R Language Definition

Version 4.2.0 beta (2022-04-08) **DRAFT**

R Core Team

This manual is for R, version 4.2.0 beta (2022-04-08).

Copyright © 2000–2021 R Core Team

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the R Core Team.

Table of Contents

1 Introduction

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 R language is a dialect of S which was designed in the 1980s and has been in widespread use in the statistical community since. Its principal designer, John M. Chambers, was awarded the 1998 ACM Software Systems Award for S.

The language syntax has a superficial similarity with C, but the semantics are of the FPL (functional programming language) variety with stronger affinities with Lisp and APL. In particular, it allows "computing on the language", which in turn makes it possible to write functions that take expressions as input, something that is often useful for statistical modeling and graphics.

It is possible to get quite far using R interactively, executing simple expressions from the command line. Some users may never need to go beyond that level, others will want to write their own functions either in an ad hoc fashion to systematize repetitive work or with the perspective of writing add-on packages for new functionality.

The purpose of this manual is to document the language *per se*. That is, the objects that it works on, and the details of the expression evaluation process, which are useful to know when programming R functions. Major subsystems for specific tasks, such as graphics, are only briefly described in this manual and will be documented separately.

Although much of the text will equally apply to S, there are also some substantial differences, and in order not to confuse the issue we shall concentrate on describing R.

The design of the language contains a number of fine points and common pitfalls which may surprise the user. Most of these are due to consistency considerations at a deeper level, as we shall explain. There are also a number of useful shortcuts and idioms, which allow the user to express quite complicated operations succinctly. Many of these become natural once one is familiar with the underlying concepts. In some cases, there are multiple ways of performing a task, but some of the techniques will rely on the language implementation, and others work at a higher level of abstraction. In such cases we shall indicate the preferred usage.

Some familiarity with R is assumed. This is not an introduction to R but rather a programmers' reference manual. Other manuals provide complementary information: in particular Section "Preface" in *An Introduction to R* provides an introduction to R and Section "System and foreign language interfaces" in *Writing R Extensions* details how to extend R using compiled code.

2 Objects

In every computer language variables provide a means of accessing the data stored in memory. 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.

In this chapter we provide preliminary descriptions of the various data structures provided in R. More detailed discussions of many of them will be found in the subsequent chapters. The R specific function `typeof` returns the *type* of an R object. Note that in the C code underlying R, all objects are pointers to a structure with typedef `SEXPREC`; the different R data types are represented in C by `SEXPTYPE`, which determines how the information in the various parts of the structure is used.

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

Users cannot easily get hold of objects of types marked with a '***'.

Function `mode` gives information about the *mode* of an object in the sense of Becker, Chambers & Wilks (1988), and is more compatible with other implementations of the S language. Finally, the function `storage.mode` returns the *storage mode* of its argument in the sense of Becker et al. (1988). It is generally used when calling functions written in another language, such as C or FORTRAN, to ensure that R objects have the data type

expected by the routine being called. (In the S language, vectors with integer or real values are both of mode `"numeric"`, so their storage modes need to be distinguished.)

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

R objects are often coerced to different types during computations. There are also many functions available to perform explicit coercion. When programming in the R language the type of an object generally doesn't affect the computations, however, when dealing with foreign languages or the operating system it is often necessary to ensure that an object is of the correct type.

2.1 Basic types

2.1.1 Vectors

Vectors can be thought of as contiguous cells containing data. Cells are accessed through indexing operations such as `x[5]`. More details are given in Section 3.4 [Indexing], page 17.

R has six basic ('atomic') vector types: logical, integer, real, complex, string (or character) and raw. The modes and storage modes for the different vector types are listed in the following table.

typeof	**mode**	**storage.mode**
`logical`	`logical`	`logical`
`integer`	`numeric`	`integer`
`double`	`numeric`	`double`
`complex`	`complex`	`complex`
`character`	`character`	`character`
`raw`	`raw`	`raw`

Single numbers, such as `4.2`, and strings, such as `"four point two"` are still vectors, of length 1; there are no more basic types. Vectors with length zero are possible (and useful).

String vectors have mode and storage mode `"character"`. A single element of a character vector is often referred to as a *character string*.

2.1.2 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. These are explained in detail in Section 3.4 [Indexing], page 17.

Lists are vectors, and the basic vector types are referred to as *atomic vectors* where it is necessary to exclude lists.

2.1.3 Language objects

There are three types of objects that constitute the R language. They are *calls*, *expressions*, and *names*. Since R has objects of type `"expression"` we will try to avoid the use of the word expression in other contexts. In particular syntactically correct expressions will be referred to as *statements*.

These objects have modes `"call"`, `"expression"`, and `"name"`, respectively.

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. Components of the parse tree can be extracted using the standard indexing operations.

2.1.3.1 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`.

Symbols have mode `"name"`, storage mode `"symbol"`, and type `"symbol"`. They can be coerced to and from character strings using `as.character` and `as.name`. They naturally appear as atoms of parsed expressions, try e.g. `as.list(quote(x + y))`.

2.1.4 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. The main difference is that an expression object can contain several such expressions. Another more subtle difference is that objects of type `"expression"` are only evaluated when explicitly passed to `eval`, whereas other language objects may get evaluated in some unexpected cases.

An expression object behaves much like a list and its components should be accessed in the same way as the components of a list.

2.1.5 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. The argument list is a comma-separated list of arguments. An argument can be a symbol, or a '*`symbol = default`*' construct, or the special argument `....` The second form of argument is used to specify a default value for an argument. This value will be used if the function is called without any value specified for that argument. The `...` argument is special and can contain any number of arguments. It is generally used if the number of arguments is unknown or in cases where the arguments will be passed on to another function.

The body is a parsed R statement. It is usually a collection of statements in braces but it can be a single statement, a symbol or even a constant.

A function's environment is the environment that was active at the time that the function was created. Any symbols bound in that environment are *captured* and available to the function. This combination of the code of the function and the bindings in its environment is called a 'function closure', a term from functional programming theory. In this document

we generally use the term 'function', but use 'closure' to emphasize the importance of the attached environment.

It is possible to extract and manipulate the three parts of a closure object using `formals`, `body`, and `environment` constructs (all three can also be used on the left hand side of assignments). The last of these can be used to remove unwanted environment capture.

When a function is called, a new environment (called the *evaluation environment*) is created, whose enclosure (see Section 2.1.10 [Environment objects], page 6) is the environment from the function closure. This new environment is initially populated with the unevaluated arguments to the function; as evaluation proceeds, local variables are created within it.

There is also a facility for converting functions to and from list structures using `as.list` and `as.function`. These have been included to provide compatibility with S and their use is discouraged.

2.1.6 NULL

There is a special object called `NULL`. It is used whenever there is a need to indicate or specify that an object is absent. It should not be confused with a vector or list of zero length.

The `NULL` object has no type and no modifiable properties. There is only one `NULL` object in R, to which all instances refer. To test for `NULL` use `is.null`. You cannot set attributes on `NULL`.

2.1.7 Builtin objects and special forms

These two kinds of object contain the builtin functions of R, i.e., those that are displayed as `.Primitive` in code listings (as well as those accessed via the `.Internal` function and hence not user-visible as objects). The difference between the two lies in the argument handling. Builtin functions have all their arguments evaluated and passed to the internal function, in accordance with *call-by-value*, whereas special functions pass the unevaluated arguments to the internal function.

From the R language, these objects are just another kind of function. The `is.primitive` function can distinguish them from interpreted functions.

2.1.8 Promise objects

Promise objects are part of R's lazy evaluation mechanism. They contain three slots: a value, an expression, and an environment. When a function is called the arguments are matched and then each of the formal arguments is bound to a promise. The expression that was given for that formal argument and a pointer to the environment the function was called from are stored in the promise.

Until that argument is accessed there is no *value* associated with the promise. When the argument is accessed, the stored expression is evaluated in the stored environment, and the result is returned. The result is also saved by the promise. The `substitute` function will extract the content of the expression slot. This allows the programmer to access either the value or the expression associated with the promise.

Within the R language, promise objects are almost only seen implicitly: actual function arguments are of this type. There is also a `delayedAssign` function that will make a promise

out of an expression. There is generally no way in R code to check whether an object is a promise or not, nor is there a way to use R code to determine the environment of a promise.

2.1.9 Dot-dot-dot

The `...` object type is stored as a type of pairlist. The components of `...` can be accessed in the usual pairlist manner from C code, but `...` is not easily accessed as an object in interpreted code, and even the existence of such an object should typically not be assumed, as that may change in the future.

The object can be captured (with promises being forced!) as a list, so for example in `table` one sees

```
    args <- list(...)
## ....
    for (a in args) {
## ....
```

Note that the implementation of `...` as a pairlist object is *not* to be considered part of the R API, and code outside base R should not rely on this current description of `....` On the other hand, the above `list(...)` access, and the other "dot-access" functions `...length()`, `...elt()`, `...names()`, and "reserved words" `..1`, `..2`, etc, see also help page `?dots`, are part of the stable R API.

If a function has `...` as a formal argument then any actual arguments that do not match a formal argument are matched with `....`

2.1.10 Environments

Environments can be thought of as consisting of two things. A *frame*, consisting of a set of symbol-value pairs, and an *enclosure*, a pointer to an enclosing environment. When R looks up the value for a symbol the frame is examined and if a matching symbol is found its value will be returned. If not, the enclosing environment is then accessed and the process repeated. Environments form a tree structure in which the enclosures play the role of parents. The tree of environments is rooted in an empty environment, available through `emptyenv()`, which has no parent. It is the direct parent of the environment of the base package (available through the `baseenv()` function).

Environments are created implicitly by function calls, as described in Section 2.1.5 [Function objects], page 4, and Section 3.5.2 [Lexical environment], page 22. In this case the environment contains the variables local to the function (including the arguments), and its enclosure is the environment of the currently called function. Environments may also be created directly by `new.env`. The frame content of an environment can be accessed by use of `ls`, `names`, `$`, `[`, `[[`, `get`, and `get0`, and manipulated by `$<-`, `[[<-`, and `assign` as well as `eval` and `evalq`.

The `parent.env` function may be used to access the enclosure of an environment.

Unlike most other R objects, environments are not copied when passed to functions or used in assignments. Thus, if you assign the same environment to several symbols and change one, the others will change too. In particular, assigning attributes to an environment can lead to surprises.

2.1.11 Pairlist objects

Pairlist objects are similar to Lisp's dotted-pair lists. They are used extensively in the internals of R, but are rarely visible in interpreted code, although they are returned by `formals`, and can be created by (e.g.) the `pairlist` function. A zero-length pairlist is `NULL`, as would be expected in Lisp but in contrast to a zero-length list. Each such object has three slots, a CAR value, a CDR value and a TAG value. The TAG value is a text string and CAR and CDR usually represent, respectively, a list item (head) and the remainder (tail) of the list with a NULL object as terminator (the CAR/CDR terminology is traditional Lisp and originally referred to the address and decrement registers on an early 60's IBM computer).

Pairlists are handled in the R language in exactly the same way as generic vectors ("lists"). In particular, elements are accessed using the same `[[]]` syntax. The use of pairlists is deprecated since generic vectors are usually more efficient to use. When an internal pairlist is accessed from R it is generally (including when subsetted) converted to a generic vector.

In a very few cases pairlists are user-visible: one is `.Options`.

2.1.12 The "Any" type

It is not really possible for an object to be of "Any" type, but it is nevertheless a valid type value. It gets used in certain (rather rare) circumstances, e.g. `as.vector(x, "any")`, indicating that type coercion should not be done.

2.2 Attributes

All objects except `NULL` can have one or more attributes attached to them. Attributes are stored as a pairlist where all elements are named, but should be thought of as a set of name=value pairs. A listing of the attributes can be obtained using `attributes` and set by `attributes<-`, individual components are accessed using `attr` and `attr<-`.

Some attributes have special accessor functions (e.g. `levels<-` for factors) and these should be used when available. In addition to hiding details of implementation they may perform additional operations. R attempts to intercept calls to `attr<-` and to `attributes<-` that involve the special attributes and enforces the consistency checks.

Matrices and arrays are simply vectors with the attribute `dim` and optionally `dimnames` attached to the vector.

Attributes are used to implement the class structure used in R. If an object has a `class` attribute then that attribute will be examined during evaluation. The class structure in R is described in detail in Chapter 5 [Object-oriented programming], page 30.

2.2.1 Names

A `names` attribute, when present, labels the individual elements of a vector or list. When an object is printed the `names` attribute, when present, is used to label the elements. The `names` attribute can also be used for indexing purposes, for example, `quantile(x)["25%"]`.

One may get and set the names using `names` and `names<-` constructions. The latter will perform the necessary consistency checks to ensure that the names attribute has the proper type and length.

Pairlists and one-dimensional arrays are treated specially. For pairlist objects, a virtual `names` attribute is used; the `names` attribute is really constructed from the tags of the list components. For one-dimensional arrays the `names` attribute really accesses `dimnames[[1]]`.

2.2.2 Dimensions

The `dim` attribute is used to implement arrays. The content of the array is stored in a vector in column-major order and the `dim` attribute is a vector of integers specifying the respective extents of the array. R ensures that the length of the vector is the product of the lengths of the dimensions. The length of one or more dimensions may be zero.

A vector is not the same as a one-dimensional array since the latter has a `dim` attribute of length one, whereas the former has no `dim` attribute.

2.2.3 Dimnames

Arrays may name each dimension separately using the `dimnames` attribute which is a list of character vectors. The `dimnames` list may itself have names which are then used for extent headings when printing arrays.

2.2.4 Classes

R has an elaborate class system[1], principally controlled via the `class` attribute. This attribute is a character vector containing the list of classes that an object inherits from. This forms the basis of the "generic methods" functionality in R.

This attribute can be accessed and manipulated virtually without restriction by users. There is no checking that an object actually contains the components that class methods expect. Thus, altering the `class` attribute should be done with caution, and when they are available specific creation and coercion functions should be preferred.

2.2.5 Time series attributes

The `tsp` attribute is used to hold parameters of time series, start, end, and frequency. This construction is mainly used to handle series with periodic substructure such as monthly or quarterly data.

2.2.6 Copying of attributes

Whether attributes should be copied when an object is altered is a complex area, but there are some general rules (Becker, Chambers & Wilks, 1988, pp. 144–6).

Scalar functions (those which operate element-by-element on a vector and whose output is similar to the input) should preserve attributes (except perhaps class).

Binary operations normally copy most attributes from the longer argument (and if they are of the same length from both, preferring the values on the first). Here 'most' means all except the `names`, `dim` and `dimnames` which are set appropriately by the code for the operator.

Subsetting (other than by an empty index) generally drops all attributes except `names`, `dim` and `dimnames` which are reset as appropriate. On the other hand, subassignment generally preserves attributes even if the length is changed. Coercion drops all attributes.

[1] actually two, but this draft manual predates the **methods** package.

The default method for sorting drops all attributes except names, which are sorted along with the object.

2.3 Special compound objects

2.3.1 Factors

Factors are used to describe items that can have a finite number of values (gender, social class, etc.). A factor has a `levels` attribute and class `"factor"`. Optionally, it may also contain a `contrasts` attribute which controls the parametrisation used when the factor is used in a modeling functions.

A factor may be purely nominal or may have ordered categories. In the latter case, it should be defined as such and have a `class` vector `c("ordered"," factor")`.

Factors are currently implemented using an integer array to specify the actual levels and a second array of names that are mapped to the integers. Rather unfortunately users often make use of the implementation in order to make some calculations easier. This, however, is an implementation issue and is not guaranteed to hold in all implementations of R.

2.3.2 Data frame objects

Data frames are the R structures which most closely mimic the SAS or SPSS data set, i.e. a "cases by variables" matrix of data.

A data frame is a list of vectors, factors, and/or matrices all having the same length (number of rows in the case of matrices). In addition, a data frame generally has a `names` attribute labeling the variables and a `row.names` attribute for labeling the cases.

A data frame can contain a list that is the same length as the other components. The list can contain elements of differing lengths thereby providing a data structure for ragged arrays. However, as of this writing such arrays are not generally handled correctly.

3 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.

This chapter describes the basic mechanisms of the evaluator, but avoids discussion of specific functions or groups of functions which are described in separate chapters later on or where the help pages should be sufficient documentation.

Users can construct expressions and invoke the evaluator on them.

3.1 Simple evaluation

3.1.1 Constants

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

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

Perhaps unexpectedly, the number returned from the expression `1` is a numeric. In most cases, the difference between an integer and a numeric value will be unimportant as R will do the right thing when using the numbers. There are, however, times when we would like to explicitly create an integer value for a constant. We can do this by calling the function `as.integer` or using various other techniques. But perhaps the simplest approach is to qualify our constant with the suffix character 'L'. For example, to create the integer value 1, we might use

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

We can use the 'L' suffix to qualify any number with the intent of making it an explicit integer. So '0x10L' creates the integer value 16 from the hexadecimal representation. The constant `1e3L` gives 1000 as an integer rather than a numeric value and is equivalent to `1000L`. (Note that the 'L' is treated as qualifying the term `1e3` and not the `3`.) If we qualify a value with 'L' that is not an integer value, e.g. `1e-3L`, we get a warning and the numeric value is created. A warning is also created if there is an unnecessary decimal point in the number, e.g. `1.L`.

We get a syntax error when using 'L' with complex numbers, e.g. `12iL` gives an error.

Constants are fairly boring and to do more we need symbols.

3.1.2 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.

In this small example `y` is a symbol and its value is 4. A symbol is an R object too, but one rarely needs to deal with symbols directly, except when doing "programming on the language" (Chapter 6 [Computing on the language], page 36).

```
> y <- 4
```

```
> y
[1] 4
```

3.1.3 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
```

In this example the function `mean` was called with one argument, the vector of integers from 1 to 10.

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.

Function calls can have *tagged* (or *named*) arguments, as in `plot(x, y, pch = 3)`. Arguments without tags are known as *positional* since the function must distinguish their meaning from their sequential positions among the arguments of the call, e.g., that `x` denotes the abscissa variable and `y` the ordinate. The use of tags/names is an obvious convenience for functions with a large number of optional arguments.

A special type of function calls can appear on the left hand side of the assignment operator as in

```
> class(x) <- "foo"
```

What this construction really does is to call the function `class<-` with the original object and the right hand side. This function performs the modification of the object and returns the result which is then stored back into the original variable. (At least conceptually, this is what happens. Some additional effort is made to avoid unnecessary data duplication.)

3.1.4 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))
```

R contains a number of operators. They are listed in the table below.

`-`	Minus, can be unary or binary
`+`	Plus, can be unary or binary
`!`	Unary not
`~`	Tilde, used for model formulae, can be either unary or binary
`?`	Help
`:`	Sequence, binary (in model formulae: interaction)
`*`	Multiplication, binary

`/`	Division, binary
`^`	Exponentiation, binary
`%x%`	Special binary operators, *x* can be replaced by any valid name
`%%`	Modulus, binary
`%/%`	Integer divide, binary
`%*%`	Matrix product, binary
`%o%`	Outer product, binary
`%x%`	Kronecker product, binary
`%in%`	Matching operator, binary (in model formulae: nesting)
`<`	Less than, binary
`>`	Greater than, binary
`==`	Equal to, binary
`>=`	Greater than or equal to, binary
`<=`	Less than or equal to, binary
`&`	And, binary, vectorized
`&&`	And, binary, not vectorized
`|`	Or, binary, vectorized
`||`	Or, binary, not vectorized
`<-`	Left assignment, binary
`->`	Right assignment, binary
`$`	List subset, binary

Except for the syntax, there is no difference between applying an operator and calling a function. In fact, `x + y` can equivalently be written `‘+‘(x, y)`. Notice that since ‘+’ is a non-standard function name, it needs to be quoted.

R deals with entire vectors of data at a time, and most of the elementary operators and basic mathematical functions like `log` are vectorized (as indicated in the table above). This means that e.g. adding two vectors of the same length will create a vector containing the element-wise sums, implicitly looping over the vector index. This applies also to other operators like `-`, `*`, and `/` as well as to higher dimensional structures. Notice in particular that multiplying two matrices does not produce the usual matrix product (the `%*%` operator exists for that purpose). Some finer points relating to vectorized operations will be discussed in Section 3.3 [Elementary arithmetic operations], page 16.

To access individual elements of an atomic vector, one generally uses the `x[i]` construction.

```
> x <- rnorm(5)
> x
[1] -0.12526937 -0.27961154 -1.03718717 -0.08156527  1.37167090
> x[2]
[1] -0.2796115
```

List components are more commonly accessed using `x$a` or `x[[i]]`.

```
> x <- options()
> x$prompt
[1] "> "
```

Indexing constructs can also appear on the right hand side of an assignment.

Like the other operators, indexing is really done by functions, and one could have used `` `[`(x, 2) `` instead of `x[2]`.

R's indexing operations contain many advanced features which are further described in Section 3.4 [Indexing], page 17.

3.2 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 statement[1] The value can always be assigned to a symbol.

Both semicolons and new lines can be used to separate statements. A semicolon always indicates the end of a statement while a new line *may* indicate the end of a statement. If the current statement is not syntactically complete new lines are simply ignored by the evaluator. If the session is interactive the prompt changes from '`>`' to '`+`'.

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

Statements can be grouped together using braces '`{`' and '`}`'. A group of statements is sometimes called a *block*. Single statements are evaluated when a new line is typed at the end of the syntactically complete statement. Blocks are not evaluated until a new line is entered after the closing brace. In the remainder of this section, *statement* refers to either a single statement or a block.

```
> { x <- 0
+ x + 5
+ }
[1] 5
```

3.2.1 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
```

First, *statement1* is evaluated to yield *value1*. If *value1* is a logical vector with first element `TRUE` then *statement2* is evaluated. If the first element of *value1* is `FALSE` then

[1] Evaluation always takes place in an environment. See Section 3.5 [Scope of variables], page 22, for more details.

statement3 is evaluated. If *value1* is a numeric vector then *statement3* is evaluated when the first element of *value1* is zero and otherwise *statement2* is evaluated. Only the first element of *value1* is used. All other elements are ignored. If *value1* has any type other than a logical or a numeric vector an error is signalled.

`if`/`else` statements can be used to avoid numeric problems such as taking the logarithm of a negative number. Because `if`/`else` statements are the same as other statements you can assign the value of them. The two examples below are equivalent.

```
> if( any(x <= 0) ) y <- log(1+x) else y <- log(x)
> y <- if( any(x <= 0) ) log(1+x) else log(x)
```

The `else` clause is optional. The statement `if(any(x <= 0)) x <- x[x <= 0]` is valid. When the `if` statement is not in a block the `else`, if present, must appear on the same line as the end of *statement2*. Otherwise the new line at the end of *statement2* completes the `if` and yields a syntactically complete statement that is evaluated. A simple solution is to use a compound statement wrapped in braces, putting the `else` on the same line as the closing brace that marks the end of the statement.

`if`/`else` statements can be nested.

```
if ( statement1 ) {
    statement2
} else if ( statement3 ) {
    statement4
} else if ( statement5 ) {
    statement6
} else
    statement8
```

One of the even numbered statements will be evaluated and the resulting value returned. If the optional `else` clause is omitted and all the odd numbered *statement*s evaluate to `FALSE` no statement will be evaluated and `NULL` is returned.

The odd numbered *statement*s are evaluated, in order, until one evaluates to `TRUE` and then the associated even numbered *statement* is evaluated. In this example, *statement6* will only be evaluated if *statement1* is `FALSE` and *statement3* is `FALSE` and *statement5* is `TRUE`. There is no limit to the number of `else if` clauses that are permitted.

3.2.2 Looping

R has three statements that provide explicit looping.[2] 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.

There are two statements that can be used to explicitly control looping. They are `break` and `next`. The `break` statement causes an exit from the innermost loop that is currently being executed. The `next` statement immediately causes control to return to the start of the loop. The next iteration of the loop (if there is one) is then executed. No statement below `next` in the current loop is evaluated.

The value returned by a loop statement is always `NULL` and is returned invisibly.

[2] Looping is the repeated evaluation of a statement or block of statements.

3.2.3 repeat

The `repeat` statement causes repeated evaluation of the body until a break is specifically requested. This means that you need to be careful when using `repeat` because of the danger of an infinite loop. The syntax of the `repeat` loop is

```
repeat statement
```

When using `repeat`, *statement* must be a block statement. You need to both perform some computation and test whether or not to break from the loop and usually this requires two statements.

3.2.4 while

The `while` statement is very similar to the `repeat` statement. The syntax of the `while` loop is

```
while ( statement1 ) statement2
```

where *statement1* is evaluated and if its value is `TRUE` then *statement2* is evaluated. This process continues until *statement1* evaluates to `FALSE`.

3.2.5 for

The syntax of the `for` loop is

```
for ( name in vector )
   statement1
```

where *vector* can be either a vector or a list. For each element in *vector* the variable *name* is set to the value of that element and *statement1* is evaluated. A side effect is that the variable *name* still exists after the loop has concluded and it has the value of the last element of *vector* that the loop was evaluated for.

3.2.6 switch

Technically speaking, `switch` is just another function, but its semantics are close to those of control structures of other programming languages.

The syntax is

```
switch (statement, list)
```

where the elements of *list* may be named. First, *statement* is evaluated and the result, *value*, obtained. If *value* is a number between 1 and the length of *list* then the corresponding element of *list* is evaluated and the result returned. If *value* is too large or too small `NULL` is returned.

```
> 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
```

If *value* is a character vector then the element of `...` with a name that exactly matches *value* is evaluated. If there is no match a single unnamed argument will be used as a default. If no default is specified, `NULL` is returned.

```
> y <- "fruit"
> switch(y, fruit = "banana", vegetable = "broccoli", "Neither")
[1] "banana"
> y <- "meat"
> switch(y, fruit = "banana", vegetable = "broccoli", "Neither")
[1] "Neither"
```

A common use of `switch` is to branch according to the character value of one of the arguments to a function.

```
> centre <- function(x, type) {
+ switch(type,
+        mean = mean(x),
+        median = median(x),
+        trimmed = mean(x, trim = .1))
+ }
> x <- rcauchy(10)
> centre(x, "mean")
[1] 0.8760325
> centre(x, "median")
[1] 0.5360891
> centre(x, "trimmed")
[1] 0.6086504
```

`switch` returns either the value of the statement that was evaluated or `NULL` if no statement was evaluated.

To choose from a list of alternatives that already exists `switch` may not be the best way to select one for evaluation. It is often better to use `eval` and the subset operator, `[[`, directly via `eval(x[[condition]])`.

3.3 Elementary arithmetic operations

In this section, we discuss the finer points of the rules that apply to basic operation like addition or multiplication of two vectors or matrices.

3.3.1 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.

As from R 1.4.0, any arithmetic operation involving a zero-length vector has a zero-length result.

3.3.2 Propagation of names

propagation of names (first one wins, I think - also if it has no names?? —- first one *with names* wins, recycling causes shortest to lose names)

3.3.3 Dimensional attributes

(matrix+matrix, dimensions must match. vector+matrix: first recycle, then check if dims fit, error if not)

3.3.4 NA handling

Missing values in the statistical sense, that is, variables whose value is not known, have the value `NA`. This should not be confused with the `missing` property for a function argument that has not been supplied (see Section 4.1.2 [Arguments], page 25).

As the elements of an atomic vector must be of the same type there are multiple types of `NA` values. There is one case where this is particularly important to the user. The default type of `NA` is `logical`, unless coerced to some other type, so the appearance of a missing value may trigger logical rather than numeric indexing (see Section 3.4 [Indexing], page 17, for details).

Numeric and logical calculations with `NA` generally return `NA`. In cases where the result of the operation would be the same for all possible values the `NA` could take, the operation may return this value. In particular, '`FALSE & NA`' is `FALSE`, '`TRUE | NA`' is `TRUE`. `NA` is not equal to any other value or to itself; testing for `NA` is done using `is.na`. However, an `NA` value will match another `NA` value in `match`.

Numeric calculations whose result is undefined, such as '`0/0`', produce the value `NaN`. This exists only in the `double` type and for real or imaginary components of the complex type. The function `is.nan` is provided to check specifically for `NaN`, `is.na` also returns `TRUE` for `NaN`. Coercing `NaN` to logical or integer type gives an `NA` of the appropriate type, but coercion to character gives the string `"NaN"`. `NaN` values are incomparable so tests of equality or collation involving `NaN` will result in `NA`. They are regarded as matching any `NaN` value (and no other value, not even `NA`) by `match`.

The `NA` of character type is as from R 1.5.0 distinct from the string `"NA"`. Programmers who need to specify an explicit string `NA` should use '`as.character(NA)`' rather than `"NA"`, or set elements to `NA` using `is.na<-`.

There are constants `NA_integer_`, `NA_real_`, `NA_complex_` and `NA_character_` which will generate (in the parser) an `NA` value of the appropriate type, and will be used in deparsing when it is not otherwise possible to identify the type of an `NA` (and the `control` options ask for this to be done).

There is no `NA` value for raw vectors.

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

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`. Negative out of bounds values for `i` are silently disregarded since R version 2.6.0, S compatibly, as they mean to drop non-existing elements and that is an empty operation ("no-op").

  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 16. 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 contain a component named `"aa"` and `"aabb"` is the only name which has prefix `"aa"`. For `[[`, partial matching can be controlled via the `exact` argument which defaults to `NA` indicating that partial matching is allowed, but should result in a warning when it occurs. Setting `exact` to `TRUE` prevents partial

matching from occurring, a `FALSE` value allows it and does not issue any warnings. Note that `[` always 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
```

Indexing matrices may not contain negative indices. `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 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 command is as if the following had been executed

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

Note that the index is first converted to a numeric index and then the elements are replaced sequentially along the numeric index, as if a `for` loop had been used. Any existing variable called `‘*tmp*‘` will be overwritten and deleted, and this variable name should not be used in code.

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

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

is equivalent to

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

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"))
rm(‘*tmp*‘)
```

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*‘
rm(‘*tmp*‘)
```

and also to

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

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 four lines

```
‘*tmp*‘ <- get(e, envir=parent.env(), inherits=TRUE)
```

```
`*tmp*`[i] <- e[i]+1
e <<- `*tmp*`
rm(`*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*`
rm(`*tmp*`)
```

and not to

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

3.5 Scope of variables

Almost every programming language has a set of scoping rules, allowing the same name to be used for different objects. This allows, e.g., a local variable in a function to have the same name as a global object.

R uses a *lexical scoping* model, similar to languages like Pascal. However, R is a *functional programming language* and allows dynamic creation and manipulation of functions and language objects, and has additional features reflecting this fact.

3.5.1 Global environment

The global environment is the root of the user workspace. An assignment operation from the command line will cause the relevant object to belong to the global environment. Its enclosing environment is the next environment on the search path, and so on back to the empty environment that is the enclosure of the base environment.

3.5.2 Lexical environment

Every call to a function creates a *frame* which contains the local variables created in the function, and is evaluated in an environment, which in combination creates a new environment.

Notice the terminology: A frame is a set of variables, an environment is a nesting of frames (or equivalently: the innermost frame plus the enclosing environment).

Environments may be assigned to variables or be contained in other objects. However, notice that they are not standard objects—in particular, they are not copied on assignment.

A closure (mode `"function"`) object will contain the environment in which it is created as part of its definition (By default. The environment can be manipulated using

`environment<-`). When the function is subsequently called, its evaluation environment is created with the closure's environment as enclosure. Notice that this is not necessarily the environment of the caller!

Thus, when a variable is requested inside a function, it is first sought in the evaluation environment, then in the enclosure, the enclosure of the enclosure, etc.; once the global environment or the environment of a package is reached, the search continues up the search path to the environment of the base package. If the variable is not found there, the search will proceed next to the empty environment, and will fail.

3.5.3 The call stack

Every time a function is invoked a new evaluation frame is created. At any point in time during the computation the currently active environments are accessible through the *call stack*. Each time a function is invoked a special construct called a context is created internally and is placed on a list of contexts. When a function has finished evaluating its context is removed from the call stack.

Making variables defined higher up the call stack available is called dynamic scope. The binding for a variable is then determined by the most recent (in time) definition of the variable. This contradicts the default scoping rules in R, which use the bindings in the environment in which the function was defined (lexical scope). Some functions, particularly those that use and manipulate model formulas, need to simulate dynamic scope by directly accessing the call stack.

Access to the call stack is provided through a family of functions which have names that start with '`sys.`'. They are listed briefly below.

`sys.call` Get the call for the specified context.

`sys.frame`
: Get the evaluation frame for the specified context.

`sys.nframe`
: Get the environment frame for all active contexts.

`sys.function`
: Get the function being invoked in the specified context.

`sys.parent`
: Get the parent of the current function invocation.

`sys.calls`
: Get the calls for all the active contexts.

`sys.frames`
: Get the evaluation frames for all the active contexts.

`sys.parents`
: Get the numeric labels for all active contexts.

`sys.on.exit`
: Set a function to be executed when the specified context is exited.

`sys.status`
: Calls `sys.frames`, `sys.parents` and `sys.calls`.

`parent.frame`
: Get the evaluation frame for the specified parent context.

3.5.4 Search path

In addition to the evaluation environment structure, R has a search path of environments which are searched for variables not found elsewhere. This is used for two things: packages of functions and attached user data.

The first element of the search path is the global environment and the last is the base package. An `Autoloads` environment is used for holding proxy objects that may be loaded on demand. Other environments are inserted in the path using `attach` or `library`.

Packages which have a *namespace* have a different search path. When a search for an R object is started from an object in such a package, the package itself is searched first, then its imports, then the base namespace and finally the global environment and the rest of the regular search path. The effect is that references to other objects in the same package will be resolved to the package, and objects cannot be masked by objects of the same name in the global environment or in other packages.

4 Functions

4.1 Writing functions

While R can be very useful as a data analysis tool most users very quickly find themselves wanting to write their own functions. This is one of the real advantages of R. Users can program it and they can, if they want to, change the system level functions to functions that they find more appropriate.

R also provides facilities that make it easy to document any functions that you have created. See Section "Writing R documentation" in *Writing R Extensions*.

4.1.1 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 '`}`').

Generally functions are assigned to symbols but they don't need to be. The value returned by the call to `function` is a function. If this is not given a name it is referred to as an anonymous function. Anonymous functions are most frequently used as arguments to other functions such as the `apply` family or `outer`.

Here is a simple function: `echo <- function(x) print(x)`. So `echo` is a function that takes a single argument and when `echo` is invoked it prints its argument.

4.1.2 Arguments

The formal arguments to the function define the variables whose values will be supplied at the time the function is invoked. The names of these arguments can be used within the function body where they obtain the value supplied at the time of function invocation.

Default values for arguments can be specified using the special form '*name* = *expression*'. In this case, if the user does not specify a value for the argument when the function is invoked the expression will be associated with the corresponding symbol. When a value is needed the *expression* is evaluated in the evaluation frame of the function.

Default behaviours can also be specified by using the function `missing`. When `missing` is called with the name of a formal argument it returns `TRUE` if the formal argument was not matched with any actual argument and has not been subsequently modified in the body of the function. An argument that is `missing` will thus have its default value, if any. The `missing` function does not force evaluation of the argument.

The special type of argument `...` can contain any number of supplied arguments. It is used for a variety of purposes. It allows you to write a function that takes an arbitrary number of arguments. It can be used to absorb some arguments into an intermediate function which can then be extracted by functions called subsequently.

4.2 Functions as objects

Functions are first class objects in R. They can be used anywhere that an R object is required. In particular they can be passed as arguments to functions and returned as values from functions. See Section 2.1.5 [Function objects], page 4, for the details.

4.3 Evaluation

4.3.1 Evaluation environment

When a function is called or invoked a new evaluation frame is created. In this frame the formal arguments are matched with the supplied arguments according to the rules given in Section 4.3.2 [Argument matching], page 26. The statements in the body of the function are evaluated sequentially in this environment frame.

The enclosing frame of the evaluation frame is the environment frame associated with the function being invoked. This may be different from S. While many functions have `.GlobalEnv` as their environment this does not have to be true and functions defined in packages with namespaces (normally) have the package namespace as their environment.

4.3.2 Argument matching

This subsection applies to closures but not to primitive functions. The latter typically ignore tags and do positional matching, but their help pages should be consulted for exceptions, which include `log`, `round`, `signif`, `rep` and `seq.int`.

The first thing that occurs in a function evaluation is the matching of formal to the actual or supplied arguments. This is done by a three-pass process:

1. **Exact matching on tags**. For each named supplied argument the list of formal arguments is searched for an item whose name matches exactly. It is an error to have the same formal argument match several actuals or vice versa.
2. **Partial matching on tags**. Each remaining named supplied argument is compared to the remaining formal arguments using partial matching. If the name of the supplied argument matches exactly with the first part of a formal argument then the two arguments are considered to be matched. It is an error to have multiple partial matches. Notice that if `f <- function(fumble, fooey) fbody`, then `f(f = 1, fo = 2)` is illegal, even though the 2nd actual argument only matches `fooey`. `f(f = 1, fooey = 2)` *is* legal though since the second argument matches exactly and is removed from consideration for partial matching. If the formal arguments contain `...` then partial matching is only applied to arguments that precede it.
3. **Positional matching**. Any unmatched formal arguments are bound to *unnamed* supplied arguments, in order. If there is a `...` argument, it will take up the remaining arguments, tagged or not.

If any arguments remain unmatched an error is declared.

Argument matching is augmented by the functions `match.arg`, `match.call` and `match.fun`. Access to the partial matching algorithm used by R is via `pmatch`.

4.3.3 Argument evaluation

One of the most important things to know about the evaluation of arguments to a function is that supplied arguments and default arguments are treated differently. The supplied arguments to a function are evaluated in the evaluation frame of the calling function. The default arguments to a function are evaluated in the evaluation frame of the function.

The semantics of invoking a function in R argument are *call-by-value*. In general, supplied arguments behave as if they are local variables initialized with the value supplied and the name of the corresponding formal argument. Changing the value of a supplied argument within a function will not affect the value of the variable in the calling frame.

R has a form of lazy evaluation of function arguments. Arguments are not evaluated until needed. It is important to realize that in some cases the argument will never be evaluated. Thus, it is bad style to use arguments to functions to cause side-effects. While in **C** it is common to use the form, `foo(x = y)` to invoke `foo` with the value of `y` and simultaneously to assign the value of `y` to `x` this same style should not be used in R. There is no guarantee that the argument will ever be evaluated and hence the assignment may not take place.

It is also worth noting that the effect of `foo(x <- y)` if the argument is evaluated is to change the value of `x` in the calling environment and not in the evaluation environment of `foo`.

It is possible to access the actual (not default) expressions used as arguments inside the function. The mechanism is implemented via promises. When a function is being evaluated the actual expression used as an argument is stored in the promise together with a pointer to the environment the function was called from. When (if) the argument is evaluated the stored expression is evaluated in the environment that the function was called from. Since only a pointer to the environment is used any changes made to that environment will be in effect during this evaluation. The resulting value is then also stored in a separate spot in the promise. Subsequent evaluations retrieve this stored value (a second evaluation is not carried out). Access to the unevaluated expression is also available using `substitute`.

When a function is called, each formal argument is assigned a promise in the local environment of the call with the expression slot containing the actual argument (if it exists) and the environment slot containing the environment of the caller. If no actual argument for a formal argument is given in the call and there is a default expression, it is similarly assigned to the expression slot of the formal argument, but with the environment set to the local environment.

The process of filling the value slot of a promise by evaluating the contents of the expression slot in the promise's environment is called *forcing* the promise. A promise will only be forced once, the value slot content being used directly later on.

A promise is forced when its value is needed. This usually happens inside internal functions, but a promise can also be forced by direct evaluation of the promise itself. This is occasionally useful when a default expression depends on the value of another formal argument or other variable in the local environment. This is seen in the following example where the lone `label` ensures that the label is based on the value of `x` before it is changed in the next line.

```
function(x, label = deparse(x)) {
    label
    x <- x + 1
```

```
    print(label)
}
```

The expression slot of a promise can itself involve other promises. This happens whenever an unevaluated argument is passed as an argument to another function. When forcing a promise, other promises in its expression will also be forced recursively as they are evaluated.

4.3.4 Scope

Scope or the scoping rules are simply the set of rules used by the evaluator to find a value for a symbol. Every computer language has a set of such rules. In R the rules are fairly simple but there do exist mechanisms for subverting the usual, or default rules.

R adheres to a set of rules that are called *lexical scope*. This means the variable bindings in effect at the time the expression was created are used to provide values for any unbound symbols in the expression.

Most of the interesting properties of scope are involved with evaluating functions and we concentrate on this issue. A symbol can be either bound or unbound. All of the formal arguments to a function provide bound symbols in the body of the function. Any other symbols in the body of the function are either local variables or unbound variables. A local variable is one that is defined within the function. Because R has no formal definition of variables, they are simply used as needed, it can be difficult to determine whether a variable is local or not. Local variables must first be defined, this is typically done by having them on the left-hand side of an assignment.

During the evaluation process if an unbound symbol is detected then R attempts to find a value for it. The scoping rules determine how this process proceeds. In R the environment of the function is searched first, then its enclosure and so on until the global environment is reached.

The global environment heads a search list of environments that are searched sequentially for a matching symbol. The value of the first match is then used.

When this set of rules is combined with the fact that functions can be returned as values from other functions then some rather nice, but at first glance peculiar, properties obtain.

A simple example:

```
f <- function() {
    y <- 10
    g <- function(x) x + y
    return(g)
}
h <- f()
h(3)
```

A rather interesting question is what happens when `h` is evaluated. When a function body is evaluated there is no problem determining values for local variables or for bound variables. Scoping rules determine how the language will find values for the unbound variables.

When `h(3)` is evaluated we see that its body is that of `g`. Within that body `x` is bound to the formal argument and `y` is unbound. In a language with lexical scope `x` will be associated with the value 3 and `y` with the value 10 local to `f` so `h(3)` should return the value 13. In R this is indeed what happens.

In S, because of the different scoping rules one will get an error indicating that `y` is not found, unless there is a variable `y` in your workspace in which case its value will be used.

5 Object-oriented programming

Object-oriented programming is a style of programming that has become popular in recent years. Much of the popularity comes from the fact that it makes it easier to write and maintain complicated systems. It does this through several different mechanisms.

Central to any object-oriented language are the concepts of class and of methods. A *class* is a definition of an object. Typically a class contains several *slots* that are used to hold class-specific information. An object in the language must be an instance of some class. Programming is based on objects or instances of classes.

Computations are carried out via *methods*. Methods are basically functions that are specialized to carry out specific calculations on objects, usually of a specific class. This is what makes the language object oriented. In R, *generic functions* are used to determine the appropriate method. The generic function is responsible for determining the class of its argument(s) and uses that information to select the appropriate method.

Another feature of most object-oriented languages is the concept of inheritance. In most programming problems there are usually many objects that are related to one another. The programming is considerably simplified if some components can be reused.

If a class inherits from another class then generally it gets all the slots in the parent class and can extend it by adding new slots. On method dispatching (via the generic functions) if a method for the class does not exist then a method for the parent is sought.

In this chapter we discuss how this general strategy has been implemented in R and discuss some of the limitations within the current design. One of the advantages that most object systems impart is greater consistency. This is achieved via the rules that are checked by the compiler or interpreter. Unfortunately because of the way that the object system is incorporated into R this advantage does not obtain. Users are cautioned to use the object system in a straightforward manner. While it is possible to perform some rather interesting feats these tend to lead to obfuscated code and may depend on implementation details that will not be carried forward.

The greatest use of object oriented programming in R is through `print` methods, `summary` methods and `plot` methods. These methods allow us to have one generic function call, `plot` say, that dispatches on the type of its argument and calls a plotting function that is specific to the data supplied.

In order to make the concepts clear we will consider the implementation of a small system designed to teach students about probability. In this system the objects are probability functions and the methods we will consider are methods for finding moments and for plotting. Probabilities can always be represented in terms of the cumulative distribution function but can often be represented in other ways. For example as a density, when it exists or as a moment generating function when it exists.

5.1 Definition

Rather than having a full-fledged object-oriented system R has a class system and a mechanism for dispatching based on the class of an object. The dispatch mechanism for interpreted code relies on four special objects that are stored in the evaluation frame. These special objects are `.Generic`, `.Class`, `.Method` and `.Group`. There is a separate dispatch mechanism used for internal functions and types that will be discussed elsewhere.

The class system is facilitated through the `class` attribute. This attribute is a character vector of class names. So to create an object of class `"foo"` one simply attaches a class attribute with the string '`"foo"`' in it. Thus, virtually anything can be turned in to an object of class `"foo"`.

The object system makes use of *generic functions* via two dispatching functions, `UseMethod` and `NextMethod`. The typical use of the object system is to begin by calling a generic function. This is typically a very simple function and consists of a single line of code. The system function `mean` is just such a function,

```
> mean
function (x, ...)
UseMethod("mean")
```

When `mean` is called it can have any number of arguments but its first argument is special and the class of that first argument is used to determine which method should be called. The variable `.Class` is set to the class attribute of `x`, `.Generic` is set to the string `"mean"` and a search is made for the correct method to invoke. The class attributes of any other arguments to `mean` are ignored.

Suppose that `x` had a class attribute that contained `"foo"` and `"bar"`, in that order. Then R would first search for a function called `mean.foo` and if it did not find one it would then search for a function `mean.bar` and if that search was also unsuccessful then a final search for `mean.default` would be made. If the last search is unsuccessful R reports an error. It is a good idea to always write a default method. Note that the functions `mean.foo` etc. are referred to, in this context, as methods.

`NextMethod` provides another mechanism for dispatching. A function may have a call to `NextMethod` anywhere in it. The determination of which method should then be invoked is based primarily on the current values of `.Class` and `.Generic`. This is somewhat problematic since the method is really an ordinary function and users may call it directly. If they do so then there will be no values for `.Generic` or `.Class`.

If a method is invoked directly and it contains a call to `NextMethod` then the first argument to `NextMethod` is used to determine the generic function. An error is signalled if this argument has not been supplied; it is therefore a good idea to always supply this argument.

In the case that a method is invoked directly the class attribute of the first argument to the method is used as the value of `.Class`.

Methods themselves employ `NextMethod` to provide a form of inheritance. Commonly a specific method performs a few operations to set up the data and then it calls the next appropriate method through a call to `NextMethod`.

Consider the following simple example. A point in two-dimensional Euclidean space can be specified by its Cartesian (x-y) or polar (r-theta) coordinates. Hence, to store information about the location of the point, we could define two classes, `"xypoint"` and `"rthetapoint"`. All the 'xypoint' data structures are lists with an x-component and a y-component. All 'rthetapoint' objects are lists with an r-component and a theta-component.

Now, suppose we want to get the x-position from either type of object. This can easily be achieved through generic functions. We define the generic function `xpos` as follows.

```
xpos <- function(x, ...)
```

```
    UseMethod("xpos")
```

Now we can define methods:

```
xpos.xypoint <- function(x) x$x
xpos.rthetapoint <- function(x) x$r * cos(x$theta)
```

The user simply calls the function `xpos` with either representation as the argument. The internal dispatching method finds the class of the object and calls the appropriate methods.

It is pretty easy to add other representations. One need not write a new generic function only the methods. This makes it easy to add to existing systems since the user is only responsible for dealing with the new representation and not with any of the existing representations.

The bulk of the uses of this methodology are to provide specialized printing for objects of different types; there are about 40 methods for `print`.

5.2 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 from th

FIXME: something is missing here

5.3 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.

In order to determine the correct method the class attribute of the first argument to the generic is obtained and used to find the correct method. The name of the generic function is combined with the first element of the class attribute into the form, *`generic.class`* and a function with that name is sought. If the function is found then it is used. If no such function is found then the second element of the class attribute is used, and so on until all the elements of the class attribute have been exhausted. If no method has been found at that point then the method *`generic.default`* is used. If the first argument to the generic function has no class attribute then *`generic.default`* is used. Since the introduction of namespaces the methods may not be accessible by their names (i.e. `get("generic.class")` may fail), but they will be accessible by `getS3method("generic","class")`.

Any object can have a `class` attribute. This attribute can have any number of elements. Each of these is a string that defines a class. When a generic function is invoked the class of its first argument is examined.

5.4 UseMethod

`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.

`UseMethod` changes the evaluation model in two ways. First, when it is invoked it determines the next method (function) to be called. It then invokes that function using the current evaluation environment; this process will be described shortly. The second way in which `UseMethod` changes the evaluation environment is that it does not return control to the calling function. This means, that any statements after a call to `UseMethod` are guaranteed not to be executed.

When `UseMethod` is invoked the generic function is the specified value in the call to `UseMethod`. The object to dispatch on is either the supplied second argument or the first argument to the current function. The class of the argument is determined and the first element of it is combined with the name of the generic to determine the appropriate method. So, if the generic had name `foo` and the class of the object is `"bar"`, then R will search for a method named `foo.bar`. If no such method exists then the inheritance mechanism described above is used to locate an appropriate method.

Once a method has been determined R invokes it in a special way. Rather than creating a new evaluation environment R uses the environment of the current function call (the call to the generic). Any assignments or evaluations that were made before the call to `UseMethod` will be in effect. The arguments that were used in the call to the generic are rematched to the formal arguments of the selected method.

When the method is invoked it is called with arguments that are the same in number and have the same names as in the call to the generic. They are matched to the arguments of the method according to the standard R rules for argument matching. However the object, i.e. the first argument has been evaluated.

The call to `UseMethod` has the effect of placing some special objects in the evaluation frame. They are `.Class`, `.Generic` and `.Method`. These special objects are used to by R to handle the method dispatch and inheritance. `.Class` is the class of the object, `.Generic` is the name of the generic function and `.Method` is the name of the method currently being invoked. If the method was invoked through one of the internal interfaces then there may also be an object called `.Group`. This will be described in Section Section 5.6 [Group methods], page 34. After the initial call to `UseMethod` these special variables, not the object itself, control the selection of subsequent methods.

The body of the method is then evaluated in the standard fashion. In particular variable look-up in the body follows the rules for the method. So if the method has an associated environment then that is used. In effect we have replaced the call to the generic by a call to the method. Any local assignments in the frame of the generic will be carried forward into the call to the method. Use of this *feature* is discouraged. It is important to realize that control will never return to the generic and hence any expressions after a call to `UseMethod` will never be executed.

Any arguments to the generic that were evaluated prior to the call to `UseMethod` remain evaluated.

If the first argument to `UseMethod` is not supplied it is assumed to be the name of the current function. If two arguments are supplied to `UseMethod` then the first is the name of the method and the second is assumed to be the object that will be dispatched on. It is evaluated so that the required method can be determined. In this case the first argument in the call to the generic is not evaluated and is discarded. There is no way to change the other arguments in the call to the method; these remain as they were in the call to the

generic. This is in contrast to `NextMethod` where the arguments in the call to the next method can be altered.

5.5 NextMethod

`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.

Unevaluated arguments remain unevaluated. Missing arguments remain missing.

The syntax for a call to `NextMethod` is `NextMethod(generic, object, ...)`. If the `generic` is not supplied the value of `.Generic` is used. If the `object` is not supplied the first argument in the call to the current method is used. Values in the `...` argument are used to modify the arguments of the next method.

It is important to realize that the choice of the next method depends on the current values of `.Generic` and `.Class` and not on the object. So changing the object in a call to `NextMethod` affects the arguments received by the next method but does not affect the choice of the next method.

Methods can be called directly. If they are then there will be no `.Generic`, `.Class` or `.Method`. In this case the `generic` argument of `NextMethod` must be specified. The value of `.Class` is taken to be the class attribute of the object which is the first argument to the current function. The value of `.Method` is the name of the current function. These choices for default values ensure that the behaviour of a method doesn't change depending on whether it is called directly or via a call to a generic.

An issue for discussion is the behaviour of the `...` argument to `NextMethod`. The White Book describes the behaviour as follows:

- named arguments replace the corresponding arguments in the call to the current method. Unnamed arguments go at the start of the argument list.

What I would like to do is:

-first do the argument matching for NextMethod; -if the object or generic are changed fine -first if a named list element matches an argument (named or not) the list value replaces the argument value. - the first unnamed list element

Values for lookup: Class: comes first from .Class, second from the first argument to the method and last from the object specified in the call to NextMethod

Generic: comes first from .Generic, if nothing then from the first argument to the method and if it's still missing from the call to NextMethod

Method: this should just be the current function name.

5.6 Group methods

For several types of internal functions R provides a dispatching mechanism for operators. This means that operators such as `==` or `<` can have their behaviour modified for members

of special classes. The functions and operators have been grouped into three categories and group methods can be written for each of these categories. There is currently no mechanism to add groups. It is possible to write methods specific to any function within a group.

The following table lists the functions for the different Groups.

'`Math`' abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cospi, cumsum, exp, floor, gamma, lgamma, log, log10, round, signif, sin, sinh, sinpi, tan, tanh, tanpi, trunc

'`Summary`' all, any, max, min, prod, range, sum

'`Ops`' `+`, `-`, `*`, `/`, `^`, `<` , `>`, `<=`, `>=`, `!=`, `==`, `%%`, `%/%`, `&`, `|`, `!`

For operators in the Ops group a special method is invoked if the two operands taken together suggest a single method. Specifically, if both operands correspond to the same method or if one operand corresponds to a method that takes precedence over that of the other operand. If they do not suggest a single method then the default method is used. Either a group method or a class method dominates if the other operand has no corresponding method. A class method dominates a group method.

When the group is Ops the special variable `.Method` is a string vector with two elements. The elements of `.Method` are set to the name of the method if the corresponding argument is a member of the class that was used to determine the method. Otherwise the corresponding element of `.Method` is set to the zero length string, `""`.

5.7 Writing methods

Users can easily write their own methods and generic functions. A generic function is simply a function with a call to `UseMethod`. A method is simply a function that has been invoked via method dispatch. This can be as a result of a call to either `UseMethod` or `NextMethod`.

It is worth remembering that methods can be called directly. That means that they can be entered without a call to `UseMethod` having been made and hence the special variables `.Generic`, `.Class` and `.Method` will not have been instantiated. In that case the default rules detailed above will be used to determine these.

The most common use of generic functions is to provide `print` and `summary` methods for statistical objects, generally the output of some model fitting process. To do this, each model attaches a class attribute to its output and then provides a special method that takes that output and provides a nice readable version of it. The user then needs only remember that `print` or `summary` will provide nice output for the results of any analysis.

6 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.

There is a number of standard applications of this facility, such as calculation of analytical derivatives of expressions, or the generation of polynomial functions from a vector of coefficients. However, there are also uses that are much more fundamental to the workings of the interpreted part of R. Some of these are essential to the reuse of functions as components in other functions, as the (admittedly not very pretty) calls to `model.frame` that are constructed in several modeling and plotting routines. Other uses simply allow elegant interfaces to useful functionality. As an example, consider the `curve` function, which allows you to draw the graph of a function given as an expression like `sin(x)` or the facilities for plotting mathematical expressions.

In this chapter, we give an introduction to the set of facilities that are available for computing on the language.

6.1 Direct manipulation of language objects

There are three kinds of language objects that are available for modification, calls, expressions, and functions. At this point, we shall concentrate on the call objects. These are sometimes referred to as "unevaluated expressions", although this terminology is somewhat confusing. The most direct method of obtaining a call object is to use `quote` with an expression argument, e.g.,

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

The arguments are not evaluated, the result is simply the parsed argument. The objects `e1` and `e2` may be evaluated later using `eval`, or simply manipulated as data. It is perhaps most immediately obvious why the `e2` object has mode `"call"`, since it involves a call to the `plot` function with some arguments. However, `e1` actually has exactly the same structure as a call to the binary operator `+` with two arguments, a fact that gets clearly displayed by the following

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

The components of a call object are accessed using a list-like syntax, and may in fact be converted to and from lists using `as.list` and `as.call`

```
> e2[[1]]
plot
```

```
> e2[[2]]
x
> e2[[3]]
y
```

When keyword argument matching is used, the keywords can be used as list tags:

```
> e3 <- quote(plot(x = age, y = weight))
> e3$x
age
> e3$y
weight
```

All the components of the call object have mode `"name"` in the preceding examples. This is true for identifiers in calls, but the components of a call can also be constants—which can be of any type, although the first component had better be a function if the call is to be evaluated successfully—or other call objects, corresponding to subexpressions. Objects of mode name can be constructed from character strings using `as.name`, so one might modify the `e2` object as follows

```
> e2[[1]] <- as.name("+")
> e2
x + y
```

To illustrate the fact that subexpressions are simply components that are themselves calls, consider

```
> e1[[2]] <- e2
> e1
x + y + 2
```

All grouping parentheses in input are preserved in parsed expressions. They are represented as a function call with one argument, so that `4 - (2 - 2)` becomes `"-"(4, "("("-"(2, 2)))` in prefix notation. In evaluations, the '`(`' operator just returns its argument.

This is a bit unfortunate, but it is not easy to write a parser/deparser combination that both preserves user input, stores it in minimal form and ensures that parsing a deparsed expression gives the same expression back.

As it happens, R's parser is not perfectly invertible, nor is its deparser, as the following examples show

```
> 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
```

Deparsed expressions should, however, evaluate to an equivalent value to the original expression (up to rounding error).

...internal storage of flow control constructs...note Splus incompatibility...

6.2 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. An example of this is seen at the beginning of `plot.default`:

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

This causes the variable or expression given as the `x` argument to `plot` to be used for labeling the x-axis later on.

The function used to achieve this is `substitute` which takes the expression `x` and substitutes the expression that was passed through the formal argument `x`. Notice that for this to happen, `x` must carry information about the expression that creates its value. This is related to the lazy evaluation scheme of R (see Section 2.1.8 [Promise objects], page 5). A formal argument is really a *promise*, an object with three slots, one for the expression that defines it, one for the environment in which to evaluate that expression, and one for the value of that expression once evaluated. `substitute` will recognize a promise variable and substitute the value of its expression slot. If `substitute` is invoked inside a function, the local variables of the function are also subject to substitution.

The argument to `substitute` does not have to be a simple identifier, it can be an expression involving several variables and substitution will occur for each of these. Also, `substitute` has an additional argument which can be an environment or a list in which the variables are looked up. For example:

```
> substitute(a + b, list(a = 1, b = quote(x)))
1 + x
```

Notice that quoting was necessary to substitute the `x`. This kind of construction comes in handy in connection with the facilities for putting math expression in graphs, as the following case shows

```
> plot(0)
> for (i in 1:4)
+   text(1, 0.2 * i,
+        substitute(x[ix] == y, list(ix = i, y = pnorm(i))))
```

It is important to realize that the substitutions are purely lexical; there is no checking that the resulting call objects make sense if they are evaluated. `substitute(x <- x + 1, list(x = 2))` will happily return `2 <- 2 + 1`. However, some parts of R make up their own rules for what makes sense and what does not and might actually have a use for such ill-formed expressions. For example, using the "math in graphs" feature often involves constructions that are syntactically correct, but which would be meaningless to evaluate, like '`{}>=40*" years"`'.

Substitute will not evaluate its first argument. This leads to the puzzle of how to do substitutions on an object that is contained in a variable. The solution is to use `substitute` once more, like this

```
> expr <- quote(x + y)
> substitute(substitute(e, list(x = 3)), list(e = expr))
substitute(x + y, list(x = 3))
> eval(substitute(substitute(e, list(x = 3)), list(e = expr)))
3 + y
```

The exact rules for substitutions are as follows: Each symbol in the parse tree for the first is matched against the second argument, which can be a tagged list or an environment frame. If it is a simple local object, its value is inserted, *except* if matching against the global environment. If it is a promise (usually a function argument), the promise expression is substituted. If the symbol is not matched, it is left untouched. The special exception for substituting at the top level is admittedly peculiar. It has been inherited from S and the rationale is most likely that there is no control over which variables might be bound at that level so that it would be better to just make substitute act as `quote`.

The rule of promise substitution is slightly different from that of S if the local variable is modified before `substitute` is used. R will then use the new value of the variable, whereas S will unconditionally use the argument expression—unless it was a constant, which has the curious consequence that `f((1))` may be very different from `f(1)` in S. The R rule is considerably cleaner, although it does have consequences in connection with lazy evaluation that comes as a surprise to some. Consider

```
logplot <- function(y, ylab = deparse(substitute(y))) {
    y <- log(y)
    plot(y, ylab = ylab)
}
```

This looks straightforward, but one will discover that the y label becomes an ugly `c(...)` expression. It happens because the rules of lazy evaluation cause the evaluation of the `ylab` expression to happen *after* `y` has been modified. The solution is to force `ylab` to be evaluated first, i.e.,

```
logplot <- function(y, ylab = deparse(substitute(y))) {
    ylab
    y <- log(y)
    plot(y, ylab = ylab)
}
```

Notice that one should not use `eval(ylab)` in this situation. If `ylab` is a language or expression object, then that would cause the object to be evaluated as well, which would not at all be desirable if a math expression like `quote(log[e](y))` was being passed.

A variant on `substitute` is `bquote`, which is used to replace some subexpressions with their values. The example from above

```
> plot(0)
> for (i in 1:4)
+   text(1, 0.2 * i,
+        substitute(x[ix] == y, list(ix = i, y = pnorm(i))))
```

could be written more compactly as

```
plot(0)
for(i in 1:4)
   text(1, 0.2*i, bquote( x[.(i)] == .(pnorm(i)) ))
```

The expression is quoted except for the contents of `.()` subexpressions, which are replaced with their values. There is an optional argument to compute the values in a different environment. The syntax for `bquote` is borrowed from the LISP backquote macro.

6.3 More on evaluation

The `eval` function was introduced earlier in this chapter as a means of evaluating call objects. However, this is not the full story. It is also possible to specify the environment in which the evaluation is to take place. By default this is the evaluation frame from which `eval` is called, but quite frequently it needs to be set to something else.

Very often, the relevant evaluation frame is that of the parent of the current frame (cf. ???). In particular, when the object to evaluate is the result of a `substitute` operation of the function arguments, it will contain variables that make sense to the caller only (notice that there is no reason to expect that the variables of the caller are in the lexical scope of the callee). Since evaluation in the parent frame occurs frequently, an `eval.parent` function exists as a shorthand for `eval(expr, sys.frame(sys.parent()))`.

Another case that occurs frequently is evaluation in a list or a data frame. For instance, this happens in connection with the `model.frame` function when a `data` argument is given. Generally, the terms of the model formula need to be evaluated in `data`, but they may occasionally also contain references to items in the caller of `model.frame`. This is sometimes useful in connection with simulation studies. So for this purpose one needs not only to evaluate an expression in a list, but also to specify an enclosure into which the search continues if the variable is not in the list. Hence, the call has the form

```
eval(expr, data, sys.frame(sys.parent()))
```

Notice that evaluation in a given environment may actually change that environment, most obviously in cases involving the assignment operator, such as

```
eval(quote(total <- 0), environment(robert$balance)) # rob Rob
```

This is also true when evaluating in lists, but the original list does not change because one is really working on a copy.

6.4 Evaluation of expression objects

Objects of mode `"expression"` are defined in Section 2.1.4 [Expression objects], page 4. They are very similar to lists of call objects.

```
> ex <- expression(2 + 2, 3 + 4)
> ex[[1]]
2 + 2
> ex[[2]]
3 + 4
> eval(ex)
[1] 7
```

Notice that 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. The evaluator will not evaluate an expression object recursively, only when it is passed directly to `eval` function as above. The difference can be seen like this:

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

The deparser represents an expression object by the call that creates it. This is similar to the way it handles numerical vectors and several other objects that do not have a specific external representation. However, it does lead to the following bit of confusion:

```
> e <- quote(expression(2 + 2))
> e
expression(2 + 2)
> mode(e)
[1] "call"
> ee <- expression(2 + 2)
> ee
expression(2 + 2)
> mode(ee)
[1] "expression"
```

I.e., `e` and `ee` look identical when printed, but one is a call that generates an expression object and the other is the object itself.

6.5 Manipulation of function calls

It is possible for a function to find out how it has been called by looking at the result of `sys.call` as in the following example of a function that simply returns its own call:

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

However, this is not really useful except for debugging because it requires the function to keep track of argument matching in order to interpret the call. For instance, it must be able to see that the 2nd actual argument gets matched to the first formal one (`x` in the above example).

More often one requires the call with all actual arguments bound to the corresponding formals. To this end, the function `match.call` is used. Here's a variant of the preceding example, a function that returns its own call with arguments matched

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

Notice that the second argument now gets matched to `x` and appears in the corresponding position in the result.

The primary use of this technique is to call another function with the same arguments, possibly deleting some and adding others. A typical application is seen at the start of the `lm` function:

```
mf <- cl <- match.call()
mf$singular.ok <- mf$model <- mf$method <- NULL
mf$x <- mf$y <- mf$qr <- mf$contrasts <- NULL
mf$drop.unused.levels <- TRUE
mf[[1]] <- as.name("model.frame")
mf <- eval(mf, sys.frame(sys.parent()))
```

Notice that the resulting call is evaluated in the parent frame, in which one can be certain that the involved expressions make sense. The call can be treated as a list object where the first element is the name of the function and the remaining elements are the actual argument expressions, with the corresponding formal argument names as tags. Thus, the technique to eliminate undesired arguments is to assign `NULL`, as seen in lines 2 and 3, and to add an argument one uses tagged list assignment (here to pass `drop.unused.levels = TRUE`) as in line 4. To change the name of the function called, assign to the first element of the list and make sure that the value is a name, either using the `as.name("model.frame")` construction here or `quote(model.frame)`.

The `match.call` function has an `expand.dots` argument, a switch which if set to `FALSE` lets all `...` arguments be collected as a single argument with the tag `....`

```
> f <- function(x, y, ...) match.call(expand.dots = FALSE)
> f(y = 1, 2, z = 3, 4)
f(x = 2, y = 1, ... = list(z = 3, 4))
```

The `...` argument is a list (a pairlist to be precise), not a call to `list` like it is in S:

```
> e1 <- f(y = 1, 2, z = 3, 4)$...
> e1
$z
[1] 3

[[2]]
[1] 4
```

One reason for using this form of `match.call` is simply to get rid of any `...` arguments in order not to be passing unspecified arguments on to functions that may not know them. Here's an example paraphrased from `plot.formula`:

```
m <- match.call(expand.dots = FALSE)
m$... <- NULL
m[[1]] <- "model.frame"
```

A more elaborate application is in `update.default` where a set of optional extra arguments can add to, replace, or cancel those of the original call:

```
extras <- match.call(expand.dots = FALSE)$...
if (length(extras) > 0) {
    existing <- !is.na(match(names(extras), names(call)))
```

```
        for (a in names(extras)[existing]) call[[a]] <- extras[[a]]
        if (any(!existing)) {
            call <- c(as.list(call), extras[!existing])
            call <- as.call(call)
        }
    }
```

Notice that care is taken to modify existing arguments individually in case `extras[[a]] == NULL`. Concatenation does not work on call objects without the coercion as shown; this is arguably a bug.

Two further functions exist for the construction of function calls, namely `call` and `do.call`.

The function `call` allows creation of a call object from the function name and the list of arguments

```
> x <- 10.5
> call("round", x)
round(10.5)
```

As seen, the value of `x` rather than the symbol is inserted in the call, so it is distinctly different from `round(x)`. The form is used rather rarely, but is occasionally useful where the name of a function is available as a character variable.

The function `do.call` is related, but evaluates the call immediately and takes the arguments from an object of mode `"list"` containing all the arguments. A natural use of this is when one wants to apply a function like `cbind` to all elements of a list or data frame.

```
is.na.data.frame <- function (x) {
    y <- do.call(cbind, lapply(x, is.na))
    rownames(y) <- row.names(x)
    y
}
```

Other uses include variations over constructions like `do.call("f", list(...))`. However, one should be aware that this involves evaluation of the arguments before the actual function call, which may defeat aspects of lazy evaluation and argument substitution in the function itself. A similar remark applies to the `call` function.

6.6 Manipulation of functions

It is often useful to be able to manipulate the components of a function or closure. R provides a set of interface functions for this purpose.

`body` Returns the expression that is the body of the function.

`formals` Returns a list of the formal arguments to the function. This is a `pairlist`.

`environment`
Returns the environment associated with the function.

`body<-` This sets the body of the function to the supplied expression.

`formals<-`
Sets the formal arguments of the function to the supplied list.

`environment<-`
: Sets the environment of the function to the specified environment.

It is also possible to alter the bindings of different variables in the environment of the function, using code along the lines of `evalq(x <- 5, environment(f))`.

It is also possible to convert a function to a list using `as.list`. The result is the concatenation of the list of formal arguments with the function body. Conversely such a list can be converted to a function using `as.function`. This functionality is mainly included for S compatibility. Notice that environment information is lost when `as.list` is used, whereas `as.function` has an argument that allows the environment to be set.

7 System and foreign language interfaces

7.1 Operating system access

Access to the operating system shell is via the R function `system`. The details will differ by platform (see the on-line help), and about all that can safely be assumed is that the first argument will be a string `command` that will be passed for execution (not necessarily by a shell) and the second argument will be `internal` which if true will collect the output of the command into an R character vector.

The functions `system.time` and `proc.time` are available for timing (although the information available may be limited on non-Unix-like platforms).

Information from the operating system environment can be accessed and manipulated with

`Sys.getenv`	OS environment variables
`Sys.putenv`	
`Sys.getlocale`	System locale
`Sys.putlocale`	
`Sys.localeconv`	
`Sys.time`	Current time
`Sys.timezone`	Time zone

A uniform set of file access functions is provided on all platforms:

`file.access`	Ascertain File Accessibility
`file.append`	Concatenate files
`file.choose`	Prompt user for file name
`file.copy`	Copy files
`file.create`	Create or truncate a files
`file.exists`	Test for existence
`file.info`	Miscellaneous file information
`file.remove`	remove files
`file.rename`	rename files
`file.show`	Display a text file
`unlink`	Remove files or directories.

There are also functions for manipulating file names and paths in a platform-independent way.

`basename`	File name without directory
`dirname`	Directory name
`file.path`	Construct path to file
`path.expand`	Expand ~ in Unix path

7.2 Foreign language interfaces

See Section "System and foreign language interfaces" in *Writing R Extensions* for the details of adding functionality to R via compiled code.

Functions `.C` and `.Fortran` provide a standard interface to compiled code that has been linked into R, either at build time or via `dyn.load`. They are primarily intended for

compiled **C** and FORTRAN code respectively, but the `.C` function can be used with other languages which can generate C interfaces, for example C++.

Functions `.Call` and `.External` provide interfaces which allow compiled code (primarily compiled **C** code) to manipulate R objects.

7.3 .Internal and .Primitive

The `.Internal` and `.Primitive` interfaces are used to call **C** code compiled into R at build time. See Section ".Internal vs .Primitive" in *R Internals*.

8 Exception handling

The exception handling facilities in R are provided through two mechanisms. Functions such as `stop` or `warning` can be called directly or options such as `"warn"` can be used to control the handling of problems.

8.1 stop

A call to `stop` halts the evaluation of the current expression, prints the message argument and returns execution to top-level.

8.2 warning

The function `warning` takes a single argument that is a character string. The behaviour of a call to `warning` depends on the value of the option `"warn"`. If `"warn"` is negative warnings are ignored. If it is zero, they are stored and printed after the top-level function has completed. If it is one, they are printed as they occur and if it is 2 (or larger) warnings are turned into errors.

If `"warn"` is zero (the default), a variable `last.warning` is created and the messages associated with each call to `warning` are stored, sequentially, in this vector. If there are fewer than 10 warnings they are printed after the function has finished evaluating. If there are more than 10 then a message indicating how many warnings occurred is printed. In either case `last.warning` contains the vector of messages, and `warnings` provides a way to access and print it.

8.3 on.exit

A function can insert a call to `on.exit` at any point in the body of a function. The effect of a call to `on.exit` is to store the value of the body so that it will be executed when the function exits. This allows the function to change some system parameters and to ensure that they are reset to appropriate values when the function is finished. The `on.exit` is guaranteed to be executed when the function exits either directly or as the result of a warning.

An error in the evaluation of the `on.exit` code causes an immediate jump to top-level without further processing of the `on.exit` code.

`on.exit` takes a single argument which is an expression to be evaluated when the function is exited.

8.4 Error options

There are a number of `options` variables that can be used to control how R handles errors and warnings. They are listed in the table below.

'`warn`' Controls the printing of warnings.

'`warning.expression`'
Sets an expression that is to be evaluated when a warning occurs. The normal printing of warnings is suppressed if this option is set.

‘`error`’ Installs an expression that will be evaluated when an error occurs. The normal printing of error messages and warning messages precedes the evaluation of the expression.

Expressions installed by `options("error")` are evaluated before calls to `on.exit` are carried out.

One can use `options(error = expression(q("yes")))` to get R to quit when an error has been signalled. In this case an error will cause R to shut down and the global environment will be saved.

9 Debugging

Debugging code has always been a bit of an art. R provides several tools that help users find problems in their code. These tools halt execution at particular points in the code and the current state of the computation can be inspected.

Most debugging takes place either through calls to `browser` or `debug`. Both of these functions rely on the same internal mechanism and both provide the user with a special prompt. Any command can be typed at the prompt. The evaluation environment for the command is the currently active environment. This allows you to examine the current state of any variables etc.

There are five special commands that R interprets differently. They are,

'`RET`' Go to the next statement if the function is being debugged. Continue execution if the browser was invoked.

'`c`'
'`cont`' Continue the execution.

'`n`' Execute the next statement in the function. This works from the browser as well.

'`where`' Show the call stack

'`Q`' Halt execution and jump to the top-level immediately.

If there is a local variable with the same name as one of the special commands listed above then its value can be accessed by using `get`. A call to `get` with the name in quotes will retrieve the value in the current environment.

The debugger provides access only to interpreted expressions. If a function calls a foreign language (such as **C**) then no access to the statements in that language is provided. Execution will halt on the next statement that is evaluated in R. A symbolic debugger such as `gdb` can be used to debug compiled code.

9.1 browser

A call to the function `browser` causes R to halt execution at that point and to provide the user with a special prompt. Arguments to `browser` are ignored.

```
> foo <- function(s) {
+ c <- 3
+ browser()
+ }
> foo(4)
Called from: foo(4)
Browse[1]> s
[1] 4
Browse[1]> get("c")
[1] 3
Browse[1]>
```

9.2 debug/undebug

The debugger can be invoked on any function by using the command `debug(`*`fun`*`)`. Subsequently, each time that function is evaluated the debugger is invoked. The debugger allows you to control the evaluation of the statements in the body of the function. Before each statement is executed the statement is printed out and a special prompt provided. Any command can be given, those in the table above have special meaning.

Debugging is turned off by a call to `undebug` with the function as an argument.

```
> debug(mean.default)
> mean(1:10)
debugging in: mean.default(1:10)
debug: {
    if (na.rm)
        x <- x[!is.na(x)]
    trim <- trim[1]
    n <- length(c(x, recursive = TRUE))
    if (trim > 0) {
        if (trim >= 0.5)
            return(median(x, na.rm = FALSE))
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort(x, partial = unique(c(lo, hi)))[lo:hi]
        n <- hi - lo + 1
    }
    sum(x)/n
}
Browse[1]>
debug: if (na.rm) x <- x[!is.na(x)]
Browse[1]>
debug: trim <- trim[1]
Browse[1]>
debug: n <- length(c(x, recursive = TRUE))
Browse[1]> c
exiting from: mean.default(1:10)
[1] 5.5
```

9.3 trace/untrace

Another way of monitoring the behaviour of R is through the `trace` mechanism. `trace` is called with a single argument that is the name of the function you want to trace. The name does not need to be quoted but for some functions you will need to quote the name in order to avoid a syntax error.

When `trace` has been invoked on a function then every time that function is evaluated the call to it is printed out. This mechanism is removed by calling `untrace` with the function as an argument.

```
> trace("[<-")
> x <- 1:10
```

```
> x[3] <- 4
trace: "[<-"(*tmp*, 3, value = 4)
```

9.4 traceback

When an error has caused a jump to top-level a special variable called `.Traceback` is placed into the base environment. `.Traceback` is a character vector with one entry for each function call that was active at the time the error occurred. An examination of `.Traceback` can be carried out by a call to `traceback`.

10 Parser

The parser is what converts the textual representation of R code into an internal form which may then be passed to the R evaluator which causes the specified instructions to be carried out. The internal form is itself an R object and can be saved and otherwise manipulated within the R system.

10.1 The parsing process

10.1.1 Modes of parsing

Parsing in R occurs in three different variants:

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

The read-eval-print loop forms the basic command line interface to R. Textual input is read until a complete R expression is available. Expressions may be split over several input lines. The primary prompt (by default '`> `') indicates that the parser is ready for a new expression, and a continuation prompt (by default '`+ `') indicates that the parser expects the remainder of an incomplete expression. The expression is converted to internal form during input and the parsed expression is passed to the evaluator and the result is printed (unless specifically made invisible). If the parser finds itself in a state which is incompatible with the language syntax, a "Syntax Error" is flagged and the parser resets itself and resumes input at the beginning of the next input line.

Text files can be parsed using the `parse` function. In particular, this is done during execution of the `source` function, which allows commands to be stored in an external file and executed as if they had been typed at the keyboard. Note, though, that the entire file is parsed and syntax checked before any evaluation takes place.

Character strings, or vectors thereof, can be parsed using the `text=` argument to `parse`. The strings are treated exactly as if they were the lines of an input file.

10.1.2 Internal representation

Parsed expressions are stored in an R object containing the parse tree. A fuller description of such objects can be found in Section 2.1.3 [Language objects], page 4, and Section 2.1.4 [Expression objects], page 4. Briefly, every elementary R expression is stored in function call form, as a list with the first element containing the function name and the remainder containing the arguments, which may in turn be further R expressions. The list elements can be named, corresponding to tagged matching of formal and actual arguments. Note that *all* R syntax elements are treated in this way, e.g. the assignment `x <- 1` is encoded as `"<-"(x, 1)`.

10.1.3 Deparsing

Any R object can be converted to an R expression using `deparse`. This is frequently used in connection with output of results, e.g. for labeling plots. Notice that only objects of mode `"expression"` can be expected to be unchanged by reparsing the output of deparsing. For

instance, the numeric vector `1:5` will deparse as `"c(1, 2, 3, 4, 5)"`, which will reparse as a call to the function `c`. As far as possible, evaluating the deparsed and reparsed expression gives the same result as evaluating the original, but there are a couple of awkward exceptions, mostly involving expressions that weren't generated from a textual representation in the first place.

10.2 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."
```

10.3 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.

10.3.1 Constants

There are five types of constants: integer, logical, numeric, complex and string.

In addition, there are four special constants, `NULL`, `NA`, `Inf`, and `NaN`.

`NULL` is used to indicate the empty object. `NA` is used for absent ("Not Available") data values. `Inf` denotes infinity and `NaN` is not-a-number in the IEEE floating point calculus (results of the operations respectively 1/0 and 0/0, for instance).

Logical constants are either `TRUE` or `FALSE`.

Numeric constants follow a similar syntax to that of the **C** language. They consist of an integer part consisting of zero or more digits, followed optionally by '`.`' and a fractional part of zero or more digits optionally followed by an exponent part consisting of an '`E`' or an '`e`', an optional sign and a string of one or more digits. Either the fractional or the decimal part can be empty, but not both at once.

Valid numeric constants: `1 10 0.1 .2 1e-7 1.2e+7`

Numeric constants can also be hexadecimal, starting with '`0x`' or '`0x`' followed by zero or more digits, '`a-f`' or '`A-F`'. Hexadecimal floating point constants are supported using C99 syntax, e.g. '`0x1.1p1`'.

There is now a separate class of integer constants. They are created by using the qualifier `L` at the end of the number. For example, `123L` gives an integer value rather than a numeric value. The suffix `L` can be used to qualify any non-complex number with the intent of creating an integer. So it can be used with numbers given by hexadecimal or scientific notation. However, if the value is not a valid integer, a warning is emitted and the numeric value created. The following shows examples of valid integer constants, values which will generate a warning and give numeric constants and syntax errors.

Valid integer constants: `1L, 0x10L, 1000000L, 1e6L`
Valid numeric constants: `1.1L, 1e-3L, 0x1.1p-2`
Syntax error: `12iL 0x1.1`

A warning is emitted for decimal values that contain an unnecessary decimal point, e.g. `1.L`. It is an error to have a decimal point in a hexadecimal constant without the binary exponent.

Note also that a preceding sign (`+` or `-`) is treated as a unary operator, not as part of the constant.

Up-to-date information on the currently accepted formats can be found by `?NumericConstants`.

Complex constants have the form of a decimal numeric constant followed by '`i`'. Notice that only purely imaginary numbers are actual constants, other complex numbers are parsed a unary or binary operations on numeric and imaginary numbers.

Valid complex constants: `2i 4.1i 1e-2i`

String constants are delimited by a pair of single ('') or double ('"') quotes and can contain all other printable characters. Quotes and other special characters within strings are specified using *escape sequences*:

`\'` single quote

`\"` double quote

`\n` newline (aka 'line feed', `LF`)

`\r` carriage return (`CR`)

`\t` tab character

`\b` backspace

`\a` bell

`\f` form feed

`\v` vertical tab

`\\` backslash itself

`\`*nnn* character with given octal code – sequences of one, two or three digits in the range `0 ... 7` are accepted.

`\x`*nn* character with given hex code – sequences of one or two hex digits (with entries `0 ... 9 A ... F a ... f`).

`\u`*nnnn* `\u{`*nnnn*`}`
(where multibyte locales are supported, otherwise an error). Unicode character with given hex code – sequences of up to four hex digits. The character needs to be valid in the current locale.

`\U`*nnnnnnnn* `\U{`*nnnnnnnn*`}`
(where multibyte locales are supported, otherwise an error). Unicode character with given hex code – sequences of up to eight hex digits.

A single quote may also be embedded directly in a double-quote delimited string and vice versa.

A 'nul' (`\0`) is not allowed in a character string, so using `\0` in a string constant terminates the constant (usually with a warning): further characters up to the closing quote are scanned but ignored.

10.3.2 Identifiers

Identifiers consist of a sequence of letters, digits, the period (‘`.`’) and the underscore. They must not start with a digit or an underscore, or with a period followed by a digit.

The definition of a letter depends on the current locale: the precise set of characters allowed is given by the C expression `(isalnum(c) || c == '.' || c == '_')` and will include accented letters in many Western European locales.

Notice that identifiers starting with a period are not by default listed by the `ls` function and that `...` and `..1`, `..2`, etc. are special.

Notice also that objects can have names that are not identifiers. These are generally accessed via `get` and `assign`, although they can also be represented by text strings in some limited circumstances when there is no ambiguity (e.g. `"x" <- 1`). As `get` and `assign` are not restricted to names that are identifiers they do not recognise subscripting operators or replacement functions. The following pairs are *not* equivalent

```
x$a<-1          assign("x$a",1)
x[[1]]          get("x[[1]]")
names(x)<-nm    assign("names(x)",nm)
```

10.3.3 Reserved words

The following identifiers have a special meaning and cannot be used for object names

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN
NA NA_integer_ NA_real_ NA_complex_ NA_character_
... ..1 ..2 etc.
```

10.3.4 Special operators

R allows user-defined infix operators. These have the form of a string of characters delimited by the ‘`%`’ character. The string can contain any printable character except ‘`%`’. The escape sequences for strings do not apply here.

Note that the following operators are predefined

```
%% %*% %/% %in% %o% %x%
```

10.3.5 Separators

Although not strictly tokens, stretches of whitespace characters (spaces, tabs and formfeeds, on Windows and UTF-8 locales other Unicode whitespace characters[1]) serve to delimit tokens in case of ambiguity, (compare `x<-5` and `x < -5`).

Newlines have a function which is a combination of token separator and expression terminator. If an expression can terminate at the end of the line the parser will assume it does so, otherwise the newline is treated as whitespace. Semicolons (‘`;`’) may be used to separate elementary expressions on the same line.

Special rules apply to the `else` keyword: inside a compound expression, a newline before `else` is discarded, whereas at the outermost level, the newline terminates the `if` construction and a subsequent `else` causes a syntax error. This somewhat anomalous behaviour occurs

[1] such as `U+A0`, non-breaking space, and `U+3000`, ideographic space.

because R should be usable in interactive mode and then it must decide whether the input expression is complete, incomplete, or invalid as soon as the user presses `RET`.

The comma (‘`,`’) is used to separate function arguments and multiple indices.

10.3.6 Operator tokens

R uses the following operator tokens

`+ - * / %% ^`	arithmetic
`> >= < <= == !=`	relational
`! & |`	logical
`~`	model formulae
`-> <-`	assignment
`$`	list indexing
`:`	sequence

(Several of the operators have different meaning inside model formulas)

10.3.7 Grouping

Ordinary parentheses—‘`(`’ and ‘`)`’—are used for explicit grouping within expressions and to delimit the argument lists for function definitions and function calls.

Braces—‘`{`’ and ‘`}`’—delimit blocks of expressions in function definitions, conditional expressions, and iterative constructs.

10.3.8 Indexing tokens

Indexing of arrays and vectors is performed using the single and double brackets, ‘`[]`’ and ‘`[[]]`’. Also, indexing tagged lists may be done using the ‘`$`’ operator.

10.4 Expressions

An R program consists of a sequence of R expressions. An expression can be a simple expression consisting of only a constant or an identifier, or it can be a compound expression constructed from other parts (which may themselves be expressions).

The following sections detail the various syntactical constructs that are available.

10.4.1 Function calls

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)

Each argument can be tagged (*`tag=expr`*), or just be a simple expression. It can also be empty or it can be one of the special tokens `...`, `..2`, etc.

A tag can be an identifier or a text string.

Examples:

```
f(x)
g(tag = value, , 5)
"odd name"("strange tag" = 5, y)
(function(x) x^2)(5)
```

10.4.2 Infix and prefix operators

The order of precedence (highest first) of the operators is

```
::
$ @
^
- +                    (unary)
:
%xyz% |>
* /
+ -                    (binary)
> >= < <= == !=
!
& &&
| ||
~                      (unary and binary)
-> ->>
<- <<-
=                      (as assignment)
```

Note that `:` precedes binary `+`/`-`, but not `^`. Hence, `1:3-1` is 012, but `1:2^3` is `1:8`.

The exponentiation operator '`^`' and the left assignment plus minus operators '`<- - = <<-`' group right to left, all other operators group left to right. That is, `2 ^ 2 ^ 3` is 2^8, not 4^3, whereas `1 - 1 - 1` is -1, not 1.

Notice that the operators `%%` and `%/%` for integer remainder and divide have higher precedence than multiply and divide.

Although it is not strictly an operator, it also needs mentioning that the '`=`' sign is used for tagging arguments in function calls and for assigning default values in function definitions.

The '`$`' sign is in some sense an operator, but does not allow arbitrary right hand sides and is discussed under Section 10.4.3 [Index constructions], page 58. It has higher precedence than any of the other operators.

The parsed form of a unary or binary operation is completely equivalent to a function call with the operator as the function name and the operands as the function arguments.

Parentheses are recorded as equivalent to a unary operator, with name `"("`, even in cases where the parentheses could be inferred from operator precedence (e.g., `a * (b + c)`).

Notice that the assignment symbols are operators just like the arithmetic, relational, and logical ones. Any expression is allowed also on the target side of an assignment, as far as the parser is concerned (`2 + 2 <- 5` is a valid expression as far as the parser is concerned. The evaluator will object, though). Similar comments apply to the model formula operator.

10.4.3 Index constructions

R has three indexing constructs, two of which are syntactically similar although with somewhat different semantics:

```
object [ arg1, ..... , argn ]
object [[ arg1, ..... , argn ]]
```

The *object* can formally be any valid expression, but it is understood to denote or evaluate to a subsettable object. The arguments generally evaluate to numerical or character indices, but other kinds of arguments are possible (notably `drop = FALSE`).

Internally, these index constructs are stored as function calls with function name `"["` respectively `"[["`.

The third index construction is

```
object $ tag
```

Here, *object* is as above, whereas *tag* is an identifier or a text string. Internally, it is stored as a function call with name `"$"`

10.4.4 Compound expressions

A compound expression is of the form

```
{ expr1 ; expr2 ; ..... ; exprn }
```

The semicolons may be replaced by newlines. Internally, this is stored as a function call with `"{"` as the function name and the expressions as arguments.

10.4.5 Flow control elements

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
```

The expressions in these constructs will typically be compound expressions.

Within the loop constructs (`while`, `repeat`, `for`), one may use `break` (to terminate the loop) and `next` (to skip to the next iteration).

Internally, the constructs are stored as function calls:

```
"if"(cond, expr)
"if"(cond, expr1, expr2)
"while"(cond, expr)
"repeat"(expr)
"for"(var, list, expr)
"break"()
"next"()
```

10.4.6 Function definitions

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.

Notice that function arguments unlike list tags, etc., cannot have "strange names" given as text strings.

Internally, a function definition is stored as a function call with function name `function` and two arguments, the *arglist* and the *body*. The *arglist* is stored as a tagged pairlist where the tags are the argument names and the values are the default expressions.

10.5 Directives

The parser currently only supports one directive, `#line`. This is similar to the C-preprocessor directive of the same name. The syntax is

```
#line nn [ "filename" ]
```

where *nn* is an integer line number, and the optional *filename* (in required double quotes) names the source file.

Unlike the C directive, `#line` must appear as the first five characters on a line. As in C, *nn* and `"filename"` entries may be separated from it by whitespace. And unlike C, any following text on the line will be treated as a comment and ignored.

This directive tells the parser that the following line should be assumed to be line *nn* of file *filename*. (If the filename is not given, it is assumed to be the same as for the previous directive.) This is not typically used by users, but may be used by preprocessors so that diagnostic messages refer to the original file.

Function and Variable Index

O

P

Q

R

S

T

U

W

Concept Index

V

Appendix A References

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), *The New S Language.* Chapman & Hall, New York. This book is often called the "*Blue Book*".