
  - history-sensitive : maintain values between successive function calls.

- Disadvantages:
  - lack of flexibility (no recursion).
  - storage cannot be shared among variables.

Stack-Dynamic Variables

- **Stack-dynamic** = storage is allocated & deallocated in last-in first-out order, from the **run-time stack**.
  - usually in conjunction with subroutine calls and returns.
  - Example: local variables in C subprograms and Java methods.

```
int  factorial(int n) {
        int result = 1;
        for (int i = 2; i ≤ n; i++)
            result *= i;
        return result;
}
```

Stack-Dynamic Variables

- Advantages:
  - Allows recursion;
  - Conserves storage.

- Disadvantages:
  - Overhead of allocation and deallocation.
  - Subprograms cannot be history sensitive.
  - Inefficient references (indirect addressing).

Stack-based Allocation

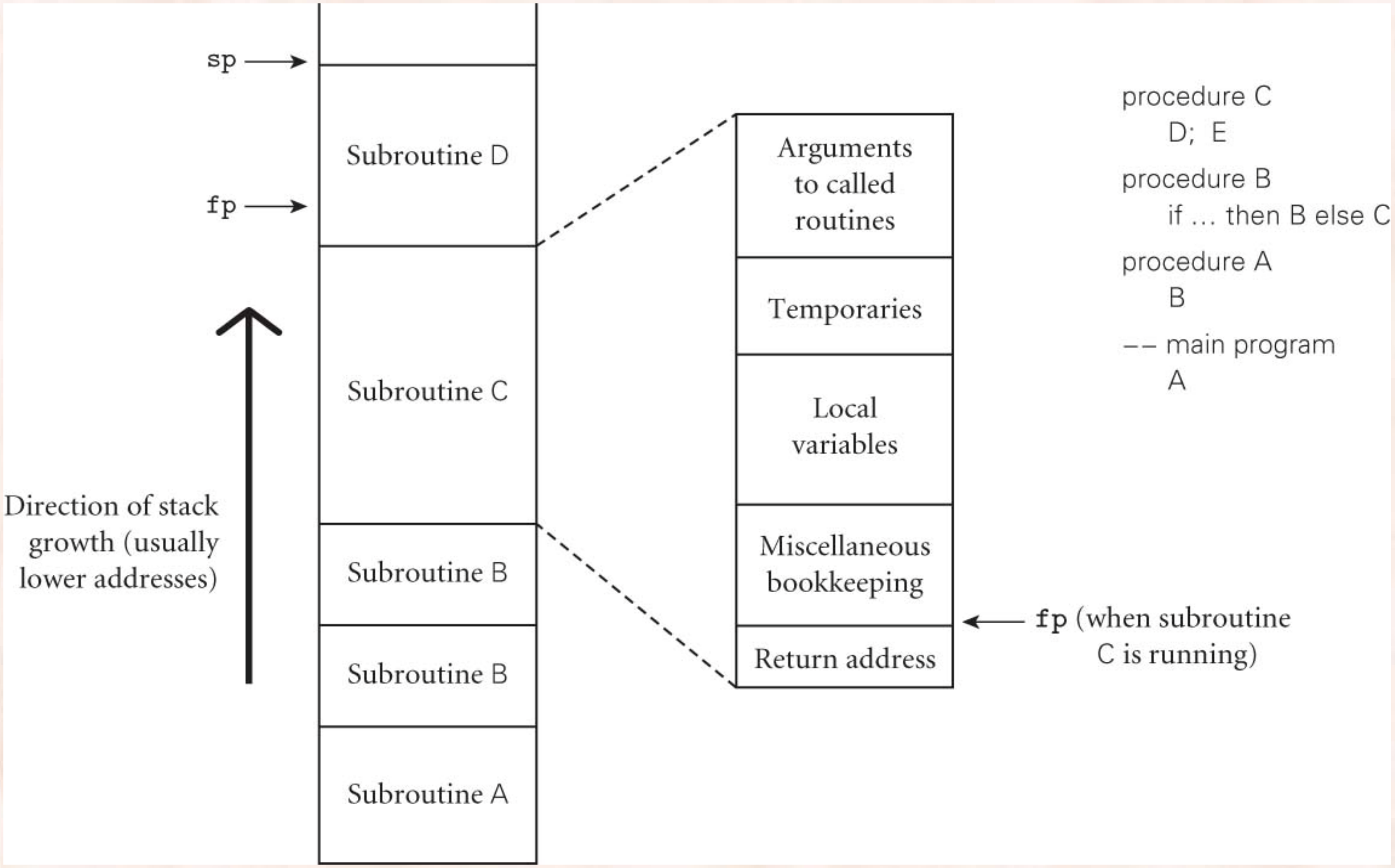
- Each instance of a subroutine has its own **frame**, or **activation record**, on the run-time stack:
  - arguments and return values.
  - local variables and temporary values.
  - bookkeeping information (e.g., saved registers, *static link*).

- Addresses computed relative to the **stack pointer (sp)** or **frame pointer (fp)**:
  - fixed OFFSETS determined at compile time.
  - **frame pointer** set to point to a known location within frame.

Stack-based Allocation

- Stack maintenance through:
  - **calling sequence**:
    - code executed by the caller, immediately before & after the call.
  - **prologue** & **epilogue**:
    - code executed by the callee, at the beginning & the end of the subroutine.
  - more details in PLP 8.2

Stack-based Allocation


sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine B
Subroutine A
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)

```
procedure C
    D;  E
procedure B
    if ... then B else C
procedure A
    B
-- main program
    A
```

Explicit Heap-Dynamic Variables

- **Explicit heap-dynamic** variables are allocated and deallocated from the heap by explicit directives, specified by the programmer, which take effect during execution:
  - The actual variables are nameless.
  - Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java.

```
int *intNode;           // create the pointer, stack-dynamic.
…
intNode = new int;      // create the heap-dynamic variable.
…
delete intNode;         // deallocate the heap-dynamic variable.
```

Explicit Heap-Dynamic Variables

- Advantages:
  - Enable the specification and construction of dynamic structures (linked lists & trees) that grow and shrink during the execution.
- Disadvantages:
  - Unreliable: difficult to use pointers & references correctly.
  - Innefficient: heap managemenet is costly and complicated.

Implicit Heap-Dynamic Variables

- **Implicit heap-dynamic** variables – allocation and deallocation caused by assignment statements:
  - All their attributes (e.g. type) are bound every time they are assigned.
  - Examples: strings and arrays in Perl, variables in JavaScript & PHP.

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

- Advantages: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic.
  - Loss of error detection by compiler.

Heap-based Allocation

- Storage management algorithms:
  - maintain one **free list** (linear cost):
    - **first fit** algorithm
    - **best fit** algorithm
  - divide heap into multiple free lists, one for each standard size:
    - static division.
    - dynamic division:
      - **buddy system**:
        - » split block of size 2^{k+1} into two blocs of size 2^k
      - **Fibonnacci heap**:
        - » split block of size F_n into two blocs of size F_{n-1}, F_{n-2}.

Heap-based Allocation

- Need to **compact** the heap to reduce external fragmentation.
- **Garbage collection** eliminates manual deallocation errors:
  - dangling references.
  - memory leaks.
- More details in PLP 7.7.2 and 7.7.3.

Heap-based Allocation

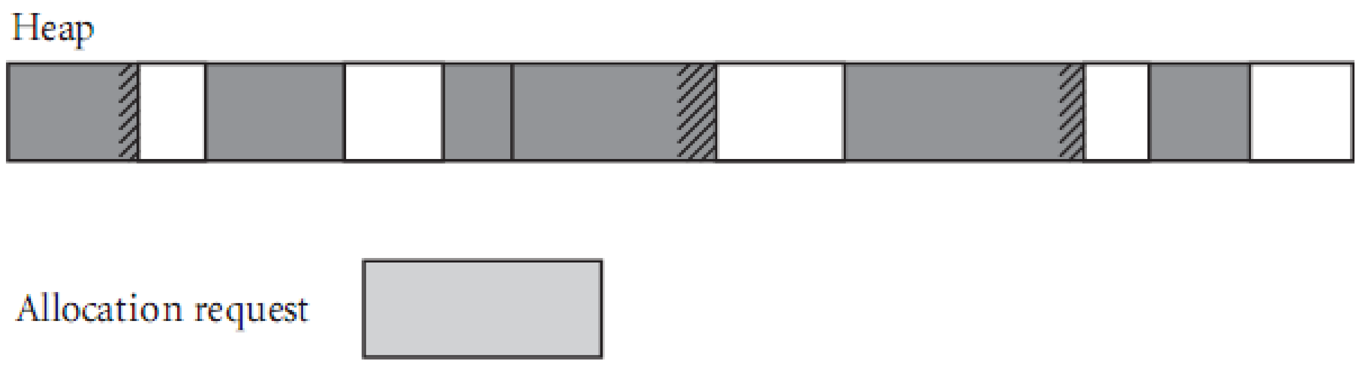


Figure 3.2 Fragmentation. The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Scope

- The **scope** of a variable is the range of statements over which it is *visible*:
  - Variable *v* is visible in statement *s* if *v* can be referenced in *s*.
- The scope rules of a language determine how occurrences of names are associated with variables:
  - **static scoping**.
  - **dynamic scoping**.
- Two types of variables:
  - **local** variables: declared inside the program unit/block.
  - **nonlocal** variable: visible, but declared outside the program unit.

Static Scope

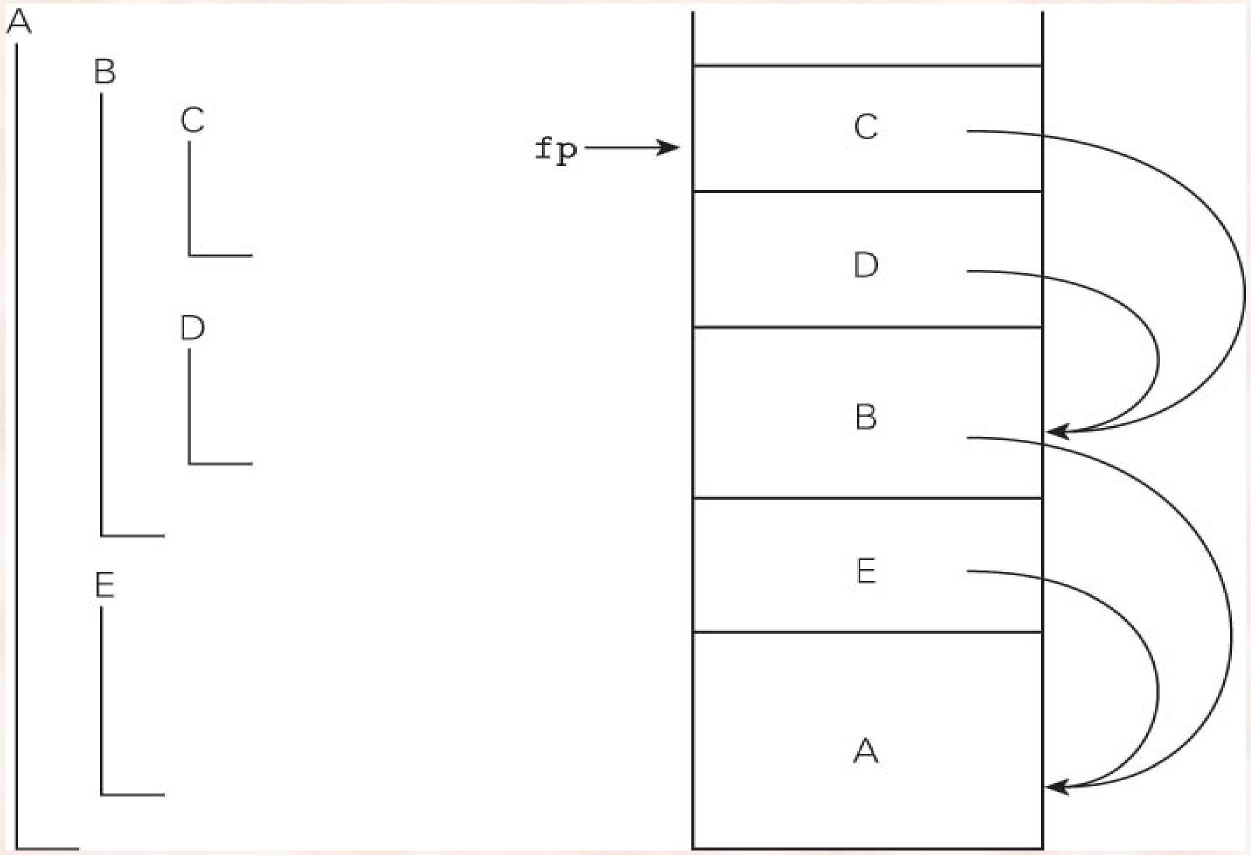
- Introduced in ALGOL 60 as a method of binding names to nonlocal variables:
  - To connect a name reference to a variable, you (or the compiler) must find the declaration.
  - Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name.
- Two ways of creating nested static scopes:
  - Nested subprogram definitions (e.g., Ada, JavaScript, and PHP).
  - Nested blocks.
- Given a specific scope:
  - Enclosing static scopes are called its static ancestors;
  - The nearest static ancestor is called its static parent.

Static Scope: Subprograms

```
procedure Big is
    X: Integer;
    procedure SUB1 is
       X: Integer;
       begin  -- of SUB1
       ...
       end;  -- of SUB1
     procedure SUB2 is
       begin  -- of SUB2
       ... X ...
       end;  -- of SUB2
   begin  -- of Big
    ...
   end;  -- of Big
```

Big calls Sub1
Sub1 calls Sub2
Sub2 uses X

Implementation: Static Chains



Static Scope: Blocks

- Blocks – a method of creating (nested) static scopes inside program units (introduced in ALGOL 60).
- Examples:
  - C-based languages:

```
while (...) {
   int index;
    ...
 }
```

  - Ada:

```
declare Temp : Float;
   begin
    ...
   end
```

Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name:
  - creates “holes” in the scope.

- C++ and Ada allow access to these "hidden" variables:
  - In Ada: **`unit.name`**
  - In C++: **`class_name::name`**
  - In Python: **`global name`**

Declaration Order

- If object *x* is declared somewhere within block B, does the scope of *x* include the portion of B before the declaration?

- Declaration order rules:
  - Algol 60, LISP (early languages):
    - all declarations appear at the beginning of their scope.
  - Pascal:
    - names must be declared before they are used.
      - the scope is still the entire block => subtle interactions.
  - Ada, C, C++, Java:
    - the scope is from the declaration to the end of the block (w/o holes).

Declaration Order

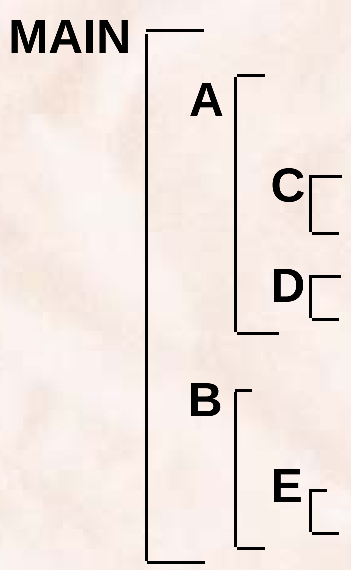
- Declaration order rules:
  - C++ and Java relax the rules in many cases:
    - members of a class are visible inside all class members.
    - classes in Java can be declared in any order.
  - Modula-3:
    - the scope is the entire block (minus holes) => can use a variable before declaring it.
  - Scheme is very flexible:
    - let
    - let* (*declaration-to-end-of-block* semantics)
    - letrec (*whole-bloc* semantics)

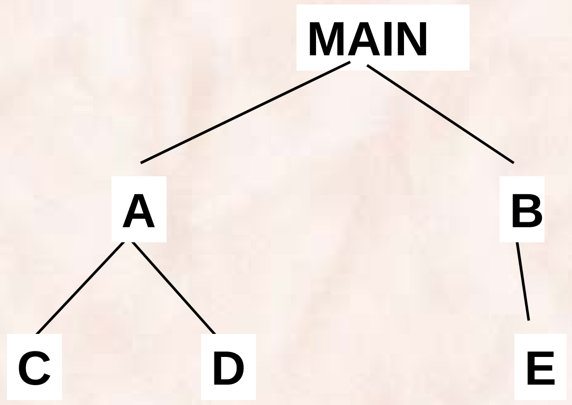
Static Scope: Evaluation

MAIN can call A and B

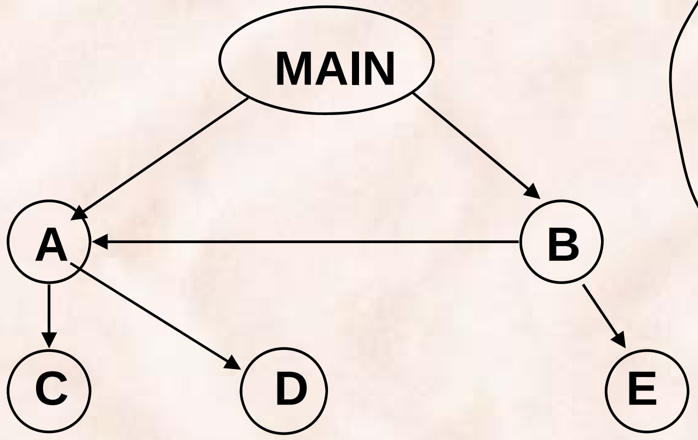
A can call C and D

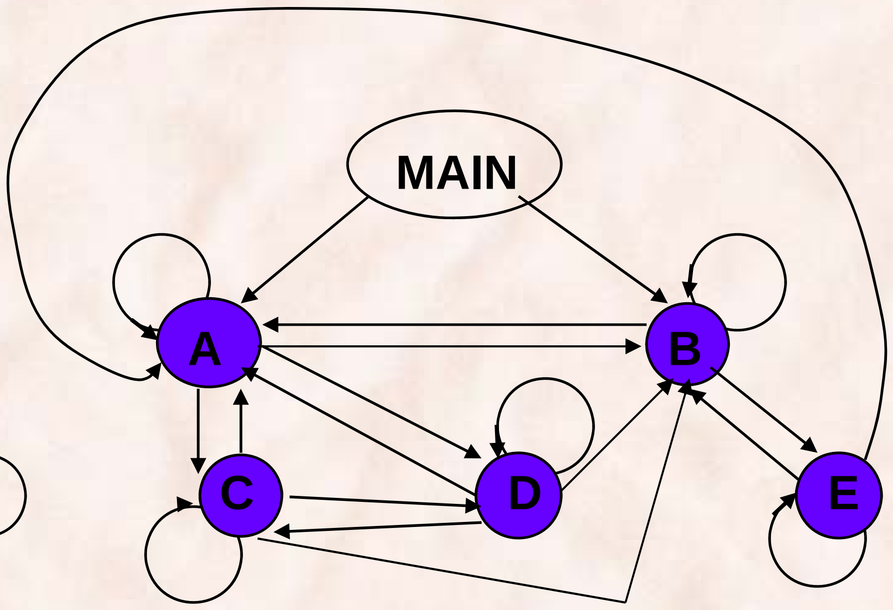
B can call A and E





Static Scope: Evaluation





Static Scope: Evaluation

- Suppose the specification is changed so that D must now access some data in B.
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables).
  - Move the data from B that D needs to MAIN (but then all procedures can access them).
- Same problem for procedure access as for data access.
- Overall: static scoping often encourages many globals.

Dynamic Scope

- **Static Scope**: names are associated to variables based on their textual layout (spatial).
- **Dynamic Scope**: names are associated to variables based on calling sequences of program units (temporal).
  - References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

Dynamic Scope Example

```
procedure Big is
    X: Integer;
    procedure SUB1 is
      X: Integer;
      begin  -- of SUB1
      ...
      end;  -- of SUB1
    procedure SUB2 is
      begin  -- of SUB2
      ... X ...
      end;  -- of SUB2
   begin  -- of Big
    ...
   end;  -- of Big
```

Big calls Sub1
Sub1 calls Sub2
Sub2 uses X

Typical Stack Frame

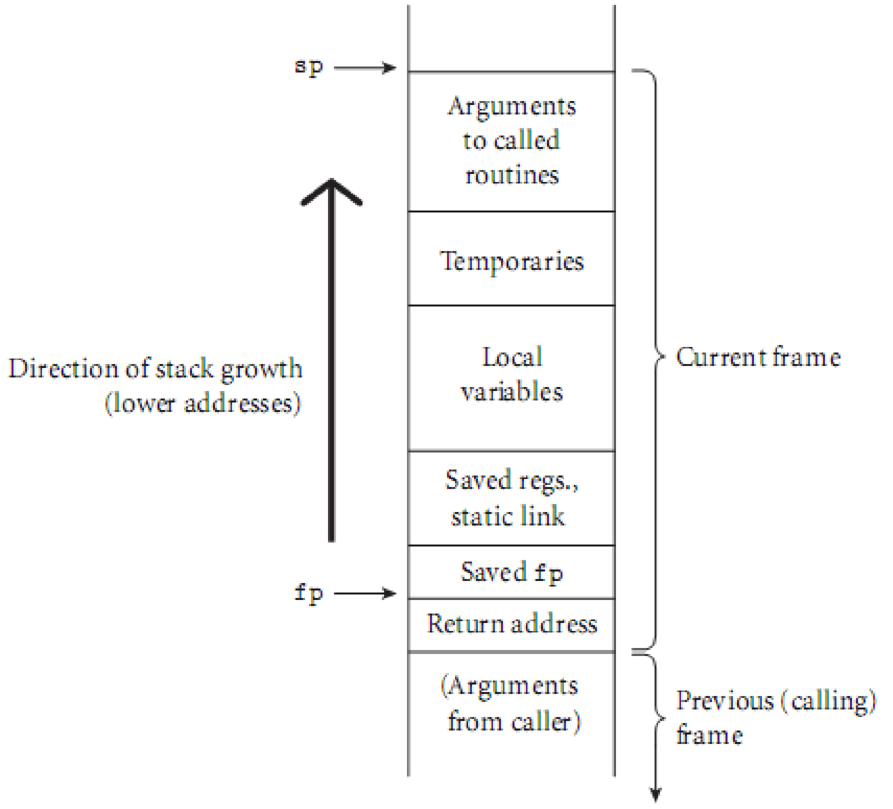


Figure 8.2 **A typical stack frame.** Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

Dynamic Scope: Implementation

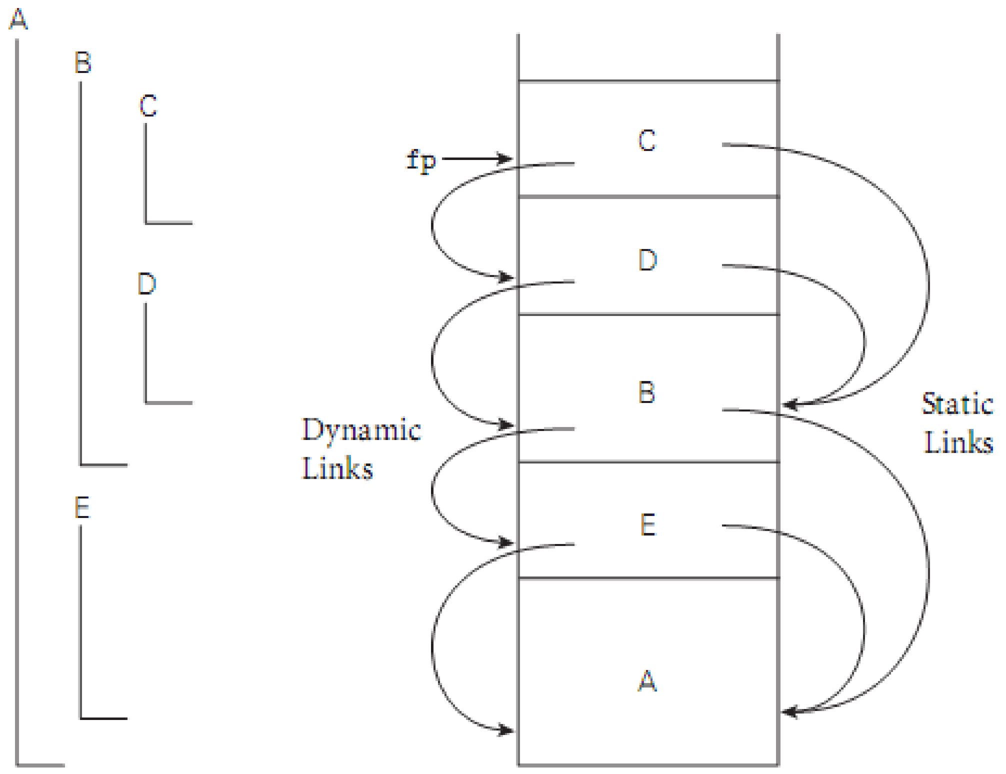


Figure 8.1 **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

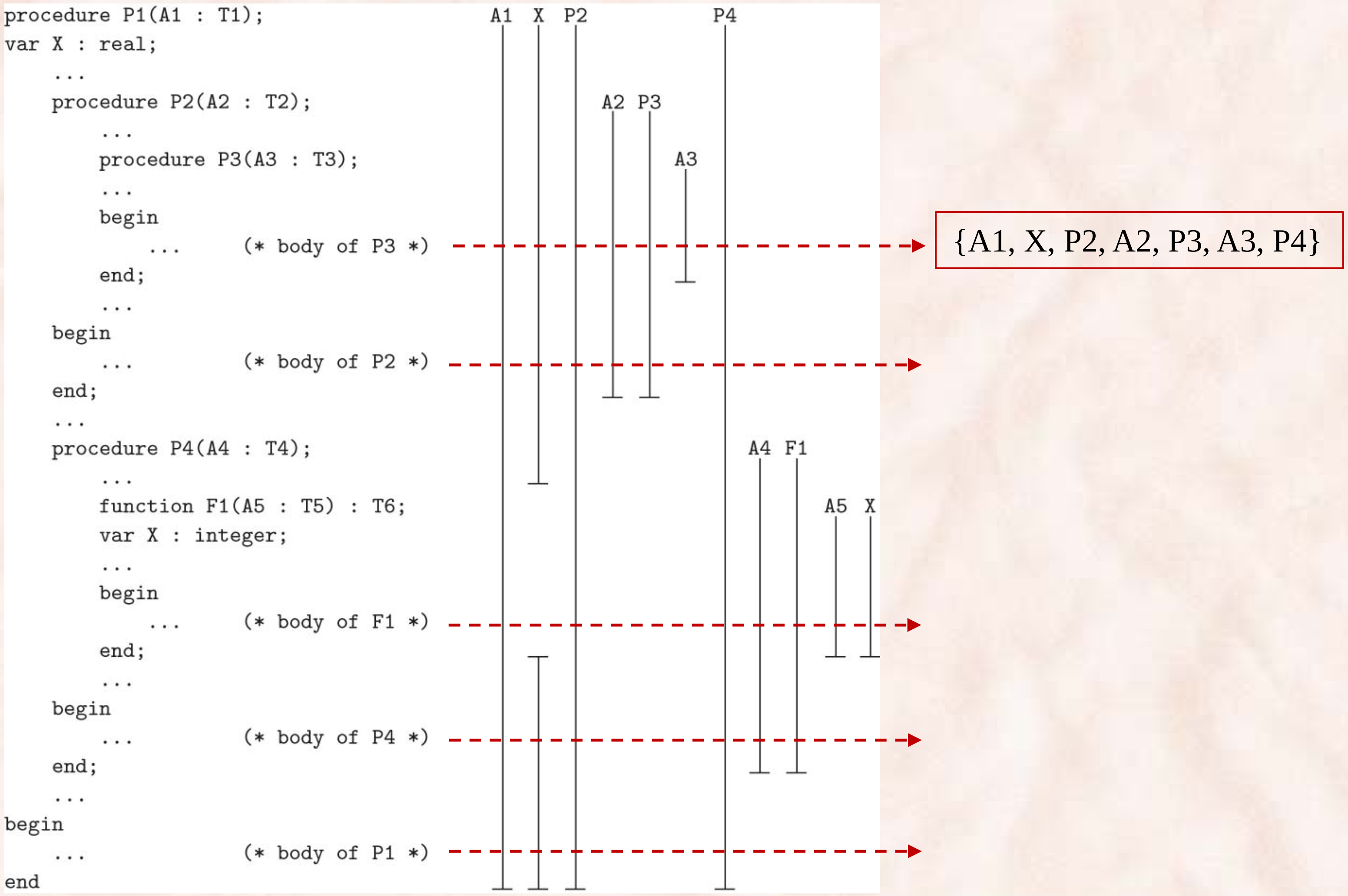
Dynamic Scope: Evaluation

- Advantages:
  - convenience: called subprogram is executed in the context of the caller $\Rightarrow$ no need to pass variables in the caller as parameters.
- Disadvantages:
  - poor readability
    - virtually impossible for a human reader to determine the meaning of references to nonlocal variables.
  - less reliable programs than with static scoping.
  - execution is slower than with static scoping.

Referencing Environments

- The **referencing environment** of a statement is the collection of all names that are visible in the statement:
  - In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes.
  - In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all **active** subprograms:
    - A subprogram is **active** if its execution has begun but has not yet terminated.
  - Variables in *enclosing scopes*/*active subprograms* can be hidden by variables with same name.

Referencing Environments



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

Binding Rules & Closures

- Some languages allow using subroutines as parameters.
- When should the scope rule be applied?
  - When the subroutine is passed as parameter:
    - $\Rightarrow$ **deep binding**
  - When the subroutine is called:
    - $\Rightarrow$ **shallow binding**

- Deep binding is default in static scoping:
  - create an explicit representation of the **referencing environment**.
  - bundle it with a reference to the **subroutine**.

  => a **closure**.

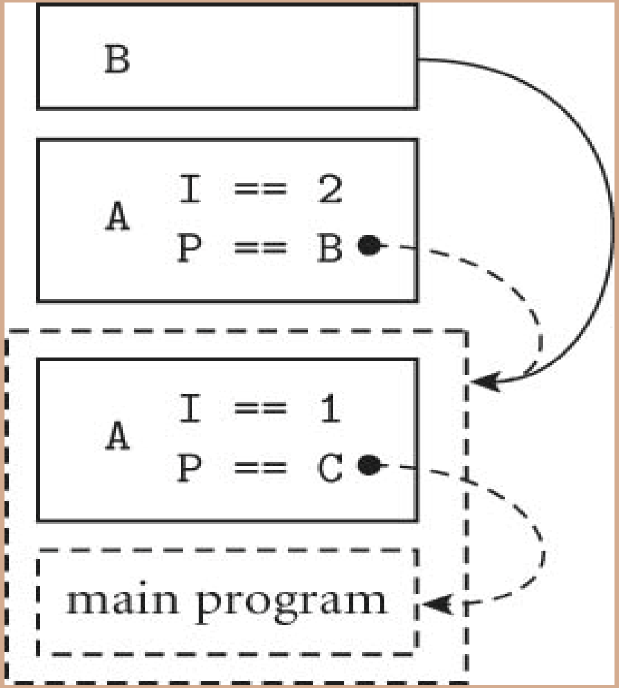
```
program main(input, output);
        procedure A(I: integer; procedure P);
             procedure B;
             begin
                writeln(I);
             end;

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

procedure C;
      begin
      end;

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

Deep Binding (Pascal)



Deep Binding => ?
Shallow Binding => ?

First-Class Values and Unlimited Extent

- First-class values can be:
  - passed as parameter;
  - returned from a subroutine;
  - assigned into a variable.

- Functions are first-class values in functional PLs.

- Scheme is a functional PL:
  - functions are first-class values & scopes may be nested.

  ⇒ functions may outlive the execution of the scope in which they were declared => local objects need to have **unlimited extent**.

Unlimited Extent

- Unlimited extent = lifetime continues indefinitely:
  - space reclaimed by garbage collector.
    - space generally allocated on the heap.

```
(define plus-x (lambda (x)
        (lambda (y) (+ x y))))
 …
(let ((f (plus-x 2)))
        (f 3))
```

Separate Compilation

- Separately-compiled files in C provide a sort of poor person's modules:
  - Rules for how variables work with separate compilation are messy.
    - Language has been jerry-rigged to match the linker.
  - *Static* on a function or variable outside a function means it is usable only in the current source file
    - Different notion from the static variables inside a function.
  - *Extern* on a variable or function means that it is declared in another source file
    - Functions headers without bodies are *extern* by default.
  - *Extern* declarations are interpreted as forward declarations if a later declaration overrides them.