==========
Binding Rules

Recall that the ***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.

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 subroutine that was passed as a parameter, returned from a function, or stored in a variable.

That is, they govern the binding of referencing environments to ***formal subroutines***.

With ***shallow binding***, the nonlocal referencing environment of a subroutine is the referencing environment in force at the time it (the subroutine) is called. Original Lisp worked this way by default.

With ***deep binding***, the nonlocal referencing environment of a subroutine is the referencing environment in force when creating a reference to the subroutine at run time -- when passing it to or returning it from some other subroutine, or when storing a reference to it in a pointer.

[
Aside: "subroutine" v "function" v "procedure" v "method
Subroutine is the general term.
A function is a subroutine that returns a value.
A procedure is a subroutine that does *not* return a value
(it is executed solely for its side effects).
A method is a subroutine that belongs to a class, with special scope rules and usually dynamic choice among subclass instances.
I am sometimes sloppy about using these; bear with me.
]

For subroutines passed as parameters, this environment captured by deep binging is the same as would be extant if the procedure were actually called at the point where the reference was created. When the reference is passed or saved, this referencing environment is passed or saved as well. When the subroutine is eventually called (through the reference), this saved referencing environment is restored.
The original Lisp made this behavior available when desired; it's the default in most modern languages.

A subroutine reference together with its bundled referencing environment is called a ***subroutine closure***. There are several possible implementations; the simplest is code address plus a copy of the static link.

First and second-class subroutines

first: can pass, return, store
second: can pass, but not return or store

Why not return or store?
Limited v. unlimited extent.

Example: (* OCaml *)

```
let plus_n n = fun k -> n + k;;
let plus_3 = plus_n 3;;
let apply_to_2 f = f 2;;

apply_to_2 plus3                => 5
```

Here the n inside `plus_n` needs to have a lifetime that extends beyond the call inside `plus_3`.

Note 1: The difference between deep and shallow binding is not apparent unless you pass subroutines as parameters, return them from functions, or store references to them in variables. Binding rules are therefore irrelevant in languages that lack formal subroutines: you don't need closures if you don't have formal subroutines.

Note 2: To the best of my knowledge, no language with static (lexical) scope rules has used shallow binding: it's possible to figure out what that combination would do, but it really doesn't make sense. Some languages with dynamic scope rules (e.g., Snobol) offered only shallow binding; others (eg. early Lisp) offered both. Hence, the issues are separable.

Note 3: In a language with lexical scope, the difference (if anybody cared) would only be noticeable for non-local references, that is, references which are neither local nor global. Binding rules would have no relevance to (lexical) local/global references since all local references are always bound to the currently executing instance and there is only one instance of the main program containing the global variables. Binding rules are therefore irrelevant in languages such as C, which lack nested subroutines, or Modula-2, which allow only outermost subroutines to be passed as parameters, and would also be irrelevant in a language with nested subroutines but no recursion (I'm not aware of any like that).
So closures are trivial with static scope and no nested subroutines.

Example of why deep binding matters for static scope (in OCaml):

Scan a list. Return the sum of element k and first negative element (if any) prior to k.

```
let foo l k =
  let rec helper l f i b =
    match l with
    | [] -> raise (Failure "list too short")
    | h :: t ->
        if i = k then f h
        else if (b && h < 0) then helper t (( + ) h) (i + 1) false
             else helper t f (i + 1) b in
  helper l (fun x -> x) 0 true;;
```

This captures the "right" h in `foo [1; -3; 2; -4; 5] 4;;`

Note that the OCaml implementation doesn't use a straightforward application of static links, because of tail recursion optimization.
The equivalent code in Python would:

```
def f(l, k):
    def helper (l, f, i, b):
        if l == []:
            return "list too short"
        elif i == k:
            return f(l[0])
        elif b and l[0] < 0:
            h = l[0]
            return helper(l[1:], (lambda x: h + x), i+1, False)
        else:
            return helper(l[1:], f, i+1, b)

    return helper(l, (lambda x: x), 0, True)

l = [1, -3, 2, -4, 5]
print(f(l, 4))
)
```

We'll return to implementation techniques for scope and binding rules in chapter 9.

NB: object-oriented languages without first-class subroutines can get some of the same effect using ***object closures***: create an object whose fields hold values that would have been in the referencing environment of a subroutine closure; pass the object to somebody; let them invoke one of its methods. The operator() mechanism of C++ makes this look like ordinary subroutine invocation.

Lambda expressions

Function ***aggregates*** -- in-line built-up values of functional type.
Familiar to users of functional languages.
Increasingly common in imperative languages as well.
Require unlimited extent to really work well, as in Ruby, C#, and Scala.
Can still be useful without it, though, as in Java 8 and C++'11.

Here a C++ example:

```
auto plusx(int x) {
      return [x](int y){ return x + y; };
  }
```

This captures x by value. Capturing by reference `[&x]` would not work here, due to limited extent.

==========
Other concepts related to naming

Aliases: using more than one name for the same thing
Problems:
potentially confusing
inhibit code improvement (e.g., promotion to registers)

What are aliases good for?
* linked data structures
x space saving -- modern data allocation methods are better
x multiple representations -- unions are better

Aliases sometime arise in parameter passing as an unfortunate side effect.
Euclid scope rules are designed to prevent this.

Overloading: using the same name for multiple things

Some overloading happens in almost all languages
integer + v. real +
built in I/O operations in some languages
function return in Pascal
Some languages get into overloading in a big way
Ada
C++

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

overloading is also known as "ad hoc polymorphism."

(True) Polymorphism

Means, literally, "having many forms."
In practice, it means "applicable to many types."
There are several different variants.

Simplest is ad hoc polymorphism, which really doesn't deserve the name.

Subtype polymorphism in OO languages allows code to do "the right thing" to parameters of different types in the same type hierarchy
by calling the virtual function appropriate to the concrete type of the actual parameter

Explicit parametric polymorphism (generics)
You specify type parameters when you declare or use the generic.

Templates in C++ are an example of this:

```
typedef set<string>::const_iterator string_handle_t;
set<string> string_map;
   ...
pair<string_handle_t, bool> p = string_map.insert(ident);
```

Here `pair.first` is the string we inserted;
`pair.second` is true iff it wasn't there before

Implemented via macro expansion in C++ v1; built-in in Standard C++.

Similar mechanisms in Clu, Ada, Java, C#, Scala, ...

May be implemented with
- a single copy of the code that always manipulates references
- a separate copy of the code for every (set of similar) type(s)

Implicit (true) parametric polymorphism
You don't have to specify the type(s) for which code works; the language implementation figures it out and won't let you perform operations on objects that don't support them.
Functional languages generally support true parametric polymorphism, either in the runtime system (Lisp and its descendants) or in the compiler (ML and its descendants, inc. Haskell).

More on polymorphism in Chapter 7.