Specifications of R Language

R is a system for statistical computation and graphics. It provides, among other things, a programming language, high level graphics, interfaces to other languages and debugging facilities. This manual details and defines the R language. The language syntax has a superficial similarity with C, but the semantics are of the FPL (functional programming language) variety.

R does not provide direct access to the computer's memory but rather provides a number of specialized data structures we will refer to as objects. These objects are referred to through symbols or variables. In R, however, the symbols are themselves objects and can be manipulated in the same way as any other object. This is different from many other languages and has wide ranging effects. The R specific function typeof returns the type of an R object. The following table describes the possible values returned by typeof and what they are:

"NULL"	NULL "symbol" a variable name
"pairlist"	a pairlist object (mainly internal)
"closure"	a function "environment" an environment
"promise"	an object used to implement lazy evaluation
"Language"	an R language construct
"special"	an internal function that does not evaluate its arguments
"builtin"	an internal function that evaluates its arguments
"char"	a 'scalar' string object (internal only)
"logical"	a vector containing logical values
"integer"	a vector containing integer values
"double"	a vector containing real values
"complex"	a vector containing complex values
"character"	a vector containing character values
"..."	the special variable length argument
"any"	a special type that matches all types: there are no objects of this type
"expression"	an expression object
"list"	a list
"bytecode"	byte code (internal only)

"externalptr" an external pointer object
"weakref" a weak reference object
"raw" a vector containing bytes
"S4" an S4 object which is not a simple object

Function mode gives information about the mode of an object.

```
> x <- 1:3
> typeof(x)
[1] "integer"
> mode(x)
[1] "numeric"
> storage.mode(x)
[1] "integer"
```

BASIC TYPES

Vectors - Vectors can be thought of as contiguous cells containing data. Cells are accessed through indexing operations such as x[5]. R has six basic (‘atomic’) vector types: logical, integer, real, complex, string (or character) and raw.

Lists - Lists (“generic vectors”) are another kind of data storage. Lists have elements, each of which can contain any type of R object, i.e. the elements of a list do not have to be of the same type. List elements are accessed through three different indexing operations.

Language Objects - There are three types of objects that constitute the R language. They are calls, expressions, and names. They can be created directly from expressions using the quote mechanism and converted to and from lists by the as.list and as.call functions.

Symbol Objects - Symbols refer to R objects. The name of any R object is usually a symbol. Symbols can be created through the functions as.name and quote.

Expression Objects - In R one can have objects of type "expression". An expression contains one or more statements. A statement is a syntactically correct collection of

tokens. Expression objects are special language objects which contain parsed but unevaluated R statements.

Function Objects - In R functions are objects and can be manipulated in much the same way as any other object. Functions (or more precisely, function closures) have three basic components: a formal argument list, a body and an environment.

Evaluation of Expressions

When a user types a command at the prompt (or when an expression is read from a file) the first thing that happens to it is that the command is transformed by the parser into an internal representation. The evaluator executes parsed R expressions and returns the value of the expression. All expressions have a value. This is the core of the language.

Constants - Any number typed directly at the prompt is a constant and is evaluated.

```
> 1
[1] 1
```

Symbol Lookup - When a new variable is created it must have a name so it can be referenced and it usually has a value. The name itself is a symbol. When a symbol is evaluated its value is returned. Later we shall explain in detail how to determine the value associated with a symbol.

```
> y <- 4
> y
[1] 4
```

Function Calls - Most of the computations carried out in R involve the evaluation of functions. We will also refer to this as function invocation. Functions are invoked by name with a list of arguments separated by commas.

```
> mean(1:10)
[1] 5.5
```

R contains a huge number of functions with different purposes. Most are used for producing a result which is an R object, but others are used for their side effects, e.g., printing and plotting functions.

Operators - R allows the use of arithmetic expressions using operators similar to those of the C programming language, for instance

```
> 1 + 2
[1] 3
```

Expressions can be grouped using parentheses, mixed with function calls, and assigned to variables in a straightforward manner

```
> y <- 2 * (a + log(x))
```

Control Structures

Computation in R consists of sequentially evaluating statements. Statements, such as x<-1:10 or mean(y), can be separated by either a semi-colon or a new line. Whenever the evaluator is presented with a syntactically complete statement that statement is evaluated and the value returned. The result of evaluating a statement can be referred to as the value of the statement1 The value can always be assigned to a symbol.

```
> x <- 0; x + 5
[1] 5
> y <- 1:10
> 1; 2
[1] 1
[1] 2
```

If - The if/else statement conditionally evaluates two statements. There is a condition which is evaluated and if the value is TRUE then the first statement is evaluated; otherwise the second statement will be evaluated. The if/else statement returns, as its value, the value of the statement that was selected. The formal syntax is

```
if ( statement1 )
        statement2
else
        Statement3
```

Looping - R has three statements that provide explicit looping. They are for, while and repeat. The two built-in constructs, next and break, provide additional control over the evaluation. R provides other functions for implicit looping such as tapply, apply, and lapply. In addition many operations, especially arithmetic ones, are vectorized so you may not need to use a loop.

while (statement1) statement2

for (name in vector)
    Statement1

switch (statement, list)

```
> x <- 3
> switch(x, 2+2, mean(1:10), rnorm(5))
[1] 2.2903605 2.3271663 -0.7060073 1.3622045 -0.2892720
> switch(2, 2+2, mean(1:10), rnorm(5)) [1] 5.5
> switch(6, 2+2, mean(1:10), rnorm(5)) NULL
```

Elementary Arithmetic Operations

Recycling Rules - If one tries to add two structures with a different number of elements, then the shortest is recycled to length of longest. That is, if for instance you add c(1, 2, 3) to a six-element vector then you will really add c(1, 2, 3, 1, 2, 3). If the length of the longer vector is not a multiple of the shorter one, a warning is given.

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).

x[i]
x[i, j]

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

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

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

x[3:5] <- 13:15

Scope of variables

Global Environment
Lexical Environment
Call Stack

Functions

Syntax and Examples - The syntax for writing a function is

function (arglist) body

The first component of the function declaration is the keyword function which indicates to R that you want to create a function. An argument list is a comma separated list of formal arguments. A formal argument can be a symbol, a statement of the form 'symbol = expression', or the special formal argument '...'. The body can be any valid R expression. Generally, the body is a group of expressions contained in curly braces ('{' and '}').

Object Oriented Programming

Inheritance - The class attribute of an object can have several elements. When a generic function is called the first inheritance is mainly handled through NextMethod. NextMethod determines the method currently being evaluated, finds the next class.

Method Dispatching - Generic functions should consist of a single statement. They should usually be of the form foo <- function(x, ...) UseMethod("foo", x). When UseMethod is called, it determines the appropriate method and then that method is invoked with the same arguments, in the same order as the call to the generic, as if the call had been made directly to the method.

Use Method - UseMethod is a special function and it behaves differently from other function calls. The syntax of a call to it is UseMethod(generic, object), where generic is the name of the generic function, object is the object used to determine which method should be chosen. UseMethod can only be called from the body of a function.

Next Method - NextMethod is used to provide a simple inheritance mechanism. Methods invoked as a result of a call to NextMethod behave as if they had been invoked from the previous method. The arguments to the inherited method are in the same order and have the same names as the call to the current method. This means that they are the same as for the call to the generic. However, the expressions for the arguments are the names of the corresponding formal arguments of the current method. Thus the arguments will have values that correspond to their value at the time NextMethod was invoked.

Computing on the Language

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, but in contrast to FORTRAN and the ALGOL family. 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 you to alter and subsequently execute them, or create entirely new functions from scratch.

```
> e1 <- quote(2 + 2)
> e2 <- quote(plot(x, y))

> quote("+"(2, 2))
2 + 2

> e2[[1]] plot
> e2[[2]] x
> e2[[3]] y

> str(quote(c(1,2)))
language c(1, 2)
> str(c(1,2))
num [1:2] 1 2
> deparse(quote(c(1,2)))
[1] "c(1, 2)"
> deparse(c(1,2))
[1] "c(1, 2)"
> quote("-"(2, 2))
2 - 2
> quote(2 - 2)
2 - 2
```

Substitutions - It is in fact not often that one wants to modify the innards of an expression like in the previous section. More frequently, one wants to simply get at an expression in order to deparse it and use it for labeling plots, for instance.

```
xlabel <- if (!missing(x)) deparse(substitute(x))
```

Evaluation of expression objects - evaluating an expression object evaluates each call in turn, but the final value is that of the last call. In this respect it behaves almost identically to the compound language object quote({2 + 2; 3 + 4}). However, there is a subtle difference: Call objects are indistinguishable from subexpressions in a parse tree. This means that they are automatically evaluated in the same way a subexpression would be. Expression objects can be recognized during evaluation and in a sense retain their quotedness.

```
> eval(substitute(mode(x), list(x = quote(2 + 2))))
[1] "numeric"
> eval(substitute(mode(x), list(x = expression(2 + 2))))
[1] "expression"
```

Parser

Parsing in R occurs in three different variants:

- The read-eval-print loop
- Parsing of text files
- Parsing of character strings

Comments - Comments in R are ignored by the parser. Any text from a # character to the end of the line is taken to be a comment, unless the # character is inside a quoted string. For example,

```
> x <- 1 # This is a comment...
> y <- " #... but this is not."
```

Tokens - Tokens are the elementary building blocks of a programming language. They are recognised during lexical analysis which (conceptually, at least) takes place prior to the syntactic analysis performed by the parser itself.

A function definition is of the form

function (arglist) body

The function body is an expression, often a compound expression. The arglist is a comma separated list of items each of which can be an identifier, or of the form ‘identifier = default’, or the special token ‘...’. The default can be any valid expression.

R contains the following control structures as special syntactic constructs
if (cond) expr
if (cond) expr1 else expr2
while (cond)
expr
repeat expr
for (var in list)
Expr

A function call takes the form of a function reference followed by a comma-separated list of arguments within a set of parentheses.

function_reference (arg1, arg2, ... , argn)

The function reference can be either:

• an identifier (the name of the function)
• a text string (ditto, but handy if the function has a name which is not a valid identifier)
• an expression (which should evaluate to a function object)