# Eine Einführung in R

Lars-Erik Kimmel
DCSM Hochschule RheinMain
Lars-Erik.Kimmel@gmx.de

28. Februar 2010

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 S

Die Sprache S wurde in den 1980er Jahren in den Bell Labs unter der Leitung von John M. Chambers entwickelt. S ist ein System zur Berechnung, Darstellung und Speicherung statistischer Modelle.

Dank seiner revolutionären Benutzbarkeit im Gebiet der Statistik, wurde die Sprache 1998 mit dem ACM-Award Software System Award ausgezeichnet. Andere vorherige ACM-Preisträger waren zum Beispiel Unix, TeX oder das WWW.

Das wichtigste Ziel der Sprache ist es, den Gedankengang des Benutzers wieder zu spiegeln. Dieses Merkmal zeigt sich deutlich in der Syntax der Sprache. S ist hauptsächlich eine interaktive Sprache, mit der man allein in der Kommandozeile schon sehr weit kommt. Seine Unix-ähnliche Umgebung erleichert die Arbeit. Die S-Syntax ähnelt C und einigen modernen Skriptsprachen.

Heute läuft S kommerziell weit verbreitet unter dem Namen S+ von der Firma TIBCO.

## 1.2 R

R hat sich entgegen S+ entwickelt. Während S+ im Prinzip S mit einigen Extras ist und eine ausgefeiltere GUI liefert, ist R nach der S-Sprachdefinition als Opensource-Projekt entwickelt worden.

Die Sprache R ist ein Dialekt von S mit absichtlichen Unterschieden in der Implementierung. Die meisten S-Programme laufen aber unter R und tatsächlich sind die meisten Funktionen in R mit S-Code implementiert. Die signifikantesten Unterschiede befinden sich im Scoping von Variablen und in der Laufzeiteffizienz, die in den meisten Fällen in R dramatisch besser ist. Eine vollständige Liste aller Unterschiede zwischen S und R wird im R-FAQ geführt. Der größte offensichtliche Unterschied ist, dass R kostenlos ist.

Wie schon in S ist es möglich, neue Modelle zu programmieren und in Paketen zur Wiederverwendung zu verpacken. R bietet schon mit der Installation ein großes Basispaket für allgemeine Modelle. Durch die Community sind viele weitere Pakete für spezielle Modelle verfügbar.

Auch R liefert eine Unix-ähnliche Kommandozeile und eine simple GUI, die weitreichend für den üblichen Gebrauch genügt. Die R-Umgebung hilft wie die Unix-Shell mit Kommandovervollständigung von Präfixen beim Druck der Tab-Taste und einer Kommandohistorie, die durch die Pfeiltasten auswählbar ist. Für die

Abwärtskompatibilität zu S gibt es eine Sprachanbindung zu Fortran und C, was die Programmierung effektiver Algorithmen erlaubt.

Diese Ausarbeitung bezieht sich hauptsächlich auf die R-Sprachdefinition. Da R aber S implementiert, gelten die meisten Definitionen auch für S. Der Text spricht nebenbei deutliche semantische Unterschiede in den Implementierungen an.

# 2 Basistypen

R besitzt 5 elementare Typen, die die Basis aller komplexeren Datentypen bilden.

- numeric
- integer
- complex
- logical
- string

## 2.1 numeric

Numerische Konstanten entsprechen in Dezimal- oder Hexadezimaldarstellung der Zahlennotation in C. Die Zahlen bestehen aus den Zeichen [0-9], Nachkommazahlen werden durch "." vom ganzzahligen Teil gekennzeichnet. Hexadezimalzahlen starten mit der Zeichenfolge "0x". Führende Vorzeichen gelten dabei nicht als Teil der Konstante, sondern als Operatoren.

Listing 1: numerische Konstanten

```
1 10
2 0.1
3 .2
4 1e-7
5 1.2e+7
6 2e
7 3e+
```

## 2.2 integer

Ganze Zahlen haben die eigene Klasse Integer. Der Suffix ”L” an numerischen nicht komplexen Zahlen erzeugt eine ganze Zahl. Ist der Wert keine korrekte Zahl wird eine Warnung ausgegeben und der numerische Anteil des Wertes verwendet.

Listing 2: ganzzahlig Konstanten

```
1  # Integer:
2  1L
3  0x10L
4  1000000L
5  1e6L
6
7  # numerische Werte:
8  1.1L
9  1e-3L
10
11 # Syntaxerror:
12 12iL
```

## 2.3 complex

Komplexe Zahlen haben einen numerischen Wert gefolgt von einem ”i”, der den imaginären Teil kennzeichnet. Nur pure imaginäre Zahlen sind vom Typ complex. Der reale numerische Teil wird per Operator dazu gerechnet.

Listing 3: komplexe Konstanten

```
1 # imaginre Zahlen:
2 2i
3 4.1i
4 # mit realem Anteil:
5 1e-2i
```

## 2.4 logical

Als logische Konstanten existieren TRUE und FALSE. Es gibt auch die kürzeren Variablen T und F, die aber wegen ihrer Veränderbarkeit für weitere Programmierung nicht empfehlenswert sind.

### 2.5 string

Strings sind wie in C Zeichenketten. Sie werden von ein- oder zweigestrichelten Anführungszeichen eingeschlossen, können aber auch in bestimmten Anwendungen ohne Anführungszeichen stehen. Strings können auch Sonderzeichen enthalten, müssen dann aber in Anführungszeichen stehen, bzw. in ihrer notiert werden. Ein Zeichen wird durch einen vorangestellten Backslash "\" in seiner escaped Sequenz geschrieben. Das nul-Zeichen "\0" sollte nicht verwendet werden. Es beendet einen String. Alle dem nul-Zeichen folgenden Zeichen werden ignoriert, bleiben aber bestehen.

## 3 vordefinierte Konstanten

Ergänzend zu den den elementaren Typen gibt es 4 vordefinierte Konstanten.

- NULL
- NA
- NaN
- Inf

### 3.1 NULL

NULL ist eine Referenz auf das NULL-Objekt. Das NULL-Objekt klassifiziert leere Objekte. Es gibt nur ein NULL-Objekt im verwendeten Speicher.

### 3.2 NA

Wie NULL für Objekte, gibt es für statistische Zahlenwerte ein ähnliches Konstrukt: NA (**n**ot **a**vailable). NA steht für einen Wert, der nicht vorhanden ist. Jeder Basistyp hat sein eigenes NA, was den internen Umgang in Berechnungen vereinfacht. Auf einen NA-Wert einer Variable kann mit *is.na(object)* geprüft werden. Die Funktion gibt auch TRUE auf NaN zurück.

Listing 4: Prüfung auf NA Werte

```
1 # ergibt TRUE:
2 is.na(NA)
3 # ergibt auch TRUE:
4 is.na(NaN)
```

## 3.3 NaN

NaN steht für "**n**ot **a** **n**umber" gemäß dem IEEE-Standard für floating point Operationen. Unmögliche Operationen wie Division durch 0 ergeben NaN.

Listing 5: Operationen mit NaN-Ergebnissen

```
1 0/0
2 Inf-Inf
```

## 3.4 Inf

Inf (**inf**inity) ist die Konstante für unendlich ($\infty$).

# 4 Syntax

Anweisungen sind per ";" oder *newline* zu trennen.

"#" leitet Kommentare bis zum Ende der Zeile ein.

Als Variablennamen ist eine Kombination aus alphanumerischen Zeichen, "." und "_" erlaubt. Namen, die mit "." beginnen gelten als besonders; "..." ist ein besonderes Objekt.

Zuweisungen können links oder rechtsseitig erfolgen. Semantisch ist jede Zuweisung linksseitig zur Variable. Üblich ist "<-" als Zuweisung.

Listing 6: einfache Zuweisungen

```
1 x = 1
2 x <- TRUE
3 1.1223 -> x
4 assign("x",  12.0E2)
```

Die numerische Arithmetik hat die üblichen Operatoren `+`, `-`, `*`, `/`, `^`, `**` und viele weitere mathematische Funktionen und deren kürzeren vordefinierten Operatoren. Eine komplette Liste steht unter Prioritäten der Operatoren auf Seite 19.

Operationen, bei denen ein Operand ein NA-Wert ist, ergeben immer NA. Analog ergibt jede Operation mit einem NaN-Wert immer NaN. Vergleiche mit NaN-Werten ergeben immer NA.

Neben den numerischen Operationen, gibt es auch die Arithmetik der komplexen Zahlen. Dabei ist es wichtig, dass der komplexe Teil der Operanden definiert wird. Definiert man den komplexen Anteil der Zahl nicht, wird in der realen Zahlenmenge gerechnet.

Listing 7: komplexe Arithmetik

```
1 # ergibt NaN:
2 sqrt(-1)
3 # ergibt i, rechnet in der komplexen Zahlenmenge
4 sqrt(-1+0i)
```

Wie man sieht erfolgen Funktionsaufrufe wie in C durch Klammern hinter dem Funktionsnamen. Innerhalb der runden Klammern zum Aufruf der Funktion stehen die Parameter durch Kommata getrennt.

## 4.1 Hilfsfunktion

Wie schon erwähnt, kommt man in R alleine in der Kommandozeile schon sehr weit. Für Fälle in denen man nicht weiter weiß, gibt es ein Hilfssystem, das den manpages von Unix ähnelt.

Listing 8: Hilfsfunktion

```
1 # Hilfeaufruf zu einer Funktion:
2 help(solve)
3 # das gleiche kürzer
4 ?solve
5 # bei Sonderzeichen:
6 help("[[")
7 # Suche nach Teilworten in der Hilfe:
8 ??solve
```

# 5 Objekte

Alles in R, das man sehen und anfassen kann, ist ein Objekt. Auch scheinbar alleinstehende Zahlen sind eigentlich Objekte.

Objekte werden primär durch ihren Typ, Modus und Speicherklasse unterschieden.

Auch Variablennamen (symbols) und Ausdrücke (expressions) sind Objekte. Ein Symbol referenziert auf ein Objekt. Ein Objekt, auf das es keine Referenz gibt wird automatisch gelöscht. Es gibt also keine Garbage-Collection die irgendwann zufällig stattfindet. Was nicht mehr verwendet wird oder verwendet werden kann, wird umgehend gelöscht. Gibt man also etwas in die Kommadozeile ohne Zuweisung an ein Symbol ein, wird das Objekt direkt nach der Ausgabe wieder gelöscht

Die Funktionen *objects()* und *ls()* listen alle vorhandenen Objekte mit Namen auf. Namen, die mit "." beginnen, sind unsichtbar. Das Löschen von Objekten kann mit *rm(objectname)* erzwungen werden. Als Parameter wird der verwendete Symbolname der Variable angegeben.

Vorhandene Objekte einer Session können in dem Arbeitspfad gesichert und später wiederverwendet werden. R fragt beim Beenden, ob die Session gesichert werden soll und stellt diese automatisch beim nächsten Start wieder her. S speichert eine Session *immer* vor dem Beenden im aktuellen Arbeitspfad.

Den Modus eines Objekts kann man mittels Funktionen mit *as.<typ>(object)* umwandeln. Eine Rückwandlung erzeugt ein nur beinahe äquivalentes Objekt zum Original. Dies passiert, da es unterschiedliche Wege gibt, ein äquivalentes Objekt zu erstellen und damit eine Rückwandlung formal nicht deterministisch wäre.

Nicht vorhandene Objekte werden durch Zuweisung des NULL-Objekts gekennzeichnet. Solche Zuweisungen beinhalten Referenzen auf dieses einzige unveränderbare NULL-Objekt.

# 6 Attribute

Jedes Objekt hat Attribute, die dessen Eigenschaften beschreiben. Auf Attribute eines Objekts wird mittels vordefinierter Methoden zugegriffen. Der Aufruf dieser Methoden sieht wie ein Funktionsaufruf aus. Diese Methoden liefern eine Referenz auf das Attribut im Speicher, dem man auf den üblichen Weg neue Werte zuweisen kann.

In R hat jedes Objekt die Attribute:

- *typeof(object)* beschreibt den Typ des Objekts.
- *mode(object)* beschreibt den Modus des Objekts.
- *storage.mode(object)* beschreibt die Speicherklasse des Objekts.
- *class(object)* beschreibt eine Liste aller abgeleiteten Klassen.
- *length(object)* beschreibt die Länge eines Objekts.

Das *class*-Attribut realisiert ein Klassensystem, auf das generische Funktionen basieren. Dieses Klassensystem ermöglicht einen OOP-ersetzenden Ansatz zum Überladen von Funktionen. R implementiert kein OOP, sondern ein Klassensystem über das *class*-Attribut.

*Length* liefert nicht immer sinnvolle Ergebnisse, z.B. wenn das Objekt eine Funktion ist. Die Funktionf kann unvorhersehbare Ergebnisse liefern.

Allgemein können Attribute über *attributes(object)* aufgelistet werden. Der allgemeine Getter und Setter von Attributen ist *attr(object, attrname)*. Diese Funktion kann neue Attribute zu einem Objekt hinzufügen, vorhandene Attribute verändern oder löschen. Das Bearbeiten von Attributen nicht selbst erstellter Objekte kann überraschende Ergebnisse liefern, die nicht vorhersehbar sind. Die Verwendung dieser Technik ist vorsichtig einzusetzen. Vorzugsweise gibt es spezielle Funktionen, die Referenzen auf Attribute liefern denen man neue Werte zuweisen kann.

# 7 Basisklassen

R ist aus einigen Basisklassen aufgebaut, wobei die wichtigste und einfachste der Vektor ist.

## 7.1 Vektoren

Vektoren enthalten nur Elemente eines Typs. Der Typ der enthaltenen Elemente bestimmt den Modus des Objekts. Auch leere Vektoren haben einen Modus durch ihre Konstruktor-Methoden.

Listing 9: leerer Vektor

```
1 x <- numeric(0)
2 mode(x) # numeric
```

Die einfachste Funktion, um Vektoren zu erzeugen ist *c(...)*, wobei "..." für eine beliebige Anzahl von Parameter steht. Die Funktion *c(...)* vereint alle Vektoren in der Parameterliste.

Listing 10: einfache Vektoren

```
1 # Ein Vektor [1, 2, 3]
2 x <- c(1, 2, 3)
3 y <- c(x, 4, 5, 6)
4 # y enthält jetzt [1, 2, 3, 4, 5, 6]
```

Wird einem noch nicht vorhandenen Index ein Wert zugewiesen, wird der Vektor mit NA-Werten bis zu dem zugewiesenen Index aufgefüllt. Der Vektor hat dann eine Länge entsprechend des zugewiesenen Indizes.

Listing 11: Erweiterbarkeit von Vektoren

```
1 x <- numeric(0)
2 x[3] <- 2
3 length(x) # 3
```

Man kann nachträglich die Länge eines Vektors direkt über das *lenght*-Attribut ändern. Dabei werden alle neuen Indizes mit NA-Werten gefüllt. Wird die Länge eines Vektors gekürzt, werden alle nachfolgenden Elemente gelöscht.

Listing 12: Verändern der Vektorlänge

```
1 # Ein Vektor [1, 2, 3]
2 x <- c(1, 2, 3)
3 length(x) <- 5
4 # x ist jetzt ein Vektor mit [1, 2, 3, NA, NA]
5 length(x) <- 2
6 # x enthält [1, 2]
```

Die wichtigsten Funktionen für Vektoren sind:

- *range(vector)* liefert einen Vektor mit dem Minimum und Maximum des Vektors. Äquivalent dazu wäre auch *c(min(vector), max(vector))*.
- *length(vector)* liefert die Anzahl der enthaltenen Elemente.
- *sum(vector)* liefert die Summe der enthaltenen Elemente.
- *prod(vector)* liefert das Produkt der enthaltenen Elemente.

Beim Rechnen mit Vektoren wird immer auf den größten Vektor aufgefüllt. Die kleineren Vektoren werden so oft in sich wiederholt, bis die maximale Vektor-Größe in dem Ausdruck erreicht wird. Ist der längste Vektor kein Vielfaches der kürzeren wird eine Warnmeldung ausgegeben.

Listing 13: Erweitern der Operanden bei Vektoroperationen

```
1 x <- c(1, 2, 3)
2 y <- c(1, 2)
3 z <- c(1, 2, 1)
4
5 e <- x * y
6 # e ist ein numerische Vektor mit [1, 4, 3]
```

```
7  f <- x * z
8  # f ist auch ein numerischer Vektor mit [1, 4, 3]
9  # e entspricht also in jedem Element f.
10 # y wurde auf c(1, 2, 1) erweitert.
```

Es gibt eine mächtige Kurzschreibweise von Sequenzen durch den Operator ":". Der ":"-Operator hat eine sehr hohe Priorität, was sehr kurze und komplexe Operationen ermöglicht. Bei Verwendung wird die Funktion *seq(...)* aufgerufen.

Listing 14: Sequenzen und äquivalente Schreibweisen

```
1  # einfache Sequenz:
2  a <- c(1, 2, 3, .., 49, 50)
3  a <- 1:50
4  a <- seq(1, 50)
5
6  # Vektoren mit den ersten 50 geraden Zahlen:
7  b <- c(2, 4, 6, 8, ..., 98, 100)
8  b <- 2*1:50
9
10 # ein komplexerer Ausdruck:
11 c <- 3*c(1, 2, .., 9, 10)-5
12 c <- 3*1:10-5
```

### 7.1.1 logische Vektoren

Elemente eines logischen Vektors sind vom Typ logical und können also die Werte TRUE, FALSE oder NA haben. NA wird in Vergleichsoperationen wie FALSE gewertet.

Logische Vektoren werden durch Vergleichsoperationen mit Vektoren erstellt. Eine Vergleichsoperation erweitert die kürzeren Vektoren wieder auf den längsten verwendeten Vektor in der Operation. Logische Operatoren sind `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `|`. Rückgabe einer Vergleichsoperation mit Vektoren ist ein Vektor der Länge des längsten verwendeten Vektors. Der Inhalt des gelieferten Vektors einer Vergleichsoperation ist der boolesche Wert des Vergleichs jeder Elemente der Vektoroperanden bezüglich des verwendeten Vergleichsoperators.

Listing 15: ein logischer Vektor

```
1  c(-1, 0, 2, 4) > 0
2  # Ausgabe ist ein logischer Vektor mit [FALSE, FALSE, TRUE, TRUE]
```

In arithmetischen Operationen zwischen logischen und numerischen Vektoren wird bei TRUE mit 1 und bei FALSE mit 0 gerechnet.

### 7.1.2 Namensgebung

Man kann jedem Element eines Vektors Namen zuweisen. Dabei wird der Parameterliste der Konstruktor-Methode eine Zuweisung der Elementwerte auf Strings mit oder ohne Anführungszeichen angegeben. Die Strings benennen die Elemente des Vektors zusätzlich zu dem numerischen Index. Mittels Selektion kann dann über den numerischen Index oder den String zugegriffen werden. Bei der Zuweisung der Elementnamen muss keine Eindeutigkeit bestehen. Bei der Selektion der Elemente über die Elementnamen jedoch wird der erste gefundene Treffer gegeben, sollte es mehrfach zugewiesene Elementnamen geben.

Mit der Funktion *names(vektor)* können die Elementnamen eines Vektors ermittelt werden. An Hand des Funktionsaufrufs erkennt man, dass die Namen ein Attribut des Vektors ist. Der Wert des *names*-Attributes ist ein Vektor aus Strings mit den zugewiesenen Namen, wobei der Index dem zugewiesenen Element entspricht. Äquivalent zur Zuweisung der Namen in der Konstruktor-Methode kann dem *names*-Attribut eines Vektors ein String-Vektor zugewiesen werden, wobei es wichtig ist, dass die Länge dieses String-Vektors dem Vektor entspricht, dessen Elemente benannt werden sollen.

Listing 16: benannte Vektoren

```
1 x <- c("eins"=1, "zwei"=2, "drei"=3)
2 x <- c(eins=1, zwei=2, drei=3)
3
4 # äquivalent dazu:
5 x <- c(1, 2, 3)
6 names(x) <- c("eins", "zwei", "drei")
```

Die Namensgebung ist in anderen Basisklassen auf die gleiche Weise möglich.

### 7.1.3 Selektion

Es ist möglich Teile eines Vektors als Teilvektoren zu extrahieren. Die Elemente eines Vektors sind über Indizes in eckigen Klammern hinter dem Symbolnamen, der auf einen Vektor referenziert, selektierbar. Indizes innerhalb der eckigen Klammern können beliebige Vektoren oder elementare Typen enthalten. Positive Indizes selektieren Elemente unter den gegebenen Indizes. Liegt ein Index außerhalb des Vektors, ist der Index also numerisch und z.B. größer als die Länge des Vektors, wird NA zurückgegeben.

Listing 17: positives Selektieren in Vektoren

```
1 x <- c(1, 2, 3)
2 x[3]
3 # Ausgabe ist ein Vektor [3]
4 x[1:2]
5 # Ausgabe ist ein Vektor [1, 2]
```

Negative Indizes schließen Elemente aus der Selektion aus. Es wird also ein Vektor mit allen Elementen zurück gegeben, deren Index nicht negativ in der Selektion steht. Liegt der Index außerhalb des Vektors, wird ein Fehler ausgegeben. Fließkommazahlen im Index werden immer abgerundet als ganze Zahl interpretiert.

Listing 18: negatives Selektieren in Vektoren

```
1 x <- c(1, 2, 3)
2 x[-2]
3 # Ausgabe ist ein Vektor [1, 3]
4 x[-(2:3)]
5 # Ausgabe ist ein Vektor [1]
```

Logische Vektoren im Index selektieren Elemente, für die der Vergleich im Index TRUE ergibt. Der logische Vektor im Index wird wie in der Vektorarithmetik auf die Größe des Vektors erweitert. Ist der logische Vektor im Index größer, wird der Vektor mit NA-Werten aufgefüllt.

Listing 19: Selektieren durch logische Vektoren

```
1 x <- c(1, 2, 3)
2 x[x > 1]
3 # Ausgabe ist ein numerischer Vektor mit [2, 3]
```

Strings in einfachen eckigen Klammern selektieren das Element, dessen Name diesem String gleicht oder den gleichen Präfix hat, wenn es keinen exakten Treffer und es nur einen Präfix-Treffer gibt. Der String muss in Anführungszeichen angegeben werden. "[[" und "$" suchen nur nach exakten Treffern im *names*-Attribut eines Vektors.

Listing 20: Selektion in benannten Vektoren

```
1 x["z"]
2 # nimmt auch das Element mit Namen "zwei"
3 x[["zwei"]]
4 # nimmt nur das Element mit Namen "zwei"
```

Ein Aufruf mit leeren Klammern "[]" gibt den reinen Vektor ohne unnütze Attribute zurück, was einem Vektor alleine mit *dim*-Attribut entspricht.

Für den selektierten Teilvektor werden immer die relevanten *names*- und *dim*-Attribute kopiert.

## 7.2 weitere wichtige Klassen

### 7.2.1 Listen

Listen sind in etwa Vektoren, die allerdings verschiedenartige Elemente unterschiedlichen Typs enthalten können. In benannten Listen kann man auf Elemente über den "$"-Operator zugreifen.

Listing 21: Selektion in benannter Liste

```
1 list$name
```

### 7.2.2 Arrays

Arrays sind mehrdimensionale Vektoren. Man ordnet Vektoren in Arrays durch anhängen eines numeric Vektors als *dim*-Attributs an einen Vektor, wobei die Anzahl der Elemente hier mit der neuen Dimension übereinstimmen muss.

Listing 22: Erstellen eines Arrays aus einem Vektor

```
1 x.dim <- c(3, 5, 2)
```

Die Funktion *array(...)* erstellt ein leeres Array. Analog zum *names*-Attribut in Vektoren gibt es das Attribut *dimnames*, die jeder Dimension des Arrays einen eigenen Namen gibt. Die Daten im Array werden spaltenweise abgelegt. Der erste Index in einer Selektion eines zweidimensionalen Arrays entspricht also der Zeile. Eindimensionale Arrays ähneln und verhalten sich größtenteils wie Vektoren, geben aber andere Fehlermeldungen. Die Selektion verhält sich wie in Matrizen.

### 7.2.3 Matrizen

Matrizen sind zweidimensionale Arrays und können nur Zahlen enthalten. Zum Erstellen einer Matrix gibt es folgende Funktionen:

- *rbind(...)* erstellt eine Matrix mit zeilenweise vorgegebenen Werten.
- *cbind(...)* erstellt Matrizen spaltenweise.
- *matrix(0, dim)* erstellt eine Nullmatrix.

Auf Elemente in Matrizen und Arrays wird über Indizes in eckigen Klammern zugegriffen wie bei Vektoren. Dabei sind die Dimensionsindizes per Komma getrennt anzugeben. Negative Indizes sind nicht erlaubt. Dafür können Indizes leer gelassen werden, um Submatrizen oder Subsets einer Matrix oder eines Arrays zu erhalten. Das Ergebnis von *[index]* ist nicht äquivalent zu *[, index]*. Ein Index kann auch eine Matrix sein, die dann komplexe Subsets in dem Array selektiert.

### 7.2.4 Faktoren

Faktoren behandeln eine abzählbare Menge von Daten z.B. für Kategorie-Einstufungen von Daten in Vektoren.

### 7.2.5 Data Frames

Data Frames sind Listen aus Vektoren, Faktoren oder Matrizen der selben Länge bzw. Zeilenzahl. Damit kann man ein Objekt mit spaltenweise unterschiedlichen Datentypen realisieren.

### 7.2.6 Funktionen

Funktionen selbst sind auch Objekte und können in einer Session gespeichert oder in Objekten abgelegt werden. "*function(arglist) block*" erstellt ein Funktionsobjekt, wobei *arglist* die eingehende Parameterliste aus Symbolen mit oder ohne Initialwerte ist. Ein besonderer Parameter in der Parameterliste kann "..." sein, das beliebig viele Parameter als Listenobjekt an die Funktion übergibt. Die Parameterübergabe wird zunächst nach exakte Treffer der Parameternamen zugewiesen. Gibt es keinen exakten Treffer der Parameternamen in der Parameterliste wird nach einem Parameternamen mit dem Namen des übergebenen Arguments als Präfix gesucht. Gibt es mehrere Treffer auf diesen Präfix in der Parameterliste, wird ein Fehler ausgegeben. Alle weiteren Parameter werden zu der Liste der nicht benannten Argumente gezählt. Nicht benannten Argumente werden nach

ihrer Position den Parametern oder als Teil des "..."-Objekts zugeordnet. Ist dies nicht möglich, wird eine Fehlermeldung ausgegeben.

Der Funktionsname ohne Klammern liefert die Referenz auf das Funktionsobjekt.

# 8 Parser

Der Parser kümmert sich um die Umwandlung der Texteingaben in die interne Darstellung von Ausdrücken.

Der Parser wartet in der Kommandozeile auf einen vollständigen Ausdruck. Trifft er einen vollständigen Ausdruck beginnt der Parser mit seiner Arbeit. Die Ausdrücke können sich über mehrere Zeilen erstrecken. Nach Bestätigen der Eingabe, werden Ausdrücke in eine interne Form gewandelt, vom Evaluierer ausgewertet und das Ergebnis in der Kommandozeile ausgegeben. Geht der Parser in einen inkompatiblen Zustand, weil die eingegebene Syntax falsch war, wirft dieser einen Syntax Error und wartet auf eine neue Eingabe.

Beim Lesen von Anweisungen aus einer Datei mit *source(filename)*, wird die komplette Datei wie eine Eingabe über die Tastatur in eine interne Form gebracht und syntaktisch geprüft, bevor sie ausgewertet wird.

Die interne Form einer Anweisungen ist eine Baumstruktur. Symbole, Ausdrücke und jede Anweisung werden in eigene Objekte gespeichert. Infixe Schreibweisen von Operatoren werden in ihre jeweiligen Funktionsaufrufe mit den Operanden als Parameter umgewandelt.

Jedes Objekt kann durch *deparse(object)* in seinen vermeintlich ursprünglichen Ausdruck umgewandelt werden. Diese Umwandlung ist nicht immer umkehrbar, da es viele Objekte gibt, die aus unterschiedlichen Operationen entstehen können. Es wird lediglich versucht, einen Ausdruck zu erstellen, der ausgewertet das selbe Ergebnis liefert wie das ursprüngliche Objekt. Es wird also ein zu dem Objekt äquivalenter Ausdruck gebildet. Diese Funktion liefert vor allem bei Texten keine guten Ergebnisse.

Der Parser ignoriert den Rest einer Zeile sobald er einen Kommentar eingeleitet durch "`#`" außerhalb von Anführungszeichen findet.

Zunächst werden durch eine lexikalische Analyse atomare Anweisungen unterschieden. Atomare Anweisungen sind Konstanten, Variablennamen, reservierte Worte, spezielle und gängige Operatoren, Separatoren, Blöcke und Selektionen. Objekte werden über ihren Variablennamen mit der Funktion *get(varname)* erfragt und mit der Funktion *assign(varname, object)* gesetzt. Dabei werden eindeutige Operationen, Funktionsaufrufe und Variablennamen, die auch Sonderzeichen wie Klammern und andere Steuerzeichen enthalten dürfen, intern eindeutig von

einander unterschieden.

Nach der lexikalischen Analyse prüft eine syntaktische Analyse die Korrektheit der Anweisungen und Ausdrücke. Ausdrücke können also eine Baumstruktur aufspannen. Ausdrücke gehen von einfachen Zuweisungen bis hin zu komplexen Ausdrücken, die wiederum aus Ausdrücken bestehen. Sie bestehen aus Funktionsaufrufen, Operatoren, komplexe Selektionen durch Indizes, Ausdrucksblöcke, Kontrollstrukturen und Funktionsdefinitionen.

Funktionsdefinitionen werden intern als Funktionsobjekt mit Funktionsnamen, Argumenten als *tagged pairlist* und Funktionsblock gespeichert. Die Argumentliste weißt den Argumentnamen definierte Standardwerte zu. Argumentnamen dürfen dabei keine Sonderzeichen enthalten, dürfen also nur alphanumerische Zeichen enthalten.

Die Kontrollstrukturen *if*, *while*, *repeat*, *for*, *break* und *next* sind intern gleichnamige Funktionen, die die bekannten Programmierparadigmen bieten.

Die Selektion mittels Indizes wird durch die Funktionen "[" und "[[" ausgeführt, wobei die Indizes die Parameter der Funktionen sind.

## 8.1 Prioritäten der Operatoren

Die Priorität der Operatoren sinkt in folgender Aufstellung von oben nach unten.

| | |
|---|---|
| `::` | |
| `$, @` | |
| `^` | |
| `-, +` | (numerisch) |
| `:` | |
| `%xyz%` | (spezielle Operatoren) |
| `*, /` | |
| `+, -` | (binär) |
| `>, >=, <, <=, ==, !=` | |
| `!` | |
| `&, &&` | |
| `\|, \|\|` | |
| `~` | (Tilde, numerisch und binär) |
| `->, ->>` | |
| `=` | (Zuweisung) |
| `<-, <<-` | |

Spezielle Operatoren werden mit "%" umrahmt. Man kann eigene Operatoren durch diese Schreibweise und generische Funktionen definieren.

Der $-Operator ist nur streng genommen ein Operator. Er ruft die Funktion "*$*"

auf und bietet somit eine Kurzschreibweise, statt einer Operation.

# 9 Evaluation

Jedes Kommando in der Kommandozeile wird vom Parser in eine interne Form transferiert und das Ergebnis berechnet. Jeder vollständiger Ausdruck in der Kommandozeile wird durch *newline* oder ";" getrennt und sofort evaluiert. Es werden immer ganze Vektoren ausgewertet. Die meisten Operationen erfolgen für jedes Vektorelement separat. Dies ist besonders wichtig bei Operationen, die man aus der linearen Algebra kennt, wie Vektor-Addition oder -Multiplikation.

Jede Funktion erzeugt implizit eine eigene Umgebung. Eine Umgebung besteht aus einem Frame mit Symbol-Wert-Paaren und dessen übergeordneten Umgebung. Die Umgebungen spannen somit eine Baumstruktur auf. Die Funktion *parent.env(...)* greift auf die darüber liegende Umgebung. Umgebungen können nie kopiert werden. Ist in einer Umgebung ein Symbol zugeordnet, wird erst in dem eigenen Gültigkeitsbereich danach in dem Gültigkeitsbereich der übergeordneten Umgebungen nach diesem Symbol gesucht. Eine Variable aus der Argumentliste wird als *bound* bezeichnet, eine in der Funktion neu definierte Variable als *local*. Alle anderen Variablen sind *unbound* und einer übergeordneten Umgebung zugeordnet. Hier liegt ein Unterschied zwischen R von S in der Scoping-Regel, die vor allem durch *function*-Objekte sichtbar wird.

Listing 23: Unterschiede im Scoping zwischen R und S

```
1 f <- function() {
2   y <- 10
3   g <- function(x) x + y
4   return(g)
5 }
6 h <- f()
7 h(3)
```

In R: 13.
In S: Fehler!
*y* wurde nicht definiert, da die Funktion *g* zurück gegeben wurde und nun in der globalen Umgebung liegt anstatt in der Umgebung, die die Funktion *f* aufspannt hat. *y* wird in der globalen Umgebung gesucht und nicht in der Umgebung von Funktion *f*.

Im Gegensatz zu den meisten Funktionen, die in der globalen Umgebung *.GlobalEnv* definiert werden, liegen Funktionen in Paketen meist in eigenen Namespaces mit ihren eigenen Umgebungen.

Funktionsaufrufe in R haben eine unsichtbare lazy evaluation. Die zentrale Klasse für diesen Mechanismus ist die Promise-Klasse. Diese Klasse verknüpft einen nicht evaluierten Ausdruck, seinen evaluierten Wert und die aufrufende Funktion. Übergibt man einen Ausdruck an eine Funktion, wird dieser Ausdruck erst ausgewertet und in dem Promise-Objekt gespeichert, wenn das Argument das erste mal verwendet wird. Alle weiteren Operationen mit diesem Argument verwenden den gesicherten Wert. Das garantiert also nicht, dass ein Ausdruck als Parameter auch ausgewertet wird. Aus C übliche Kurzschreibweisen wie `foo(x = y+z)` sind in R nicht empfehlenswert. Eine Auswertung eines Promise-Objekts kann weitere nicht ausgewertete Promise-Objekte beeinflussen. Promise-Objekte sind unsichtbar und können nicht manipuliert werden.

Semantisch sind die Funktionsaufrufe in R call-by-value. Ein Ausdruck wird durch ein Promise-Objekt in der Umgebung der aufrufenden Funktion ausgewertet und dessen Wert dann in der aufgerufenen Funktion kopiert.

Scheinbar elementare Operationen wie `x + y` sind intern auch Funktionsaufrufe der Form `'x'(x, y)`. Der Parser expandiert erst die Operatoren rekursiv zu ihren äquivalenten Funktionen bevor er sie auswertet. Dies ermöglicht ein Überladen von Operatoren durch eigene Klassen.

Blöcke können in geschweifte Klammern erzeugt werden. Blöcke werden in der Kommandozeile erst evaluiert, wenn nach der schließenden Klammer ein *newline* folgt.

# 10 Klassensystem

Es gibt kein OOP in R. Stattdessen implementiert R ein Klassensystem.

Jedes Objekt hat ein *class*-Attribut, das eine hierarchische Liste aller abgeleiteten Klassennamen enthält. Dieses *class*-Attribut ist veränderbar und kann jedem Objekt zugewiesen werden. Methoden sind in R allein durch generische Funktionen realisiert, die je nach Werte in diesem *class*-Attribut reagieren. Dieses rudimentäre Klassensystem öffnet eine ausgefeilte Benutzung generischer Funktionen. Die Programmierung eigener Klassen ist kompliziert und wird schnell unübersichtlich.

Generische Funktionen arbeiten mit 4 Objekten in ihrer Umgebung (*.Generic*, *.Class*, *.Method*, *.Group*) und 2 delegierenden Funktionen (*UseMethod*, NextMethod).

Wird innerhalb einer generische Funktion die *UseMethod* aufgerufen, wird das *class*-Attribut des ersten Parameters im *.Class*-Objekt gesichert und in das *.Generic*-Objekt der Namen der generischen Funktion gesichert. Dann wird in der Reihenfolge der Klassen in der *class*-Liste die passende Funktion gesucht. Die zu einer Klasse passende Funktion enthält den Namen der Klasse in der Form "*gene-*

*ricFunctionName.ClassName*". Funktionen, die auf diese Weise definiert werden, sind Methoden. Wird keine Methode gefunden, wird die *default*-Methode aufgerufen. Gibt es keine *default*-Method wird ein Fehler ausgegeben. *UseMethod* übergibt vollständig die Kontrolle an die Methode, so dass nachfolgende Anweisungen nicht mehr aufgerufen werden. Die Methode erhält normalen Zugriff auf die Umgebung der generische Funktion, so dass Zuweisungen vor dem UseMethod gültig sind, die Nutzung solcher Zuweisungen der Lesbarkeit halber aber nicht empfohlen wird.

*NextMethod* kann überall in einer generischen Funktion auftauchen. *NextMethod* ruft die Methode der nächst übergeordneten Klasse im *.Class*-Objekt auf. Wichtig ist dabei, dass *NextMethod* nicht selbst das *.Class*-Objekt setzt, sondern es vor dem Aufruf von *NextMethod* gesetzt werden muss.

In *.Method* wird zuletzt der Name der gefunden Methode gespeichert.

Eine generische Funktion wird meist als 2-Zeiler definiert, dahinter werden die Methoden definiert:

Listing 24: Definition einer generische Funktion

```
1 generic <- function(x, ...)
2         UseMethod("generic")
3 generic.class1  <- function(args) ...
4 generic.class2  <- function(args) ...
```

Sei $x$ ein Objekt der Klasse *class1*, dann wird mit *generic(x)* die Methode *generic.class1(x)* aufgerufen. Sei nun $y$ ein Objekt der Klasse *myclass*, die abgeleitet wurde von der Klasse *class1*, wird durch *generic(y)* wieder *generic.class1(y)* aufgerufen. Die Methoden von *generic* können nicht direkt aufgerufen werden.

Um eigene Methoden zu schreiben, muss man also nur seinem Objekt einen eigenen Klassennamen dem *class*-Attribut anhängen und die generische Funktion um eine eigene Methode mit dem selben Namen erweitern.

Nach R-Definition haben die Funktionen *UseMethod* und *NextMethod* die 2 Argumente *generic* und *object*. Das Argument *generic* definiert den Namen der generischen Funktion und *object* das Objekt, dessen *class*-Attribut verwendet wird, um eine Methode zu finden. Wird kein *object* als Parameter angegeben wird automatisch das *class*-Attribut des ersten Parameters der generischen Funktion benutzt. In S kann es noch weitere Argumente geben, die Standardparameter an die Methoden übergibt. R ignoriert alle weiteren Argumente und gibt eine Warnung aus.

Das *.Group*-Objekt gibt es nur durch Aufrufe vordefinierter Funktionen im Kern. Man kann keine eigenen Gruppen definieren. Es gibt 3 vordefinierte Standardgruppen:

| Gruppe | Funktionen |
|---|---|
| Math | `abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cumsum, exp, floor, gamma, lgamma, log, log10, round, signif, sin, sinh, tan, tanh, trunc` |
| Summary | `all, any, max, min, prod, range, sum` |
| Ops | `+, -, *, /, ^, < , >, <=, >=, !=, ==, %%, %/%, &, \|, !` |

Die Gruppen sind durch Überladen der entsprechenden Methoden erweiterbar. Es können auch eigene Methoden in die Gruppen gesetzt werden.

In der Ops-Gruppe wird nach einer Methode gesucht, die zu den Klassen beider Operanden passt. Wird keine oder mehrere Methoden für beide Operanden gefunden, wird die *default*-Methode aufgerufen. In der Suche nach der Methode, wird erst nach der Klasse, dann nach der Gruppe gesucht.

# Literatur

[1] The Comprehensive R Archive Network, http://cran.r-project.org/.

[2] The R Project for Statistical Computing, http://www.r-project.org/.

[3] An Introduction to R, http://cran.r-project.org/doc/manuals/R-intro.html, ISBN 3-900051-12-7.

[4] R Language Definition, http://cran.r-project.org/doc/manuals/R-lang.html, ISBN 3-900051-12-7.

[5] Seite über die Anwendung von S, http://www.burns-stat.com/.

[6] An Introduction to the S Language, http://www.burns-stat.com/pages/Tutor/slanguage.html.

[7] A Guide for the Unwilling S User, http://www.burns-stat.com/pages/Tutor/unwilling_S.pdf.

# Listings