Chapter 3:: Names, Scopes, and Bindings

Programming Language Pragmatics

Michael L. Scott

Name, Scope, and Binding

- A name is exactly what you think it is
  - Most names are identifiers
  - symbols (like '+') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually)in which the binding is active

Binding

- Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
  - language design time
    - program structure, possible type
  - language implementation time
    - I/O, arithmetic overflow, type equality (if unspecified in manual)

Binding

- Implementation decisions (continued):
  - program writing time
    - algorithms, names
  - compile time
    - plan for data layout
  - link time
    - layout of whole program in memory
  - load time
    - choice of physical addresses

Binding

- Implementation decisions (continued):
  - run time
    - value/variable bindings, sizes of strings
    - subsumes
      - program start-up time
      - module entry time
      - elaboration time (point a which a declaration is first "seen")
      - procedure entry time
      - block entry time
      - statement execution time

Binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively
  - “static” is a coarse term; so is "dynamic"
- *IT IS DIFFICULT TO OVERSTATE THE IMPORTANCE OF BINDING TIMES IN PROGRAMMING LANGUAGES*

Binding

- In general, early binding times are associated with greater efficiency
- Later binding times are associated with greater flexibility
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times
- Today we talk about the binding of identifiers to the variables they name

Binding

- Scope Rules - control bindings
  - Fundamental to all programming languages is the ability to name data, i.e., to refer to data using symbolic identifiers rather than addresses
  - Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names

Lifetime and Storage Management

- Key events
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Lifetime and Storage Management

- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding it's garbage
  - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is *active* is its scope
- In addition to talking about the *scope of a binding*, we sometimes use the word *scope* as a noun all by itself, without an indirect object

Lifetime and Storage Management

- Storage Allocation mechanisms
  - Static
  - Stack
  - Heap
- Static allocation for
  - code
  - globals
  - static or own variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions

Lifetime and Storage Management

- Central stack for
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines (not necessary in FORTRAN – no recursion)
  - reuse space (in all programming languages)

Lifetime and Storage Management

- Contents of a stack frame (cf., Figure 3.1)
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Lifetime and Storage Management

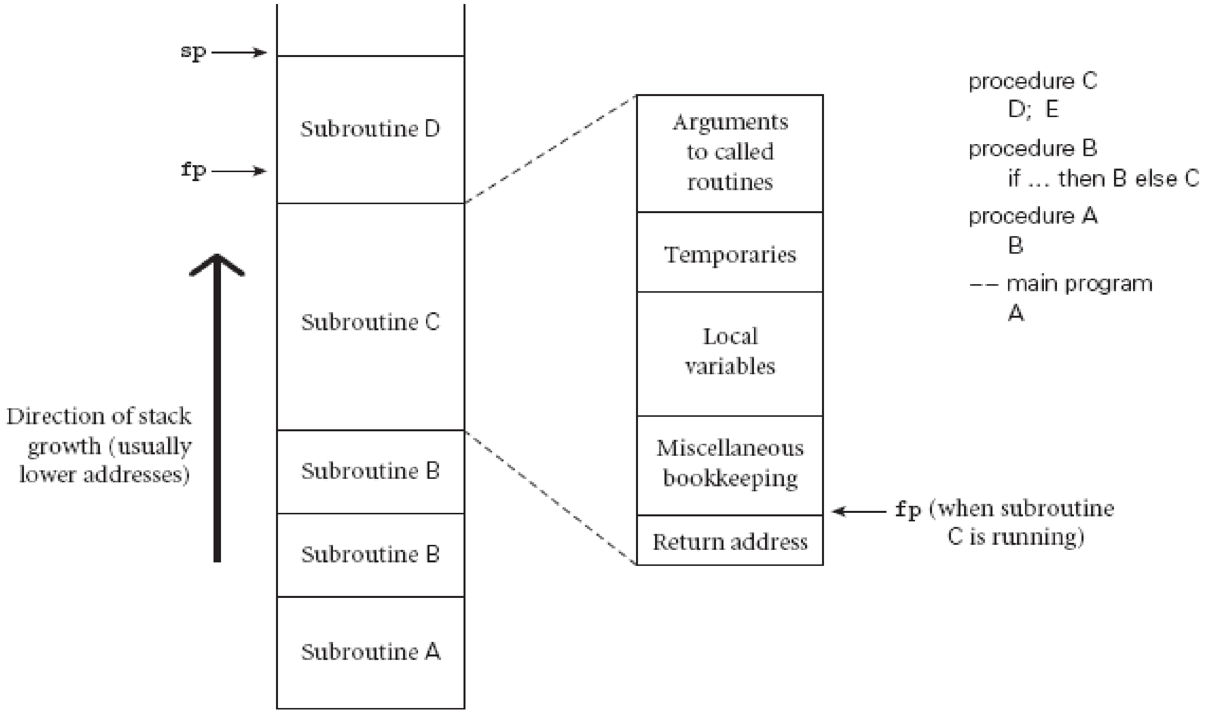

Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (sp) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (fp) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Lifetime and Storage Management

- Maintenance of stack is responsibility of *calling sequence* and subroutine *prolog* and *epilog*
  - space is saved by putting as much in the prolog and epilog as possible
  - time *may* be saved by
    - putting stuff in the caller instead

      or
    - combining what's known in both places (interprocedural optimization)

Lifetime and Storage Management

- Heap for dynamic allocation

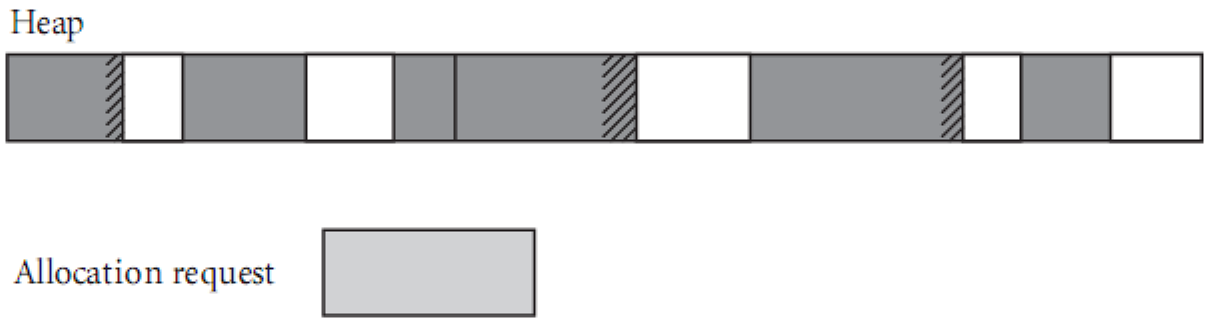

Figure 3.2 **Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy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

Scope Rules

- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted (see below)
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
  - make references to variables

Scope Rules

- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
- Algol 68:
  - ELABORATION = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
  - storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations

Scope Rules

- With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the physical (lexical) structure of the program
  - The determination of scopes can be made by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time
  - Most compiled languages, C and Pascal included, employ static scope rules

Scope Rules

- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Algol 60 and Pascal
  - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope
  - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found

Scope Rules

- We will see classes - a relative of modules - later on, when discussing abstraction and object-oriented languages
  - These have even more sophisticated (static) scope rules
- Euclid is an example of a language with lexically-nested scopes in which all scopes are closed
  - rules were designed to avoid ALIASES, which complicate optimization and correctness arguments

Scope Rules

- Note that the bindings created in a subroutine are destroyed at subroutine exit
  - The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime
  - Bindings to variables declared in a module are inactive outside the module, not destroyed
  - The same sort of effect can be achieved in many languages with *own* (Algol term) or *static* (C term) variables (see Figure 3.5)

Scope Rules

- Access to non-local variables STATIC LINKS
  - Each frame points to the frame of the (correct instance of) the routine inside which it was declared
  - In the absence of formal subroutines, *correct* means closest to the top of the stack
  - You access a variable in a scope k levels out by following k static links and then using the known offset within the frame thus found

Scope Rules

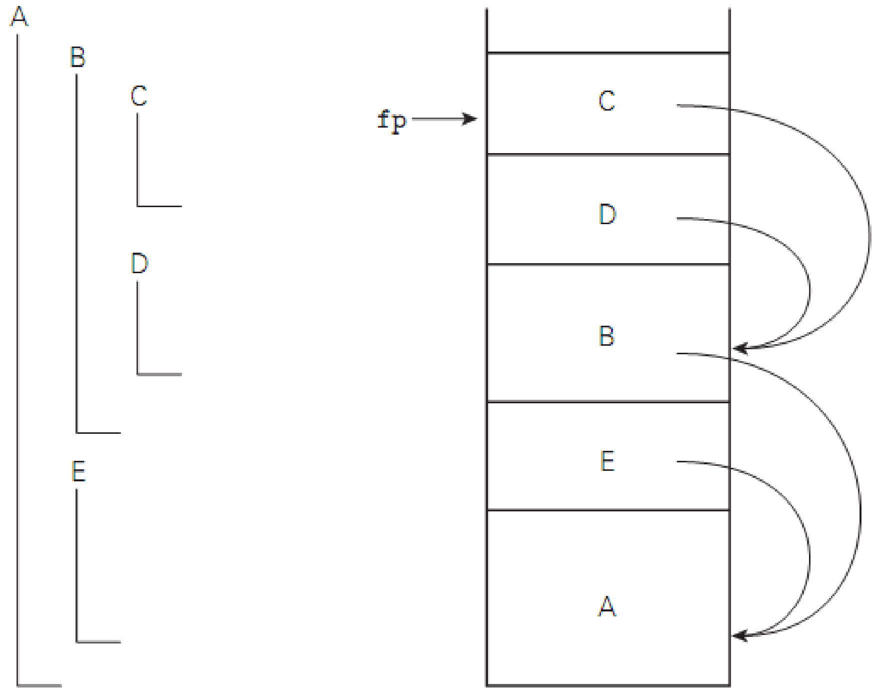

Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Scope Rules

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time

Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
  - early LISP dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

Scope Rules
Example: Static vs. Dynamic

```
program scopes (input, output );
var a : integer;
procedure first;
   begin a := 1; end;
procedure second;
   var a : integer;
   begin first; end;
begin
   a := 2; second; write(a);
end.
```

Scope Rules
Example: Static vs. Dynamic

- If static scope rules are in effect (as would be the case in Pascal), the program prints a 1
- If dynamic scope rules are in effect, the program prints a 2
- Why the difference? At issue is whether the assignment to the variable a in procedure *first* changes the variable a declared in the main program or the variable a declared in procedure *second*

Scope Rules
Example: Static vs. Dynamic

- Static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable `a`
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding at run time
  - Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
  - This is generally considered bad programming practice nowadays
    - Alternative mechanisms exist
      - static variables that can be modified by auxiliary routines
      - default and optional parameters

Scope Rules
Example: Static vs. Dynamic

- At run time we create a binding for a when we enter the main program.
- Then we create another binding for a when we enter procedure *second*
  - This is the most recent, active binding when procedure *first* is executed
  - Thus, we modify the variable local to procedure *second*, not the global variable
  - However, we write the global variable because the variable a local to procedure second is no longer active

The Meaning of Names within a Scope

- Aliasing
  - What are aliases good for? (consider uses of FORTRAN equivalence)
    - space saving - modern data allocation methods are better
    - multiple representations - unions are better
    - linked data structures   - legit
  - Also, aliases arise in parameter passing as an unfortunate side effect
    - Euclid scope rules are designed to prevent this

The Meaning of Names within a Scope

- Overloading
  - some overloading happens in almost all languages
    - integer + v. real +
    - read and write in Pascal
    - function return in Pascal
  - some languages get into overloading in a big way
    - Ada
    - C++

The Meaning of Names within a Scope

- It's worth distinguishing between some closely related concepts
  - overloaded functions - two different things with the same name; in C++
    - overload norm

```
int norm (int a){return a>0 ? a : -a;)
complex norm (complex c ) { // ...
```

  - polymorphic functions -- one thing that works in more then one way
    - in Modula-2: function min (A : array of integer); …
    - in Smalltalk

The Meaning of Names within a Scope

- It's worth distinguishing between some closely related concepts (2)
  - generic functions (modules, etc.) - a syntactic template that can be instantiated in more than one way *at compile time*
    - via macro processors in C++
    - built-in in C++
    - in Clu
    - in Ada

Binding of Referencing Environments

- Accessing variables with dynamic scope:
  - (1) keep a stack (*association list*) of all active variables
    - When you need to find a variable, hunt down from top of stack
    - This is equivalent to searching the activation records on the dynamic chain

Binding of Referencing Environments

- Accessing variables with dynamic scope:
  - (2) keep a central table with one slot for every variable name
    - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
    - Otherwise, you'll need a hash function or something to do lookup
    - Every subroutine changes the table entries for its locals at entry and exit.

Binding of Referencing Environments

- (1) gives you slow access but fast calls
- (2) gives you slow calls but fast access
- In effect, variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
- Because static scope rules tend to be more complicated, however, the data structure and lookup algorithm also have to be more complicated

Binding of Referencing Environments

- REFERENCING ENVIRONMENT of a statement at run time is the set of active bindings
- A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding

Binding of Referencing Environments

- SCOPE RULES determine that collection and its order
- BINDING RULES determine which instance of a scope should be used to resolve references when calling a procedure that was passed as a parameter
  – they govern the binding of referencing environments to formal procedures

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 behavior of the linker
  - *Static* on a function or variable *outside* a function means it is usable only in the current source file
    - This *static* is a different notion from the *static* variables inside a function

Separate Compilation

- Separately-compiled files in C (continued)
  - *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

Separate Compilation

- Separately-compiled files in C (continued)
  - Variables or functions (with bodies) that don't say *static* **or** *extern* are either *global* or *common* (a Fortran term)
    - Functions and variables that are given initial values are *global*
    - Variables that are not given initial values are *common*
  - Matching common declarations in different files refer to the same variable
    - They also refer to the same variable as a matching *global* declaration

Conclusions

- The morals of the story:
  - language features can be surprisingly subtle
  - designing languages to make life easier for the compiler writer *can* be a GOOD THING
  - most of the languages that are easy to understand are easy to compile, and vice versa

Conclusions

- A language that is easy to compile often leads to
  - a language that is easy to understand
  - more good compilers on more machines (compare Pascal and Ada!)
  - better (faster) code
  - fewer compiler bugs
  - smaller, cheaper, faster compilers
  - better diagnostics