# A Critique of ***R***
from the Perspective of Programming Language Theory

Sungwoo Park
Pohang University of Science and Technology, Korea

Japanese ***R*** Use***R***'s meeting

Dec 8, 2006

First Encounter with R

- A regional workshop on R in May, 2006
  - motto:

    "Don't teach SAS. Teach R instead."

- An invited talk at the workshop
  - supposed to say ***"SAS is bad. R is good."***
  - actually said ***"SAS is really bad, R is also bad."***

- R seemed to have quite a few flaws in its design.

'Towards 2020 Science'

- A report on

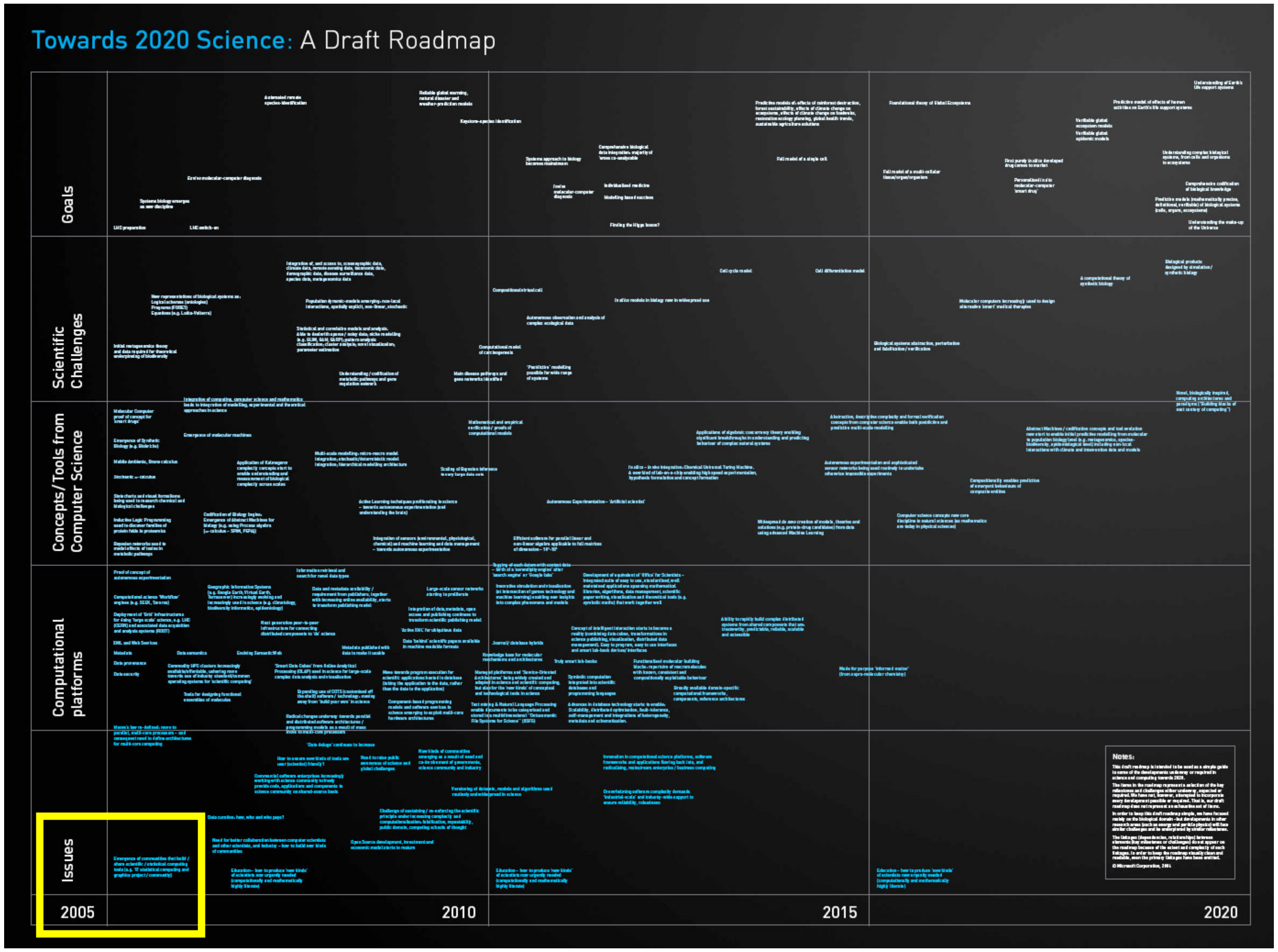
  "the role and future of science over the next 14 years"

  – by the 2020 Science Group
    - over 30 scientists elected for their expertise
    - met over an intense 3 days in July 2005

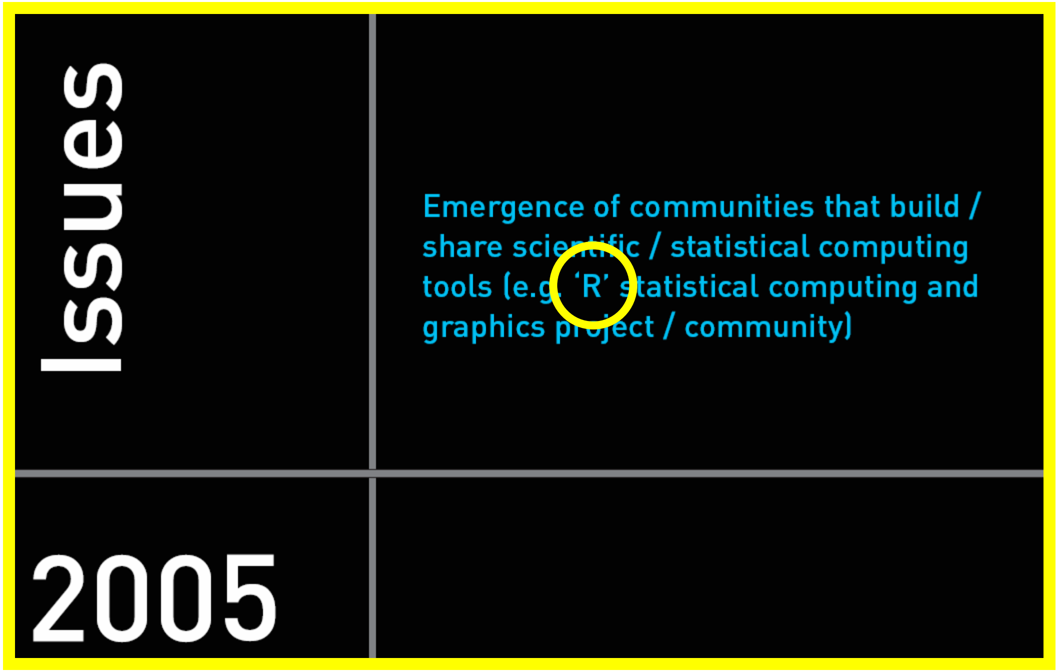
  – 86 pages

  – sponsored by Microsoft

Towards 2020 Science: A Draft Roadmap



R to Be Reckoned With



Issues

Emergence of communities that build / share scientific / statistical computing tools (e.g. 'R' statistical computing and graphics project / community)

2005

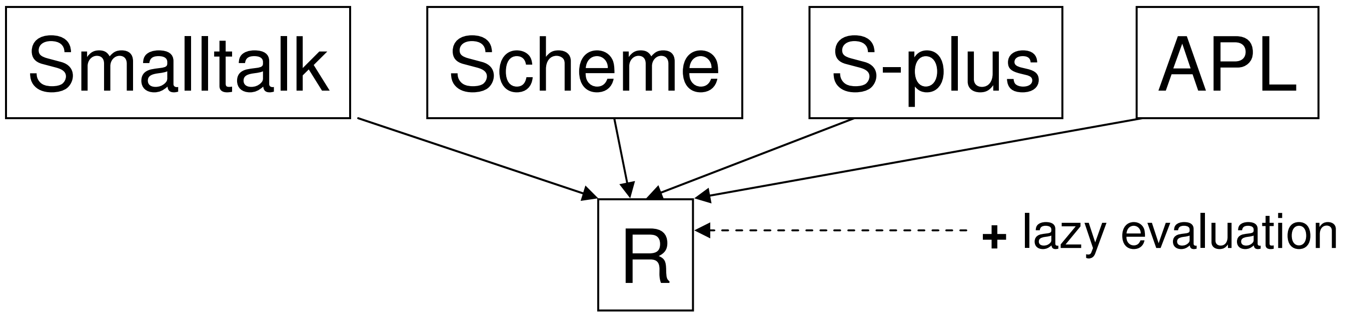
*"Many niche areas of software development exist where alternatives and/or enhancements of managed platforms are deployed and used by scientists, including ... and **the language R**."*

R Dissected

- Popularity of R in the statistics community
  - statistical computing
  - high level graphics

    "Many users will come to R mainly for its graphical facilities." – An introduction to R

- R as a hybrid language



Caveat

- A technical debate on
  "Why is your programming language good/bad?"
  $\approx$
  A religious debate on
  "What is the best religion?"

  $\Rightarrow$ **Take this presentation with a grain of salt.**

- One thing is certain, however:
  "More features do not always mean
  a better programming language."

Outline

- Introduction ✓
- **Programming paradigm for R**
  - **Imperative language?**
  - **Functional language?**
  - **Both?**
  - **Or neither?**
- Lexical scoping
- Further analysis
- A functional language for R users
- Conclusion

Imperative vs Functional

Imperative languages
- Everything denotes a command.
- Variables are mutable.
- Functions are not first-class objects.

Functional languages
- Everything denotes a value.
- Variables are immutable.
- **<u>Functions as first-class objects.</u>**

Functions are first-class objects in R.
Does this mean that R is a functional language?

Imperative Languages

- A program consists of **commands**.
  - command = "do something"
- Nothing wrong:

```
if (x == 1)
    x ← x + 1
else
    x ← x - 1
```

- Nothing wrong either:

```
if (x == 1)
    x ← x + 1
```

Functional Languages

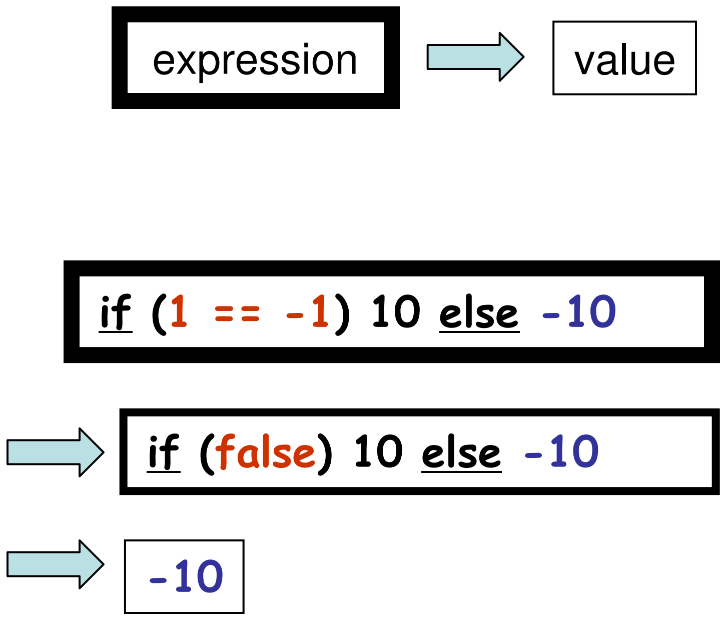
- A program consists of **expressions**.
  - expression = "obtain a **value**"
- Nothing wrong:

```
if (x == 1)
    x + 1
else
    x - 1
```

- But this does not make sense:

```
if (x == 1)
    x + 1
```

  - What is the value if $x \neq 1$?



```
if (1 == -1) 10 else -10
if (false) 10 else -10
-10
```

R: Not Functional

```
> foo
function (x) {
    if (x <= 0) 1
  }
> foo (0)
[1] 1
> foo (1)
> foo (0) + foo (1)
numeric(0)
```

if (x <= 0) 1 is not an expression: it does not always evaluate to a value. **foo** is not a function: it is not defined on positive integers.

numeric(0) is what?

Variable Binding

```
> x = 1 + 1
> x
[1] 2
```

- A variable x is “**bound**” to value **2**.
- From now on, any occurrence of **x** is replaced by **2**.

```
> y = x + x
> y
[1] 4
```

Variables are NOT Variable?

Imperative languages

- The contents of a variable can change.

```
> x ← 0
> x
[1] 0
> x ← 1
> x
[1] 1
```

Functional languages

- The contents of a variable **never** change.

⇒ You **cannot** assign a new value to a variable.

- Surprise?

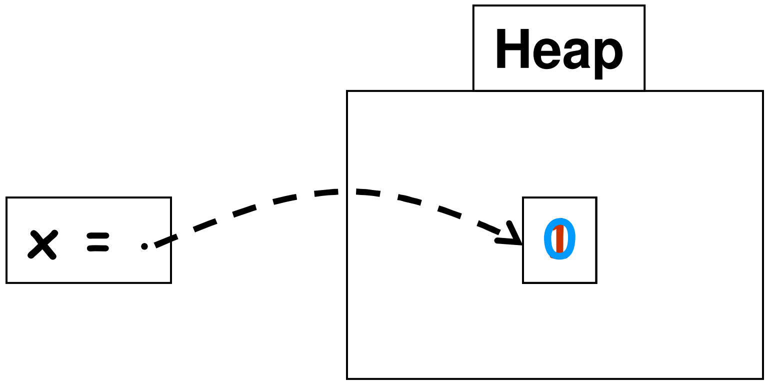
⇒ nothing special in functional languages

So, R is an imperative language?

References in Functional Languages

- There are assignments, but not to variables.
  ⇒ assignments to **references.**
- Reference (≈ pointer in C)
  – points to a heap cell.

```
- val x = ref 0;          // initialization
val x = ref 0 : int ref
- !x;                     // dereferencing
val it = 0 : int
- x := 1;                 // assignment
val it = () : unit
- !x;                     // dereferencing
val it = 1 : int
```



R: Neither Functional Nor Imperative

Imperative languages

- Everything denotes a command.
- **Variables are mutable.**
- Functions are not first-class objects.

Functional languages

- Everything denotes a value.
- Variables are immutable.
- **Functions as first-class objects**

- Functions are first-class objects, but
  no clear definition of **<u>commands</u>** or **<u>expressions</u>**
  no distinction between **<u>variables</u>** and **<u>references</u>**
- A fatal design decision
  $\Rightarrow$ engenders many idiosyncrasies in the definition.

Outline

- Introduction ✓
- Programming paradigm for R ✓
- **Lexical scoping**
- Further analysis
- A functional language for R users
- Conclusion

Lexical Scoping

- Uses bindings that are active
  at the time of creating a function.

```
y ← 0
foo ←
  function () {
    y ← 100
    function (x) x + y
  }
> foo () (0)
[1] 100
```

- Useful in R because functions are first-class objects.
- Unfortunately R fails to implement lexical scoping correctly.

No Lexical Scoping

- R

```
> x ← 1
> foo ← function (y) x + y
> x ← 100
> foo (0)
[1] 100
```

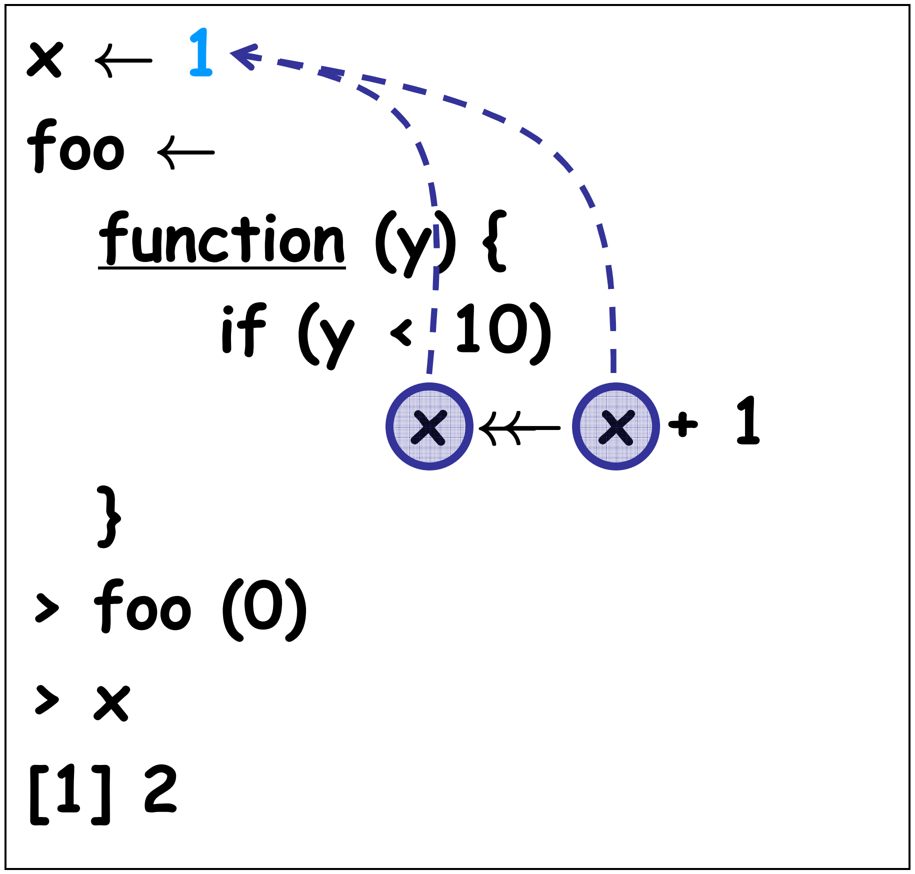
- Standard ML

```
val x = 1
val foo = fn y => x + y
val x = 100
- foo 0;
val it = 1 : int
```

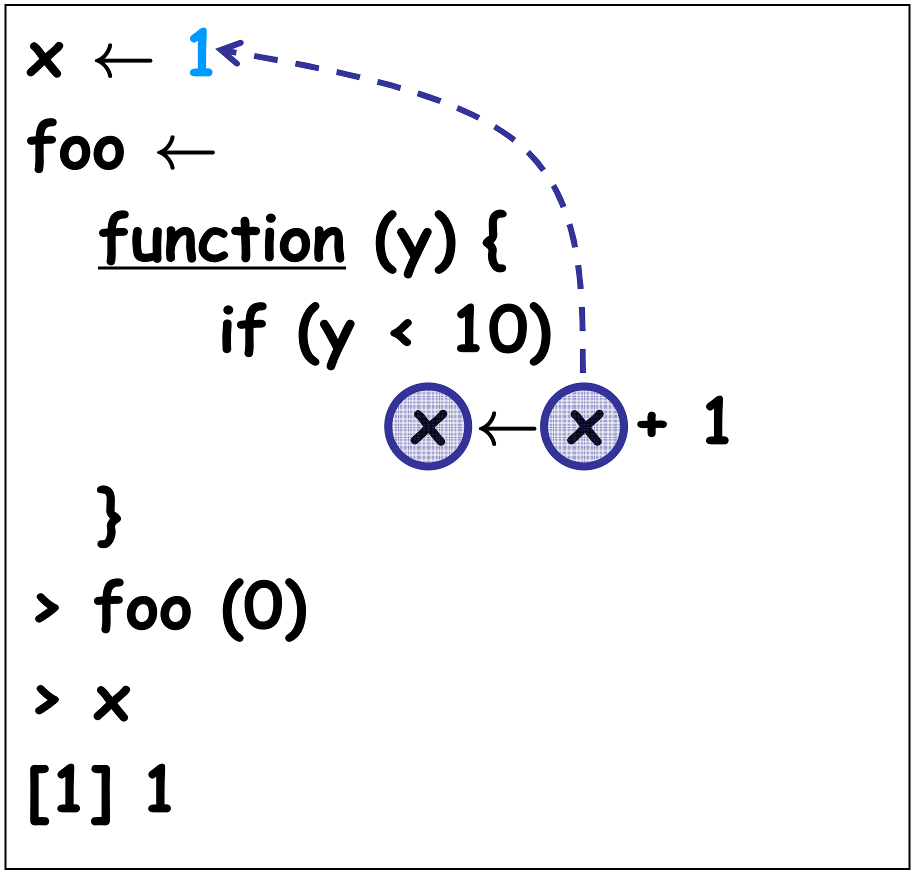
- Dynamic scoping at the top-level
- lexical scoping at inner levels

  ⇒ for the sake of compatibility?

↞ VS ←



```
x ← 1
foo ←
   function (y) {
        if (y < 10)
              x ↞ x + 1
   }
> foo (0)
> x
[1] 2
```



```
x ← 1
foo ←
   function (y) {
        if (y < 10)
              x ← x + 1
   }
> foo (0)
> x
[1] 1
```

Special Top Level?

*"While purely functional languages do not allow assignment, **<u>they allow it at top-level</u>**; otherwise the user could not define new functions."*

– Lexical Scope and Statistical Computing

$\Rightarrow$ Wrong!

- There is nothing special for the top level.
- assignment at the top level?
  - No, it's just a **<u>binding</u>**.

- Due to failure to distinguish between
  <u>variables</u> and **<u>references</u>**, or
  <u>bindings</u> and **<u>assignments</u>**.

Lexical Scoping in CS

*"Although the usual definition of static or lexical scope in computer science is that ..., **<u>this definition is not specific enough.</u>** Computer scientists tend not to differentiate as finely because **<u>their concerns are different.</u>**"*

– Lexical Scope and Statistical Computing

$\Rightarrow$ This is absolutely wrong.

$$\begin{array}{rrcl} \text{expression} & e & ::= & x \mid \lambda x{:}A.\,e \mid e\ e \\ \text{environment} & \eta & ::= & \cdot \mid \eta, x \hookrightarrow v \end{array}$$

$$\frac{x \hookrightarrow v \in \eta}{\eta \vdash x \hookrightarrow v}\ \mathbf{Var}_{\mathsf{e}} \qquad \frac{}{\eta \vdash \lambda x{:}A.\,e \hookrightarrow [\eta, \lambda x{:}A.\,e]}\ \mathbf{Lam}_{\mathsf{e}}$$

$$\frac{\eta \vdash e_1 \hookrightarrow [\eta', \lambda x{:}A.\,e] \quad \eta \vdash e_2 \hookrightarrow v_2 \quad \eta', x \hookrightarrow v_2 \vdash e \hookrightarrow v}{\eta \vdash e_1\ e_2 \hookrightarrow v}\ \mathbf{App}_{\mathsf{e}}$$

Outline

- Introduction *V*
- Programming paradigm for R *V*
- Lexical scoping *V*
- **Further analysis**
- A functional language for R users
- Conclusion

R-TYPE

- Dynamic type binding
  - An R object can change its type during the computation.

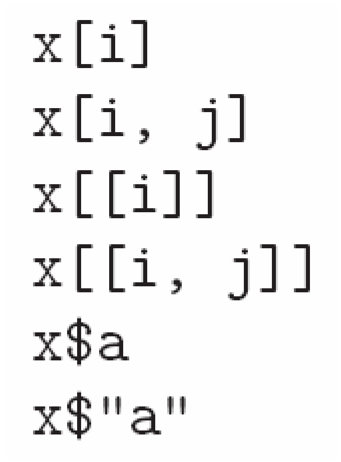
```
x ← c(1.0, 2.0, 3.0)
x ← 47
```

  - **typeof** returns the type of an R object.
    - **symbol**, **pairlist**, **closure**, **environment**

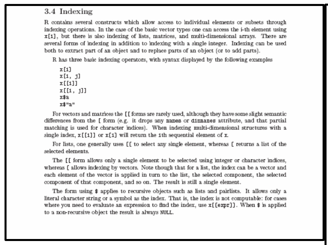
- Is it good? ⇒ philosophical debate
  - dynamic type binding is good for:
    - quick, small programming tasks
  - static type binding is good for:
    - large programming tasks

Complex Semantics

- Ex. *Section 3.4 Indexing* in *R Language Definition*



```
x[i]
x[i, j]
x[[i]]
x[[i, j]]
x$a
x$"a"
```



3.4 Indexing

R contains several constructs which allow access to individual elements or subsets through indexing operations. In the case of the basic vector types one can access the i-th element using `x[i]`, but there is also indexing of lists, matrices, and multi-dimensional arrays. There are several forms of indexing in addition to indexing with a single integer. Indexing can be used both to extract part of an object and to replace parts of an object (or to add parts).

R has three basic indexing operators, with syntax displayed by the following examples

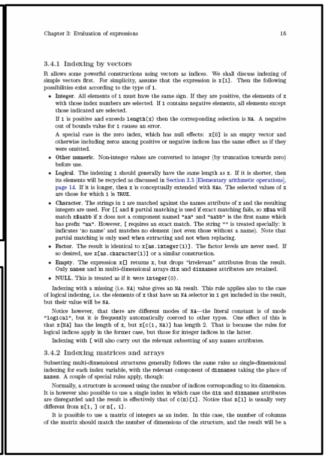
```
x[i]
x[i, j]
x[[i]]
x[[i, j]]
x$a
x$"a"
```

For vectors and matrices the `[[` forms are rarely used, although they have some slight semantic differences from the `[` form (e.g. it drops any `names` or `dimnames` attribute, and that partial matching is used for character indices). When indexing multi-dimensional structures with a single index, `x[[i]]` or `x[i]` will return the `i`th sequential element of `x`.

For lists, one generally uses `[[` to select any single element, whereas `[` returns a list of the selected elements.

The `[[` form allows only a single element to be selected using integer or character indices, whereas `[` allows indexing by vectors. Note though that for a list, the index can be a vector and each element of the vector is applied in turn to the list, the selected component, the selected component of that component, and so on. The result is still a single element.

The form using `$` applies to recursive objects such as lists and pairlists. It allows only a literal character string or a symbol as the index. That is, the index is not computable: for cases where you need to evaluate an expression to find the index, use `x[[expr]]`. When `$` is applied to a non-recursive object the result is always `NULL`.



3.4.1 Indexing by vectors

R allows some powerful constructions using vectors as indices. We shall discuss indexing of simple vectors first. For simplicity, assume that the expression is `x[i]`. Then the following possibilities exist according to the type of `i`.

- **Integer**. All elements of `i` must have the same sign. If they are positive, the elements of `x` with those index numbers are selected. If `i` contains negative elements, all elements except those indicated are selected.

  If `i` is positive and exceeds `length(x)` then the corresponding selection is `NA`. A negative out of bounds value for `i` causes an error.

  A special case is the zero index, which has null effects: `x[0]` is an empty vector and otherwise including zeros among positive or negative indices has the same effect as if they were omitted.
- **Other numeric**. Non-integer values are converted to integer (by truncation towards zero) before use.
- **Logical**. The indexing `i` should generally have the same length as `x`. If it is shorter, then its elements will be recycled as discussed in Section 3.3 [Elementary arithmetic operations], page 14. If it is longer, then `x` is conceptually extended with `NA`s. The selected values of `x` are those for which `i` is `TRUE`.
- **Character**. The strings in `i` are matched against the names attribute of `x` and the resulting integers are used. For `[[` and `$` partial matching is used if exact matching fails, so `x$aa` will match `x$aabb` if `x` does not a component named `"aa"` and `"aabb"` is the first name which has prefix `"aa"`. However, `[` requires an exact match. The string `""` is treated specially: it indicates 'no name' and matches no element (not even those without a name). Note that partial matching is only used when extracting and not when replacing.
- **Factor**. The result is identical to `x[as.integer(i)]`. The factor levels are never used. If so desired, use `x[as.character(i)]` or a similar construction.
- **Empty**. The expression `x[]` returns `x`, but drops "irrelevant" attributes from the result. Only `names` and in multi-dimensional arrays `dim` and `dimnames` attributes are retained.
- **NULL**. This is treated as if it were `integer(0)`.

Indexing with a missing (i.e. `NA`) value gives an `NA` result. This rule applies also to the case of logical indexing, i.e. the elements of `x` that have an `NA` selector in `i` get included in the result, but their value will be `NA`.

Notice however, that there are different modes of `NA`—the literal constant is of mode `"logical"`, but it is frequently automatically coerced to other types. One effect of this is that `x[NA]` has the length of `x`, but `x[c(1, NA)]` has length 2. That is because the rules for logical indices apply in the former case, but those for integer indices in the latter.

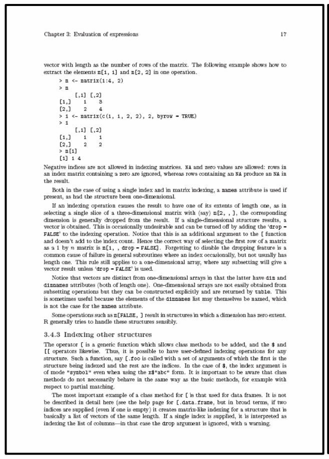
Indexing with `[` will also carry out the relevant subsetting of any names attributes.

3.4.2 Indexing matrices and arrays

Subsetting multi-dimensional structures generally follows the same rules as single-dimensional indexing for each index variable, with the relevant component of `dimnames` taking the place of `names`. A couple of special rules apply, though:

Normally, a structure is accessed using the number of indices corresponding to its dimension. It is however also possible to use a single index in which case the `dim` and `dimnames` attributes are disregarded and the result is effectively that of `c(m)[i]`. Notice that `m[1]` is usually very different from `m[1, ]` or `m[, 1]`.

It is possible to use a matrix of integers as an index. In this case, the number of columns of the matrix should match the number of dimensions of the structure, and the result will be a



vector with length as the number of rows of the matrix. The following example shows how to extract the elements `m[1, 1]` and `m[2, 2]` in one operation.

```
> m <- matrix(1:4, 2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> i <- matrix(c(1, 1, 2, 2), 2, byrow = TRUE)
> i
     [,1] [,2]
[1,]    1    1
[2,]    2    2
> m[i]
[1] 1 4
```

Negative indices are not allowed in indexing matrices. `NA` and zero values are allowed: rows in an index matrix containing a zero are ignored, whereas rows containing an `NA` produce an `NA` in the result.

Both in the case of using a single index and in matrix indexing, a `names` attribute is used if present, as had the structure been one-dimensional.

If an indexing operation causes the result to have one of its extents of length one, as in selecting a single slice of a three-dimensional matrix with (say) `m[2, , ]`, the corresponding dimension is generally dropped from the result. If a single-dimensional structure results, a vector is obtained. This is occasionally undesirable and can be turned off by adding the '`drop = FALSE`' to the indexing operation. Notice that this is an additional argument to the `[` function and doesn't add to the index count. Hence the correct way of selecting the first row of a matrix as a 1 by n matrix is `m[1, , drop = FALSE]`. Forgetting to disable the dropping feature is a common cause of failure in general subroutines where an index occasionally, but not usually has length one. This rule still applies to a one-dimensional array, where any subsetting will give a vector result unless '`drop = FALSE`' is used.

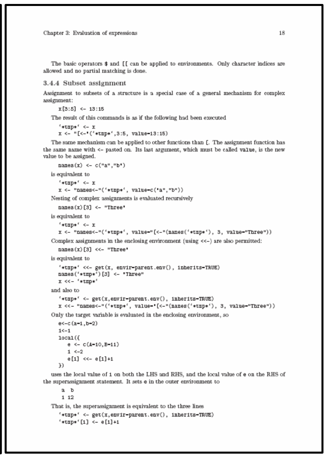
Notice that vectors are distinct from one-dimensional arrays in that the latter have `dim` and `dimnames` attributes (both of length one). One-dimensional arrays are not easily obtained from subsetting operations but they can be constructed explicitly and are returned by `table`. This is sometimes useful because the elements of the `dimnames` list may themselves be named, which is not the case for the `names` attribute.

Some operations such as `m[FALSE, ]` result in structures in which a dimension has zero extent. R generally tries to handle these structures sensibly.

3.4.3 Indexing other structures

The operator `[` is a generic function which allows class methods to be added, and the `$` and `[[` operators likewise. Thus, it is possible to have user-defined indexing operations for any structure. Such a function, say `[.foo` is called with a set of arguments of which the first is the structure being indexed and the rest are the indices. In the case of `$`, the index argument is of mode `"symbol"` even when using the `x$"abc"` form. It is important to be aware that class methods do not necessarily behave in the same way as the basic methods, for example with respect to partial matching.

The most important example of a class method for `[` is that used for data frames. It is not be described in detail here (see the help page for `[.data.frame`, but in broad terms, if two indices are supplied (even if one is empty) it creates matrix-like indexing for a structure that is basically a list of vectors of the same length. If a single index is supplied, it is interpreted as indexing the list of columns—in that case the `drop` argument is ignored, with a warning.



The basic operators `$` and `[[` can be applied to environments. Only character indices are allowed and no partial matching is done.

3.4.4 Subset assignment

Assignment to subsets of a structure is a special case of a general mechanism for complex assignment:

```
x[3:5] <- 13:15
```

The result of this commands is as if the following had been executed

```
'*tmp*' <- x
x <- "[<-"('*tmp*',3:5, value=13:15)
```

The same mechanism can be applied to other functions than `[`. The assignment function has the same name with `<-` pasted on. Its last argument, which must be called `value`, is the new value to be assigned.

```
names(x) <- c("a","b")
```

is equivalent to

```
'*tmp*' <- x
x <- "names<-"('*tmp*', value=c("a","b"))
```

Nesting of complex assignments is evaluated recursively

```
names(x)[3] <- "Three"
```

is equivalent to

```
'*tmp*' <- x
x <- "names<-"('*tmp*', value="[<-"(names('*tmp*'), 3, value="Three"))
```

Complex assignments in the enclosing environment (using `<<-`) are also permitted:

```
names(x)[3] <<- "Three"
```

is equivalent to

```
'*tmp*' <<- get(x, envir=parent.env(), inherits=TRUE)
names('*tmp*')[3] <- "Three"
x <<- '*tmp*'
```

and also to

```
'*tmp*' <- get(x,envir=parent.env(), inherits=TRUE)
x <<- "names<-"('*tmp*', value="[<-"(names('*tmp*'), 3, value="Three"))
```

Only the target variable is evaluated in the enclosing environment, so

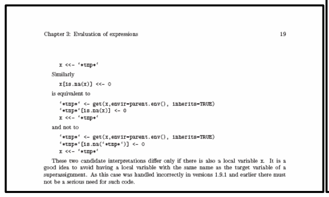
```
e<-c(a=1,b=2)
i<-1
local({
   e <- c(A=10,B=11)
   i <-2
   e[i] <<- e[i]+1
})
```

uses the local value of `i` on both the LHS and RHS, and the local value of `e` on the RHS of the superassignment statement. It sets `e` in the outer environment to

```
 a  b
 1 12
```

That is, the superassignment is equivalent to the three lines

```
'*tmp*' <- get(x,envir=parent.env(), inherits=TRUE)
'*tmp*'[i] <- e[i]+1
```



```
x <<- '*tmp*'
```

Similarly

```
x[is.na(x)] <<- 0
```

is equivalent to

```
'*tmp*' <- get(x,envir=parent.env(), inherits=TRUE)
'*tmp*'[is.na(x)] <- 0
x <<- '*tmp*'
```

and not to

```
'*tmp*' <- get(x,envir=parent.env(), inherits=TRUE)
'*tmp*'[is.na('*tmp*')] <- 0
x <<- '*tmp*'
```

These two candidate interpretations differ only if there is also a local variable `x`. It is a good idea to avoid having a local variable with the same name as the target variable of a superassignment. As this case was handled incorrectly in versions 1.9.1 and earlier there must not be a serious need for such code.

- Why on earth such "a" complex semantics for statistical computing?

So Many Complex/Special Cases

- From *R Language Definition*
  - *"Another more subtle difference is ..."*
  - *"... evaluated in some unexpected cases."*
  - *"... can lead to surprises."*
  - *"In a very few cases, ..."*
  - *"... in certain (rather rare) circumstances, ..."*
  - *"... are treated specially."*
  - *"... should be done with caution."*
  - *"A couple of special rules apply, though:"*
  - *"... is not guaranteed to hold in all implementations."*
  - *"is not generally handled correctly."*
  - *"The special exception for ... is admittedly peculiar."*

Evolution or Degeneration?

"R appears to be working fine."

"??? seems often useful, so let's add it to R."

"Now ??? is available, but there is something fishy going on."

- Example of ??? = first-class functions

  "This ability is rarely used even though it is ***potentially very powerful****." – Lexical Scope and Statistical Computing*

  – incorporating first-class functions without expressions and bindings

  ⇒ fitting a square peg into a round hole

- The worst example of ??? is yet to come, however.

Lazy Evaluation

*"A policy of lazy arguments is **very useful** because ... This **can be very useful** for specifying functions or models in symbolic form."*
– R: A Language for Data Analysis and Graphics

- Evaluation strategy of R
  - eager evaluation for built-in functions: fully evaluate arguments
  - lazy evaluation for *promise* objects: evaluate only when necessary
- Yes, lazy evaluation is a great idea.
  - Ex. Haskell
  - **But only if all functions are pure mathematical functions.**
- Lazy evaluation + computational effects ⇒ total complete mess
  - computational effects (= side effects)
    - **plot**, **print**, vector update, assignments
  - Functions in R are not mathematical functions anyway.
  - Solution from programming language theory = monad
- Besides lazy evaluation in R is **not** really lazy evaluation!

Meta-programming in R

- **quote** creates unevaluated expressions.
- **eval** treats programs as data.

```
> e <- quote (2 + 2)
> v <- eval (e)
```

- Useful constructs? Yes!
  - implementing compilers, staged computation, and so on
- But do you really need **quote**, **eval**, **deparse**, **substitute** for statistical computing?

*"More frequently, one wants to ... in order to deparse it and use it for **labeling plots**, for instance." – R Language definition*

$\Rightarrow$ launching a nuclear missile to kill a fly

Why Not Use First-Class Functions?

- A weird program exploiting lazy evaluation and **eval**

```
curve ← function (expr, from, to) {
    x ← seq (from, to, length=500)
    y ← eval (substitute (expr))
    plot(x, y, type="l")
}
curve (x^2 - 1, -2, 2)
```

This function call does not make sense.
⇒ misunderstanding of lazy evaluation!

- A quick fix = use a first-class function

```
curve ← function (f, from, to) ...
curve (function (x) x^2- 1, -2, 2)
```

Other Minor (Yet Serious) Points

- Maintaining state within functions

  *"The ability to preserve state information between function invocations is a **very useful** feature ..."*

  – R: A Language for Data Analysis and Graphics

  $\Rightarrow$ a trivial exercise in functional programming

- Confusion between definition and implementation

  "To understand completely the rules ..., the reader needs to be familiar with the notion of an evaluation frame."

  – An Introduction to R

  – Specific implementation strategies are taken as part of the definition.
    - environment, closure, call stack, evaluation frame, ...

Outline

- Introduction *V*
- Programming paradigm for R *V*
- Lexical scoping *V*
- Further analysis *V*
- **A functional language for R users**
- Conclusion

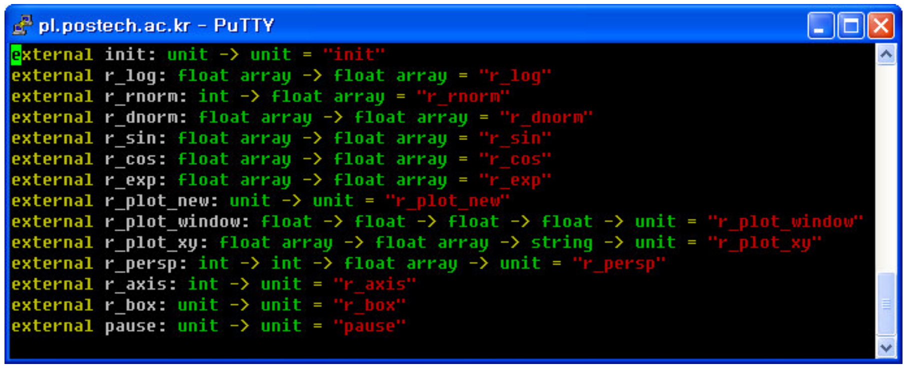
Next Generation R?

- Claim
  1. Admit it or not, R is an ill-designed language.
  2. Nevertheless, R is too juicy to give up:
     - statistical computing
     - high level graphics
  3. R shares a lot in common with functional languages.
- Plan
  – extend an existing functional language with an interface to the R base library.

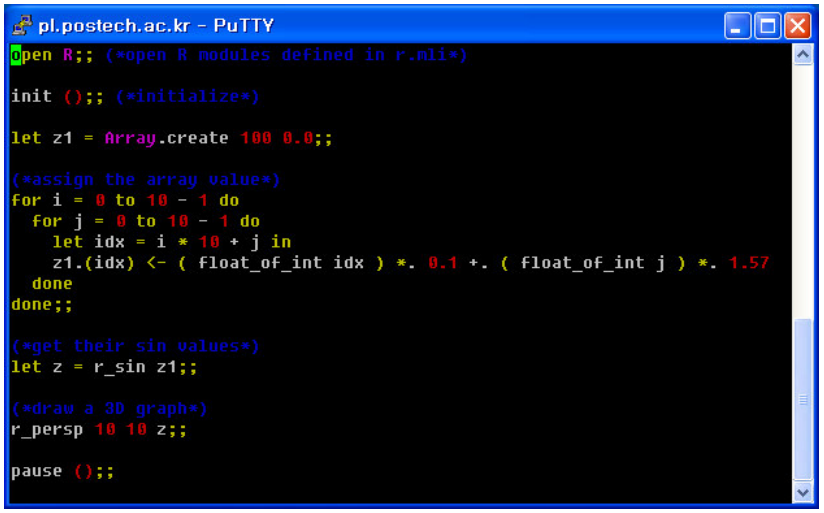
Objective CAML with R

- Objective CAML
  – industrial strength functional language
  – rough speed comparison
    - nearly as fast as, or sometimes faster than, C
    - consistently faster than C++
    - about 10 times faster than Matlab
  – strong type system (based on type theory)
    - **significantly** less development time than in C
    - more reliable code than in C
  – huge library contributed by users
  – free!
- Let's develop an Objective CAML interface to R!

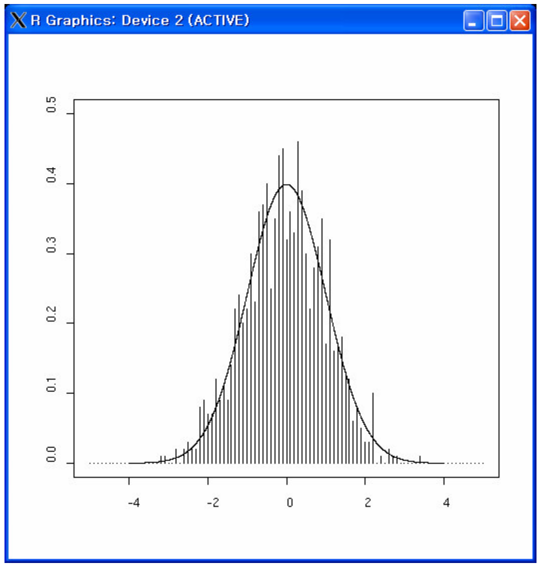
Preliminary Results

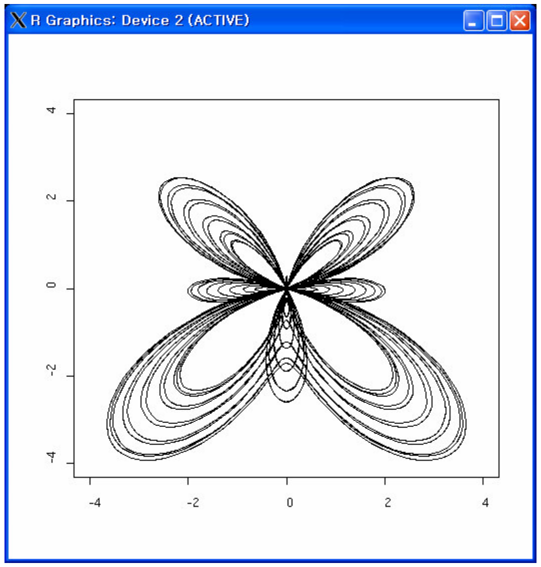


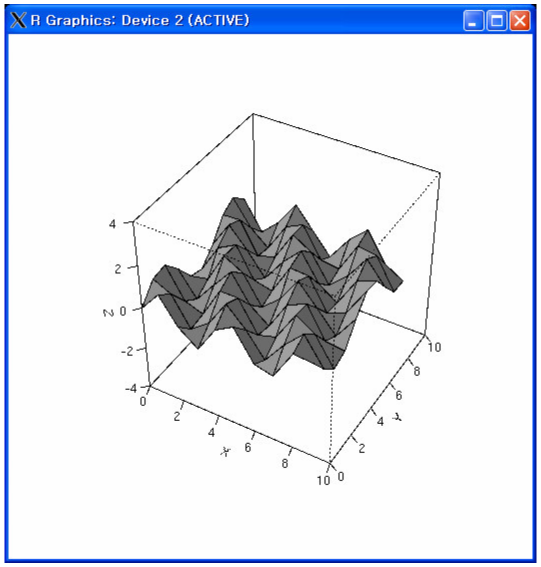
```
external init: unit -> unit = "init"
external r_log: float array -> float array = "r_log"
external r_rnorm: int -> float array = "r_rnorm"
external r_dnorm: float array -> float array = "r_dnorm"
external r_sin: float array -> float array = "r_sin"
external r_cos: float array -> float array = "r_cos"
external r_exp: float array -> float array = "r_exp"
external r_plot_new: unit -> unit = "r_plot_new"
external r_plot_window: float -> float -> float -> float -> unit = "r_plot_window"
external r_plot_xy: float array -> float array -> string -> unit = "r_plot_xy"
external r_persp: int -> int -> float array -> unit = "r_persp"
external r_axis: int -> unit = "r_axis"
external r_box: unit -> unit = "r_box"
external pause: unit -> unit = "pause"
```



```
open R;; (*open R modules defined in r.mli*)

init ();; (*initialize*)

let z1 = Array.create 100 0.0;;

(*assign the array value*)
for i = 0 to 10 - 1 do
  for j = 0 to 10 - 1 do
    let idx = i * 10 + j in
    z1.(idx) <- ( float_of_int idx ) *. 0.1 +. ( float_of_int j ) *. 1.57
  done
done;;

(*get their sin values*)
let z = r_sin z1;;

(*draw a 3D graph*)
r_persp 10 10 z;;

pause ();;
```







Outline

- Introduction ✓
- Programming paradigm for R ✓
- Lexical scoping ✓
- Further analysis ✓
- A functional language for R users ✓
- **Conclusion**

Summary

- R is great!
  - library for statistical computing
  - library for publication quality graphics
  - the whole statistics community actively contributing new libraries
- R is an ill-designed language, however.
- So, it's time to act.
  - just use programming language theory!

Thanks a lot!