V09 – Umgebungen

10. Mai 2021

Contents

1 Motivation

Welche Ausgaben erzeugen folgende Code-Beispiele?

```
# Beispiel 1
x <- 2
y <- 2
f1 <- function() {
  x <- 1
  c(x, y)
}
f1()
### Error, c(1, 1), c(1, 2), c(2, 2) ?
```

```r
# Beispiel 2
f2 <- function() {
  if (!exists("a")) { # Existiert eine Variable mit dem Namen a?
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
f2()
## 1
f2()
### Error, 1, 2, 3 ?
```

```r
# Beispiel 3
x <- 10
f3 <- function() x + 1
f3()
## 11
x <- 100
f3()
### Error, 1, 11, 101 ?
```

2 Grundbegriffe

Namensbindung (*name binding*): Mit `NAME <- WERT` wird der Wert `WERT` an den Namen `NAME` gebunden. In diesem Zusammenhang werden Namen auch als *Symbol* bezeichnet.

Der Gültigkeitsbereich einer Namensbindung wird auch mit **Scope** bezeichnet.

```r
x <- 5
f(x)
# hier endet der Scope der Namensbindung von x mit 5
# es beginnt der Scope von x gebunden mit 6
x <- 6
g(x)
```

Die **Auflösung** eines Namens (*name resolution*) bezeichnet das Ersetzen eines Namens mit dem gebundenen Wert.

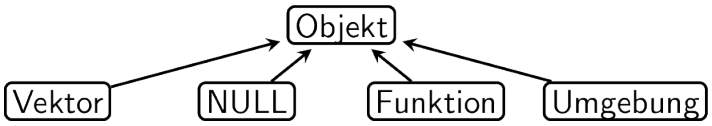
Scoping-Regeln steuern die Auflösung von Namen. Die Scoping-Regeln von R basieren auf **Umgebungen** (*environments*), einer weiteren Datenstruktur.

Jeder Namensbindung ist genau eine Umgebung zugeordnet. Umgebungen enthalten eine beliebige Anzahl an Namensbindungen.

3 Umgebungsobjekte

In diesem Kapitel nutzen wir das Paket `rlang`, welches unter anderem einige Funktionen für den Umgang mit Umgebungen bereitstellt.

```r
# install.packages("rlang")
library(rlang)
```



Umgebungen (*environments*) sind Objekte, die Listen mit benannten Elementen ähneln, jedoch 4 Unterschiede aufweisen:

- Jeder Name darf nur einmal auftreten.
- Elemente sind nicht geordnet (kein "i-ter Eintrag").
- Jede Umgebungen (außer der leeren Umgebung) hat eine Eltern-Umgebung (*parent environment*).
- Umgebungen werden nicht kopiert, wenn sie geändert werden (kein *copy-on-modify* sondern *Referenz-Semantik*).

Mit `rlang::env()` wird eine neue Umgebung analog zu `list()` erzeugt.

```
l1 <- list(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3
)
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3
)
```

Umgebungen haben Typ und Klasse `environment`. `length()` gibt die Anzahl der Elemente zurück.

```
c(typeof(e1), class(e1))
## [1] "environment" "environment"
length(e1)
## [1] 4
```

Für Umgebungen können die Subsetting-Operatoren `[[` und `$` genutzt werden, `[[` jedoch nur mit Strings nicht mit Zahlen, da die Elemente einer Umgebung keine Ordnung haben.

Da Umgebungen anders als etwa Listen einer *Referenz-Semantik* folgen, können zwei Namen die gleiche Umgebung als Wert haben.

```
e1$d
## [1] 1 2 3
e1other <- e1
# e1other ist keine Kopie von e1, sondern eine 'Referenz' auf das selbe Objekt
e1other$d <- 0 # Änderung von e1other bedeutet Änderung des referenzierten Objektes
e1$d # Daher ist auch das von e1 referenziert Objekt geändert
## [1] 0

# vergleiche
l1$d
## [1] 1 2 3
l1other <- l1
l1other$d <- 0
l1$d
## [1] 1 2 3
```

Gleichheit von Umgebungen wird mit `identical()` getestet, nicht mit `==`.

```
e1 == e1other
## Error in e1 == e1other: comparison (1) is possible only for atomic and list types
identical(e1,e1other)
## [1] TRUE
```

Wegen der Referenz-Semantik können Umgebungen sich selbst (bzw eine Referenz auf sich selbst) enthalten. Dies ist nicht möglich für Listen.

```
e1$d <- e1
identical(e1$d$d$d$d$d$d, e1)
## [1] TRUE
l1$d <- l1
l1$d$d
## [1] 1 2 3
```

Da `print()` für Umgebungen nicht besonders informativ ist, nutzen wir oft `rlang::env_print()`.

```
e1 # gleich print(e1)
## <environment: 0x00000000015353b08>
env_print(e1)
## <environment: 0000000015353B08>
## parent: <environment: global>
## bindings:
##  * a: <lgl>
##  * b: <chr>
##  * c: <dbl>
##  * d: <env>
```

`rlang::env_names()` gibt die Namen zurück, die eine Umgebung enthält.

```
env_names(e1)
## [1] "a" "b" "c" "d"
```

Die meisten Umgebungen werden nicht vom Nutzer erstellt, sondern von R selbst.

`rlang::empty_env()` gibt die **leere Umgebung** zurück.

```
env_names(empty_env())
## character(0)
```

`rlang::global_env()` gibt die **globale Umgebung**. Die globale Umgebung speichert die Namensbindungen, die in der Konsole erzeugt werden.

```
env_names(global_env())
## [1] "l1"           "e1"           "l1other"      ".Random.seed" "e1other"
```

`rlang::env_label()` gibt das Label einer Umgebung aus. Für spezielle Umgebungen ist das Label ein spezifischer Name, für alle anderen ein *address code*.

```
env_label(empty_env())
## [1] "empty"
env_label(global_env())
## [1] "global"
env_label(e1)
## [1] "0000000015353B08"
```

`rlang::current_env()` ist die **aktuelle Umgebung**, in der die aktuelle Code-Ausführung stattfindet.

Im Gegensatz zur leeren und globalen ist dies keine feste Umgebung. In der Konsole ist `rlang::global_env()` gleich `rlang::current_env()`, im Inneren von Funktionen nicht.

```
env_label(current_env())
## [1] "global"
f <- function() env_label(current_env())
f()
## [1] "000000001C625EB8"
```

Stößt R bei der Code-Ausführung auf einen Variablennamen, muss dieser *aufgelöst* werden, um den zugehörigen Wert zu finden. Dazu wird der Name zuerst in der aktuellen Umgebung gesucht. Falls er dort nicht gefunden wird, wird in der Eltern-Umgebung der aktuellen gesucht. Dies wird wiederholt, bis der Name (und damit der zugehörige Wert) in einer Umgebung gefunden wird (ansonsten ERROR).

Jede Umgebung außer der leeren hat genau eine Eltern-Umgebung. Die Eltern-Umgebung kann explizit mit `rlang::env()` bei der Erzeugung einer Umgebung gesetzt werden. Der Default-Wert ist `rlang::current_env()`.

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```

`rlang::env_parent()` erlaubt den Zugriff auf die Eltern-Umgebung.

```
env_label(e2a)
## [1] "000000001C8813B0"
env_label(env_parent(e2b))
## [1] "000000001C8813B0"
env_label(env_parent(e2a))
## [1] "global"
env_parent(empty_env())
## Error: The empty environment has no parent
```

`rlang::env_parents()` gibt eine Liste aller "Vorfahren" einer Umgebung bis zur globalen oder leeren Umgebung zurück.

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
env_parents(e2b)
## [[1]]   <env: 000000001C8813B0>
## [[2]] $ <env: global>
env_parents(e2d)
## [[1]]   <env: 000000001D1ADFF8>
## [[2]] $ <env: empty>
```

Um Namensbindungen von Umgebungen zu verwalten, sind die Subsetting Operatoren `[[`, `[[<-`, `$`, `$<-` und die folgende Funktionen aus dem Paket `rlang` von Nutzen: `env_get()`, `env_get_list()`, `env_bind()`, `env_has()`, `env_unbind()`.

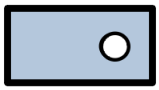
```
e <- empty_env()
e$x <- 1
## Error in e$x <- 1: cannot assign values in the empty environment
e <- env() # erstelle neue Umgebung mit der globalen Umgebung als parent
e$x <- 1
e[["y"]] <- "one"
env_bind(e, z=T)
e$y
## [1] "one"
e[["z"]]
## [1] TRUE
env_get(e, "x")
```

```
## [1] 1
str(env_get_list(e, c("x", "y", "z")))
## List of 3
##  $ x: num 1
##  $ y: chr "one"
##  $ z: logi TRUE
env_has(e, c("w", "x", "y", "z"))
##     w     x     y     z
## FALSE  TRUE  TRUE  TRUE
env_unbind(e, c("x", "z"))
env_has(e, c("w", "x", "y", "z"))
##     w     x     y     z
## FALSE FALSE  TRUE FALSE
```

4 Umgebungsdiagramme

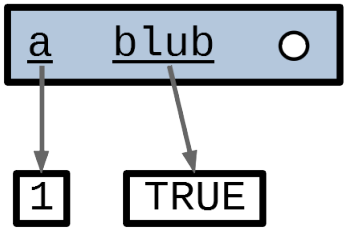
Wir visualisieren die Struktur der Umgebungen und ihrer Elemente mit Umgebungsdiagrammen.

Eine Umgebung wird als Rechteck mit einem Kreis im Inneren dargestellt. Der Kreis steht für die Elternumgebung.



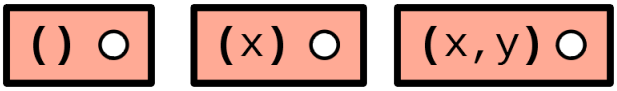
Namen der Umgebung stehen im Inneren des Rechtecks und sind unterstrichen. Von ihnen gehen Pfeile auf Rechtecke außerhalb des Umgebungsrechtecks aus. In diesen äußeren Rechtecken steht der Wert der entsprechenden Namensbindung.

```
env(a = 1, blub = TRUE)
```



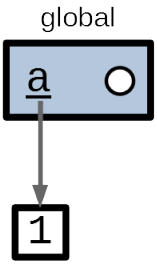
Funktionsobjekte werden dargestellt als Rechteck mit einem Kreis und `(`, `)` im Inneren. Argumentnamen stehen ggf durch `,` getrennt zwischen den Klammern.

```
function() {
  # ...
}
function(x) {
  # ...
}
function(x, y) {
  # ...
}
```



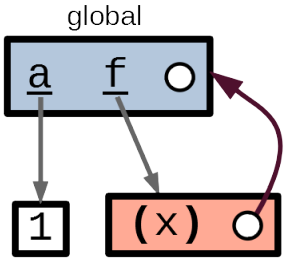
Die globale Umgebung ist mit dem Wort “global” überschrieben.

```
a <- 1 # Namensbindung in der globalen Umgebung
```



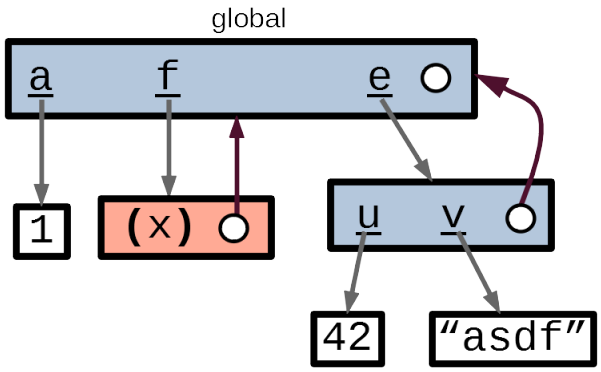
Vom Kreis einer Funktion geht ein Pfeil zu der Umgebung, in der die Funktion erstellt wurde (die sogenannte **Funktionsumgebung** dieser Funktion).

```
a <- 1
f <- function(x) {
  # ...
}
```



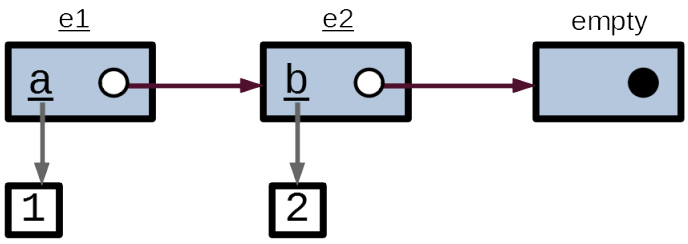
Vom Kreis einer Umgebung geht ein Pfeil zu ihrer Elternumgebung. Bei der globalen Umgebung lassen wir diesen Pfeil meist weg.

```
a <- 1
f <- function(x) {
  # ...
}
e <- env(u = 42, v = "asdf") # default parent is global env
```



Die leere Umgebung ist mit dem Wort “empty” überschrieben. Da sie keine Elternumgebung hat, ist ihr Kreis ausgefüllt. Wenn die globale Umgebung nicht wichtig ist, lassen wir sie weg und schreiben ggf über wichtige Umgebungen ihren unterstrichenen Variablennamen.

```
e1 <- env(empty_env(), b = 2)
e2 <- env(e1, a = 1)
```



5 Umgebungen für Funktionen

Bei der Erstellung und Ausführung von Funktionen sind drei Umgebungen wichtig:

- die **Funktionsumgebung** (*function environment*)
- die **Ausführungsumgebung** (*execution environment*)
- die **aufrufende Umgebung** (*caller environment*)

Sie steuern Scoping für Funktionen. Dh, diese Umgebungen legen fest, wie Namen von Variablen im Inneren einer Funktion aufgelöst werden.

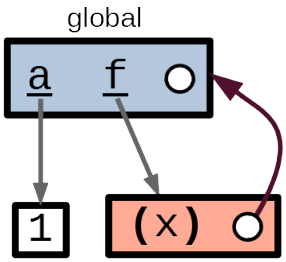
5.1 Funktionsumgebung

`rlang::fn_env()` gibt die **Funktionsumgebung** an, welche bei der Erstellung einer Funktion auf die zu diesem Zeitpunkt aktuelle Umgebung gesetzt wird. Sie ist Teil des Funktionsobjektes (außer bei primitiven Funktionen).

Beispiel 1:

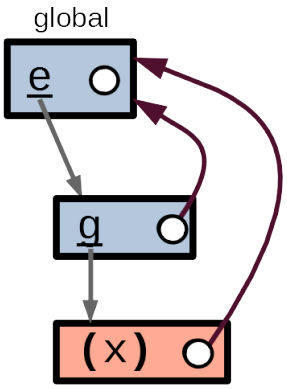
```
a <- 1
f <- function(x) x + a
env_label(current_env())
## [1] "global"
env_label(fn_env(f))
## [1] "global"
```

`f` ist ein Name in der globalen Umgebung. An diesen Name ist ein Wert gebunden. Der Wert ist ein Funktionsobjekt. Das Funktionsobjekt hat eine Funktionsumgebung, nämlich die globale Umgebung.



Beispiel 2:

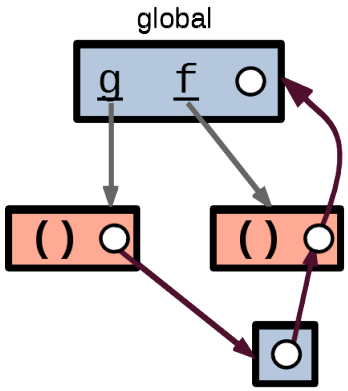
```
e <- env() # erzeuge Umgebung mit globaler Umgebung als Eltern-Umgebung
e$g <- function(x) x+2
```



Beispiel 3:

Wird eine Funktion `g()` in einer anderen Funktion `f()` erzeugt, ist die Funktionsumgebung von `g()` gleich der sogenannten *Ausführungsumgebung* beim Aufruf von `f()`.

```
f <- function() {
  print(env_label(current_env()))
  function(){} # Rückgabewert: ein Funktionsobjekt mit leerem body()
}
g <- f() # print current env inside f
## [1] "0000000015614678"
env_label(fn_env(g)) # function env of g equals current env inside f
## [1] "0000000015614678"
```



5.2 Ausführungsumgebung

Wird eine Funktion aufgerufen, wird eine neue Umgebung erstellt: eine **Ausführungsumgebung** (*execution environment*). In ihr leben die Argumente der Funktion und die Namen, die im Inneren der Funktion gebunden werden.

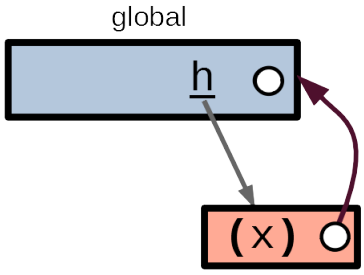
Die Eltern-Umgebung einer Ausführungsumgebung ist die Funktionsumgebung.

Bei jedem Funktionsaufruf wird eine neue Ausführungsumgebung erstellt.

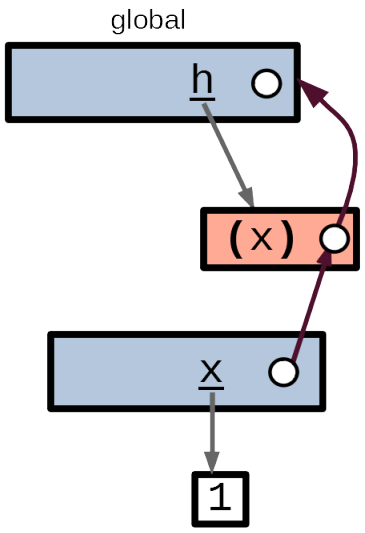
Illustrieren wir den Vorgang des Funktionsaufrufs am Beispiel der Funktion `h()`.

```
h <- function(x) {
  # 2.
  a <- 2
  # 3.
  x + a
}
# 1.
y <- h(1)
# 4.
```

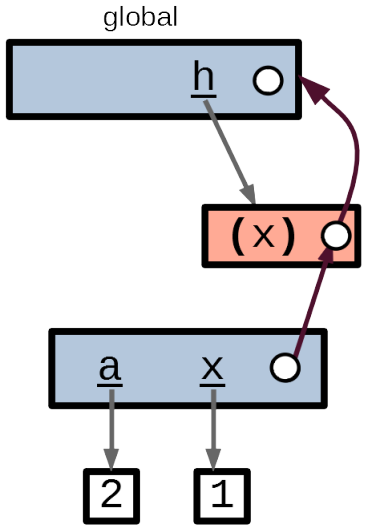
Umgebungsdiagramm bei `# 1.`:



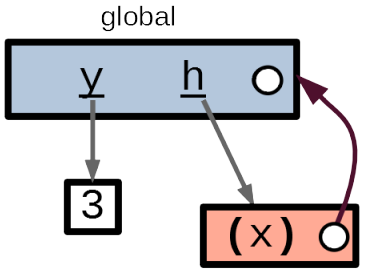
Umgebungsdiagramm bei `# 2.`: Die Ausführungsumgebung entsteht. In ihr ist der übergebene Wert `1` and den Namen des zugehörigen Argumentes `x` gebunden.



Umgebungsdiagramm bei `# 3.`:



Umgebungsdiagramm bei `# 4.`: Die Ausführungsumgebung verschwindet.



Hinweis: Wir machen hier noch eine kleine Vereinfachung bzgl wann genau die Namensbindung zwischen `x` und `1` entsteht, siehe letzter Abschnitt dieses Kapitels (*Lazy Evaluation 2*).

Normalerweise wird die Ausführungsumgebung gelöscht, sobald die Funktionsausführung beendet ist.

Sind jedoch andere, weiter existierende Objekte von einer Ausführungsumgebung abhängig, bleibt sie weiter bestehen.

Zum Beispiel kann die Ausführungsumgebung von der Funktion zurückgegeben und dann weiter verwendet werden.

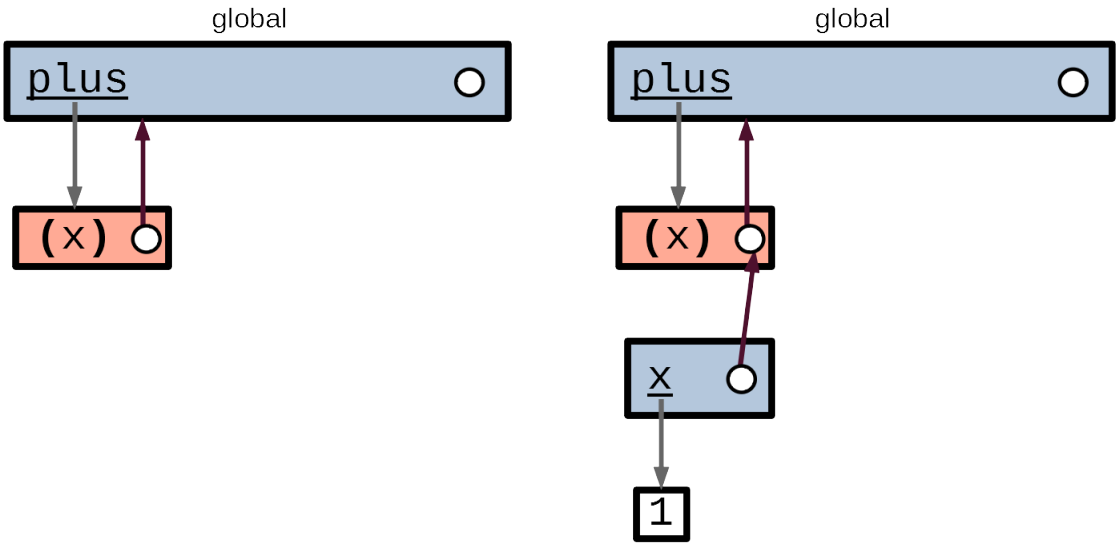
```
h <- function(x) {
  a <- x * 2
  current_env()
}
e <- h(10)
env_print(e)
## <environment: 000000001CA77CA0>
## parent: <environment: global>
## bindings:
##  * a: <dbl>
##  * x: <dbl>
env_label(fn_env(h))
## [1] "global"
```

Zu einer Funktion können daher zur selben Zeit mehrere Ausführungsumgebungen (von verschiedenen Funktionsaufrufen) existieren.

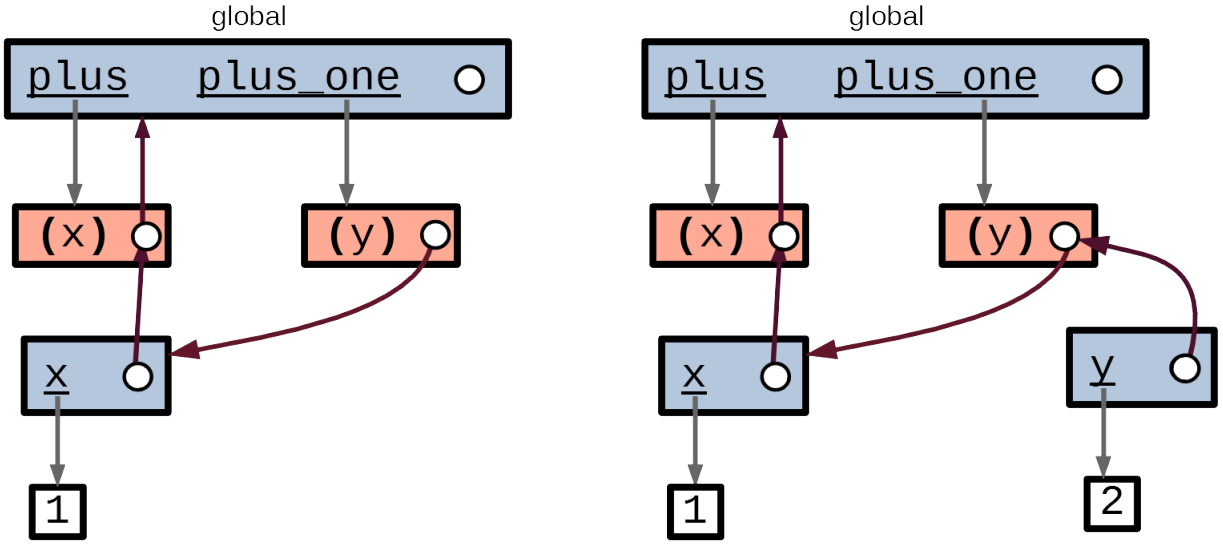
Wird eine Funktion im Innern einer anderen Funktion kreiert und zurückgegeben, wird die Ausführungsumgebung der äußeren Funktion ebenfalls nicht gelöscht, da diese als Funktionsumgebung der inneren Funktion dient.

```
plus <- function(x) {
  # 2.
  function(y) {
    # 4.
    x + y
  }
}
# 1.
plus_one <- plus(1)
# 3.
plus_one(2)
## [1] 3
```

Umgebungsdiagramm bei `# 1.` und `# 2.`



Umgebungsdiagramm bei `# 3.` und `# 4.`



Bei der Berechnung von `x + y` wird `y` direkt in einer Ausführungsumgebung von `plus_one()` gefunden. `x` wird dort nicht gefunden. Also wird in der Elternumgebung gesucht. Dies ist eine Ausführungsumgebung von `plus()`. Sie enthält den Namen `x`.

Hinweis: `rlang::env_parent()` ist keine Ersetzungsfunktion. Das Base-R Äquivalent `parent.env()` schon.

```
library(rlang)
f <- function() x
e1 <- env()
fn_env(f) <- e1
e2 <- env(e1)
parent.env(e1) <- e2 # evil
f()
```

5.3 Aufrufende Umgebung

Die Umgebung, aus der heraus eine Funktion aufgerufen wird, heißt **aufrufende Umgebung** (*caller environment*). Sie wird von `caller_env()` zurückgegeben.

```
library(stringr)
f <- function() {
  cat(str_c("f: ",
    "current: ", env_label(current_env()), ", ",
    "caller: ", env_label(caller_env()), "\n"))
  g()
}
g <- function() {
  cat(str_c("g: ",
    "current: ", env_label(current_env()), ", ",
    "caller: ", env_label(caller_env()), "\n"))
  h()
}
h <- function() {
  cat(str_c("h: ",
    "current: ", env_label(current_env()), ", ",
    "caller: ", env_label(caller_env()), "\n"))
}
f()
## f: current: 000000001DB36CA0, caller: global
## g: current: 000000001DB3AED0, caller: 000000001DB36CA0
## h: current: 000000001DB3BC38, caller: 000000001DB3AED0
```

Bei einem Funktionsaufruf werden Argumente in der aufrufenden Umgebung ausgewertet (nicht in Auführungs- oder Funktionsumgebung).

```
a <- 1 # a in aufrufender Umgebung
f <- function(x) {
  a <- 2 # a in Ausführungsumgebung
  x + 10
}
e <- env(a = 3) # a in Funktionsumgebung
fn_env(f) <- e
f(a)
## [1] 11
```

6 Scoping-Prinzipien

Die Auflösung von Namen in R kann durch drei Prinzipien beschrieben werden: *Name-Masking*, *Neuanfang*, *dynamic Lookup*.

6.1 Name-Masking

Zum Auflösen eines Namens wird dieser zuerst in der aktuellen Umgebung gesucht. Wird der Name dort nicht gefunden, werden nacheinander die Vorfahren (Elternumgebung, Elternumgebung der Elternumgebung, ...) der aktuellen Umgebung durchsucht.

Ein Name im Inneren einer Funktion wird zuerst in der Ausführungsumgebung, dann in der Funktionsumgebung (bei Erzeugung der Funktion aktuelle Umgebung) gesucht. Dadurch werden ggf Namen in den Vorfahren überdeckt (**Name-Masking**).

```r
# Beispiel 1
x <- 10
y <- 20
f1 <- function() {
  x <- 1
  c(x, y)
}
f1()
## [1]  1 20
```

Funktionen sind normale Objekte. Dementsprechend gilt *Name-Masking* prinzipiell auch für Funktions-Variablen.

```r
f <- function(x) x + 1
g <- function() {
  f <- function(x) x + 100
  f(10)
}
g()
## [1] 110
```

Bei einem Funktionsaufruf wird jedoch explizit nach einem Funktionsobjekt gesucht. Nicht-Funktions-Objekte überdecken keine Funktionsobjekte.

```r
f <- function(x) x + 1
g <- function() {
  f <- 100
  f(10)
}
g()
## [1] 11

c <- 3
c(c, 1)
## [1] 3 1
```

6.2 Dynamic Lookup

Die Auflösung der Namen geschieht erst dann, wenn die Werte wirklich benötigt werden. Dh, die beim Funktionsaufruf aktuellen Werte (nicht bei der Funktionserzeugung) werden benutzt (**dynamic Lookup**).

```r
# Beispiel 3
x <- 10
f3 <- function() x + 1
f3()
## [1] 11
x <- 100
f3()
## [1] 101
```

Mit `codetools::findGlobals()` können die externen Abhängigkeiten einer Funktion gefunden werden.

```r
f <- function() x + 1
codetools::findGlobals(f)
## [1] "+" "x"
```

Um jegliche Abhängigkeit zu unterbinden, könnte man die Funktionsumgebung auf die leere Umgebung

setzen. Dies ist aber in der Regel nicht sinnvoll.

```
fn_env(f) <- empty_env()
f() # ERROR
## Error in x + 1: could not find function "+"
```

Dynamic Lookup ermöglicht Rekursion.

```
infinite_recursion <- function() infinite_recursion()
infinite_recursion() # evil
```

6.3 Neuanfang

Die Ausführungsumgebung wird bei jedem Aufruf neu erzeugt. Alte Werte werden nicht übertragen (**Neuanfang**).

```
# Beispiel 2
f2 <- function() {
  if (!env_has(current_env(), "a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
f2()
## [1] 1
f2()
## [1] 1
```

Achtung: Namen werden nicht in der aufrufenden Umgebung gesucht (außer sie ist ein Vorfahre der Ausführungsumgebung). Jedoch entstammen die Werte der Argumente einer Funktion aus der aufrufenden Umgebung.

```
x <- 4
y <- 4
z <- 4
create_function <- function() {
  y <- 2 # Ausführungsumgebung von create_function() und Funktionsumgebung von
  function() {
    x <- 1 # Ausführungsumgebung von f()
    c(x=x, y=y, z=z)
  }
}
call_function <- function(fun) {
  x <- 3
  y <- 3
  z <- 3
  print(fun())
}
f <- create_function()
call_function(f)
## x y z
## 1 2 4
z <- 5
call_function(f)
```

```
## x y z
## 1 2 5
```

7 Zuweisungsoperatoren

Mittels `?`<-`` erfahren wir, dass es insgesamt 5 Zuweisungsoperatoren gibt: `<-`, `<<-`, `->`, `->>`, `=`.

Bei der Zuweisung `x <- value`, `value -> x`, `x = value` wird eine Namensbindung zwischen dem Namen `x` und dem Wert `value` in der aktuellen Umgebung erzeugt.

```
# äquivalent sind:
x <- 42
x = 42
42 -> x
env_bind(current_env(), x = 42)
```

Die Operatoren sind Funktionen mit dem Rückgabewert `value`. `<-` und `=` werden wie `^` und im Gegensatz zu den meisten anderen Operatoren von rechts nach links gruppiert (Ausführungsreihenfolge), weswegen `x <- y <- value` sowohl `x` als auch `y` den Wert `value` zuweist.

`=` ist größtenteils synonym zu `<-`. Allerdings hat `=` noch eine zusätzlich Bedeutung bei der Benennung von Funktionsargumenten. Beachte den Unterschied zwischen `f(y <- 1)` und `f(y = 1)`.

```
f <- function(x=0, y=0) print(c(x,y))
y <- 1
f(y = 2)
## [1] 0 2
y
## [1] 1
f(y <- 2)
## [1] 2 0
y
## [1] 2
```

Achtung: Was bedeutet `x<-2`? `x <- 2` oder `x < -2`? Guter Stil: Setzte immer Leerzeichen um Zuweisungsoperatoren. In RStudio werden bei der Nutzung von `Alt + -` Leerzeichen automatisch gesetzt.

Der Zuweisungsoperator `<-` erzeugt (oder ändert) eine Namensbindung in der aktuellen Umgebung `current_env()`, wohingegen `x <<- 5` (oder `5 ->> x`) nach dem Namen `x` in der Eltern-Umgebung der aktuellen Umgebung sucht. Wird er dort nicht gefunden, werden die weiteren Vorfahren durchsucht.

Wird der Name `x` in einer der Vorfahren-Umgebungen gefunden, wird (in der zuerst gefunden Namensbindung) der Wert entsprechend geändert (hier: auf `5` gesetzt).

Wird der Name nicht gefunden, wird eine neue Namensbindung in der globalen Umgebung angelegt.

```
cnt1 <- 0
count1 <- function() {
  cnt1 <- cnt1 + 1
}
count1()
count1()
cnt1
## [1] 0
cnt2 <- 0
count2 <- function() {
  cnt2 <<- cnt2 + 1
}
```

```
count2()
count2()
cnt2
## [1] 2
```

8 Rekursion über Umgebungen

Wird ein Variablenname in einer Umgebung nicht gefunden, wird in deren Eltern-Umgebung gesucht, wobei jede Umgebung außer der leere Umgebung (`rlang::empty_env()`) genau eine Eltern-Umgebung hat. Ausgestattet mit diesem Wissen, können wir selbst Funktionen schreiben, die uns die Umgebung ausgeben, in der ein gegebener Name gebunden ist.

```
where <- function(name, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    env
  } else {
    where(name, env_parent(env))
  }
}
x <- 2
y <- 3
env_label(where("x"))
## [1] "global"
f <- function() {
  x <- 1
  cat("x:", env_label(where("x")), "\n")
  cat("y:", env_label(where("y")), "\n")
}
f()
## x: 000000001CA5D840
## y: global
env_label(where("sd"))
## [1] "package:stats"
```

Die Folge von Umgebungen, in denen ein Name gesucht werden würde, heißt **Suchpfad** (*search path*).

```
searchpath_labels <- function(env = caller_env()) {
  lbl <- env_label(env)
  if (identical(env, empty_env())) {
    return(lbl)
  } else {
    return(c(lbl, searchpath_labels(env_parent(env))))
  }
}
searchpath_labels()
##  [1] "global"           "package:stringr"   "package:rlang"
##  [4] "package:stats"    "package:graphics"  "package:grDevices"
##  [7] "package:utils"    "package:datasets"  "package:methods"
## [10] "Autoloads"        "package:base"      "empty"
f <- function() searchpath_labels()
f()
##  [1] "000000001D3E6F28"  "global"           "package:stringr"
```

```
##  [4] "package:rlang"     "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"
## [13] "empty"
```

Der Suchpfad, ausgehend von der globalen Umgebung, enthält Umgebungen einiger Pakete und endet mit der leeren Umgebung.

Da es nur eine leere Umgebung gibt und diese die einzige ohne Eltern-Umgebung ist, gilt: Alle existierenden Umgebungen bilden mit der Verbindung durch die "Eltern-Relation" einen Baum mit Wurzel `empty_env()`.

9 Umgebungen für Pakete

9.1 Suchpfad

Wird ein Paket mit `library()` geladen, wird es zu einer Vorfahren-Umgebung der globalen Umgebung. Die direkte Eltern-Umgebung der globalen Umgebung ist das zuletzt geladene Paket. Dessen Eltern-Umgebung ist wiederum das zuvor geladene Paket.

```
library(magrittr) # to use pipe %>%
##
## Attaching package: 'magrittr'
## The following object is masked from 'package:rlang':
##
##     set_names
global_env() %>% env_label()
## [1] "global"
global_env() %>% env_parent() %>% env_label()
## [1] "package:magrittr"
global_env() %>% env_parent() %>% env_parent() %>% env_label()
## [1] "package:stringr"
```

Geht man weiter zurück, findet man alle Pakete in der Reihenfolge, in der sie geladen wurden. Dies entspricht dem Suchpfad. Alle Objekte, die von der Konsole aus aufgerufen werden können, sind in einem der Umgebungen des Suchpfades zu finden.

```
searchpath_labels()
##  [1] "global"            "package:magrittr"  "package:stringr"
##  [4] "package:rlang"     "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"
## [13] "empty"
```

Die letzte Umgebung vor der leeren ist immer `package:base`, die auch mittels `rlang::base_env()` abrufbar ist. Sie enthält elementare Bausteine von R. Die Eltern-Umgebung von `package:base` ist die leere Umgebung.

```
head(env_names(base_env()), 20)
##  [1] "!"                    "$"                    "&"
##  [4] "("                    "*"                    "+"
##  [7] "-"                    "/"                    ":"
## [10] "<"                    "="                    ">"
## [13] "as.matrix.data.frame" "@"                    "F"
## [16] "I"                    "registerS3methods"    "as.POSIXct.Date"
## [19] "T"                    "["
```

Die Sammlung von Pakete, die beim Start von R automatisch geladen werden, nennen wir **Base-R**: `stats`,

`graphics`, `grDevices`, `utils`, `datasets`, `methods`, `base`.

`Autoloads` taucht ebenfalls im Suchpfad auf, erlaubt On-Demand-Laden von Paketen und kann ignoriert werden.

9.2 Namespace

Falls Funktionen aus einem Paket Funktionen eines anderen Paketes aufrufen, scheint unsere Beschreibung des Suchpfades zu suggerieren, dass die richtige Auflösung der Funktionsnamen von der Ladereihenfolge der Pakete abhängt. Namespaces sorgen dafür, dass diese Funktionen immer richtig funktionieren.

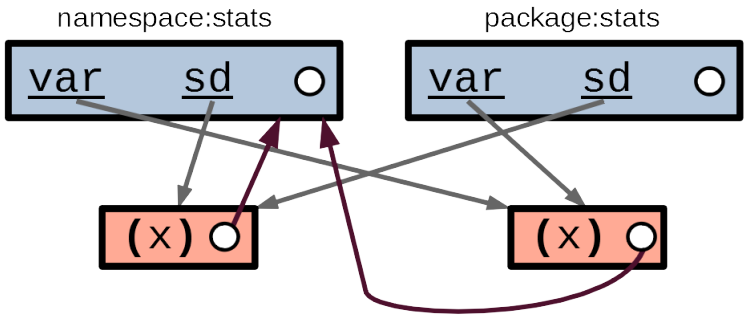
Die Funktion `sd()` (aus dem Paket `stats`) ist mittels `var()` definiert.

```
sd
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x0000000015539478>
## <environment: namespace:stats>
```

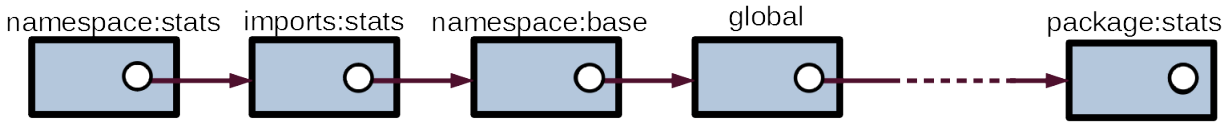
Angenommen wir selbst haben eine Funktion `var` definiert. Wie findet `sd()` die richtige `var()`-Funktion?

```
var <- function(x) 0
sd(1:5) # nutzt offensichtlich nicht unsere Definition von var()
## [1] 1.581139
```

Jede Funktion eines Paketes ist sowohl in einer Namespace- als auch in einer Paket-Umgebung gebunden. Die Funktionsumgebung der Funktion ist gleich der Namespace-Umgebung.

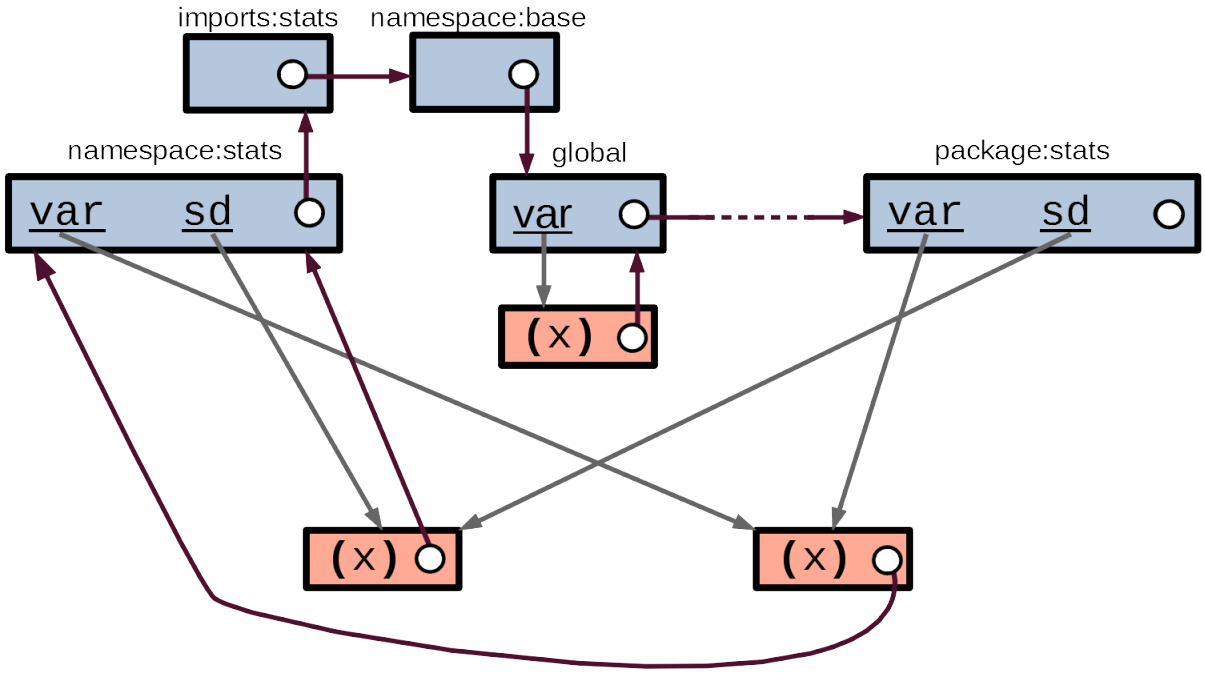


Die Namespace-Umgebung ist (über zwei andere Umgebungen) Nachkomme der globalen Umgebung. Die Paket-Umgebung ist Vorfahre der globalen Umgebung.




```
searchpath_labels(fn_env(sd))
##  [1] "namespace:stats"   "imports:stats"     "namespace:base"
##  [4] "global"            "package:magrittr"  "package:stringr"
##  [7] "package:rlang"     "package:stats"     "package:graphics"
## [10] "package:grDevices" "package:utils"     "package:datasets"
## [13] "package:methods"   "Autoloads"         "package:base"
## [16] "empty"
```

Zusammen erhalten wir folgendes Bild.



Die Namensbindung unserer selbst definierten Funktion `var()` besteht in der globalen Umgebung. In `package:stats` sind die Namen `sd` und `var`, die mit den entsprechenden Funktionen des Paketes `stats` verbunden sind. Wird `sd()` aufgerufen, wird der Name `sd` zuerst in der globalen Umgebung gesucht, nicht gefunden und dann in `package:stats` gesucht und gefunden. Die Funktion wird ausgeführt. Dabei muss der Name `var` aufgelöst werden. Dieser wird (nach einer Ausführungsumgebung) zuerst in der Funktionsumgebung `fn_env(sd)` gesucht. Dies ist `namespace:stats`, wo der Name `var` gefunden wird. Dieser ist mit der entsprechenden Funktion aus dem Paket `stats` verbunden.

9.3 Locked Environments

Namensbindungen in Paket-Umgebungen sind *locked*, dh die Namensbindungen in diesen Umgebungen können nicht ohne Weiteres von außen geändert werden.

```
env_bind(base_env(), "any"=all)
## Error in env_bind(base_env(), any = all): cannot change value of locked binding for 'any'
```

Siehe auch `?lockBinding`.

```
unlockBinding("any", base_env())
base_any <- any
env_bind(base_env(), "any"=all) # evil
any(T, T, F)
## [1] FALSE
env_bind(base_env(), "any"=base_any)
lockBinding("any", base_env())
any(T, T, F)
## [1] TRUE
```

10 Übersicht Umgebungen

Es gibt spezielle Umgebungsobjekte, die immer existieren: die globale Umgebung (`rlang::global_env()`), die leere Umgebung (`rlang::empty_env()`), die Umgebung von Base-R (`rlang::base_env()`).

Die aktuelle Umgebung `rlang::current_env()` hängt vom Zustand des Programms ab.

Jede Umgebung, außer der leeren Umgebung, hat eine Eltern-Umgebung (`rlang::parent_env()`). Die Eltern-Umgebung zusammen mit deren Eltern usw werden als Vorfahren bezeichnet. Andersherum können die Begriffe Kind- und Nachkommen-Umgebung genutzt werden.

Teil einer jeden Funktion vom Typ Closure ist die Funktionsumgebung (`rlang::fn_env()`). Eine Funktion wird aus der aufrufenden Umgebung `rlang::caller_env()` aufgerufen. Der Funktionscode wird in der Ausführungsumgebung aufgerufen. Die Eltern-Umgebung der Ausführungsumgebung ist die Funktionsumgebung.

Für ein Paket mit dem Namen `packnm` werden folgende Umgebungen angelegt: `namespace:pckgnm`, `imports:pckgnm`, `namespace:base` als Nachkommen der globalen Umgebung und `package:pckgnm` als Vorfahre der globalen Umgebung.

11 Lazy Evaluation 2

Argumente einer Funktion werden erst dann aufgelöst, wenn sie gebraucht werden. Zuvor sind sie nur **Versprechen** (dass der Name aufgelöst werden kann, engl *Promises*). Dies ermöglicht die sogenannte *lazy Evaluation*.

```
f1 <- function(x) {
  10
}
f1(stop("This is an error!"))
## [1] 10
```

Um die Auswertung von Argumenten zu erzwingen kann `force()` genutzt werden.

```
f2 <- function(x) {
  force(x)
  10
}
f2(stop("This is an error!"))
## Error in force(x): This is an error!
```

Versprechen sind Verknüpfungen zwischen Namen der Argumente einer Funktion und Ausdrücken in der aufrufenden Umgebung. Sie sind also nicht direkt mit Werten verbunden.

Wird der Name eines Versprechens aufgelöst, wird das Versprechen eingelöst und eine Namensbindung zwischen dem Namen und dem Wert des Ausdrucks erstellt. Danach existiert das Versprechen nicht mehr, nur noch die "normale" Namensbindung zwischen Name und Wert.

Im Inneren von `f1` ist `x` ein Versprechen, das nie eingelöst wird, weswegen der Ausdruck `stop("This is an error!")` nie ausgewertet wird.

In `f2` wird durch `force(x)` die Einlösung des Versprechens erzwungen, wodurch der Ausdruck `stop("This is an error!")` ausgewertet wird und in dem Fehler resultiert.

Statt `force(x)` könnte man auch einfach `x` schreiben. `force()` verdeutlicht jedoch, dass man einen bestimmten Effekt erwartet.

```
force
## function (x)
## x
```

```
## <bytecode: 0x00000000146688e8>
## <environment: namespace:base>
```

Das folgende Beispiel der Funktion `capture()` zeigt, dass Werte zwischen dem Erstellen des Versprechens und dem Einlösen noch geändert werden können.

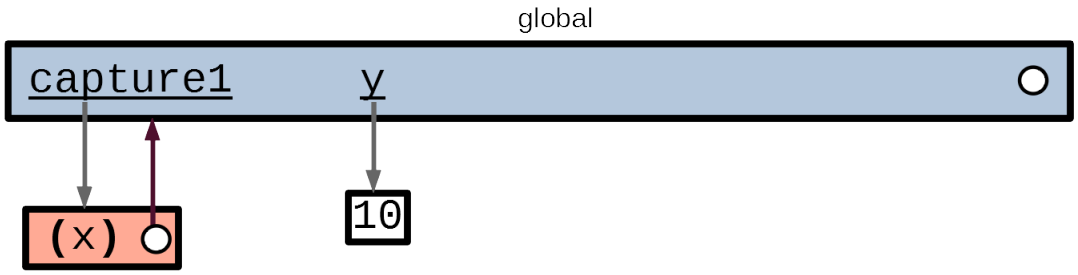
```
# gib eine Funktion zurück, die immer den Wert von x ausgibt
capture1 <- function(x) {
  function() print(x)
}
y <- 10
g1 <- capture1(y)
g2 <- capture1(y)
g1()
## [1] 10
y <- 20
g2()
## [1] 20
y <- 30
g1()
## [1] 10
g2()
## [1] 20
```

Wir gehen den Code Schritt für Schritt durch und zeichnen Umgebungsdiagramme.

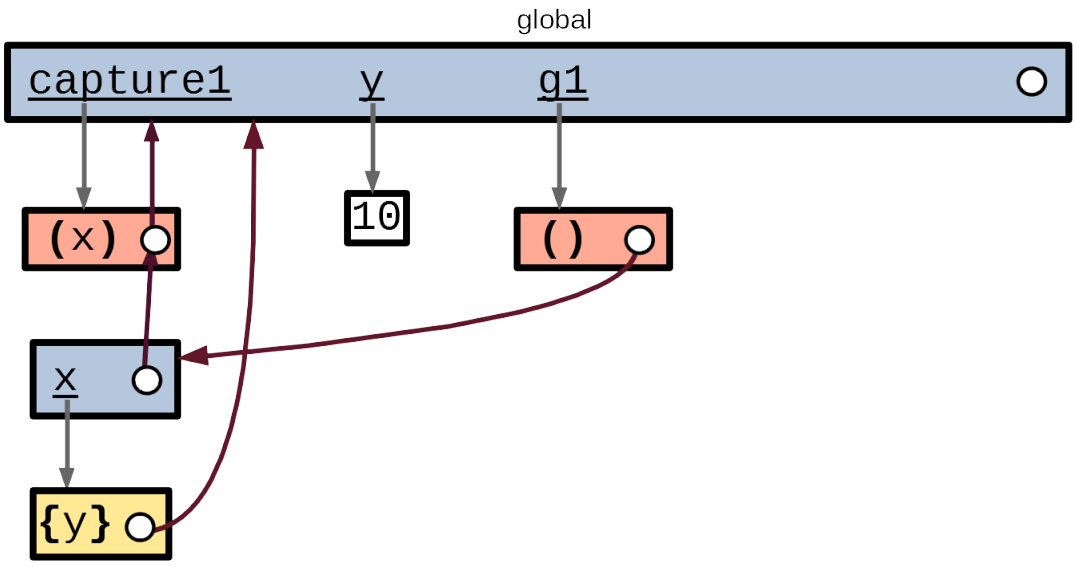
Wir oben angemerkt, haben wir Umgebungsdiagramme bisher in soweit vereinfacht, dass wir Versprechen immer sofort aufgelöst haben. Jetzt machen wir sie sichtbar.

Ein Versprechen wird durch ein Rechteck mit einem Kreis und `{`, `}` im Inneren dargestellt. Der Ausdruck, der beim Auflösen evaluiert werden soll, steht zwischen `{` und `}`. Vom Kreis geht ein Pfeil zu der Umgebung, in der dieser Ausdruck ausgewertet wird (die aufrufende Umgebung).

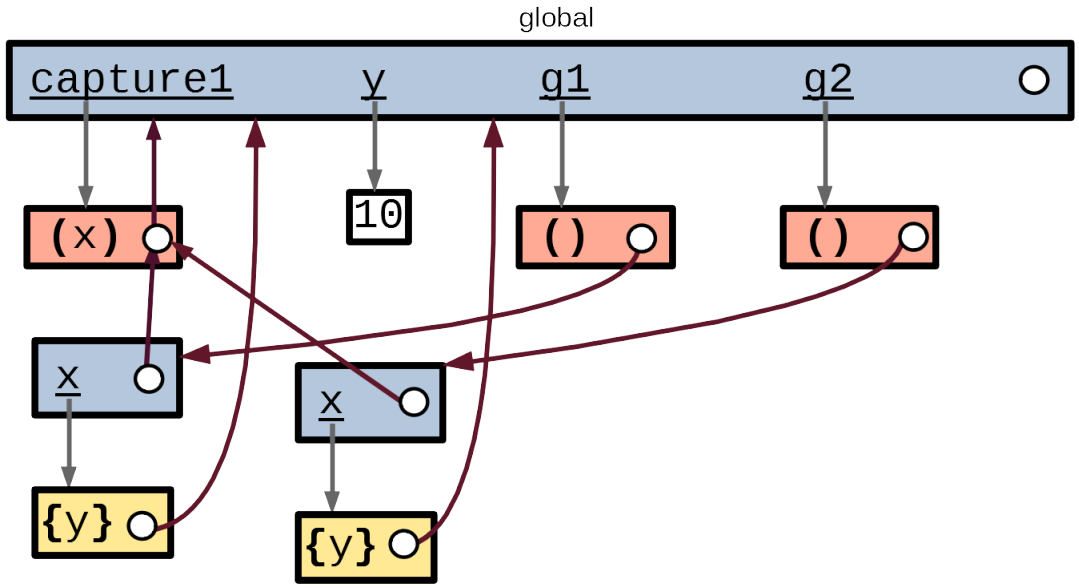
```
capture1 <- function(x) {
  function() print(x)
}

y <- 10
# Namensbindung y --> 10 in der globalen Umgebung
# 1.
```



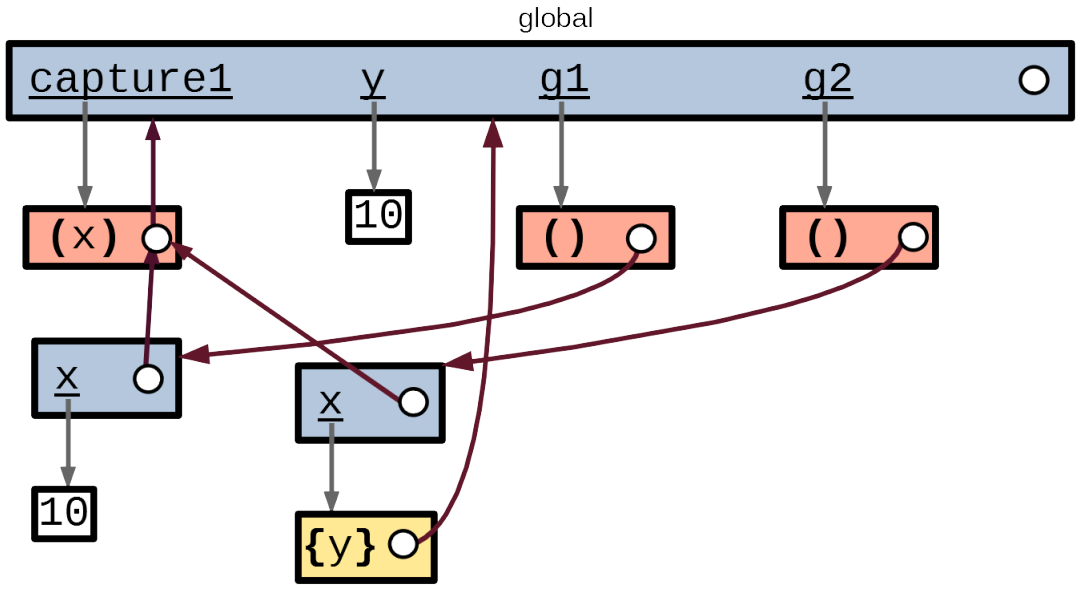

```
g1 <- capture1(y)
# In der Ausführungsumgebung [envA1] von capture wird ein Versprechen zwischen
# dem Namen x und dem Ausdruck y (mit Auswertung in der aufrufenden Umgebung
# [global]) erstellt.
# 2.
```



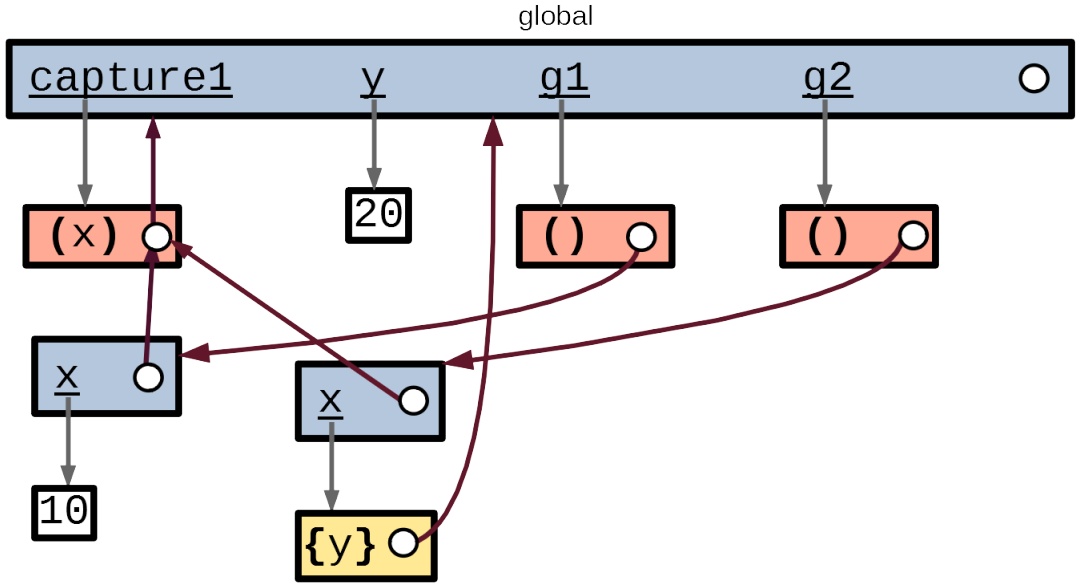

```
g2 <- capture1(y)
# Gleiches wie oben, aber
# neue Ausführungsumgebung [envA2] (Neuanfang)
# 3.
```



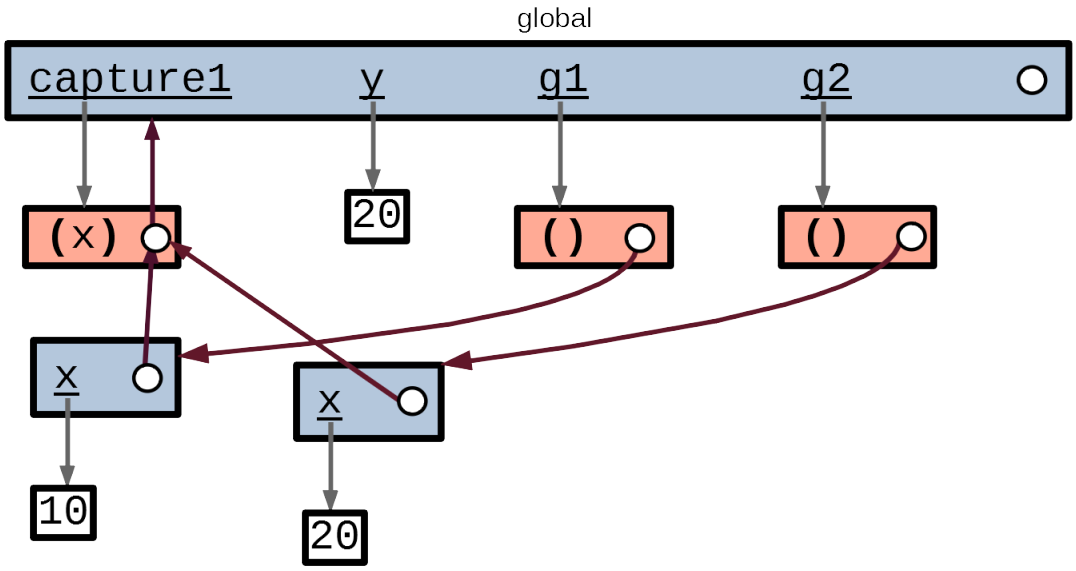

```
g1()
## [1] 10
# x muss Aufgelöst werden. Dazu wird
# das Versprechen in [envA1] eingelöst, y zu 10 aufgelöst, und
# in [envA1] die Namensbindung x --> 10 erstellt.
# 4.
```



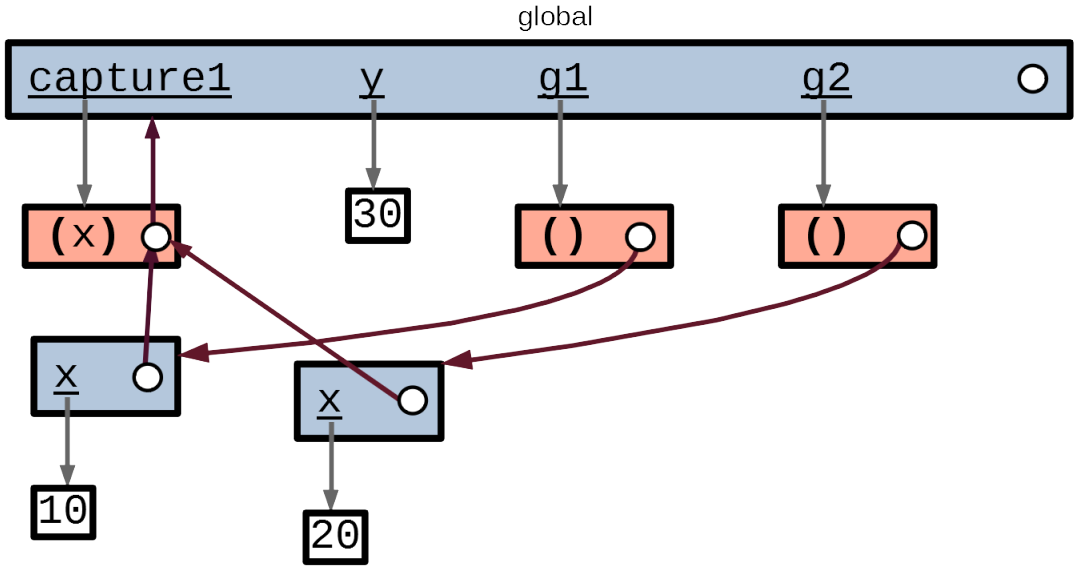

```
y <- 20
# Namensbindung y --> 20 in der globalen Umgebung
# 5.
```




```
g2()
## [1] 20
# x muss aufgelöst werden. Dazu wird
# das Versprechen in [envA2] eingelöst, y zu 20 aufgelöst, und
# in [envA2] die Namensbindung x --> 20 erstellt.
# 6.
```




```
y <- 30
# Namensbindung y --> 30 in der globalen Umgebung
# 7.
```




```
g1()
## [1] 10
# x muss aufgelöst werden.
# Es wird die Namensbindung x --> 10 in [envA1] gefunden.

g2()
## [1] 20
# x muss aufgelöst werden.
# Es wird die Namensbindung x --> 20 in [envA2] gefunden.
```

Wir möchten jedoch den Wert von y beim Aufruf der Funktion `capture()` aufnehmen (nicht beim Aufruf der

Funktion `g1()` bzw `g2()`). Lösung: Erzwinge Auflösung mittels `force()`.

```
capture2 <- function(x) {
  force(x)
  function() print(x)
}
y <- 10

g3 <- capture2(y)
# durch force() wird hier schon das Versprechen
# von x eingelöst und die Namensbindung x --> 10 erstellt.

y <- 20

g3()
## [1] 10
```

Mit der Funktion `rlang::env_bind_lazy()` können wir selbst Versprechen geben und somit *lazy evaluation* nicht nur für Funktionsargumente nutzen.

```
x <- 1
y <- 2
env_bind_lazy(current_env(), v = c(x, y)) # Versprechen des Namens v zu dem Ausdruck c(x, y)
x <- 5
v
## [1] 5 2
```

Bemerkung: Intern gibt es einen eigenständigen Datentyp für Versprechen namens `promise`. Jedoch kann die Funktion `typeof()` diesen nie ausgeben, da bei `typeof(x)` ein etwaiges Versprechen `x` eingelöst werden würde, bevor dessen Typ bestimmt werden kann.

```
env_bind_lazy(current_env(), x = 1)
typeof(x)
## [1] "double"
```