Module 6 (Advanced R programming)

Contents

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN SOCIAL FUND

Project co-financed by the European Union under the European Social Fund

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund

1 Environments

1.1 Introduction

In this section we will discuss yet another basic recursive type, `environment`. Each object of type `environment` consists of:

- a **frame** (collection of named objects),
- a pointer to an **enclosing environment**.

1.2 Frames

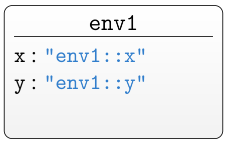
Firstly, we will focus only on the frame-part of an environment. The `new.env()` function creates a new environment.

```
e <- new.env()
print(e)
## <environment: 0x3939948>
typeof(e)
## [1] "environment"
class(e)
## [1] "environment"
is.recursive(e)
## [1] TRUE
is.vector(e)
## [1] FALSE
```

Each frame acts like a **set of named objects**. Among useful functions we find:

- `assign()` – inserts a named object into a frame,
- `get()` – extracts a named object,
- `ls()` – lists named objects,
- `exists()` – checks if a named object exists,
- `rm()` – removes a named object.

```
env1 <- new.env()
assign("x", "env1::x", envir = env1)
assign("y", "env1::y", envir = env1)
length(env1)
## [1] 2
ls(envir = env1)
## [1] "x" "y"
```



Accessing elements:

```
get("x", envir = env1)
## [1] "env1::x"
env1$x
## [1] "env1::x"
env1[["x"]]
## [1] "env1::x"
env1[["nosuchelement"]]
## NULL
get("nosuchelement", envir = env1)
## Error:  object 'nosuchelement' not found
```

Other methods to assign new objects:

```
env1$z <- "env1::z"
ls(env1)
## [1] "x" "y" "z"
env1[[".w"]] <- "env1::.w"
ls(env1)
## [1] "x" "y" "z"
ls(env1, all.names = TRUE)
## [1] ".w" "x"  "y"  "z"
```

Note that names starting with a dot are “hidden”. Let us get rid of “`.w`”:

```
rm(".w", envir = env1)
exists(".w", envir = env1)
## [1] FALSE
```

1.2.1 Frames vs named lists

Frames, just like lists, may be used to **store objects of any type**. In fact, it is easy to convert an environment to a named list and vice versa:

```
str(as.list(env1))
## List of 3
##  $ x: chr "env1::x"
##  $ y: chr "env1::y"
##  $ z: chr "env1::z"
(test <- list2env(list(a = 1, b = mean, c = TRUE)))
## <environment: 0x4046190>
ls(test)
## [1] "a" "b" "c"
test$b
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x4045810>
## <environment: namespace:base>
```

There are, however, substantial differences:

1. Elements in a frame are **not numbered**:

   ```
   env1[[1]]

   ## Error:  wrong arguments for subsetting an environment

   env1[1]

   ## Error:  object of type 'environment' is not subsettable
   ```

   A list represents a sequence. On the other hand, an environment’s frame rather a dictionary- or set-like data structure.

2. Each name in a frame must be **unique**:

   ```
   str(list(a = 1, b = 2, a = 3))

   ## List of 3
   ##  $ a: num 1
   ##  $ b: num 2
   ##  $ a: num 3
   ```

```
test <- list2env(list(a = 1, b = 2, a = 3))
ls(test)

## [1] "a" "b"

test$a

## [1] 3
```

3. Time complexity of searching for a named element in a list is $O(n)$. Frames base upon a **hash table** gives amortized $O(1)$.

```
L <- structure(as.list(1:10000), names=paste0("X", 1:10000))
E <- list2env(L)
microbenchmark::microbenchmark(
   L$X1, L$X100, L$X10000, E$X1, E$X100, E$X10000)

## Unit: nanoseconds
##      expr    min       lq   median       uq     max neval
##      L$X1    338    575.0    904.0   1309.5    4309   100
##    L$X100   4769   5144.5   6085.5   8950.5   53048   100
##  L$X10000 447302 453194.5 466690.0 752565.5 1174221   100
##      E$X1    398    793.0   1210.5   1886.5   25258   100
##    E$X100    328    691.0   1023.0   1627.0    4767   100
##  E$X10000    339    680.5    928.5   1549.0    5698   100
```

4. Environments acting as a function's arguments, are passed “like” **by reference**:

```
add1x <- function(o) o$x <- o$x + 1
```

```
o1 <- list(x=1)
add1x(o1)
print(o1$x) # pass by value

## [1] 1
```

```
o2 <- list2env(o1)
add1x(o2)
print(o2$x) # pass by reference

## [1] 2
```

Exercise: *Write a function* `words2()` *that reads the contents of a given text file,* `fname`*. The function should return a data frame with 2 columns:*

- `word` *– all the unique words occurring in the text file,*
- `count` *– total number of occurrences of each corresponding word.*

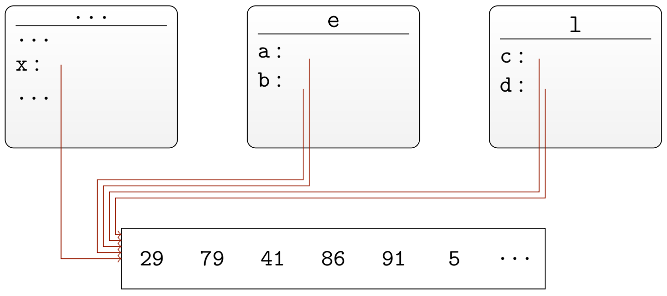
The data frame should be sorted from the most to the least common word.

The function should be able to summarize a very large text file (use connections, read file in chunks, use an environment to count the words).

Note some R memory management issues:

```
x <- sample(1:100)
e <- new.env()
e$a <- x; e$b <- x
l <- list()
l$c <- x; l$d <- x
.Internal(inspect(x))
## @1f21720 13 INTSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 29,79,41,86,91,...
.Internal(inspect(e$a))
## @1f21720 13 INTSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 29,79,41,86,91,...
.Internal(inspect(e$b))
## @1f21720 13 INTSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 29,79,41,86,91,...
.Internal(inspect(l$c))
## @1f21720 13 INTSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 29,79,41,86,91,...
.Internal(inspect(l$c))
## @1f21720 13 INTSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 29,79,41,86,91,...
```

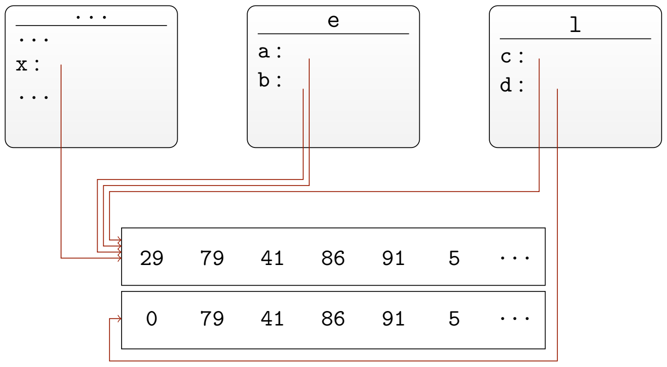
There is only one vector in RAM here. `x`, `e$a`, `e$b`, `l$c`, `l$d` **points at** the same address in RAM.



By the way, operations on pointers are not computationally costly. Copying an address is equivalent to copying an integer.

Copy only is done on demand:

```
l$d[1] <- 0
.Internal(inspect(l$c))
## @1f21720 13 INTSXP g1c7 [MARK,NAM(2)] (len=100, tl=0) 29,79,41,86,91,...
.Internal(inspect(l$d))
## @22eb3b0 14 REALSXP g0c7 [NAM(1)] (len=100, tl=0) 0,79,41,86,91,...
```



1.3 Enclosing environments

Let us study **enclosing environments**.

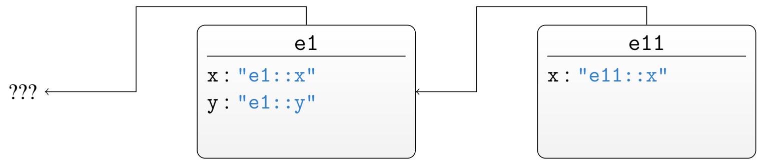
```
(e1 <- new.env())
## <environment: 0x3b814c8>
(e11 <- new.env(parent = e1))
## <environment: 0x3cfbaa8>
parent.env(e11)  # e1
```

```
## <environment: 0x3b814c8>
```

We add some elements:

```
assign("x", "e1::x", envir = e1)
assign("y", "e1::y", envir = e1)
assign("x", "e11::x", envir = e11)
```

Now we have:



Let us read some values:

```
get("x", envir = e1)
## [1] "e1::x"
get("x", envir = e11)
## [1] "e11::x"
get("y", envir = e1)
## [1] "e1::y"
```

Note that there is no `y` in `e11`:

```
get("y", envir = e11)
## [1] "e1::y"
```

However, a value from `e1` was returned. If we look at the man page of the `get()` function, we will find that its `inherits` argument defaults to `TRUE`.

```
get("y", envir = e11, inherits = FALSE)
## Error: object 'y' not found
e11$y  # inherits=FALSE
## NULL
e11[["y"]]  # inherits=FALSE
## NULL
```

Thus, if `inherits` is `TRUE` and a name is not found, an enclosing environment will be examined.

1.3.1 Search path

We surely wonder what is the enclosing environment (if any) of `e1`.

```
parent.env(e1)
## <environment: R_GlobalEnv>
```

It is the **global environment**, also known as the *user's workspace*. Intuitively, it is an environment used to store all the objects we create "in the R console".

```
exists("e1", envir = globalenv(), inherits = FALSE)
## [1] TRUE
exists("e11", envir = globalenv(), inherits = FALSE)
## [1] TRUE
```

Note that:

```
exists("mean", envir = e11, inherits = FALSE)
## [1] FALSE
```

However. . .

```
exists("mean", envir = e11, inherits = TRUE)
## [1] TRUE
```

```
exists("mean", envir = e1, inherits = FALSE)
## [1] FALSE
```

```
exists("mean", envir = globalenv(), inherits = FALSE)
## [1] FALSE
```

Let us examine the enclosing environment of the global environment (and all its predecessors, . . .)

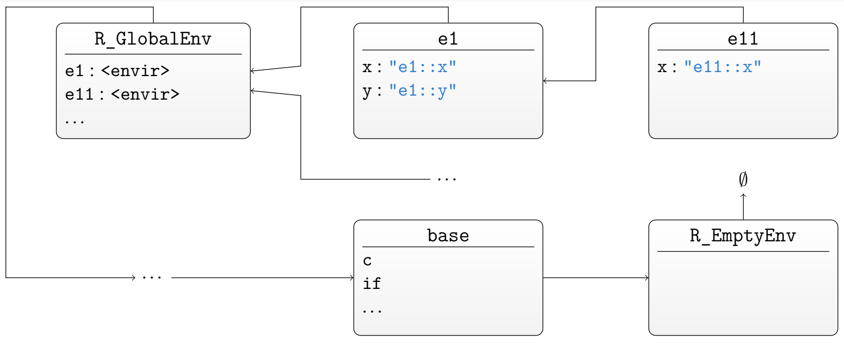
```
etmp <- globalenv()
repeat {
    cat(format(etmp), "(", attr(etmp, "name"), ")\n")
    etmp <- parent.env(etmp)
}
## <environment: R_GlobalEnv> ( )
## <environment: package:tikzDevice> ( package:tikzDevice )
## <environment: package:filehash> ( package:filehash )
## <environment: package:methods> ( package:methods )
## <environment: package:knitr> ( package:knitr )
## <environment: package:stats> ( package:stats )
## <environment: package:graphics> ( package:graphics )
## <environment: package:grDevices> ( package:grDevices )
## <environment: package:utils> ( package:utils )
## <environment: package:datasets> ( package:datasets )
## <environment: 0x211c7f8> ( Autoloads )
## <environment: base> ( )
## <environment: R_EmptyEnv> ( )
## Error:  the empty environment has no parent
```

It is the **base environment** in which all the “base” functions are “defined”:

```
sapply(c("mean", "c", "as.character", "for"), exists,
   envir=baseenv(), inherits=FALSE)
##         mean            c as.character          for
##         TRUE         TRUE         TRUE         TRUE
length(ls(baseenv()))
## [1] 1177
length(ls(emptyenv()))
## [1] 0
```

The **seach path** may also be determined by calling `search()`:

```
cat(stringi::stri_wrap(paste(search(), collapse=", ")), sep="\n")
## .GlobalEnv, package:tikzDevice, package:filehash, package:methods,
## package:knitr, package:stats, package:graphics, package:grDevices,
## package:utils, package:datasets, Autoloads, package:base
```



Each name is stored in some frame. When we type in the console:

```
mean
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x4045810>
## <environment: namespace:base>
```

R in facts does:

```
get("mean", envir = globalenv(), inherits = TRUE)
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x4045810>
## <environment: namespace:base>
```

By the way, "`envir=globalenv()`" is for simplicity here only, see below.

Before:

```
exists("stri_reverse")
## [1] FALSE
search()[1:4]
## [1] ".GlobalEnv"         "package:tikzDevice" "package:filehash"
## [4] "package:methods"
```

After attaching a package:

```
library("stringi")
exists("stri_reverse")
## [1] TRUE
search()[1:4]
## [1] ".GlobalEnv"         "package:stringi"    "package:tikzDevice"
## [4] "package:filehash"
```

Let us note the `attach()` function:

```
search()[1:4]
## [1] ".GlobalEnv"         "package:stringi"    "package:tikzDevice"
## [4] "package:filehash"
test <- list(test1 = 1, test2 = 2)
attach(test)
search()[1:4]
## [1] ".GlobalEnv"         "test"               "package:stringi"
## [4] "package:tikzDevice"
test1
## [1] 1
detach(test)
search()[1:4]
## [1] ".GlobalEnv"         "package:stringi"    "package:tikzDevice"
## [4] "package:filehash"
```

Be aware of the possibility of name shadowing:

```
"+" <- function(x, y) x * y
exists("+", envir = globalenv(), inherits = FALSE)
## [1] TRUE
exists("+", envir = baseenv(), inherits = FALSE)
## [1] TRUE
10 + 2  # globalenv
## [1] 20
rm("+")
10 + 2  # baseenv
## [1] 12
```

Moreover:

```
c <- sum
c(1, 2, 3)
## [1] 6
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund

```
rm(c)
```

However:

```
c <- 100
c(1, 2, 3)
## [1] 1 2 3
get("c", mode = "function")
## function (..., recursive = FALSE)  .Primitive("c")
rm(c)
```

1.3.2 Namespaces

Namespaces are environments associated with packages. A package `PKG` defines 2 environments:

- `package:PKG` – exported, user-visible names; can be attached,
- `namespace:PKG` – all objects defined in `PKG`'s source files (some are for `PKG`'s internal use only).

Let us focus on the `TurtleGraphics` package. The `turtle_move()` function calls `.turtle_check()`.

```
.turtle_check
## Error:  object '.turtle_check' not found
TurtleGraphics::.turtle_check  # package:TurtleGraphics
## Error:  '.turtle_check' is not an exported object from 'namespace:TurtleGraphics'
TurtleGraphics:::.turtle_check  # namespace:TurtleGraphics
## function()
## {
##    if (!exists("width", envir=.turtle_data))
##       stop("The turtle has not been initialized, please call turtle_init() first.")
## }
## <environment: namespace:TurtleGraphics>
```

Note that a package's namespaces are locked for any modification.

1.4 Summary

An environment consists in a frame (a hash table) and a pointer to an enclosing environment. Environments are passed to functions "by reference". Each named object we may access in the R console may be found in some environment.

1.5 Bibliography

- R Core Team, *R internals*, 2014, Sec. 1.2
- R Core Team, *An introduction to R*, 2014, Sec. 13
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 17

2 Computing on the language

2.1 Introduction

R belongs to a class of programming languages in which subroutines have the ability to modify or construct other subroutines and evaluate the result as an integral part of the language itself.

This is similar to Lisp and Scheme and other languages of the "functional programming" variety (...). The Lisp family takes this feature to the extreme by the "everything is a list" paradigm in which there is no distinction between programs and data.

(...) R presents a friendlier interface to programming than Lisp does, at least to someone used to mathematical formulas and C-like control structures, but the engine is really very Lisp-like.

R allows direct access to parsed expressions and functions and allows us to alter and subsequently execute them, or create entirely new functions from scratch.

[R Core Team, R Language Definition, Sec. 6]

2.2 Language objects

Syntactically, we may distinguish the following types of **R expressions**:

- **simple (irreducible) expressions**:
  - **constants**, e.g. `1`, `2L`, `0x1fb0`, `.4e-1`, `32.9i`, `"string"`, `FALSE`, `NA`, `NULL`, `Inf`;
  - **names**;
- **compound expressions**, each of which is a combination of $n+1$ expressions, $n \geq 0$, of the form:

$$\langle \mathtt{f}, \mathtt{e}_1, \ldots, \mathtt{e}_n \rangle.$$

Note that $\langle \mathtt{f}, \mathtt{e}_1, \ldots, \mathtt{e}_n \rangle$ means: apply `f` (call, operation) on $\mathtt{e}_1, \ldots, \mathtt{e}_n$ (arguments). Thus, $\langle \mathtt{f}, \mathtt{e}_1, \ldots, \mathtt{e}_n \rangle$ may also be written as $\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$

There are 3 types of **language objects** in R:

- `name` (or. `symbol`),
- `call`,
- `expression`.

They represent (or are used for this purpose) **unevaluated R expressions**.

2.2.1 Parser

R language objects are created by the R **parser** (interpreter). Each "command" we input e.g. in the console is just a string. The parser must process (interpret) such a string. This is done in two steps:

- **lexical analysis** (lexical rules → tokens),
- **syntactic analysis** (grammar rules → unevaluated R expressions).

See e.g. Aho A.V. et al, *Compilers: Principles, techniques, and tools*, Pearson, 2006 and R Core Team, *R Language Definition*, Sec. 10.

Then such an expression may be **evaluated**, but let us not go that far now.

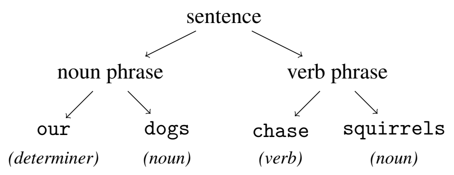
An example – a source code:

```
Our dogs chase squirrels. # cute little doggies!
```

Intuitively, **lexical analysis** finds out that we have 1 sentence here, `Our dogs chase squirrels`, with:

`our`	`dogs`	`chase`	`squirrels`
(determiner)	*(noun)*	*(verb)*	*(noun)*

On the other hand, **syntactic analysis** results in a **syntax tree**:



Thus, we have:

$$\langle \mathtt{chase}, \mathtt{dogs}, \mathtt{our}, \mathtt{squirrels} \rangle$$

To parse a string (perhaps containing a textual representation of an R expression), we call `parse()`:

```
parse(text="c(1, 2, 3)")
## expression(c(1, 2, 3))
parse(text="
   c(1,       2,
3); 5-> x")
## expression(c(1, 2, 3), x <- 5)
parse(text="c(1,2,3") # syntax error
## Error:  <text>:2:0:  unexpected end of input
## 1:  c(1,2,3
##    ^
```

The parse() function returns a parsed but unevaluated expression.

```
x <- parse(text = "sin(3.14); y; 4")
typeof(x)
## [1] "expression"
class(x)
## [1] "expression"
is.language(x)
## [1] TRUE
is.recursive(x)
## [1] TRUE
is.vector(x)
## [1] TRUE
is.list(x)
## [1] FALSE
```

expression denotes a recursive vector-like object. An expression object is a sequence of *expressions* (simple or compound ones):

```
x <- parse(text = "sin(3.14); y; 4")
length(x)
## [1] 3
x[[1]]
## sin(3.14)
as.list(x)
## [[1]]
## sin(3.14)
##
## [[2]]
## y
##
## [[3]]
## [1] 4
```

Let us study the following object:

```
x <- parse(text = "sin(3.14); y; 4")
```

Here is some basic information on elements of x:

```
##           typeof     mode      class
## sin(3.14) "language" "call"    "call"
## y         "symbol"   "name"    "name"
## 4         "double"   "numeric" "numeric"
```

Moreover:

```
##           is.language is.recursive is.atomic is.vector
## sin(3.14)        TRUE         TRUE     FALSE     FALSE
## y                TRUE        FALSE     FALSE     FALSE
## 4               FALSE        FALSE      TRUE      TRUE
```

Simple expressions are not really *syntactically* interesting:

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
(n <- as.name("x"))
## x
class(n)
## [1] "name"
as.name(c("a", "b"))  # must be a single string
## a
```

By the way, each unevaluated R expression may be *deparsed*, i.e. converted into a character vector:

```
test <- parse(text = "(function(x) { '+'(x, 2) -> y; y }) -> f")
deparse(test[[1]])
## [1] "f <- (function(x) {" "    y <- x + 2"       "    y"
## [4] "})"
cat(deparse(test[[1]]), sep = "\n")
## f <- (function(x) {
##     y <- x + 2
##     y
## })
```

2.2.2 Quoting expressions

If we input an expression in the R console, it will be parsed and evaluated automatically:

```
sin(3.14)
## [1] 0.001592653
```

To prevent R from evaluating it, we call e.g.:

```
quote(sin(3.14))
## sin(3.14)
expression(sin(3.14))
## expression(sin(3.14))
expression(sin(3.14))[[1]]
## sin(3.14)
```

2.2.3 Calls, i.e. compound expressions

Let us discuss **call** objects. They represent *compound expressions*. Recall that: $\mathtt{f}(\mathtt{e}_1, \dots, \mathtt{e}_n)$ is equivalent to $\langle \mathtt{f}, \mathtt{e}_1, \dots, \mathtt{e}_n \rangle$, where $\mathtt{f}, \mathtt{e}_1, \dots, \mathtt{e}_n$ are some expressions (simple or compound).

`call`s are sequences of language objects. The first expression, `f`, plays a special role:

```
as.call(expression(f, 1, 2, 3))  # <f,1,2,3> -> f(1, 2, 3)
## f(1, 2, 3)
as.call(expression(function(x) x^2, 1:5))  # here f - compound
## (function(x) x^2)(1:5)
as.call(expression(1, 2, 3))  # that's syntactically valid
## 1(2, 3)
```

Even though `!is.vector(`*some_call*`)`, we may use "`[[`".

```
test <- quote(
   if (TRUE) { x <- 7; print(x)} else print(2+3)
)
mode(test)
## [1] "call"
test[[1]]
## `if`
str(as.list(test), give.attr=FALSE)
## List of 4
##  $ : symbol if
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund

```
##  $ : logi TRUE
##  $ :length 3 {  x <- 7; print(x) }
##  $ : language print(2 + 3)
sapply(test, mode)
## [1] "name"    "logical" "call"    "call"
```

We have: $\langle$`if`$,$ `TRUE`$, \{$`x <- 7`$;$`print(x)`$\},$ `print`$(2+3)\rangle$.

```
test <- quote(
   if (TRUE) { x <- 7; print(x)} else print(2+3)
)
test[[3]]
## {
##     x <- 7
##     print(x)
## }
mode(test[[3]])
## [1] "call"
test[[3]][[1]]
## `{`
test[[3]][[2]]
## x <- 7
test[[3]][[3]]
## print(x)
```

and so on.

Our expression may be written as:

$$\texttt{if (TRUE) \{ x <- 7; print(x) \} else print(2+3)}$$

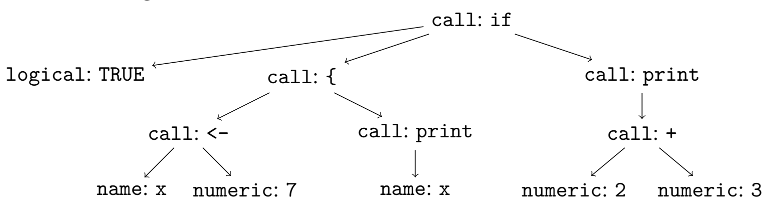
$$\equiv$$

$$\langle\texttt{if, TRUE, \{ x <- 7; print(x) \}, print(2+3)}\rangle$$

$$\equiv$$

$$\Big\langle \texttt{if}, \texttt{TRUE}, \langle \{, \langle \texttt{<-}, \texttt{x}, 7\rangle, \langle \texttt{print}, \texttt{x}\rangle\rangle, \langle \texttt{print}, \langle +, 2, 3\rangle\rangle \Big\rangle$$

Its syntax tree has the following form:



Exercise: *Implement the* `parse.tree.print()` *function. Input: an R expression* `e`*. Result: "image" of* `e`*'s parse tree printed on the console. Hint: use recursion.*

Here is another way to create a call:

```
call("f", 2 + 2, sin(pi))
## f(4, 1.22464679914735e-16)
call("f", quote(2 + 2), quote(sin(pi)))
## f(2 + 2, sin(pi))
```

`substitute()` allows to access a function's argument as-is:

```
f <- function(x) {
    list(substitute(x), deparse(substitute(x)), x)
}
```

```
str(f(2 + 2))
## List of 3
##  $ : language 2 + 2
##  $ : chr "2 + 2"
##  $ : num 4
```

By the way, the `do.call()` function creates and evaluates a call.

```
do.call("sum", list(1, 2, 3))
## [1] 6
do.call("sapply", list(1:4, get("^"), 2))
## [1]  1  4  9 16
```

2.3 Evaluation

An unevaluated R expression has no *meaning*. In a sentence "She is so clever!" – What is "She"? Is "clever" sarcastic? Does this mean that "Sophie solved the P=NP problem"? Or maybe that "Anne tried to fix the laptop herself"?

Here is an unevaluated expression:

```
exp <- quote(2 + 3)
```

In a first context, it may mean:

```
eval(exp)
## [1] 5
```

However, in the second context, its meaning can be totally different:

```
"+" <- get("*", envir = baseenv())
eval(exp)
## [1] 6
rm("+")
```

`name` objects are always **context-dependent**. Each evaluation takes place in **some environment** (this is the environment for `get(name, envir=..., inherits=TRUE)`). For example:

```
exp <- quote(print(x))
x <- 5  # global env. here
eval(exp)  # eval(exp, envir=globalenv()) here
## [1] 5
env <- new.env()
env$x <- 6
eval(exp, envir = env)
## [1] 6
eval(exp, envir = emptyenv())
## Error:  could not find function "print"
```

2.3.1 Current evaluation environment

Anytime the **current evaluation environment** may be fetched by calling:

```
sys.frame(sys.nframe())
## <environment: R_GlobalEnv>
```

In the R console it is most often the global environment. Thus:

```
x <- 1
```

is equivalent to:

```
assign("x", 1, envir = sys.frame(sys.nframe()))
```

By the way, note the `local()` function:

HUMAN CAPITAL NATIONAL COHESION STRATEGY EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
local({
    print(sys.frame(sys.nframe()))
    yyy <- "test"
    print(yyy)
})
## <environment: 0x378a568>
## [1] "test"
print(sys.frame(sys.nframe()))
## <environment: R_GlobalEnv>
print(yyy)
## Error:  object 'yyy' not found
```

Here, a temporary environment has been created. It became a new current evaluating environment.

2.3.2 Evaluation of expressions within functions

A function (`closure`) object consists of:

- parameter list,
- body expression,
- parent environment pointer.

Here is some function:

```
f <- function(x, y) x + y
```

Note that the name “`f`” is bound to the global environment here.

```
str(formals(f))  # a pair list, may be converted to a list
## Dotted pair list of 2
##  $ x: symbol
##  $ y: symbol
body(f)  # just a call object
## x + y
```

What happens when we call `f(u, v)`?

1. `f` is evaluated in the current evaluation environment. If it is not a function, the process fails.
2. Arguments `u`, `v` are matched to `f`’s params `x`, `y`.
3. `f`’s body is evaluated on `x` $\mapsto$ `u`, `y` $\mapsto$ `v`.

For each function call, a new temporary environment is created:

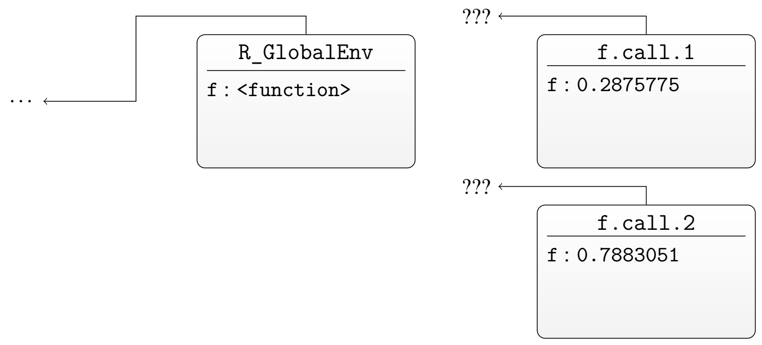
```
f <- function() {
    print(sys.frame(sys.nframe()))
    print(exists("y"))
    y <- runif(1)
    y
}

f()
## <environment: 0x375f258>
## [1] FALSE
## [1] 0.2875775
f()
## <environment: 0x36d31f0>
## [1] FALSE
## [1] 0.7883051
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund



What is the parent environment?

```
f <- function() {
    cat("Cur: ", format(sys.frame(sys.nframe())), "\n")
    cat("Par: ", format(parent.env(sys.frame(sys.nframe()))), "\n")
}

f()
## Cur:  <environment: 0x3dc2098>
## Par:  <environment: R_GlobalEnv>
f()
## Cur:  <environment: 0x3ddf4e0>
## Par:  <environment: R_GlobalEnv>
```

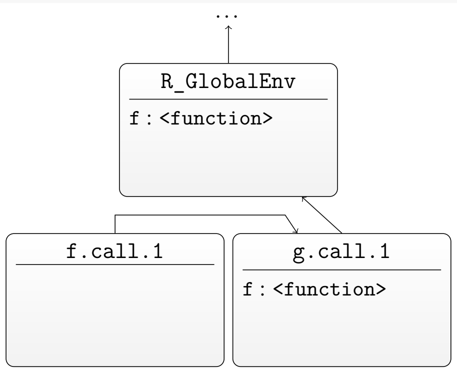
From `f()` we may access all names in the global environment and in all its predecessors.

How is the parent environment set?

```
g <- function() {
  f <- function() {
    cat("f.Cur:",format(sys.frame(sys.nframe())),"\n")
    cat("f.Par:",format(parent.env(sys.frame(sys.nframe()))),"\n")
  }

  cat("g.Cur:",format(sys.frame(sys.nframe())),"\n")
  cat("g.Par:",format(parent.env(sys.frame(sys.nframe()))),"\n")
  f()
}

g()
## g.Cur: <environment: 0x3b21140>
## g.Par: <environment: R_GlobalEnv>
## f.Cur: <environment: 0x3b14408>
## f.Par: <environment: 0x3b21140>
```

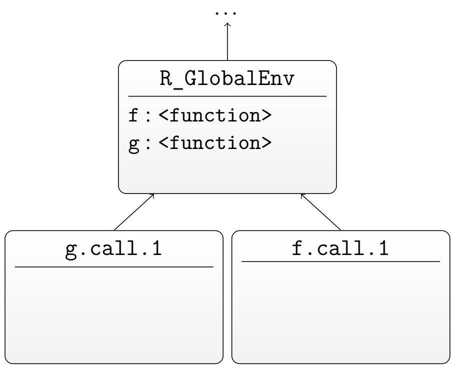


So far so good. However:

```
f <- function() {
   cat("f.Cur:", format(sys.frame(sys.nframe())), "\n")
   cat("f.Par:", format(parent.env(sys.frame(sys.nframe()))), "\n")
}

g <- function() {
   cat("g.Cur:", format(sys.frame(sys.nframe())), "\n")
   cat("g.Par:", format(parent.env(sys.frame(sys.nframe()))), "\n")
   f()
}

g()
## g.Cur: <environment: 0x3c698a8>
## g.Par: <environment: R_GlobalEnv>
## f.Cur: <environment: 0x3c4eea0>
## f.Par: <environment: R_GlobalEnv>
```



R uses **lexical scoping**. It is important **where** a function was created, and not where it is called.

Note that a function's default parent environment may be changed:

```
f <- function() {
   cat("f.Cur:", format(sys.frame(sys.nframe())), "\n")
   cat("f.Par:", format(parent.env(sys.frame(sys.nframe()))), "\n")
}

f()
## f.Cur: <environment: 0x3b66cd8>
## f.Par: <environment: R_GlobalEnv>
environment(f)
## <environment: R_GlobalEnv>
environment(f) <- baseenv()
f()
## f.Cur: <environment: 0x39e29f0>
## f.Par: <environment: base>
```

Lexical scoping is the reason why we should not refer to "global variables" from within a function. It is because we may not be sure what is its parent environment.

By the way, the "<<-" operator is equivalent to a call to `assign(`*name*`, `*value*`, inherits=TRUE)`. There are (rare) cases, when it is useful:

```
err <- FALSE
tryCatch({
    if (runif(1) < 0.5)
        stop("error!")
}, error = function(e) err <<- TRUE)
err
## [1] TRUE
```

Please check that the above does not work as expected with the use of "<-".

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

Another interesting question is: Do we have access to the calling environment? It turns out that the answer is positive:

```
f <- function() {
 cat("f.Cur:", format(sys.frame(sys.nframe())), "\n")
 cat("f.Par:", format(parent.env(sys.frame(sys.nframe()))), "\n")
 cat("f.Cal:", format(parent.frame()), "\n")
}

f()
## f.Cur: <environment: 0x39a2720>
## f.Par: <environment: R_GlobalEnv>
## f.Cal: <environment: R_GlobalEnv>
environment(f) <- baseenv()
f()
## f.Cur: <environment: 0x38302e8>
## f.Par: <environment: base>
## f.Cal: <environment: R_GlobalEnv>
```

Let us take a look at the arguments' evaluation. Recall that R is **lazy**:

```
f <- function(x) {
   cat("BEFORE\n")
   cat("x=", x, "\n")
   cat("AFTER\n")
   x
}
f(f({cat("NOW!\n"); 5}))
## BEFORE
## BEFORE
## NOW!
## x= 5
## AFTER
## x= 5
## AFTER
## [1] 5
```

In fact, arguments are represented by **promise expressions**, which consist of:

- an expression to be evaluated,
- a value (calculated on demand),
- an environment.

The arguments are evaluated in the **calling environment**. Thanks to that e.g. `substitute()` or `missing()` functions may work as expected.

Default arguments, on the other hand, are evaluated in the function's local environment:

```
a <- 1
f <- function(x={cat("NOW!\n"); a}) {
   a <- 2
   print(x)
}
f(5)
## [1] 5
f()
## NOW!
## [1] 2
```

By the way, the `match.call()` function:

```
f <- function(x, y, ...) match.call()
f(y = 1, 2, z = 3, 4)
## f(x = 2, y = 1, z = 3, 4)
```

Here, a `call` object is returned.

An exemplary usage:

```
multiapply <- function(x, ...) {
   funs <- match.call()[-c(1,2)]
   structure(sapply(funs, function(f) eval(f)(x)),
      names=sapply(funs, as.character))
}
multiapply(quote(test), typeof, mode, class)
##    typeof      mode     class
## "symbol"   "name"    "name"
multiapply(quote(f(10)), typeof, mode, class)
##      typeof        mode       class
## "language"      "call"      "call"
```

Moreover, note the `substitute()`'s function `env` argument:

```
substitute(x + y, env = list(x = 10, y = quote(sin(pi))))
## 10 + sin(pi)
```

Computing on the language is a very powerful tool:

- new expressions may be created on-the-fly,
- existing expressions may be freely modified.

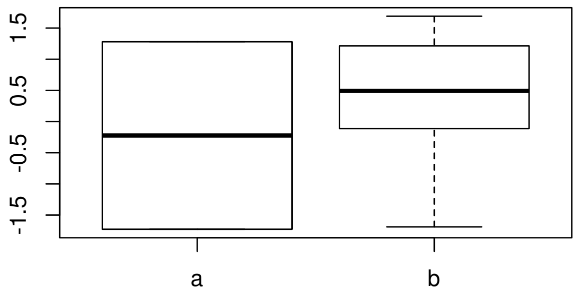
Exercise: *Check out how the %<% operator (magrittr 1.0.1 package) is implemented.*

Exercise: *Check out the source code of the pipeR 0.3-3 package.*

2.4 Formulas

Formulas are perhaps one of the most distinctive R features.

```
x <- rnorm(10)
f <- factor(sample(c("a", "b"), 10, replace = TRUE))
boxplot(x ~ f)
```



Moreover:

```
x <- rnorm(100)
y <- 10 * x + rnorm(100, 0, 0.3) + 8
lm(y ~ x)
##
## Call:
## lm(formula = y ~ x)
##
```

```
## Coefficients:
## (Intercept)            x
##       7.993        9.954
```

Formulas are created with the tilde operator.

```
frm <- (y ~ x)
class(frm)
## [1] "formula"
mode(frm)
## [1] "call"
```

It turns out that we already know how to deal with such objects.

```
frm <- (y ~ x)
unclass(frm)
## y ~ x
## attr(,".Environment")
## <environment: R_GlobalEnv>
str(as.list(frm))
## List of 3
##  $ : symbol ~
##  $ : symbol y
##  $ : symbol x
##  - attr(*, "class")= chr "formula"
##  - attr(*, ".Environment")=<environment: R_GlobalEnv>
```

Some helper functions:

```
frm <- (y ~ x)
all.vars(frm)
## [1] "y" "x"
str(terms(frm))
## Classes 'terms', 'formula' length 3 y ~ x
##   ..- attr(*, "variables")= language list(y, x)
##   ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "y" "x"
##   .. .. ..$ : chr "x"
##   ..- attr(*, "term.labels")= chr "x"
##   ..- attr(*, "order")= int 1
##   ..- attr(*, "intercept")= int 1
##   ..- attr(*, "response")= int 1
##   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

See also: the Formula package.

2.5 Summary

You now have full understanding of and control on how R expressions are evaluated. You may create, modify, and evaluate new language objects on-the-fly.

2.6 Bibliography

- R Core Team, *R language definition*, 2014, Sec. 6, 10
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 18
- Abelson H., Sussman G.J., Sussman, J., *Structure and interpretation of computer programs*, MIT Press, 1996
- Aho A.V., Lam M.S., Sethi R., Ullman J.D., *Compilers: Principles, techniques, and tools*, Pearson, 2006
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 7

3 Object-oriented programming

3.1 Introduction

Object-oriented programming makes it easier to create and maintain complicated software. Generally, OOP consists in developing:

- new data types (**classes**),
- functions dealing with objects of these classes (**methods**).

See R Core Team, *R language definition*, 2014, Sec. 5 for more details.

For instance, $A = [0, 1] \times [0, 2]$ and $B = [2, 5] \times [-1, 4]$ represent two exemplary rectangles. We may say that A, B are objects of `Rectangle` class.

What properties are common to all the rectangles? What constitutes a rectangle? The answer may be: e.g. bottom-left and top-right coordinates. Such coordinates (data) are called **slots**.

What can we do with all the rectangles? For instance, we may draw them on the screen. Such operations are called **methods**

Here are some basic OOP-related concepts:

1. **Inheritance**

   What all rectangles and all circles have in common? All of them are geometric shapes. We may say that `Rectangle` and `Circle` **inherits from** (more abstract) `GeometricShape`.

   And what is $C = [0, 1] \times [0, 1]$? It is a `Rectangle`. More precisely, it is a special kind of `Rectangle`, a `Square`. `Square` inherits from `Rectangle`.

2. **Polymorphism**

   Each `GeometricShape` can be drawn. Let $A = [0, 1] \times [0, 2]$, $D = \{(x, y) : x^2 + y^2 \le 1\}$. The `draw`$(A)$ and `draw`(D) operations are quite different. Drawing a rectangle is not the same as drawing a circle. Thus, one method name (`draw()`) may point to different class-dependent implementations.

Some examples:

- `t.test()`, `shapiro.test()`, `prop.test()` – return objects of class `htest`
- `POSIXct` and `POSIXlt` classes – different representations, similar interface (e.g. `diff()`)
- `TrapezoidalFuzzyNumber`, `PowerFuzzyNumber` – different kinds of `FuzzyNumbers` (see FuzzyNumbers package)

Note the following:

```
o1 <- shapiro.test(rnorm(100))
o2 <- Sys.time()
o3 <- FuzzyNumbers::TrapezoidalFuzzyNumber(0, 1, 2, 3)
print(o1)
##
##  Shapiro-Wilk normality test
##
## data:  rnorm(100)
## W = 0.9822, p-value = 0.1962
print(o2)
## [1] "2014-08-13 20:51:59 CEST"
print(o3)
## Trapezoidal fuzzy number with:
##    support=[0,3],
##       core=[1,2].
```

There are 3 object-oriented programming systems in R:

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN
SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

- S3 ("old-style classes"),
- S4 ("formal classes"),
- Reference Classes (R5, see `?ReferenceClasses`).

S3 and S4 are so popular that we may be using them without even realizing it. Note that S3 and S4 are build upon R's function-centric philosophy. C++, Java, C# are object-centric – this provides a very different look-and-feel. Also keep in mind that *encapsulation* has not been implemented in R's OOP systems.

3.2 S3

An **S3 object** is not defined in any formal way. The **class** attribute indicates an object's S3 class.

```
(y <- structure("testY", class = "classY"))
## [1] "testY"
## attr(,"class")
## [1] "classY"
(x <- structure("testX", class = c("classX", "classY")))
## [1] "testX"
## attr(,"class")
## [1] "classX" "classY"
is(x, "classX")  # cf. also inherits()
## [1] TRUE
is(x, "classY")
## [1] TRUE
is(x, "classZ")
## [1] FALSE
```

`y` is an object of the `classY` class.

```
class(y)
## [1] "classY"
```

`x` is an object of the `classX` and `classY` classes (in that order).

```
class(x)
## [1] "classX" "classY"
```

Recall that R objects may also have an **implicit class**:

```
z <- 1:3
class(z)
## [1] "integer"
attr(z, "class")
## NULL
```

3.2.1 Method dispatch

A very simple S3 method dispatch mechanism implements the polymorphism property:

```
o1 <- o2 <- o3 <- c(15888, 3652)
class(o2) <- "Date"
class(o3) <- c("POSIXct", "POSIXt")
print(o1)
## [1] 15888  3652
print(o2)
## [1] "2013-07-02" "1980-01-01"
print(o3)
## [1] "1970-01-01 05:24:48 CET" "1970-01-01 02:00:52 CET"
```

Why does `print()` behaves like above?

```
print
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x269d0b8>
## <environment: namespace:base>
```

This is because it is an S3 **generic function**. `UseMethod()` is responsible for S3 method dispatch.

Let `fun()` be a generic function. Additionally, let `x` be an object of class `("c`$_1$`",...,"c`$_n$`")`. Let us imagine that we call `fun(x)`.

- For each `c`$_i$, $i = 1, \ldots, n$, check if `fun.c`$_i$ exists. If that is true, call `fun.c`$_i$`(x)` and we are done.
- Otherwise, call `fun.default(x)` (the **default method**).

For example, check out the source codes of:

```
print.default
print.Date
print.POSIXlt
```

Note that e.g. `print.htest` has been hidden. Call `getS3method("print", "htest")` to get access.

We may also implement some generic functions and associated methods ourselves:

```
fun <- function(obj) UseMethod("fun")
fun.default <- function(obj) cat("fun.default\n")
fun.classX <- function(obj) cat("fun.classX\n")
fun.classY <- function(obj) cat("fun.classY\n")

fun(structure("testY", class="classY"))
## fun.classY
fun(structure("testX", class=c("classX", "classY")))
## fun.classX
fun(structure("testZ", class=c("classY", "classX")))
## fun.classY
fun(structure("testW", class=c("classZ")))
## fun.default
fun(structure("testQ", class=c("classZ", "classX")))
## fun.classX
```

In fact, S3 method dispatch is an extensible substitute for:

```
fun <- function(obj) {
   for (c in class(obj)) {
      if (c == "classX")
         return(fun.classX(obj))
      else if (c == "classY")
         return(fun.classY(obj))
   }
   return(fun.default(obj))
}
```

Also, note the `NextMethod()` function:

```
fun <- function(obj) UseMethod("fun")
fun.default <- function(obj) cat("fun.default\n")
fun.classX <- function(obj) { cat("fun.classX\n"); NextMethod() }
fun.classY <- function(obj) { cat("fun.classY\n"); NextMethod() }

fun(structure("testY", class="classY"))
## fun.classY
## fun.default
fun(structure("testX", class=c("classX", "classY")))
## fun.classX
## fun.classY
## fun.default
fun(structure("testW", class=c("classZ")))
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
## fun.default
```

Didn't we say that S3 is build upon existing R language facilities?

```
fun <- function(x, ...) {
   cat("---- fun ----\n")
   y <- 100
   print(sys.frame(sys.nframe()))
   print(parent.env(sys.frame(sys.nframe())))
   UseMethod("fun")
}

fun.default <- function(x, z, ...) {
   cat("---- fun.default ----\n")
   print(sys.frame(sys.nframe()))
   print(parent.env(sys.frame(sys.nframe())))
   cat(stringi::stri_wrap(paste(ls(all=TRUE),
      collapse=", "), width=20), sep="\n")
}
```

```
fun(1, 2)
## ---- fun ----
## <environment: 0x2579220>
## <environment: R_GlobalEnv>
## ---- fun.default ----
## <environment: 0x2df19a0>
## <environment: R_GlobalEnv>
## ..., .Class, .Generic, .GenericCallEnv, .GenericDefEnv, .Method,
## x, y, z
```

The calling environment of the generic function is copied into the calling environment of the method.

```
fun <- function(x, z, ...) UseMethod("fun")
fun.default <- function(x, z, ...) {
   cat("---- fun.default ----\n")
   print(.Class)
   print(.Generic)
   print(.GenericCallEnv)
   print(.GenericDefEnv)
   print(.Method)
}
fun(1, 2)
## ---- fun.default ----
## NULL
## [1] "fun"
## <environment: R_GlobalEnv>
## <environment: R_GlobalEnv>
## [1] "fun.default"
```

Here are some rules concerning **S3 method overloading**. A method's definition must be concordant with its generic function:

- the same parameter names,
- inclusion of "..." in the param list,
- the same default arguments.

Exercise: *Implement the* `plot()` *and* `print()` *methods for the* `Circle`, `Rectangle`, `Square`, `Triangle`, `Pentagon` *classes. Moreover, create* `circle()`, `rectangle()`, ... *functions, which construct objects of appropriate classes.*

By the way,

```
plot.new()
plot.window(c(0, 1), c(0, 1))
```

sets up a graphical device. Interestingly, these are not methods, but ordinary functions. The `lines()` *and* `polygon()` *functions may be used draw graphical primitives*

3.3 S4

S3 if quite fast and simple but objects are not defined in any formal way.

```
structure("broken", class = "POSIXct")
## Warning:  NAs introduced by coercion
## [1] NA
structure("broken", class = "htest")
```

```
## Error:  $ operator is invalid for atomic vectors
```

Let us now discuss the S4 (formal classes) systems. It is provided by the auto-loaded methods packages. An exemplary class:

```
setClass("Circle",
   slots=c(x="numeric", y="numeric", r="numeric")
)
showClass("Circle")
## Class "Circle" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        x       y       r
## Class: numeric numeric numeric
getSlots("Circle")
##         x         y         r
## "numeric" "numeric" "numeric"
```

Here is how we may create an object from the above class:

```
(obj <- new("Circle", x=0, y=0, r=1))
## An object of class "Circle"
## Slot "x":
## [1] 0
##
## Slot "y":
## [1] 0
##
## Slot "r":
## [1] 1
class(obj)
## [1] "Circle"
## attr(,"package")
## [1] ".GlobalEnv"
typeof(obj)
## [1] "S4"
isS4(obj)
## [1] TRUE
```

Accessing an object's slots:

```
obj@x
## [1] 0
slot(obj, "r")
## [1] 1
```

```
obj@y <- 2
```

Interestingly:

```
unclass(obj)
## <S4 Type Object>
## attr(,"x")
## [1] 0
## attr(,"y")
## [1] 2
## attr(,"r")
## [1] 1
```

We may create functions to test the validity of an object:

```
setValidity("Circle", function(object) {
  if (!is.finite(object@x) || length(object@x)!=1)
    return("x should be a single finite real value")
  if (!is.finite(object@y) || length(object@y)!=1)
    return("y should be a single finite real value")
  if (!is.finite(object@r) || length(object@r)!=1 || object@r<0)
    return("x should be a single nonnegative real value")
  TRUE
})
```

```
(obj <- new("Circle", x=1, y=2, r=-4))
## Error:  invalid class "Circle" object:  x should be a single nonnegative real value
```

However:

```
obj@x <- 1:10 # no validity check here, see also ?validObject
```

Note how we may create classes the inherit from other ones:

```
setClass("Rectangle",
   slots=c(bottomleft="numeric", topright="numeric"))
setClass("Square",
   contains="Rectangle",
   validity=function(object) {
      if (diff(object@topright-object@bottomleft) != 0)
         return("not a square")
      TRUE
   })
new("Square", bottomleft=c(0,0), topright=c(1,1))
## An object of class "Square"
## Slot "bottomleft":
## [1] 0 0
##
## Slot "topright":
## [1] 1 1
new("Square", bottomleft=c(0,0), topright=c(1,2))
## Error:  invalid class "Square" object:  not a square
```

An S4 generic function must be registered:

```
test <- function(x) {
   cat("default method\n")
}

setGeneric("test")
## [1] "test"
test
## standardGeneric for "test" defined from package ".GlobalEnv"
##
## function (x)
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN
SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
## standardGeneric("test")
## <environment: 0x3507710>
## Methods may be defined for arguments: x
## Use  showMethods("test")  for currently available ones.
```

See also: `?getGeneric`, `?getGenerics`.

The `show` method is a S4 `print()`'s counterpart:

```
setMethod("show",
   signature=c(object="Circle"),
   function(object) {
      validObject(object)
      cat(sprintf(
         "a circle at the origin (%g, %g) of radius %g\n",
         object@x, object@y, object@y))
   })
## [1] "show"
new("Circle", x=0, y=0, r=1)
## a circle at the origin (0, 0) of radius 0
```

S3 methods may also be overloaded:

```
setMethod("as.list",
   signature=c(x="Circle"),
   function(x, ...) { # params here must match as.list's params
      validObject(x)
      list(x=x@x, y=x@y, r=x@r)
   })
## Creating a generic function for 'as.list' from package 'base' in the global environment
## [1] "as.list"
str(as.list(new("Circle", x=0, y=0, r=1)))
## List of 3
##  $ x: num 0
##  $ y: num 0
##  $ r: num 1
```

Method dispatch may be based on more than 2 classes:

```
setGeneric("transpose", function(w, delta) w+delta)
## [1] "transpose"
setMethod("transpose",
   signature=c(w="Circle", delta="numeric"),
   function(w, delta) {
      validObject(w)
      new("Circle", x=w@x+delta, y=w@y+delta, r=w@r)
   })
## [1] "transpose"
transpose(1:5, 0.1)
## [1] 1.1 2.1 3.1 4.1 5.1
transpose(new("Circle", x=1, y=2, r=3), 0.5)
## a circle at the origin (1.5, 2.5) of radius 2.5
```

The default method has signature `c(w="ANY", delta="ANY")`.

```
attr(transpose, "default") # default method
## Method Definition (Class "derivedDefaultMethod"):
##
## function (w, delta)
## w + delta
##
## Signatures:
##          w
## target  "ANY"
## defined "ANY"
showMethods("transpose")
## Function: transpose (package .GlobalEnv)
## w="ANY", delta="ANY"
## w="Circle", delta="numeric"
## w="integer", delta="numeric"
##     (inherited from: w="ANY", delta="ANY")
```

3.4 Reference classes

Reference classes (a.k.a.) R5 are available since R 2.12. They have not become very popular yet. The main idea behind introducing R5 was to provide a C++/Java look-and-feel. R5 objects are represented by environments, which mean that they may be modified within methods. Read more: `?ReferenceClasses`

3.5 Group generics & replacement functions

We have used **replacement functions** many times:

```
x <- 1:6
length(x) <- 8  # replacement function
dim(x) <- c(2, 4)  # replacement function
x[2, 4] <- 100  # replacement function
x
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5   NA
## [2,]    2    4    6  100
```

Replacement functions modify a given named object.
A replacement function is named `funname<-()`.

```
change(name, ...) <- value
```

The above is just a syntactic sugar for:

```
name <- "change<-"(name, ..., value)
```

An example:

```
"add<-" <- function(x, value) {
    # 'value' is a fixed name
    x + value
}

z <- 1:10
add(z) <- 2
z
##  [1]  3  4  5  6  7  8  9 10 11 12
```

Here is list of overloadable operators:

group	arguments	operators
subsetting / extraction	`x`, `i`, (`j`, ...)	`[`, `[[`, `$`, `[<-`, `[[<-`, `$<-`
S4: `Arith` S3: `Ops`	`e1`, `e2`	`+`, `-`, `*`, `/`, `^`, `%%`, `%/%`
S4: `Logic` S3: `Ops`	`e1`, `e2`	`&`, `|`, `!`
S4: `Compare` S3: `Ops`	`e1`, `e2`	`==`, `!=`, `<`, `<=`, `>=`, `>`

```
"*.Interval" <- function(e1, e2)
   structure(c(
      range(e1[1]*e2[1], e1[1]*e2[2], e1[2]*e2[1], e1[2]*e2[2])
   ), class="Interval")
"print.Interval" <- function(x, ...)
   cat(sprintf("[%g, %g]\n", x[1], x[2]))

x <- structure(c(1,2), class="Interval")
y <- structure(c(-1,1), class="Interval")
x*y
## [-2, 2]
```

```
setMethod("==",
   signature(e1="Rectangle", e2="Rectangle"),
   function(e1, e2) {
      all(e1@bottomleft == e2@bottomleft) &&
         all(e1@topright == e2@topright)
   }
```

By the way, we may create our own binary operators (“`%name%`”):

```
"%+%" <- function(x, y) sum(x) + sum(y)
c(1, 2, 3) %+% c(1, 2)
## [1] 9
```

Some other forms are also available:

```
":=" <- function(x, y) {x[is.na(x)] <- y; x}
c(1, NA, 3, 4) := 0
## [1] 1 0 3 4
```

Other overloadable S3 methods:

```
cat(stringi::stri_wrap(paste(names(.knownS3Generics),
   collapse=", ")), sep="\n")
## Math, Ops, Summary, Complex, as.character, as.data.frame,
## as.environment, as.matrix, as.vector, cbind, labels,
## print, rbind, rep, seq, seq.int, solve, summary, t, edit,
## str, contour, hist, identify, image, lines, pairs, plot,
## points, text, add1, AIC, anova, biplot, coef, confint,
## deviance, df.residual, drop1, extractAIC, fitted, formula,
## logLik, model.frame, model.matrix, predict, profile,
## qqnorm, residuals, se.contrast, terms, update, vcov
cat(stringi::stri_wrap(paste(.S3PrimitiveGenerics,
   collapse=", ")), sep="\n")
## anyNA, as.character, as.complex, as.double, as.environment,
## as.integer, as.logical, as.numeric, as.raw, c, dim,
## dim<-, dimnames, dimnames<-, is.array, is.finite,
## is.infinite, is.matrix, is.na, is.nan, is.numeric, length,
## length<-, levels<-, names, names<-, rep, seq.int, xtfrm
```

Note that they do not call `UseMethod()` explicitly (in R code):

```
dim
## function (x)  .Primitive("dim")
as.character
## function (x, ...)  .Primitive("as.character")
```

Some generic functions are grouped:

```
getGroupMembers("Arith")
## [1] "+"   "-"   "*"   "^"   "%%"  "%/%" "/"
getGroupMembers("Summary")
## [1] "max"   "min"   "range" "prod"  "sum"   "any"   "all"
```

They may be overloaded as a group at once.

group	args	functions
`Math`	`x`	`abs()`, `sign()`, `sqrt()`, `ceiling()`, `floor()`, `trunc()`; `cummax()`, `cummin()`, `cumprod()`, `cumsum()`; `log()`, `log10()`, `log2()`, `log1p()`, `exp()`, `expm1()`; `acos()`, `acosh()`, `asin()`, `asinh()`, `atan()`, `atanh()`, `cos()`, `cosh()`, `sin()`, `sinh()`, `tan()`, `tanh()`; `gamma()`, `lgamma()`, `digamma()`, `trigamma()`
S4: `Math2` S3: `Math`	`x`, `digits`	`round()`, `signif()`

group	args	functions
`Summary`	`...`	`max()`, `min()`, `range()`, `prod()`, `sum()`, `any()`, `all()`
`Complex`	`z`	`Arg()`, `Conj()`, `Im()`, `Mod()`, `Re()`
props	`x`	`length()`, `length<-()`, `dimnames()`, `dimnames<-()`, `dim()`, `dim<-()`, `names()`, `names<-()`, `levels<-()`
join	`...`	`c()`, `cbind()`, `rbind()`
coertion	`x`	`as.character()`, `as.complex()`, `as.double()`, `as.environment()`, `as.integer()`, `as.logical()`, `as.numeric()`, `as.raw()`, `as.real()`, `as.vector()`, `unlist()`

group	args	functions
type/val check	`x`	`is.array()`, `is.finite()`, `is.infinite()`, `is.matrix()`, `is.na()`, `is.nan()`, `is.numeric()`
other	`x`	`rep()`, `seq()`, `xtfrm()`

3.6 Summary

OOP is a programming paradigm that bases on the concept of objects and methods. There are 3 OOP systems in R: S3, S4, and RefClasses. Objects are designed in class hierarchies and may correspond to things found in the real world. OOP may make writing complex applications easier – it tries to mimic the way people conceive reality and organize their knowledge.

3.7 Bibliography

- R Core Team, *R language definition*, 2014, Sec. 5
- R Core Team, *R internals*, 2014, Sec. 1.12
- R Core Team, *Writing R extensions*, 2014, Sec. 7
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 20
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 9

4 UNIX-like command line

4.1 Introduction

In this section we will learn some UNIX-like shell basics. Let us start a **Terminal Emulator**. We will work in Bash – it is default on most Linuxes and OS X. Windows users may for example install git for Windows and then run Start → git → git Console. Git for Windows comes with a minimal GNU run-time environment.

4.2 UNIX

The **UNIX** system appeared in 1972 and was developed by AT&T in the C programming language. Originally, it was **a system for developing software to be run on multiple platforms**. Notable properties:

- portable,
- multi-tasking,
- multi-user,
- a hierarchical file system (fs),
- devices as files,
- large number of software tools, i.e. small programs that can be strung together through a command line interpreter using pipes, as opposed to using a single monolithic program that includes all of the same functionality.

Here are some famous quotes explaining the UNIX philosophy:

> The idea that the power of a system comes more from the relationships among programs than from the programs themselves.
>
> [Brian Kernighan and Rob Pike]

> *Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.*
>
> [Brian Kernighan and Rob Pike]

> *This is the UNIX philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*
>
> [Doug McIlroy]

A UNIX system consists of the following components:

- kernel,
- development environment (compilers, linker, libraries, make, etc.),
- commands (shell, utilities like cp, ls, grep, kill, etc.),
- documentation.

Moreover, it may provide the following interfaces:

- command line (CLI),
- graphical (e.g. X Windows System, XOrg) (GUI).

UNIX has many **commercial instances**, e.g. BSD, System V, HP-UX, Solarix AIX, Xenix, OS X, SCO etc. There are also a number of different standards, like POSIX, Single UNIX Specification (SUS), Executable and Linkable Format (ELF), etc. For example, **SUSv3 (UNIX 03)** consists in:

- Base Definitions (XBD) – a list of C header files which must be provided by compliant systems. [84 header files]
- Shell and Utilities (XCU) – a list of utilities and a description of the shell, sh. [160 utilities]
- System Interfaces (XSH) – various functions which are implemented as system calls or library functions. [1123 items]

Among free UNIX instances we may find:

- FreeBSD, NetBSD, OpenBSD,
- Darwin (open source subset of OS X),
- GNU/Linux (rewritten from scratch to avoid copyright issues: UNIX-like).

The above are not registered as UNIX-compliant, as certification is expensive. There are also UNIX-like compatibility layers, for example:

- Cygwin & MSYS (Windows) – sufficient for most common open source software to be compiled and run

HUMAN CAPITAL NATIONAL COHESION STRATEGY EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

4.3 Linux

The GNU Project (started in 1983) aimed to create a complete UNIX-compatible software system. Users are free to run the software, share it, study it, and modify it. As of June 2014, 97% of the world's 500 fastest supercomputers run some variant of Linux, including all the 59 fastest, see top500.org.

Here is a list of some popular **Linux distributions (packaged)**:

- RedHat, Fedora, CentOS,
- Debian, Ubuntu, Mint,
- Skackware, SuSE,
- Gentoo,
- Arch,
- Android.

However, some distributions are not free of charge, e.g. RedHat Enterprise.

Among some of the GNU packages (maintained by the Free Software Foundation) we find:

- bash (GNU UNIX-compatible shell),
- coreutils:
  - fileutils: chgrp, chown, chmod, cp, dd, df, dir, du, ln, ls, mkdir, mkfifo, mknod, mv, rm, etc.,
  - textutils: cat, cksum, head, tail, md5sum, nl, od, pr, tsort, join, wc, tac, paste, etc.,
  - shellutils: basename, chroot, date, dirname, echo, env, groups, hostname, nice, nohup, printf, sleep, etc.,
- diffutils (e.g. diff), findutils (e.g. find, locate), grep,
- GRUB, gzip, readline, tar, time,
- GNU build system (autoconf, automake), GNU make, GNU C library (glibc), GNU debugger (gdb), GNU Binutils (as, ld), GNU Compiler Collection (gcc, g++, f90, ...), GNU gettext, GNU bison,
- Utilities: GNU Emacs, GNU nano, GNU aspell, GNU wget.

4.4 Working in the command line

Bash (Bourne-again shell, 1989) is a Unix shell written for the GNU Project. It is a free software replacement for the Bourne shell (sh). It includes:

- a command processor (type in commands and cause actions; read commands from a file /script/),
- filename wildcarding, piping, command substitutions, variables, control structures, regular expresions, ...

Here are some useful keyboard shortcuts:

- ⟨CTRL+C⟩ – send SIGINT (e.g. stop execution of a program),
- ⟨←⟩, ⟨→⟩, ⟨HOME⟩, ⟨END⟩ – move the cursor
- ⟨↑⟩, ⟨↓⟩ – navigate history,
- ⟨TAB⟩ – autocompletes from the current position.

Let us begin our brief overview of commands by discussing how to work with files and directories:

```
$ ls # list current directory's contents
$ ls --help # basic usage
$ ls -l --color
```

Note that "$" is the default command prompt character. Moreover, "`ls -l --color`" means: execute the `ls` command with two arguments: `-l` and `--color`.

To get some help on a command, use:

```
$ man ls # opens the `ls` man page
```

Press ⟨/⟩ to search within the man page; then ⟨n⟩ for more matches; ⟨q⟩ quits the viewer.

Aliases may also be defined:

```
$ alias ll='ls -l --color' # very useful
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
$ ll
```

Aliases may be added to ~/.bashrc for future use.

Here is how we may change the current working directory:

```
$ pwd # print working directory
$ mkdir test # make a directory, rmdir removes it
$ cd test # change working directory, a Bash builtin
$ cd # go home, the same as cd ~
$ cd .. # one level up
$ exit # goodbye
```

Some basic operations:

```
$ cd test
$ touch file1 # changes file timestamp
              # (or creates a new one)
$ cp file1 file3 # copy files
$ cp -i file1 file3 # ask
$ cp -f file1 file3 # force
$ ls t* # wildcard, all starting with t
$ ls test? # test1, test2, testX, but not test123
$ mv file3 file2 # move/rename
$ rm file2 # remove
$ du # disk usage
$ df # disk free
$ wget http://.... # download
```

Note that, by default, `cp`, `mv`, and `rm` overwrite/remove files without asking the user to confirm. For safety reasons, we may add `alias cp='cp -i'`, `alias mv='mv -i'`, `alias rm='rm -i'` to ~/.bashrc.

Here is a list of notable directories in the file system:

- / – root directory
- /tmp – temporary files
- /etc – config files
- /bin – executables
- /home – users' home dirs
- /usr, /opt – "apps"
- etc.

There are a number of text file viewers available. Some Linux users prefer `vi` or `emacs` to edit text files. These editors may be difficult to learn. Just note that ⟨:,q,!⟩ quits `vi`, and ⟨CTRL+X,CTRL+C⟩ quits `emacs`. The `nano` editor is much more user-friendly.

Let us issue the following command:

```
$ nano testfile.txt
```

Now press ⟨CTRL+O⟩ to save the file, ⟨CTRL+W⟩ to search, and ⟨CTRL+X⟩ to exit. Moreover, if we work in GUI, we may of course use e.g. `gedit`, `kate`, or even `RStudio` to edit text files.

```
$ cat file2 # print contents
```

See also: `head`, `tail`, `paste`, `wc`

Execute the `chown` command to change a file's owner (`whoami` gives your username). `chgrp` changes group ownership and `chmod` changes file/directory mode bits.

For example, let us create a bash script named `test.sh`:

```
#!/bin/bash    # invoke an appropriate interpreter
echo "Hello␣world!"
```

The script consists of Bash commands and program calls.

Now let us call:

```
$ chmod 755 test.sh # add exec perms for all users
$ ./test.sh # run; note the `./`
```

We may also create an R script in a similar manner. First let us get the location of the `Rscript` command:

```
$ whereis -b Rscript # it's /usr/bin/Rscript on my PC
```

Now create `r.sh`:

```
#!/usr/bin/Rscript
x <- rand(10)
print(x)
```

We can run now:

```
$ chmod 755 r.sh
$ ./r.sh
```

By the way, batch R script's command-line arguments may be read. Here is a file named `cmdargtest.sh`:

```
#!/usr/bin/Rscript
print(commandArgs(TRUE))
```

Run:

```
$ chmod 755 cmdargtest.sh
$ ./cmdargtest.sh  arg1 --arg2="xxx" -f
## [1] "arg1"        "--arg2=xxx" "-f"
```

We should also know some basic information about streams, pipes and redirects.

First of all, here is how we may redirect the `stdout`:

```
$ echo "Hello␣world" > test3 # creates a new file
```

also:

```
$ cat > test4
...
type
something
in the
console
...
<CTRL+D>
```

Note that ⟨CTRL+D⟩ sends an EOF, i.e. end of file "signal".

Here is how how we may redirect the `stderr` (e.g. error list):

```
$ command 2> file
```

. . . and the `stdin`:

```
$ command < file
```

This may be used if input data to some program should rather be taken from a file and not from keyboard.

Pipes allow us to make `stdout` of `cmd1` be `stdin` of `cmd2`:

```
$ cmd1 | cmd2
```

Some examples:

```
$ ps aux # list working processes
$ ps aux | grep gagolews # gagolews's processes
                         # (grep does text searching)
$ ps aux | grep gagolews | grep -P -e '\s1[0-9]{3}\s'
                         # regex search (PCRE)
$ man sed # stream editor
$ ps aux | grep gagolews | sed 's/gagolews/notme/g'
```

HUMAN CAPITAL NATIONAL COHESION STRATEGY
EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

Some interesting text-editing tools: `wc`, `tr`, `paste`, `split`, `sort`, `sed`

Note that each program returns an integer value (silently). A value equal to zero means that a program has been successfully executed. On the other hand, a non-zero value denotes some kind of execution error. To apply a conditional execution, we call:

```
$ someprogram1 && someprogram2
```

Here, `someprogram2` is run if and only if `someprogram1` succeeded. For example, an R batch script will return a non-zero value if it was aborted by a `stop()` call.

To list currently working processes, we call:

```
$ ps
$ top # q to exit
```

To kill a process, we issue the following command(s):

```
$ kill process_number # SIGINT
$ kill −9 process_number # SIGKILL (force)
```

By the way, in Bash, jobs may be run in the background, see `bg`, `fg`, `jobs`, and `&`.
Other utilities:

```
$ tar zxvf filename.tar.gz # unarchive a gzipped file
$ tar jxvf filename.tar.bz2 # bzip2 compression
$ unzip filename.zip
```

Moreover:

- `mc` – Midnight commander (if installed)
- `links` / `links2` – text-based web browsers (if installed)
- `time` – measures run-time of a program

Run-time measurement:

```
$ time ./test.R
```

Compiling (well-written & portable) programs:

```
$ wget ...
$ tar ....
$ cd ...
$ ./configure # try also: ./configure −−help
$ make
$ sudo make install # run as root == super user, admin
```

Exercise: *Linux/OS X Users: download, decompress, and compile the latest development version of R.*

By the way, the `system2` function in R:

```
system2("ls", "~/Dydaktyka/R-training/", stdout = TRUE)
##  [1] "about_me"                  "bash_linux"
##  [3] "esf.pdf"                   "hadoop"
##  [5] "human_capital.pdf"         "human_capital.ps"
##  [7] "ibib_ipi_ibs.pdf"          "knitr-kompiluj-adasdr.sh"
##  [9] "multithread"               "R_and_RStudio"
## [11] "rexamine-brackets.pdf"     "rexamine-logo-wide.pdf"
## [13] "R_lang_advanced"           "R_lang_intro"
## [15] "R_Rcpp"                    "R_strings_and_files"
## [17] "R_training-beameropts.R"   "R_training-defs.R"
## [19] "R_training-title.R"        "R_training-title.R.backup"
## [21] "Shiny"                     "testthat"
## system2("tar", "zxf input.tar.gz")  # etc.
```

4.5 Summary

According to the UNIX philosophy we may access small programs that can be strung together through a command line interpreter, as opposed to using a single monolithic program that includes all of the same functionality. R has a very similar spirit. Knowing how to perform basic operations in the terminal can make your work more productive.

4.6 Bibliography

- Newham C., *Learning the bash shell*, O'Reilly, 2005
- Albing C., Vossen J.P., Newham C., *bash cookbook*, O'Reilly, 2007
- linuxcommand.org/
- www.linuxlinks.com/Beginners/
- freevideolectures.com/blog/2012/04/5-websites-learning-linux/
- en.tldp.org/LDP/abs/html/index.html
- mywiki.wooledge.org/BashGuide

5 Collaborative software development with Git

5.1 Introduction

Git is a distributed[1] **revision control** and **source code management** (SCM) system. It was designed and developed by Linus Torvalds for Linux kernel development. The development of Git began on April, 3 2005. The project was announced on 6th of April, and became self-hosting the next day.

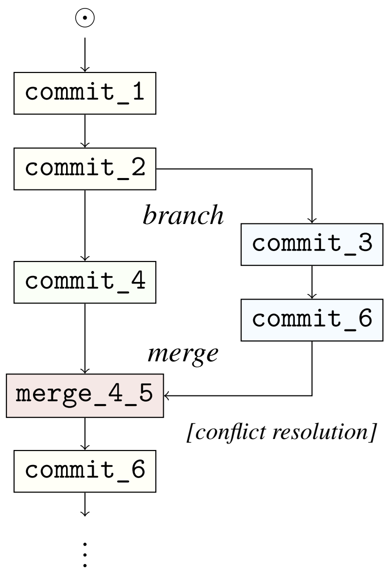
Git allows many developers to work on a common project, work on the same documents at the same time "without stepping on each other's toes." Git keeps track of software revisions (manages changes to documents, source codes, websites, etc.). A **revision** includes the information on:

- changes made (diffs, deltas),
- revision number/id, (a SHA256 checksum)
- author,
- timestamp.

Importantly, Git gives us the ability to revert a document to a previous revision.

Git allows for nonlinear development: branching and merging:

[1]Every Git working directory is an independent repository with complete history and full version tracking capabilities and does not depend on neither network access nor a central server (but can be synchronized with other machines) "Distributed" is opposed to local-only or client-server solutions.



Other SCMs:

- CVS
- Subversion (SNV)
- Mercurial
- Fossil

etc.

5.2 Local Git repositories

Let us set up Git:

```
$ git config --global user.name "My␣Name"
$ git config --global user.email "user@email.com"
$ git config --global color.ui auto
```

A Git **repository** is just a directory where Git has been initialized to start version controlling our files. The `git init` command initializes a new repository in the current working directory.

```
$ cd ~
$ mkdir git_test
$ cd git_test
$ git init
$ ls -a -l
```

Note the `.git` directory.

Create two text files named `file1.txt` and `file2.txt`. To display the working tree status, we issue the following command:

```
$ git status
```

We see that there are 2 untracked files. Generally, there are 4 possible file states:

- **staged**
  Staged for the next commit.
- **unstaged**
  Changed files, not yet staged for the next commit.
- **untracked**
  Files that are not tracked by Git yet (e.g. new files).

HUMAN CAPITAL NATIONAL COHESION STRATEGY EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

- **deleted**
  Deleted files, pending removal from a Git repository.

The `git add` command stages an untracked or modified file (shapshots the file in preparation for versioning).

```
$ git add file1.txt # untracked -> staged
$ git status
$ git reset file1.txt # -> staged
$ git status
$ git add -A # stage all
$ git status
```

Two files are staged now, but they are not in the repository yet. We now must **commit** the changes made. In other words, let us save the current repository's snapshot and record file snapshots permanently in version history.

```
$ git commit -m 'a comment (required)'
$ git status
```

Please modify the contents of `file1.txt` and create `file3.R`.

```
$ git diff # file differences not yet staged
$ git add '*.txt'
$ git diff --staged
$ git reset file1.txt # -> unstaged
$ git checkout -- file1.txt # reverts changes
```

By the way, the `.gitignore` file may be added to the repository.

Other operations:

```
$ git mv ... ... # move/rename
$ git rm filename # remove
$ git rm -r foldername
```

If we delete a file without using `git rm`, we will find that we still have to `git rm` the deleted files from the working tree.

```
$ git commit -a -m 'a comment' # git-adds all
$ git log # shows i.a. SHA256 commit reference (id)
$ git log --summary
$ git commit --amend # when you commit too early
```

5.3 Remote Git repositories

It is a good idea for our main repository to be on a remote server in case the computer is broken or we want to work on a common project with others. There are a few Git hosting services available, see git.wiki.kernel.org/-index.php/GitHosting. For example, we may use GitHub, Bitbucket, GitLab Cloud, Gitorious, etc. GitHub is the most popular among R users:

Please create a GitHub repository (online). Then we may clone this remote repository locally:

```
$ cd .....
$ git clone https://github.com/....
```

Alternatively, if we start with an empty remote repository and we want to send the local repository to the server, we may try:

```
$ git remote add origin https://...
```

Note that `origin` usually denotes the name of the main remote repository.

Let us push our local changes to our `origin` repository on GitHub. The name of our remote is `origin` and the default local branch name is `master`.

```
$ git push origin master
```

To download the changes:

```
git pull origin master # fetch and merge
git diff HEAD
```

If something went wrong:

```
git fetch origin
git reset --hard origin/master
```

5.3.1 Resolving conflicts

Git tries to auto-merge changes. Unfortunately, this is not always possible and results in conflicts. We are responsible to merge these conflicts manually by editing the files shown by Git.

After editing the files appropriately, we need to mark them as merged with:

```
$ git add filename
```

Merge conflicts can occur when changes are made to a file at the same time. A lot of people get frustrated when a conflict happens, but it is not that difficult to resolve it.

By the way, we may call:

```
devtools::install_github("user/repo")  # e.g. Rexamine/stringi
```

to install the most recent development version of an R package hosted on GitHub. Note that in some cases we will need a working C/C++ compiler (see e.g. Rtools for Windows) and a LaTeX distribution installed.

5.4 Summary

You may use git for anything connected with text files (documents, source files, etc.) – even for backing up your files. It is almost perfect for collaborative software development. Make sure you know how to resolve possible conflicts.

5.5 Bibliography

- Chacon S., *Pro Git*, Apress, 2009 ← git-scm.com/book
- try.github.io – an interactive tutorial
- git-scm.com/documentation – Git docs
- www.molecularecologist.com/2013/11/using-github-with-r-and-rstudio
- help.github.com/articles/using-pull-requests

6 Writing R packages

6.1 Introduction

In this section we will demonstrate how to write an **R package**. The *Writing R extensions* manual states that “Packages provide a mechanism for loading optional code, data and documentation as needed.” R packages give us perhaps the most efficient way to organize “big R projects” and to share code with others.

We will need the roxygen2 and devtools packages. It is strongly suggested to Windows users that they install `Rtools` and some LaTeX distribution, like MiKTeX for Windows or TeX Live.

6.2 Creating a source package

A package is a directory of files which extend R,

- *a* **source package** *(the master files of a package),*
- *or a* **tarball** *[*`.tar.gz` *file] containing the files of a source package,*
- *or an* **installed package***, the result of running R CMD INSTALL on a source package.*

On some platforms (notably OS X and Windows) there are also **binary packages***, a zip file or tarball containing the files of an installed package which can be unpacked rather than installing from sources.* [R Core Team, *Writing R extensions*, 2014, Sec.1]

Let is write a simple **source package**.

1. Create a directory for the source package, e.g. `testpkg`.

   This will also be the name of our package. You may also think of setting up a separate GitHub repository for this project.

2. Setup a new RStudio project for the package: Click File → New project → Existing directory.

3. Now edit Tools → Project options. Select Build Tools → Project build tools: Package. Then check Generate documentation with Roxygen and all the checkboxes under the Roxygen options. In the check package text box, input `R CMD check` additional options: `--as-cran`.

4. In the package's root directory, create a file named `DESCRIPTION`. It will provide basic information on the package. The format of the file is "Debian Control File", see `?read.dcf`.

   Here is a list of required fields in the `DESCRIPTION` file:

   - `Package` – name of the package, `[A-Za-z][A-Za-z0-9.]*[A-Za-z0-9]`
   - `Version`, e.g. `0.1-1`
   - `License`, e.g. `GPL-2`, `GPL (>= 2)`, or `LGPL-3`
   - `Description` – a comprehensive description of what the package does
   - `Title` – a (very) short description of the package
   - `Author` – who wrote the package (free text)
   - `Maintainer` – a contact person (bug reports, CRAN submission issues, etc.), e.g. `John Smith <jsmith@domain.net>`

   Note that `Author` and `Maintainer` may be automatically generated by specifying the `Authors@R` field, see `?person`. For example:

```
Authors@R: c(person("John", "Greedy",
        role = c("aut", "cre"),
        email = "jgreed@some.domain.net"),
    person("Giovanni", "Zadeh", role = "aut"),
    person("Lucian", "Popescu", role = "ctb",
        email = "popluci@gmail.com"))
```

   Note the roles: `cre` denotes a package's maintainer, `aut` identifies an author, `ctb` – a contributor, etc.

   Other useful fields:

   - `Encoding` – e.g. for `DESCRIPTION` and package's manual (ASCII usage recommended)
   - `URL` – e.g. the package's homepage
   - `BugReports` – a single URL to which bug reports about the package should be submitted
   - `Date` – the release date of the current version of the package, preferably `yyyy-mm-dd`

   Package dependencies:

   - `Depends` – a comma-separated list of package names which this package depends on. Those packages will be attached before the current package when `library()` or `require()` is called.

- `Imports` – packages whose namespaces are imported from but which do not need to be attached.
- `Suggests` – lists packages that are not necessarily needed (e.g. packages used only in examples, tests or vignettes)

Other: `LinkingTo`, `Enhances`.

Here is an exemplary `DESCRIPTION` file:

```
Package: testpkg
Version: 0.1-1
Date: 2015-01-01
Title: My First Collection of Functions
Authors@R: c(person("John", "Greedy",
        role = c("aut", "cre"),
        email = "jgreed@some.domain.net"))
Depends: R (>= 2.15.0), stringi
Description: A short (one paragraph) description of
    what the package does and why it may be useful.
License: GPL (>= 2)
URL: http://github.com/user/projectname
BugReports: http://github.com/user/projectname
```

5. To build the package, press ⟨`CTRL+SHIFT+B`⟩. Note that `R CMD INSTALL testpkg` is called automatically and the package becomes attached.

We just build our first package.

By the way, the `.Rbuildignore` file may consist of a list of regular expressions. All the matching file names will not be considered during a package's build. For example:

```
^.*\.Rproj$
^\.Rproj\.user$
^\.Rhistory$
^\.RData$
^\.git
^\.gitignore$
^README.md$
```

6.3 Adding R scripts and documenting a package's entries

Let us add some R scripts. R source scripts will be located in the `R/` directory. We must pay attention to documenting each package's entry. Man pages (`.Rd` files, LaTeX-like) are placed in the `man/` dir. We will use roxygen2 to auto-generate them. In this case, all "`#'`" comments in `.R` scripts will be processed by roxygen2.

Let us generate the package's "home page". Create a file `testpkg-package.R` with the following contents:

```
#' @title package title...
#'
#' @description
#' package description....
#'
#' @name testpkg-package
#' @docType package
invisible(NULL)
```

Now build the package and examine the contents of `man/testpkg-package.Rd`. Check out also:

```
help(package = "testpkg")
```

It is time to add an R function. In `R/test1.R` let us write:

```
#' @title
#' What the Function Does
#' @description
#' Short description
#' @details
#' A few paragraphs with more detailed info
#'
#' @family fungroup
#' @export
test1 <- function() {
   cat("test1")
}
```

Please build the package and call `test1()`. Thanks to the `@export` entry the function will be available to the package's users; see the `NAMESPACE` file.

Here is another function:

```
#' ...
#' @param x character vector;
#' @param y integer; number of iterations
#' @return Returns the value of...
#'
#' @examples
#' test2(1:10, 51)
#' \dontrun{test2(numeric(1000000000), 1000000)}
#'
#' @family fungroup
#' @export
test2 <- function(x, y) {
   # ...
   result
}
```

Build the package and check out the generated `man/*.Rd` files.

Here are some useful text formatting commands:

- `\code{...}` – typewriter font
- `\emph{...}`, `\strong{...}` – emphasize text
- `\link{...}` – link to another man entry
- `\url{...}` – link to an URL on the Web

Read more: R Core Team, *Writing R extensions*, 2014, Sec. 2. For example, we may also add: enumerated lists, tables, images, math formulæ, etc.

6.3.1 Adding data files

A package may consists in additional data files. We put their definitions to the `data/` directory. Among allowed file formats we find e.g. R scripts (see e.g. `dput()`), CSV, or `.rda` snapshots (cf. `save()`, preferred).

An example:

```
some_object <- iris  # just an example
save(some_object, file = "data/some_object.rda")
rm(some_object)
```

Here is a way to create `some_data`'s documentation. In the `R/some_object.R` file include the following:

```
#' @title ...
#' @description ...
#' @details ...
#'
#' @docType data
#' @format A data frame with 150 rows and 5 variables
#' @name some_object
invisible(NULL)
```

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

Now:

```
data(some_object)
some_object
```

Note that if we add `LazyData: TRUE` to the `DESCRIPTION` file, a call to `data()` will not be necessary.

6.3.2 testthat tests in R packages

Unit tests may be included in the `tests/testthat` directory.

In the `DESCRIPTION` file, add `Suggests: testthat`. Now create a `tests/test-all.R` file:

```
library("testthat")
test_check("testpkg")
```

Click Build → Test package in RStudio.

6.3.3 Vignettes

Apart from man files, additional documentation may be provided in the form of so-called **vignettes**. For example, these may be `.Rnw` (LaTeX) files processed by knitr. Read more: *Writing R extensions*, Sec. 1.4.

6.4 Publishing a package

To "compile" a package's source tarball, use `R CMD build`. This command may be executed by clicking Build → Build source package in RStudio. To check if a package conforms to a list of known "good development practices", use `R CMD CHECK --as-cran`. In this case, the most recent development version of R (compile it from sources) should be used. Also, win-builder.r-project.org/ may be used for that purposes. This command may be invoked by clicking Build → Check package in RStudio.

If a package is on GitHub, everybody may install it by calling `devtools::install_github()`. If we are sure we have made a contribution that is valuable to the R community, we may consider submitting the package to CRAN. Note that this is a peer-reviewed repository, a submission does not necessarily imply its acceptance for publication. The CRAN Repository Policy is quite strict: it tries to guarantee high quality submissions. Also, BioConductor has its own package guidelines.

Exercise: *Do not hesitate to study the source code of other packages for inspiration. For example, download and examine the contents of source tarballs of: testthat, stringdist, etc.*

Exercise: *Create a package containing your own R functions' toolbox.*

6.5 Summary

R packages provide a way to organize "big R projects" and share code with others. For more details, consult the *Writing R extensions* manual. Later on we will discuss now to include Rcpp code in R packages.

6.6 Bibliography

- R Core Team, *Writing R extensions*, 2014, Sec. 1, 2

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund