Informatics

The R language and system
with programming examples
Part 1

Claudio Sartori
Department of Computer Science and Engineering
claudio.sartori@unibo.it
https://www.unibo.it/sitoweb/claudio.sartori/

Learning Objectives

- The types of R
- Structure of an expression
- Operators and operands
- Priority of operators
- The logical values
- Logical expressions
  - De Morgan laws
- The programming constructs of R
- The structured variables of R
- I/O
- Last but not least: algorithms and translation in to R programs

Facts about R

- a system and language for statistic computing and high-quality graphics
- allows interactive usage for medium-complexity computations
- includes a programming language based on the functional paradigm
- open--source https://www.r-project.org
- available powerful open-source front-ends to help the programmer
  - e.g. Rstudio https://www.rstudio.com

R: style of use

- Interactive
  - in this way the user sends a command line to the system and receives an immediate answer or result
- Batch
  - in this way the user prepares a program that can contain any number of command lines and sends it for execution; the program will manage input, output and computations
- Rstudio allows a perfect integration of the two styles

Interactive computing

- At the prompt **>** you can type a *statement* and hit the **<return key>**
- The system will show some reaction, depending on the kind of statements
- The ↑↓ keys allow to roll by the history of the statements previously sent
- A statement can span over several lines

Examples

```
> 1.09 + 1.01
[1] 2.1
> exp(0)
[1] 1
> exp(1)
[1] 2.718282
> log(0)
[1] -Inf
> log(1)
[1] 0
```

Batch computing

- A *program*, i.e. a *sequence of statements,* is written in a text file
  - e.g. using a text editor or a tool of the front-end
- The program is executed in the system
  - The program can/should contain input and output statemens

```
source('relative_or_absolute_path/file_name.R')
```

System environment

- It is the content of the R system memory
- It includes all the *variables* defined in the interactive actions or in the batch program
- Session
  - Starts when R is started
  - The environment, possibly changed during the session, can be saved at the end of the session, when R is closed
  - The saved environment can be restored in a subsequent session

The R system

Main memory

R Studio environment

R Program Editor

View of Session variables

Console

R System environment

Session variable

Session variable

Session variable

What is a variable?

- an area in main memory where we can
  - store a value
  - read the stored value and use it in a computation
- the variable can be accessed through its name
  - the name must be unique
- the variable has a type
  - number
  - string
  - ... more to discuss about types

What is a *statement*

- It is either
  - simple: the smallest executable unit of code that expresses an action to be performed
  - complex: one of
    - sequence of simple statements
    - control statement (more to say...)

Types of statements

- Expression
  - A value computation
- Assignment
  - The name of a *variable*
  - The *assignment operator*
    <-
  - An expression

```
> 3 * 4^2 +12
[1] 60
> x <- 3 * 4^3 + 8
> x
[1] 200
```

Typing an expression

- The expression is computed, giving a value
- The value is shown in ouput

Typing an assignment

- The expression is computed
- If the variable name was not included in the environment then the variable is created in the environment
- The expression value and type are assigned to the variable

Expressions

Expression

- Defines a computation
- An alternate sequence of operands and operators
  - Associate two operands by means of an operator and compute the result
  - E.g. x + 2 uses adds two to the current value of x
- Priority
  - As in standard arithmetics, some operators must be executed before others
  - Each operator has a *priority value*, some operators have the same priority
    - Operators with the same priority are executed left to right
  - Parentheses increase the priority of the included operators

Operand

- A constant
  - It has an immediate value, e.g. a number or a sequence of characters
- A variable
  - Its value is stored in the environment
    - If the variable is not included in the environment, an error is raised
- A function
  - Similar to matematical functions
  - Has *arguments*, that are expressions
  - It is evaluated and gives a value
  - It can have side effects
    - Modifications to the environment

Operand types: constants

- The type of an expression is inspected with the function **`typeof()`**

```
> typeof(2.35)
[1] "double"
> typeof(2)
[1] "double"
> typeof(2L)
[1] "integer"
> typeof("Hello world!")
[1] "character"
> typeof(TRUE)
[1] "logical"
```

The L suffix *forces* the integer type

Operand types: expressions

- The resulting type depends on the operand types and on the operators
- The interpreter checks the compatibility of types and operands

```
> typeof(2.3*5/8)
[1] "double"
> typeof(2L*6L)
[1] "integer"
> typeof(4L/2L)
[1] "double"
> typeof(2L*6)
[1] "double"
> typeof(2<3 & 10<7)
[1] "logical"
```

The *division* operator takes out of the integer domain

Mixed operands result in the more *complex* type

Operand types: variables

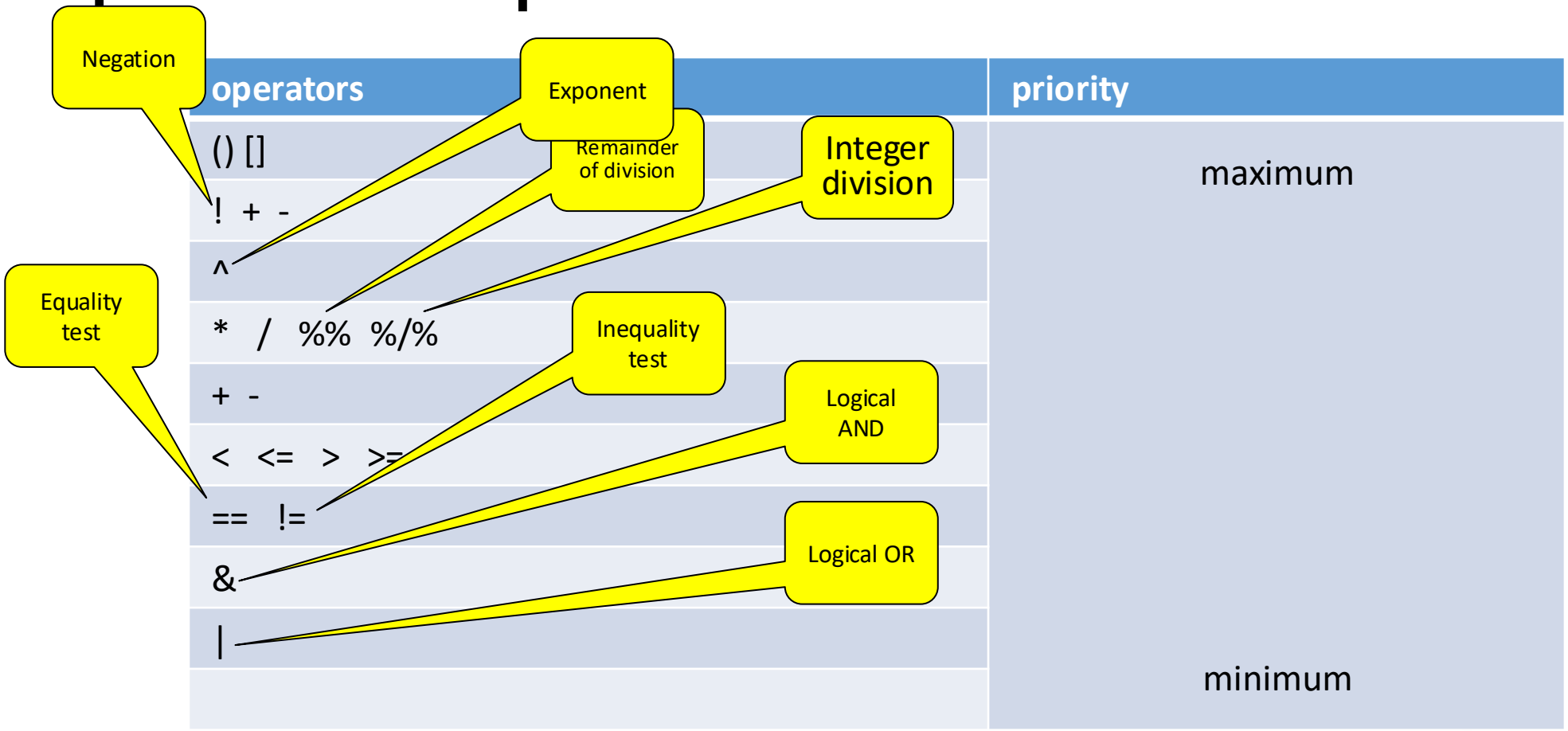
- The type of a variable *is* the type of the last assigned expression

```
> x <- 3/5
> typeof(x)
[1] "double"
> x <- "Hello world!"
> typeof(x)
[1] "character"
```

The interpreter checks the expression and executes it

- Evaluation of formal correctness
  - Alternance of operands and operators
  - Balanced parentheses
  - Well-formed constants
  - Variables already defined
- Preparation of the executable code to implement the operations

Operators priorities



The list is not exhaustive

example

Some examples

```
x * j < j + i * 3
```

```
x == j
```

```
x * y / j + i - 10
```

Data types and expressions

- you cannot apply numeric operators to non-numbers

```
> "a"+1
Error in "a" + 1
```

- TRUE and FALSE, besides being *logical* can also be used as *numbers*, and evaluate to 1 and 0, respectively

```
> TRUE == 1
[1] TRUE
```

Overflow in R

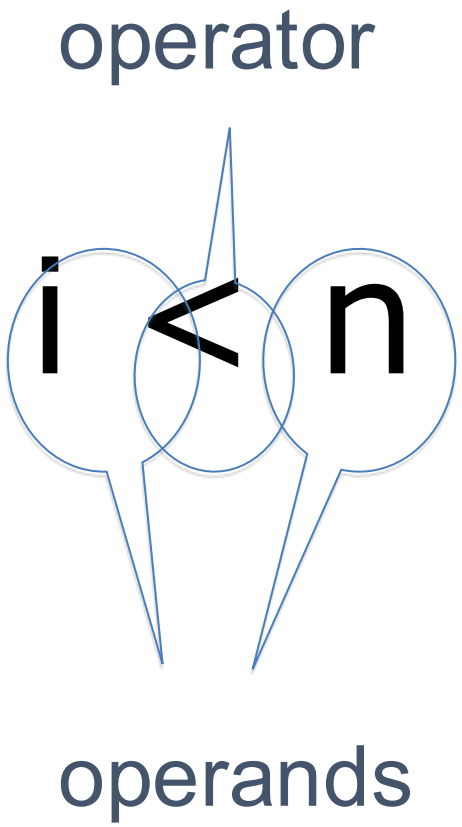
- **When an "integer operation" exceeds the limits the result is Integer Overflow**
- **When a "double" operation exceeds the limits, the result is Inf**

Attention!

```
> x <- 6                  # store in x the value 6
> 1<=x<=10                # is wrong
Error: unexpected `<=' in "0<=x<="
> 1<=x & x<=10            # is correct
[1] TRUE
```

Relational expression

- The possible results are *TRUE* and *FALSE*



Logical-relational expression

```
> i <- 6
> 0 < i & i <=9
[1] TRUE
> i <= 0 | 9 < i
[1] FALSE
```

De Morgan equivalence laws

```
!(a==k & b==h)
!(a==k | b==h)
  h<=x & x<=k
```

$\Leftrightarrow$

```
a!=k | b!=h
a!=k & b!=h
!(x<h | k<x)
```

In general

```
!(condition & condition)
!(condition | condition)
```

$\Leftrightarrow$

```
!condition | !condition
!condition & !condition
```

Logical operators for *conjunction* and *disjunction*

- & and | compute logical conjunction and disjunction (respectively) on both elementary values *and* vectors
  - we will study vectors later on
- && and || compute logical conjunction and disjunction (respectively) only on elementary values, when applied to vectors they consider only the first element

Input from *console*

readline("prompt string")

- Shows the prompt string and waits for input from the user
- The function returns the string typed by the user
- The returned value can be used in an expression and, therefore, stored in a variable
- The returned value is of type *character*

```
> readline("Please, type something: ")
Please, type something: Hello world!
[1] "Hello world!"
> x <- readline("Please, type something: ")
Please, type something: 23
> typeof(x)
[1] "character"
> x <- as.integer(readline(("Please, type something: "))
```

`readline`

- `readline` is a function used to make an interaction from the *user* to the R system
- the argument is a message sent from the system to the user, to specify what is expected

Changing the type of an expression

- A wide set of functions to transform types
- Among others, dates are internally represented as numbers and can be added/subtracted

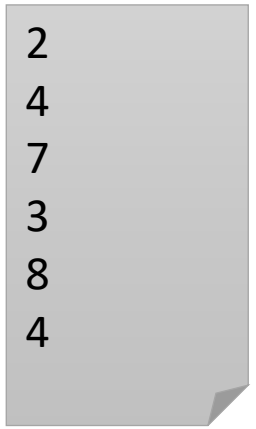
```
as.integer()
as.double()
as.character()
as.Date()
…
> x<-as.Date("2016/2/29")
> y<-as.Date("2016/2/28")
> x-y
Time difference of 1 days
```

Input from file

Generates as many elements as found in the file

```
> v <- scan("text1.txt")# by default reads double
Read 6 items
> print (v)
[1] 2 4 7 3 8 4
```

text1.txt



Input from file – more types

```
> cv <- scan("text2.txt", what = "character")
Read 4 items
> print(cv)
[1] "aa" "bb" "cc" "dd"
```

- Lots of scan options, see for full reference https://stat.ethz.ch/R-manual/R-devel/library/base/html/scan.html

Output

- Display the value of one or more expressions
- The **`cat()`** function
  - Concatenate and print
  - Similar to print, but with more detailed control on new lines
  - More suitable for batch programs
- The **`print()`** function
  - displays one expression
  - displays a complex object
    - examples later

Forces *new line*

```
> A <- 25
> cat("Value of A: ",A,"\n")
Value of A:  25
> print(paste("Value of A: ",A))
[1] "Value of A:  3"
```

Sequence

- statements written one after the other
- they will be executed one after the other, till the end
  - unless *special statements* are encoutered, such as `break` (explained later)
- a sequence can be grouped inside a pair of *braces* **{}**
- a sequence in braces can be included in *control statements*
  - *repetition*
  - *conditional*
  - *function definition*