2. R Fundamentals

Chuanhai Liu

Department of Statistics, Purdue University

2016

Table of Contents

2.1 Introduction 3
2.2 Data structure 8
2.3 Environments 12
2.4 Expressions 15
2.5 Functions 17
2.6 Lazy evaluation 23
2.7 Package basics 24
2.8 Package development 28
2.9 Exercises 35

Additional References:

1. R Internals, https://cran.r-project.org/doc/manuals/r-release/R-ints.html
2. Writing R extensions, https://cran.r-project.org/doc/manuals/r-release/R-exts.html

2.1 Introduction: what is R?

S [or R] is a programming language and environment for all kinds of computing involving data. It has a simple goal:

To turn ideas into software, quickly and faithfully

— *John M. Chambers*

To understand computations in R, two slogans are helpful:

- *Everything that exists is an object.*
- *Everything that happens is a function call.*

— *John Chambers*

2.1 Introduction: how to work with R?

- You interact with the R system by typing your commond, a function call, using the R language (with a C-like syntax) at the user command prompt '>'
- For help, type the command `?` '?' or `help()`.
- The R interpreter iteratively reads your input commands to
  - Compile your input into R expression,
  - Evaluate the compiled expression, and
  - Return the value.

```
> # ?  str
> str(2)
num 2
> 1 + 2
[1] 3
> # Let's take a closer look at the two internal steps
> str(quote(1+2))
language 1 + 2
> eval(quote(1+2))
[1] 3
> str(expression(1+2))
expression(1 + 2)
> eval(expression(1+2))
[1] 3
```

2.1 Introduction: where are you and what do you have?

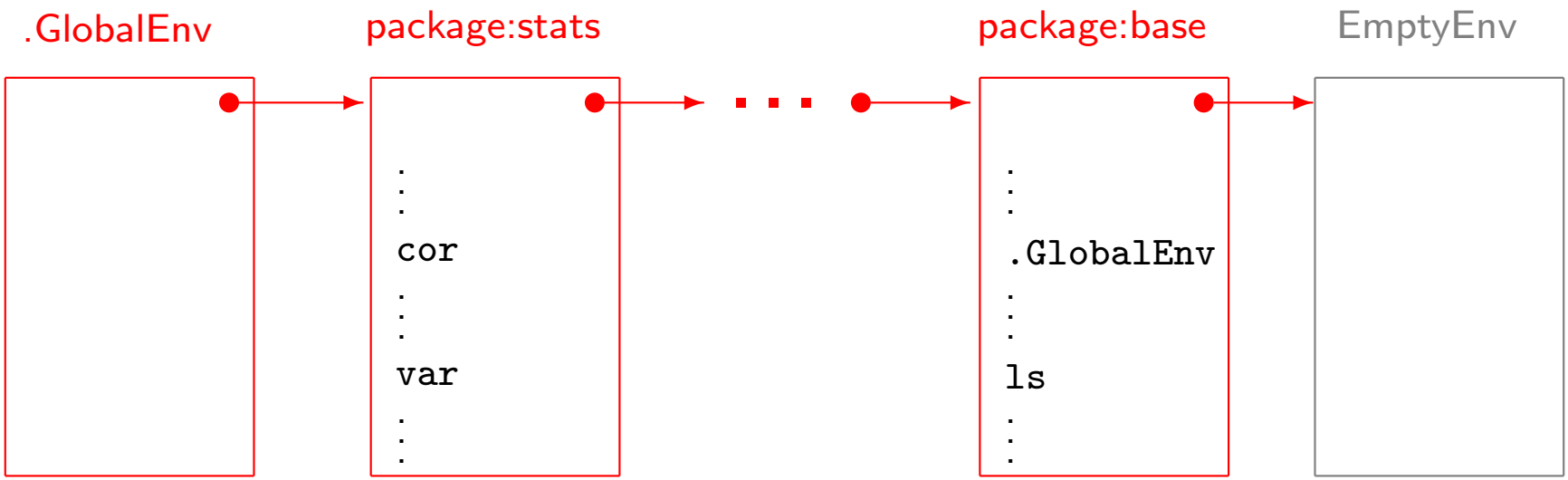
- Function calls are made at some place called *environment*. You can find the current location or environment with the `environment()` function.

  More discussion on `environment` objects is given in the Environments section.
- Environments are linked containers that contain names binded to objects. You can get the linked environments starting with the `.GlobalEnv` environment using the `search()` function.
- You can get all the names in an environment with the `ls()` and `ls.str()` functions.

  For commonly used R functions, see, for example, R Reference Card 2.0 at
  https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf
- You can get help with the `help()` and `?()` functions.

```
> environment()
<environment: R_GlobalEnv>
> search()
[1]          ".GlobalEnv"      "package:stats"   "package:graphics"
[4]  "package:grDevices"       "package:utils"   "package:datasets"
[7]    "package:methods"           "Autoloads"       "package:base"
> # Exercises
> ?  globalenv
> ?  baseenv
> ?  ls
>
```

2.1 Introduction: search path



A schematic representation of the search path from `.GlobalEnv`

See the Package section on search paths in evaluating functions in packages and name spaces.

2.1 Introduction: syntax

Unlike C variables, R variables do not need to be declared and typed. Otherwise, R language syntax is similar to that of the C programming language. For example,

- A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as "'.2way"' are not valid, and neither are the reserved words.
- Binary operators such as `+`, `-`, `*`, `/`, `&&`, and `||` have the same precedence
- It has `if` and `if-else` control flow.
- It has `while` and `for` loop expressions, although the syntax of the `for` expression is different. See `?  'for'`, `?  'break'`, and `?  'repeat'`.

In addition,

- The `try()` function is handy in handling errors: 'try' is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.
- More basic or 'standard library' functions can be found with the `ls(getNamespace("base"))` function call.

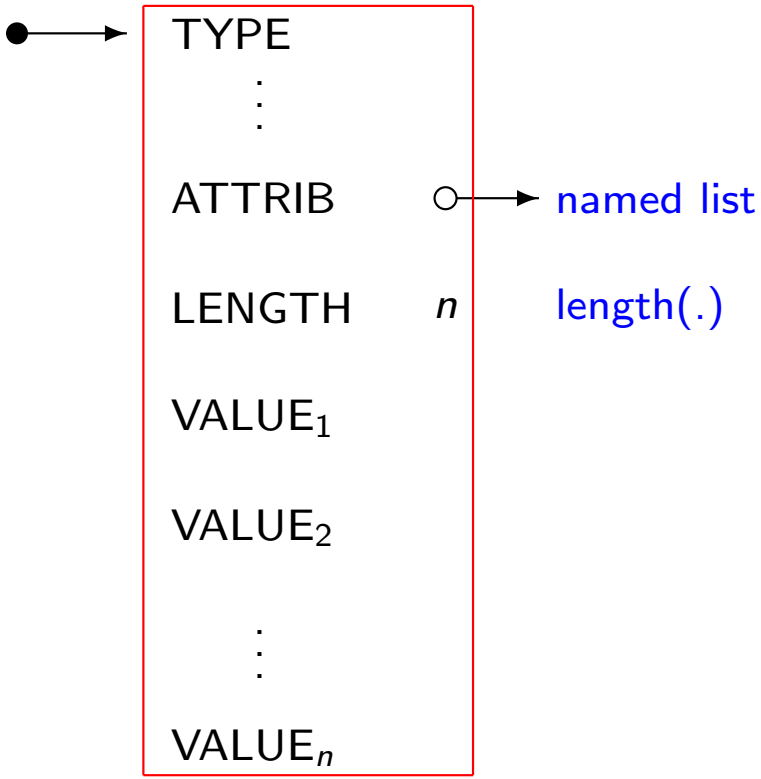
2.2 Data structure: vector-type objects

An object of vector type is an array of objects of the same type. Its length can be obtained using the `length()` function.

```
> length(2)
[1] 1
> typeof(2)
[1] "double"
> mode(2)
[1] "numeric"
> class(2)
[1] "numeric"
> attributes(2)
NULL
> structure(2, names = "C_1")
C_1
  2
> str(structure(2, names = "C_1"))
Named num 2
- attr(*, "names")= chr "C_1"
```

Types: NULL, integer, numeric, complex, character, logical, list, pairlist, expression, function, environment, ...

Internal structure:



A schematic representation

"Standard" attributes: names, class, dim, ...

2.2 Data structure: accessing vector components

Vector components, as sub-vectors, are accessed with the subsetting `'['()` function. Single objects of vectors of objects, which are called `list` objects, are accessed using the index/named-based `'[['()` and name-based `'$'()` functions.

```
> x = 0:9 # ':'(0, 9)
> x
[1] 0 1 2 3 4 5 6 7 8 9
> x[2:5] # '['(x, 2:5)
[1] 1 2 3 4
> names(x) <- letters[1:10]
> x[c("a", "d")]
a d
0 3
> my.list = as.list(x)
> str(my.list)
List of 10
$ a:  int 0
$ b:  int 1
$ c:  int 2
...
> my.list[2]
$b
[1] 1
```

```
> my.list[[2]]
[1] 1
> my.list$b
[1] 1
> d=data.frame(x=2:1, y=c(T, F))
> d
  x     y
1 2  TRUE
2 1 FALSE
> str(d)
'data.frame':  2 obs.  of 2
variables:
$ x:  int 1 2
$ y:  logi TRUE FALSE
> # Exercises
> # A=matrix(1:10,nrow=2,ncol=5)
> # print(str(A)); typeof(A)
> # print(mode(A)); class(A)
> # attributes(A)
```

2.2 Data structure: linked list or pairlist objects

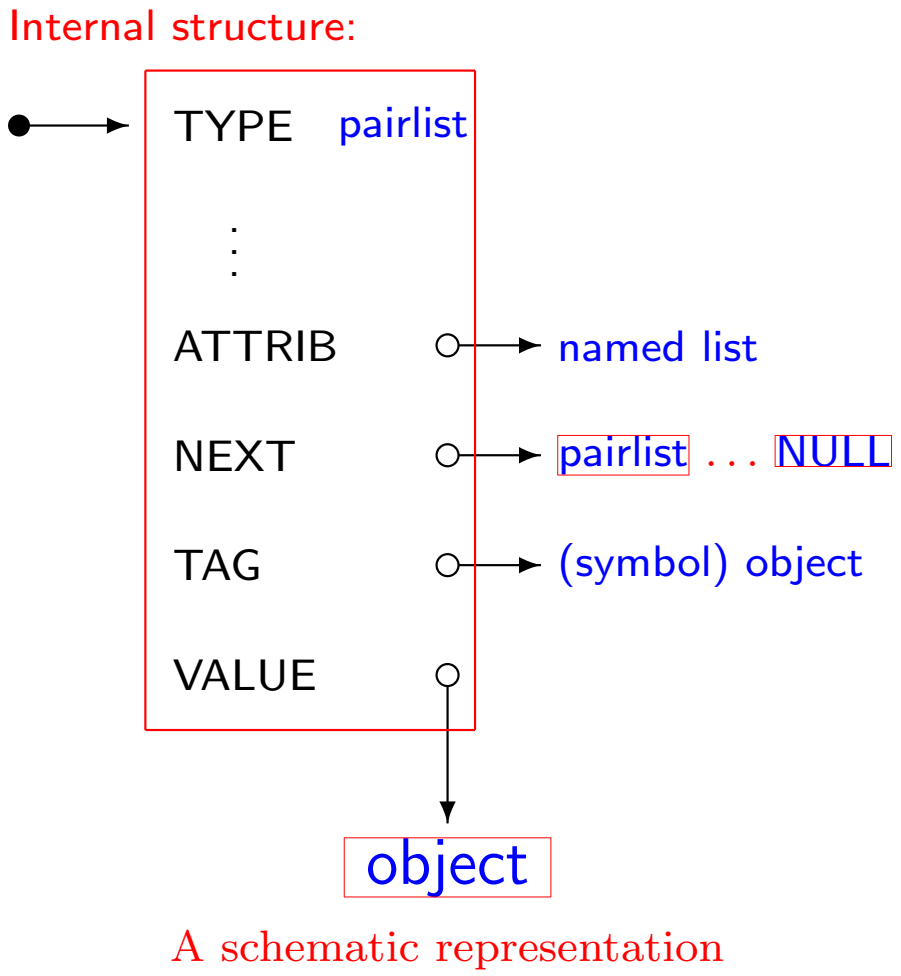
Almost all lists in R internally are *Generic Vectors*, whereas traditional *dotted-pair* lists remain available but rarely seen by users (except as formals of functions).

```
>
> pairlist(a=1:3, b="ABC")
$a
[1] 1 2 3

$b
[1] "ABC"

> str(formals(ls))
Dotted pair list of 5
$ name :  symbol
$ pos :  language -1L
$ envir :  language
as.environment(pos)
$ all.names:  logi FALSE
$ pattern :  symbol

>
```



A schematic representation

2.2 Data structure: constructor, type test, and coercion

Constructor You can create objects of a specific type with a constructor function, e.g., `character()`, `numeric()`, `logical()`, `integer()`, and `vector()`.

Type test You can check if an object is a specific type with an "is" function, e.g., `is.character()`, `is.numeric()`, `is.integer()`, and `is.atomic()`.

Coercion You can coerce objects to some specific type with an "as" function, e.g., `as.character()`, `as.numeric()`, `as.logical()`, and `as.integer()`.

Remark All elements of an atomic vector must be the same type. When combined and operated by some mathematical functions, atomic vectors will be coerced to the most flexible type.

```
> x = logical(5)
> y = as.character(x)
> is.character(y)
[1] TRUE
> y
[1] "FALSE" "FALSE" "FALSE" "FALSE" "FALSE"
> c("A", 1:10)
[1] "A" "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
> # sin(1L)
```

2.3 Environments: binding names to objects

```
> ?  environment
```

Environments consist of

- a *frame* to store a collection of named objects and
- a pointer to an *enclosing environment*.

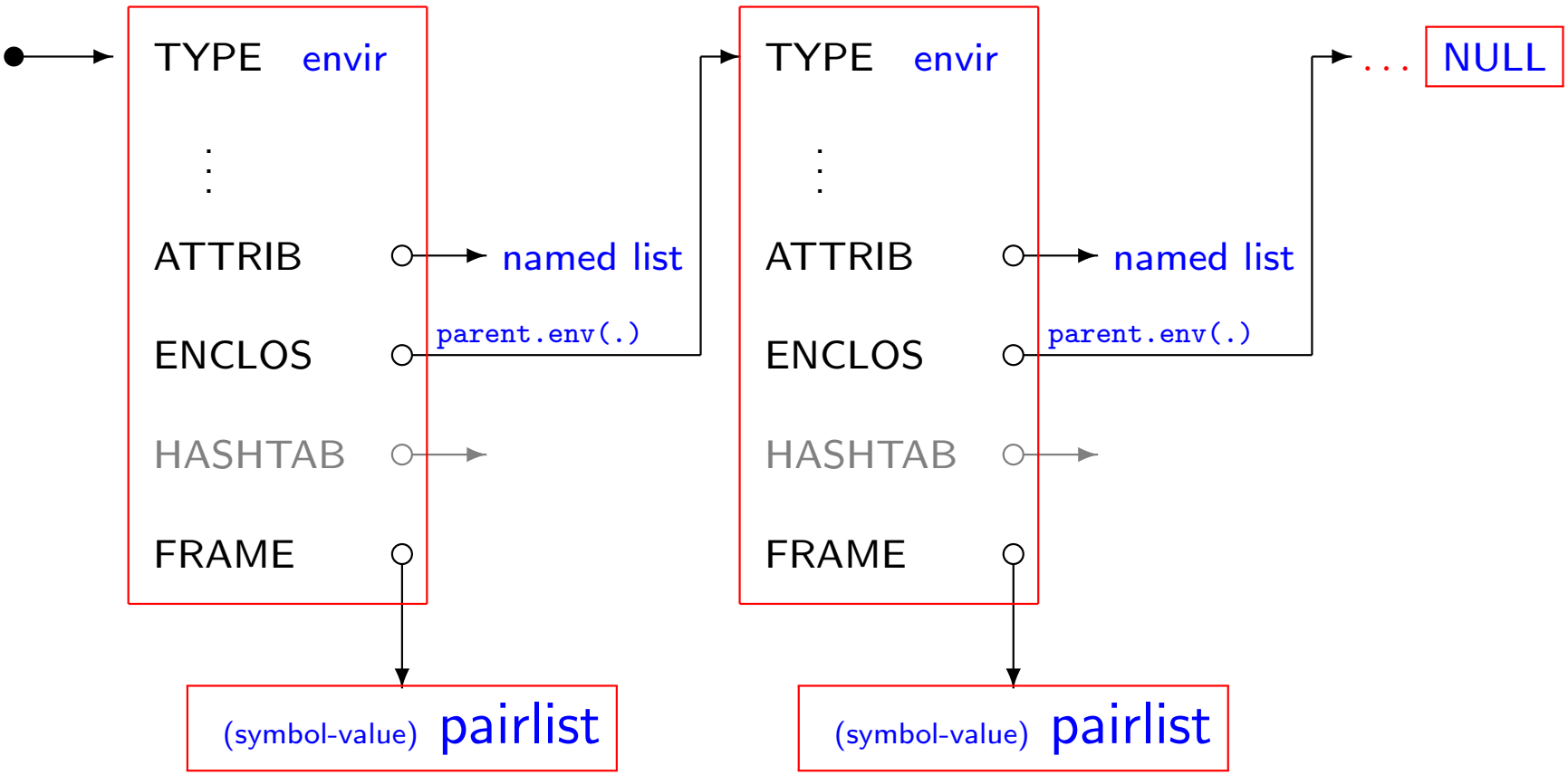
Environments provide a context for you to bind names to objects and to access objects by names.

```
> ?  make.names
```

A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as `.2way` are not valid, and neither are the reserved words.

- Any string enclosed in a pair of backticks ‘ and ‘, such as ‘ `1` ‘, can be used as variable names.
- You bind names to values with the `assign()` function and the `=`, `<-`, and `<<-` binary operators.
- You get the object binded to a name with the name itself, the `get()` function, and the `$` binary operator.
- See also the `globalenv()`, `baseenv()`, `emptyenv()`, `environment()`, `new.env()`, `ls()`, and `exists()` functions.

2.3 Environments: internal structure



A schematic representation of a chain of environments

2.3 Environments: code examples

```
> environment()
<environment: R_GlobalEnv>
> search()
[1]         ".GlobalEnv"      "package:stats"   "package:graphics"
[4]  "package:grDevices"      "package:utils"   "package:datasets"
[7]    "package:methods"          "Autoloads"       "package:base"
> e = new.env()
> parent.env(e)
<environment: R_GlobalEnv>
> e$a = 1:5
> assign("b", sin, env=e)
> ls(e)
[1] "a" "b"
> str(as.list(e))
List of 2
$ a: int [1:5] 1 2 3 4 5
$ b:function (x)
> e$b(1L)
[1] 0.841471
> sin(1L)
[1] 0.841471
>
```

2.4 Expressions

An expression is called an abstract syntax tree (AST) because it represents the hierarchical tree structure of the code.

```
> ex <- expression(x <- 10, y <- x + 1)
> eval(ex)
> y
[1] 11
> str(as.list(ex[[2]]))
List of 3
$ :  symbol <-          # binded to the assignment function of two (3 - 1) arguments
$ :  symbol y
$ :  language x + 1
> str(as.list(ex[[2]][[3]]))
List of 3
$ :  symbol +            # binded to the addition function of two (3 - 1) arguments
$ :  symbol x
$ :  num 1
> str(as.list(ex[[1]]))                                    # = ?  Exercise
```

2.4 Expressions: computing on calls

Creating You can create a call with the `parse()`, `expression()`, `quote()`, `call()`, and `as.call()` functions.

```
> eval(call("mean", quote(1:10)))
[1] 5.5
> eval(as.call(list(quote(mean), quote(1:10))))
[1] 5.5
> eval(as.call(list(as.symbol("mean"), quote(1:10))))
[1] 5.5
> str(as.list(ex[[1]]))                          # = ?  Exercise
```

Modifying You can modify call objects by modifying their components in the same way as modifying list objects.

```
> my.call = parse(text="mean(1:10)")[[1]]
> eval(my.call)
[1] 5.5
> my.call[[2]] = quote(1:3)
> eval(my.call)
[1] 2
> str(as.list(my.call))                          # = ?  Exercise
```

Capturing You can use the `sys.call()` to obtain what the user typed, and the `match.call()` to get the call that uses named arguments; See Functions.

2.5 Functions

Functions are fundamental R objects because you interact with R through function calls. Function objects have three components

Arguments A pairlist, known as the formal arguments. You can get the formal arguments with the `formals()` function.

```
> str(formals(ls))
Dotted pair list of 5
$ name :  symbol
$ pos :  language -1L
$ envir :  language as.environment(pos)
$ all.names:  logi FALSE
$ pattern :  symbol
```

Body A language or call object. You can get the body with the `body()` function.

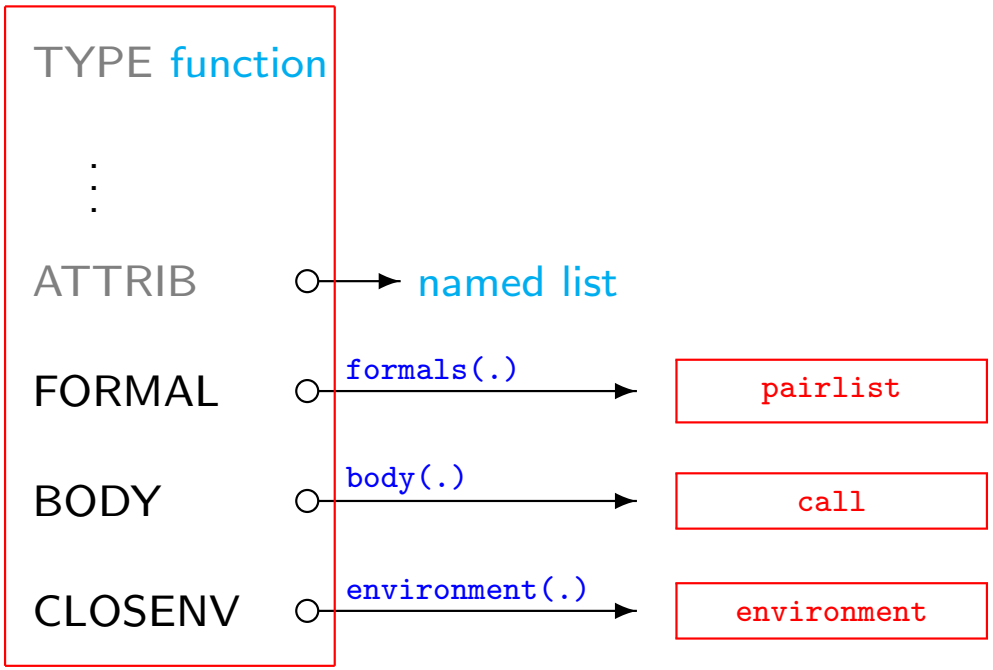
```
> typeof(body(ls))
[1] "language"
```

Environment The context or place to find named objects. You can get the environment with the `environment()` function.

```
> environment(ls)
<environment:  namespace:base>
```

2.5 Functions: internal structure

There are three types of R functions, primitive functions, internal functions, and closures. It is the closure object we discuss here. Its main components are depicted in the following diagram.



A schematic representation of function objects

2.5 Functions: creating and modifying

Creating The syntax of defining a function is given by

```
function( arglist ) expr
```

where `expr` makes a call, which can contains the `return()` function call. When reached, the `return(value)` call returns the `value` object as the value of the function call. Otherwise, the evaluated value of `expr` is the value of the function call.

```
> norm <- function(x) sqrt(sum(x*x))
> norm(1:2)
[1] 2.236068
```

Modifying If necessary, you can modify function objects with the `formals<-()`, `body<-()`, and `environment<-()` functions.

```
> norm
function (x) sqrt(sum(x * x))
> formals(norm) <- alist(x = 1:3)
> norm
function (x = 1:3) sqrt(sum(x * x))
> environment(norm)
<environment: R_GlobalEnv>
> environment(norm) <- emptyenv()
> # Now try norm()
```

2.5 Functions: evaluation

Conceptually speaking, evaluation of function calls proceeds in three steps:

- Create a temporary environment, called the working environment, with the environment of the function as its parent/enclosing environment.
- Evaluate the actual arguments in the calling environment, known as the *parent frame*, and bind the argument names to the evaluated values of their actual arguments in the working environment, via first perfect/exact name matching, then prefix matching, and finally position. See also the Lazy evaluation section.
- Evaluate the function body in the working environment. If not returned as the value of the function call, the temporary working environment is left to be handled by the Garbage collector.

```
> foo = function(x=1:3) environment()
> str(as.list(foo()))
List of 1
$ x:  int [1:3] 1 2 3
> foo = function(x, y=1:3) {
        if(missing(x)) return(environment())
        z = x + y
        return(environment())
    }
> str(as.list(foo())) # or ls.str(foo())
List of 2
$ x:  symbol
$ y:  int [1:5] 1 2 3 4 5
> # str(as.list(foo(10)))                                    = ?  Exercise
```

2.5 Functions: special functions

Here we learn three functions that provide a set of useful tools for defining functions.

`match.call()`	returns a call in which all of the specified arguments are specified by their full names.
`substitute()`	returns the parse tree for the (unevaluated) expression expr, substituting any variables bound in env.
`deparse()`	turns unevaluated expressions into character strings.
See also	Infix functions, such as %*%, `::`, and `:::`.

```
> foo = function(x=1:3) match.call()
> str(foo(x=sin(1:10)))
language foo(x = sin(1:10))
>
> foo = function(x=1:3) substitute(x)
> str(foo(x=sin(1:10)))
language sin(1:10)
> str(deparse(foo(x=sin(1:10))))
chr "sin(1:10)"
> `%+%` = function(a, b) paste0(a, b)
> "New " %+% "string"
"New string"
```

2.5 Functions: an example of simple OO-like programming

The problem Create an environment object that contains two functions:

`has.next()` checks if there is more data available, and
`get.next()` returns the next available data value.

```
> iterator <- function(x) {
        x = as.vector(x)
        has.next <- function() length(x) > 0
        get.next <- function() {
            value = if(is.atomic(x)) x[1] else x[[1]]
            x <<- x[-1]
            value
        }
        structure(environment(), class = "iterator")
    }
> a = iterator(5:1)
> i=0
> while(a$has.next()) {
        i = i + 1; cat(" [" , i, "] ", a$get.next(), sep="")
    }
 [1] 5 [2] 4 [3] 3 [4] 2 [5] 1>
```

See topics on S3 and S4 classes.

2.6 Lazy evaluation

Lazy evaluation means that expressions are only evaluated if they are actually used. Similarly, there is a lazy loading. That is, objects are only loaded if they are actually used. These operations are internally implemented with the promise data structure.

By default, R function arguments are lazy.

```
> foo = function(x) 10
> foo(stop("Whoops, this argument is evaluated"))
[1] 10
```

If you want to ensure that a argument is evaluated you can use the `force()` function.

```
> foo = function(x) {
        force(x) # or simply, x
        10
    }
> foo(stop("Whoops, this argument is evaluated"))
Error in force(x) :  Whoops, this argument is evaluated
```

You can also create promise objects with the `delayedAssign()` function.

```
> delayedAssign("x", {cat("Evaluating\n"); 1+2})
> x
Evaluating
[1] 3
```

2.7 Package basics

You have seen that many environments in the search path starting from `.GlobalEnv` are named `package:name`, where `name` stands for the name of the package, such as `base`, `graphics`, and `stats`.

The R distribution comes with about 30 packages. But the Comprehensive R Archive Network (CRAN) has thousands packages available.

You can even create packages yourself and contribute your packages to CRAN.

Packages basics:

Install You can install packages from CRAN and elsewhere with the `install.packages()` function.

attach You can load and attach packages with the `library()` function.

help You can get help on packages with `package?name`.

```
> library(lattice)
> as.environment("package:lattice")
<environment: package:lattice>
...
> is.function(as.environment("package:lattice")$xyplot)
[1] TRUE
```

2.7 Package basics: name spaces

A `library()` function call does two things: load and attach a package.

Upon loading packages, an environment is created and serves as package's name space. You can access this environment with the `getNamespace()` function. You can access objects in namespaces with the `::` and `:::` functions. Care must be taken in using the `:::` function.

When attached, a separate environment is created and inserted into the global search path, the search path from `.GlobalEnv`. The objects intended for the user, referred to as exports, are exported to this environment. It should be noted that the `detach()` function call only removes the package environment from the global search path.

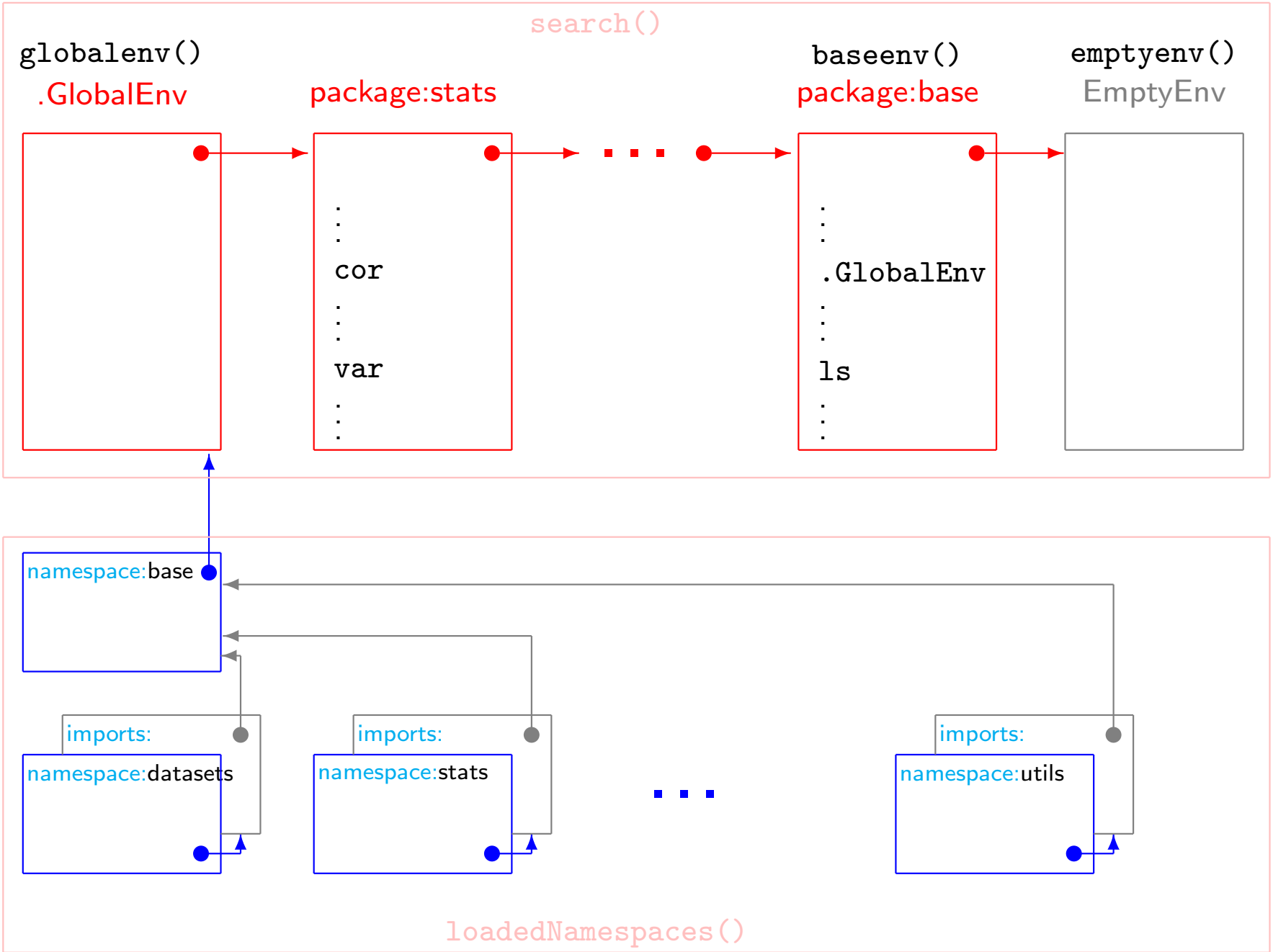
```
> library(lattice)
> as.environment("package:lattice") # get the package environment
<environment:  package:lattice>
attr(,"name")
[1] "package:lattice"
attr(,"path")
[1] ".../library/lattice"
> getNamespace("lattice") # get the namespace environment
<environment:  namespace:lattice>
> detach(lattice)
> environment(lattice::xyplot)
<environment:  namespace:lattice>
```

2.7 Package basics: search paths from name spaces

What are the search paths from loaded name spaces? The following code provides an answer, in terms of the search path for a typical name space.

```
> loadedNamespaces()
[1]     "base"   "datasets"   "graphics"  "grDevices"      "grid"
[6]  "lattice"    "methods"      "stats"      "tools"    "utils"
> parents <- function(env = globalenv()) {
          Name = paste0(if(isNamespace(env)) "namespace:" else "",
              environmentName(env))
          if(identical(env, emptyenv()))
              Name
          else
              c(Name, parents(parent.env(env)))
      }
> parents(getNamespace("lattice"))
 [1]  "namespace:lattice"  "imports:lattice"   "namespace:base"
 [4]        "R_GlobalEnv"    "package:stats" "package:graphics"
 [7]  "package:grDevices"    "package:utils" "package:datasets"
[10]    "package:methods"        "Autoloads"             "base"
[13]         "R_EmptyEnv"
>
```

2.7 Package basics: where are your objects



2.8 Package development

A *package* that can be loaded into R sessions is a directory of files which extend R. Thus package development is to create package directories. See ≪Writing R Extensions≫ for more information.

Here we discuss a simple step-by-step process to build a simple package on a Unix-type machine.

The objective:

> Create a package that contains objects to simulate data from the so-called *robit* linear regression model
>
> $$y_i = \text{sign}(x_i'\beta + e_i) \qquad (i = 1, ..., n)$$
>
> where the error terms e_i are iid with $e_i \sim t(0, 1, \nu)$, the standard Student-t distribution with ν degrees of freedom.

The objects:

`rt()`	the `rt()` function to be imported from `namespace:stats`.
`rnorm()`	the `rnorm()` function to be imported from `namespace:stats`.
`X.default()`	the `X.default(p, n)` function for creating the default design matrix, as an internal function.
`rrobit()`	the main function, `rrobit(beta, n, X = X.default, df = 7)`, to be created and exported to the user.

2.8 Package development (1): code your functions

```
> # rm(list=ls(all=TRUE))
> # getwd()
> X.default <- function(p, n) {
        p <- as.integer(p)
        n <- as.integer(n)
        if(p < 1) stop("'p' must be a positive integer")
        X <- cbind(rep(1, n), matrix(rnorm(n*(p-1)), nrow=n))
        dimnames(X) <- list(as.character(1:n),
                paste("X",as.character(0:(p-1)),sep="_"))
        X
    }
> rrobit <- function(beta, n, X = X.default, df = 7) {
        beta = as.numeric(beta)
        n = as.integer(n)
        X = if(is.function(X)) X.default(length(beta), n) else
                as.matrix(X)
        Z = as.numeric(X %*% beta) + rt(dim(X)[1], df = df)
        data.frame(X, Y = 2*((Z > 0) - 0.5 ))
    }
> ls.str()
rrobit :  function (beta, n, X = X.default, df = 7)
X.default :  function (p, n)
```

2.8 Package development(2): create package skeleton

```
> ?package.skeleton
   'package.skeleton' automates some of the setup for a new source
   package.  It creates directories, saves functions, data, and R
   code files to appropriate places, and creates skeleton help files
   and a Read-and-delete-me file describing further steps in
   packaging.

   package.skeleton(name = "anRpackage", list,
         environment = .GlobalEnv,
         path = ".", force = FALSE,
         code_files = character())
         ...
> package.skeleton("robit", list=c("rrobit", "X.default"))
Creating directories ...
...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './robit/Read-and-delete-me'.
> system("ls ./robit")
DESCRIPTION         R                   man
NAMESPACE           Read-and-delete-me
```

2.8 Package development(3): edit DESCRIPTION

Use your editor to edit the generated ./robit/DESCRIPTION file:

```
> system("cat ./robit/DESCRIPTION")
Package:  robit
Type:  Package
Title:  What the package does (short line)
Version:  1.0
Date:  2016-05-09
Author:  Who wrote it
Maintainer:  Who to complain to <yourfault@somewhere.net>
Description:  More about what it does (maybe more than one line)
License:  What license is it under?
```

2.8 Package development(4): edit NAMESPACE

Use your editor to edit the generated ./robit/NAMESPACE file:

```
> system("cat ./robit/NAMESPACE")
# exportPattern("^[[:alpha:]]+")

importFrom(stats, rnorm, rt)

export(rrobit)
```

2.8 Package development(5): edit the help files

Use your editor to edit the generated help files in ./robit/man.

Help See ≪Writing R Extensions≫.

Check Use the `tools::parse_Rd()` `tools::Rd2txt()` functions.

```
> library(tools)
> # Rd2txt(parse_Rd("./robit/man/robit-package.Rd"))
> Rd2txt("./robit/man/robit-package.Rd")
...
> # Rd2txt(parse_Rd("./robit/man/X.default.Rd"))
> Rd2txt("./robit/man/X.default.Rd")
...
> # Rd2txt(parse_Rd("./robit/man/rrobit.Rd"))
> Rd2txt("./robit/man/rrobit.Rd")
...
```

2.8 Package development(6): check, build, and install

```
> system("R CMD check robit")
...
> system("R CMD build robit")
...
> # system("R CMD REMOVE robit")
> system("R CMD INSTALL robit")
...
> library(robit)
> # rm(list = ls(all = T))
> ls.str(as.environment("package:robit"))
rrobit :  function (beta, n, X = X.default, df = 7)
> ls.str(getNamespace("robit"))
rrobit :  function (beta, n, X = X.default, df = 7)
X.default :  function (p, n)
> ls.str(parent.env(getNamespace("robit")))
rnorm :  function (n, mean = 0, sd = 1)
rt :  function (n, df, ncp)
> parent.env(parent.env(getNamespace("robit")))
<environment:  namespace:base>
> package ?  rrobit
> ?  rrobit
> rrobit(c(0, 4, 0), n=20)
...
```

2.9 Exercises

1. Install the pry-R package, `pryr`, from CRAN and use it to explore internal structures of different types of R objects.
2. Create a simple package of your choice by following Friedich Leisch (2009).

Friedich Leisch (2009). Creating R Packages: A Tutorial,
https://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf

— TO BE CONTINUED