Scoping Rules

How does R know which value to assign to which symbol?

```
> lm <- function(x) (x * x)
> lm
function(x) (x * x)
```

How does R know what value to assign to the symbol `lm` ? Why doesn't it give it the value of `lm` that is in the *stats* package(linear model has a function called `lm`)?

When R tries to bind a value to a symbol, it searches thorough a series of `environments` to find the appropriate value. When we are working on the command line and need to retrieve the value of an R object, the order is roughly.

1. Search the global environment for a symbol name matching the one requested. (Time sequence)
2. Search the numberspace of each of the packages on the search list.

The search list can be found by using `search` function.

```
> search()
 [1] ".GlobalEnv"        "tools:rstudio"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

Binding Values to Symbol

- The *global environment* or the user's workspace is always the first element of the search list and the base package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named c and a function named c.

Scoping Rule

The scoping rules for R are the main feature that make it different from the original S language.

- The scoping rules determine **how a value is associated with free variable in a function.**

- R uses *lexical scoping* or *static scooping*. A common alternative is dynamic scoping.
- Related to the scoping rules is how R uses the search *list* to bind a value to a symbol.
- Lexical scoping turns out to be particularly useful for simplifying statistical computations.

Lexical Scoping

```
f <- function(x, y) {
    x^2 _ y / z
}
```

The function has 2 formal arguments `x` and `y`. In the body of the function there is another symbol `z`. In this case `z` is called a free variable(not defined in the header). The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and not local variables(assigned inside the function body)

Lexical scoping in R means that

> the value of free variables are searched for in the environment in which the function was defined

What is an environment?

- An environment is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value
- Every environment has a parent environment; it is possible for an environment to have multiple "children"
- the only environment without a parent is the empty environment
- A function + an environment = a *closure* or *function closure*

Searching for the value for a free variable

- If the value of a symbol is not found in the environment in wchich a function was defined, then the search is continued in the *parent environment*
- The search continues down the sequence of parent environments until we hit the *top-level environment*, this usually the global environment(work space) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the `empty environemnt`. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Why does all this matter?

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace.
- This behavior is logical for most people and is usually "the right thing" to do
- However, in R you can have function defined *inside other functions*
  - Languages like C don't let you do this
- Now things are getting interesting - in this case the environment in which a function is defined is the body of another function.

```
make.power <- function(n) {
    pow <- function(x) {
        x^n
    }
    pow ## can find the value of n, even if n is a free variable
}
```

This function returns another function as its value.

```
> cube <- make.power(3)
> square <- make.power(2)
> cube(3) ## 3^3
[1] 27
> square(2)## 3^2
[1] [9]
```

What is in a function's environment?

```
> ls(environment(cube))
[1] "n" "pow"
> get("n", environment(cube))
[1] 3

>ls(environment(square))
[1] "n" "pow"
> get("n", envionemnt)
[1] [2]
```

Lexical vs. Dynamic Scoping

```
y <- 10

f <- function(x) {
    y <- 2
    y^2 + g(x)
}

g <- function(x) {
    x*y
}
```

What is the value of (compound function?)

```
f(3)
```

- With lexical scoping the value of `y` in the function `g` is looked up in the environment in which the function was defined, in this case the global environment, so the value of `y` is 10
- With dynamic scoping, the value of `y` is looked up in the environment from which the function was *called* (sometimes referred to as the *calling environment*)
  - In R the calling environment is known as the parent frame
- So the value of `y` would be 2

When a function is *defined* in the global environment an is subsequently *called* from the global environment, then the defining and the calling environment are the same. This can sometimes give the appearance of the dynamic scoping.

```
> q <- function(x) {
+ a <- 3
+ x + a + y ## x is formal argument, a is local variable, y is free variable
+ }
> g(2)
Error in g(2) : object "y" not found
> y <- 3
> g(2)
[1] 8
```

Other languages that support lexical scoping

- Scheme
- Perl
- Python
- Common Lisp(all languages converge to Lisp)

Consequences of Lexical Scoping

- In R, all objects must be stored in memory
- All function must carry a pointer to their respective defining *environments*, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.