1 Programmieren "auf der Sprache"

R ist eine Programmiersprache, bei der die Elemente der Sprache selbst Gegenstand von Operationen in dieser Sprache sein können.

In R ist alles ein Objekt: Vektoren, Matrizen, Funktionen, das Ergebnis ausgeführter Funktionen.... Auch die Elemente der Sprache selbst sind Objekte. Auf Objekte können Funktionen angewendet werden, die auf diese Objekte zugeschnitten sind.

Objekt-Typen aus der Sprache

name (Name eines Objekts)

expression: unevaluierter R-Code

call (unevaluierter Funktionsaufruf)

Objekt-Typen, die im Zusammenhang mit Programmieren auf der Sprache interessieren:

character (z.B. Text irgendwelcher Kommandos)

list (z.B. Liste von Funktionsargumenten)

Funktionen, die auf Sprachobjekte angewendet werden können:

Funktion	Argument(e) vom Typ	Wirkung	Wert vom Typ
parse	Textfile oder Argument text=<character>	Umwandlung in gültigen R-Code	expression
eval	expression	Ausführung der expression	wie Wert der ausgeführten expression
deparse	expression	Umwandlung der weitmöglichst evaluierten expression in ihren Text	character
as.name	Character	Umwandlung in den Datentyp name	name
substitute oder quote[1]	Objekt	Umwandlung in den Name des Objekts	name
call	Funktionsname (character) + Funktionsargumente	Konstruktion eines Funktionsaufrufs	call
do.call	Funktionsname (character) + Argumentliste	Konstruktion eines calls und Aufruf von eval	wie Wert der ausgeführten Funktion
as.formula	character	Umwandlung in Formel	call

Hilfsfunktionen für die Verarbeitung von Text (character): paste, substring, nchar

Funktionsweise der Operationen auf der Sprache:

1.1 Umwandlung in Expression

```
> anweisung = parse(text="x=sqrt(5)")
> anweisung
expression(x = sqrt(5))
> mode(anweisung)
[1] "expression"
```

der Parameter text muss explizit angegeben werden, weil R sonst eine Eingabe aus einem File erwartet.

to parse = "grammatikalisch zerlegen"

1.2 Ausführen einer Expression

```
> eval(anweisung)
> x
[1] 2.236068
```

1.3 Funktionsaufruf

```
> aufruf = call("sqrt",5)
Funktionsname + beliebige Aufeinanderfolge von Argumenten
> aufruf
sqrt(5)
> mode(aufruf)
[1] "call"
```

1.4 Ausführen eines Funktionsaufrufs

```
> eval(aufruf)
[1] 2.236068
```

andere Möglichkeit zum Aufrufen von Funktionen:

```
Funktionsname + Liste von Argumenten
> do.call("sqrt",list(5))
[1] 2.236068
```

1.5

1.6 Definition einer Formel

```
> formel = y~x
> formel
y ~ x
> mode(formel)
[1] "call"
> as.formula("y~x")
y ~ x
```

1.7 Datentyp name für die Übergabe von Funktionsargumenten

```
> fuenf = 5
```

```
> eval(call("sqrt","fuenf"))
Error in sqrt("fuenf") : Non-numeric argument to mathematical function
> eval(call("sqrt",fuenf))
[1] 2.236068
> eval(call("sqrt",as.name("fuenf")))
[1] 2.236068
```

umständlich, aber manchmal nützlich: wenn die Objektnamen selbst in einer Funktion erzeugt werden

Zugriff auf das Objekt selbst aus seinem Namen:

```
> eval("fuenf")
[1] "fuenf"
eval(fuenf)
[1] 5
```

> as.name("fuenf")

fuenf

> as.name(fuenf)

5

1.8 Gewinnung des Namens eines Objekts als eigenes Objekt

```
> zahlen = 1:10
> substitute(zahlen)
zahlen
> mode(substitute(zahlen))
[1] "name"
```

identisch:

```
> quote(zahlen)
zahlen
```

Gewinnung des Namens als Character:

```
> deparse(substitute(zahlen))
[1] "zahlen"
```

nützlich für die Beschriftung von Plots und Output

Aber:

```
> deparse(zahlen)
[1] "c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)"
```

Obacht mit quote und substitute in Funktionen:

```
testfunktion = function(x)
{
print(quote(x))
print(substitute(x))
print(deparse(substitute(x)))
print(as.character(substitute(x)))
print(deparse(x))
}
```

```
> testfunktion(zahlen)
x
zahlen
[1] "zahlen"
[1] "zahlen"
[1] "c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)"
```

Für bloße Namen sind deparse(substitute()) und as.character(substitute()) äquivalent. Aber:

```
> deparse(substitute(sqrt(5)))
[1] "sqrt(5)"
> as.character(substitute(sqrt(5)))
[1] "sqrt" "5"
```

1.9 Beispiele

1.10 Beschriftung von Plots

```
plot.mit.beschriftung = function(x,y,...)
{
plot(x,y,xlab=deparse(substitute(x)),ylab=deparse(substitute(y)),...)
title(paste(deparse(substitute(y)),"vs.",deparse(substitute(x))))
}
```

1.11 Erzeugung mehrerer Plots (Steuerung von sich wiederholenden Vorgängen)

```
for(metall in names(test)[2:4])
{
win.graph()
eval(call("plot",as.name("jahr"),as.name(metall),type="l"))
}
```

Das gleiche "zu Fuß":

```
for(metall in names(test)[2:4])
{
win.graph()
eval(parse(text=paste("plot(jahr,",metall,",type='l')")))
}
```

beachte die einfachen Anführungszeichen!

1.12 Name in Großschreibung

```
Name = function(objekt)
{#---erzeugt Name des Objekts in Großschreibung
if(is.character(objekt)&length(objekt)==1) name = objekt
if(is.name(objekt)) name = as.character(objekt) else
if(!is.name(objekt)) name = as.character(substitute(objekt))
Name = paste(
LETTERS[letters==substring(name,1,1)],substring(name,2,nchar(name)),sep="")
Name
}
```

Anwendung:

```
for(metall in names(test)[2:4])
{
win.graph()
eval(call("plot",jahr,as.name(metall),type="l",
      xlab=Name(jahr),ylab=Name(as.name(metall))))
```

```
}
```

1.13 Anwendung der Namensfunktion auf Vektoren von Namen

```
> metalle = c("vanadium","chrom","nickel")
> Name(metalle)
[1] "Metalle"
> as.name(metalle)
vanadium
as.name funktioniert nicht für Vektoren
> sapply(metalle,as.name)
$vanadium
vanadium

$chrom
chrom

$nickel
nickel
komisch: Liste statt Vektor

> sapply(sapply(metalle,as.name),Name)
  vanadium      chrom     nickel
"Vanadium"    "Chrom"   "Nickel"
```

1.14 Anwendung: Plotten mehrerer Linien mit Legende

das ist die vektorisierte Form von Linienplots:

```
matplot(jahr,test[,c("vanadium","chrom","nickel")],type="l")

linienplot.mit.legende = function(x,y,...)
{#---Mehrlinien-Plot mit Legende
 #---y ist Matrix oder Data Frame mit nrow wie length(x)
y = data.frame(y)
matplot(x,y,xlab=Name(substitute(x)),ylab="",type="l",...)
cat("linken oberen Eckpunkt der Legende in Plotkoordinaten eingeben:\n")
cat("x-Koordinate:\n")
ecke.x = as.numeric(readline())
cat("y-Koordinate:\n")
ecke.y = as.numeric(readline())
legend(ecke.x,ecke.y,
      sapply(sapply(names(y),as.name),Name),lty=1:ncol(y),col=1:ncol(y))
}

2
3 Formel editieren und in Modell aufrufen
```

```
kandidaten = names(test)[-1]
varzahl = 3
formel = "arsen ~"
for(i in 1:varzahl)
```
formel = paste(formel,"+",kandidaten[i])
```
formel
```
[1] "arsen ~ aluminium + titan + vanadium"

as.formula(formel)
```
arsen ~ aluminium + titan + vanadium
```
äquivalent ist

formel = eval(parse(text=formel))

Ausreißer identifizieren und ausschließen

```
ausreisser = function(frame=test)
{
attach(frame)
cat("Ungleichung fuer Ausreisser eingeben:\n")
ungleichung = parse(text=readline())
logi = eval(ungleichung)
logi = logi & !is.na(logi)
zeile = row.names(frame)[logi]
cat(paste("Datensatz mit",ungleichung,":\n"))
print(frame[zeile,])
frame.bereinigt = test[!logi,]
frame.bereinigt
}
```

Interaktive Bildung von abgeleiteten Variablen

```
abgeleitete.variable = function(frame=test)
{
attach(frame)
cat("Formel fuer die Bildung der abgeleiteten Variablen eingeben: \n")
formel= readline()
transformation = parse(text=formel)
varneu = eval(transformation)
varneu
}
```

Literatur: W.N.Venables, B.D. Ripley (2000): S Programming. Springer

---

[1] siehe R Language Definition. Beachte dort auch match.call