# P08 – Meta-Programmierung

31. Mai 2021

## Contents


**1 Pipe Assign (6 Punkte)** **1**

**2 Aussagenlogik (30 Punkte)** **2**

**3 Tidy-Eval-Framework (24 Punkte)** **5**


**Hinweise zur Abgabe:**

Erstelle pro Aufgabe eine R-Code-Datei (in diesem Fall 3 Dateien) und benenne diese nach dem Schema `P<Woche>-<Aufgabe>.R` also hier `P08-1.R`, `P08-2.R` und `P08-3.R`. Schreibe den Code zur Lösung einer Aufgabe in die jeweilige Datei.

Es ist erlaubt (aber nicht verpflichtend) zu zweit abzugeben. Abgaben in Gruppen von drei oder mehr Personen sind nicht erlaubt. Diese Gruppierung gilt nur für die Abgabe der Programmierprobleme.

Bei Abgaben zu zweit gibt nur eine der beiden Person ab. Dabei müssen in **jeder** abgegebenen Datei in der **ersten Zeile** als Kommentar **beide** Namen stehen also zB

```
# Ada Lovelace, Charles Babbage

1+1
# ...
```

Die Abgabe der einzelnen Dateien (kein Archiv wie `.zip`) erfolgt über Moodle im Element namens `P08`. Die Abgabe muss bis spätestens Sonntag, 6. Juni 2021, 23:59 erfolgen.

# 1 Pipe Assign (6 Punkte)

Wir wollen Zwischenergebnisse bei Funktionsaufrufen mit dem Pipeoperator speichern. Wir versuchen zuerst `->`. Wegen unterschiedlicher Operatorpriorität bringt das allerdings die Ausführungsreihenfolge durcheinander.

```
library(magrittr) # lädt %>%
c(3,4) -> x1 %>%
  `^`(2) -> x2 %>%
  sum() -> x3 %>%
  sqrt() -> res
## Error in x1 %>% `^`(2) <- c(3, 4): object 'x1' not found
```

Das Problem können wir dadurch lösen, dass wir unseren eigenen Zuweisungsoperator `%->%` schreiben, da alle Nutzeroperatoren der Form `%ANY%` die gleiche Priorität haben und dann von links nach rechts ausgeführt werden. Erzeuge `%->%`, sodass wir Zwischenergebnisse speichern können.

```
c(3,4) %->% x1 %>%
  `^`(2) %->% x2 %>%
  sum() %->% x3 %>%
  sqrt() -> res
```

```
x1
## [1] 3 4
x2
## [1]  9 16
x3
## [1] 25
res
## [1] 5
```

## 2 Aussagenlogik (30 Punkte)

Wir wollen R-Ausdrücke nutzen, um aussagenlogische Formeln zu repräsentieren und auszuwerten.

Wir definieren den Begriff **Proposition**:

- 0 und 1 sind Propositionen. (0 werden wir als *falsch* und 1 als *wahr* interpretieren.)
- $A, B, C, D, E, F, G$ sind Propositionen. (Wir beschränken uns hier auf diese 7 Variablensymbole.)
- Sind $\varphi$ und $\psi$ Propositionen, dann sind es auch
  - $(\neg\varphi)$
  - $(\varphi \wedge \psi)$
  - $(\varphi \vee \psi)$
  - $(\varphi \rightarrow \psi)$
  - $(\varphi \leftarrow \psi)$
  - $(\varphi \leftrightarrow \psi)$

Klammern lassen wir ggf weg, falls dadurch keine Mehrdeutigkeiten entstehen.

Beispiele für Propositionen sind:

- $A \vee B$
- $0 \rightarrow C$
- $\neg(A \wedge (\neg A))$
- $(A \wedge 1 \wedge D) \vee C$
- $(\neg D) \rightarrow (E \leftrightarrow D)$

In R wollen wir Propositionen mit S3-Objekten der Klasse `Prop` repräsentieren, welche auf Ausdrücken basieren. Konkret übersetzen wir:

- 0 und 1 als die `double`-Werte `0` und `1`
- $A, \ldots, G$ als die Objekte `A`, ..., `G` vom Typ `symbol`.
- $\neg, \wedge, \vee$ als `!`, `&`, `|`
- $\rightarrow$, $\leftarrow$, $\leftrightarrow$ als `>=`, `<=`, `==`

**a)** Schreibe einen Validator `validate_Prop(prop)`, der überprüft, ob eine übergebenes Objekt eine Proposition ist, dh ein Ausdruck-Objekt mit S3-Klasse `Prop`, welches nur aus den obig genannten Symbolen und Operatoren besteht (und ggf Klammern). Falls `prop` keine Proposition ist, soll ein Fehler ausgegeben werden.

```
validate_Prop(expr(A >= B))
## Error in validate_Prop(expr(A >= B)): class must be Prop
validate_Prop(structure(expr(A >= B), class="Prop"))
## A >= B
## attr(,"class")
## [1] "Prop"
validate_Prop(structure(expr(A < B), class="Prop"))
## Error in validate_alphabet(x): operator not allowed
validate_Prop(structure(expr(A <= B), class="Prop"))
## A <= B
```

```
## attr(,"class")
## [1] "Prop"
validate_Prop(structure(expr(a <= B), class="Prop"))
## Error in FUN(X[[i]], ...): only symbols A to G are supported
validate_Prop(structure(expr(AA <= B), class="Prop"))
## Error in FUN(X[[i]], ...): only symbols A to G are supported
validate_Prop(structure(expr((!A) <= (B & Y)), class="Prop"))
## Error in FUN(X[[i]], ...): only symbols A to G are supported
validate_Prop(structure(expr((!A) <= (B & A)), class="Prop"))
## (!A) <= (B & A)
## attr(,"class")
## [1] "Prop"
```

**b)** Schreibe eine Funktion `Prop(x)`, welche ihr Argument als Ausdruck aufnimmt, es mit der Klasse `Prop` versieht und mit `validate_Prop()` validiert.

```
Prop(A >= B)
## A >= B
## attr(,"class")
## [1] "Prop"
Prop(((A >= B) & C) <= B)
## ((A >= B) & C) <= B
## attr(,"class")
## [1] "Prop"
Prop(A < B)
## Error in validate_alphabet(x): operator not allowed
```

**Wichtig:** Repräsentiere eine Proposition, die aus einem einzelnen Symbol `X` besteht, als den Ausdruck (Typ: `language`) `(X)`. Dies ist notwendig, da der Ausdruck `X` (Typ: `symbol`) keine Attribute haben kann.

```
ex1 <- expr(A)
typeof(ex1)
## [1] "symbol"
structure(ex1, class="Prop")
## Error in attributes(.Data) <- c(attributes(.Data), attrib): cannot set attribute on a symbol

ex2 <- expr((A))
typeof(ex2)
## [1] "language"
structure(ex2, class="Prop")
## (A)
## attr(,"class")
## [1] "Prop"

Prop(A)
## (A)
## attr(,"class")
## [1] "Prop"
```

**c)** Schreibe eine Funktion, die dafür sorgt, dass wir mit `print()` ein Proposition-Objekt so ausgeben, dass die mathematischen Operatoren $\neg, \wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow$ anstatt der R-Operatoren `!`, `&`, `|`, `>=`, `<=`, `==` angezeigt werden. Nutze die vorgegebenen Variablen `ops_expr_str` und `ops_print_str`. Sie enthalten die entsprechenden Ersetzungen.

```
ops_expr_str <- c("==", "<=", ">=", "&", "|", "!")
ops_print_str <- c("\u2194","\u2190","\u2192","\u2227","\u2228","\u00AC")
cat(ops_print_str) # Teste dies in der Konsole aus
## <U+2194> <U+2190> <U+2192> <U+2227> <U+2228> ¬

# Sollte in der Konsole funktionieren. Hier werden leider nur die Unicode-Werte angezeigt...
Prop(A >= B)
## A <U+2192> B
Prop(((A >= B) & C) <= B)
## ((A <U+2192> B) <U+2227> C) <U+2190> B
Prop(!A & (B >= ((C == C) | B)))
## ¬A <U+2227> (B <U+2192> ((C <U+2194> C) <U+2228> B))
```

**d)** Schreibe eine Funktion `interprete(prop, vars, append=FALSE)`, die eine Proposition `prop` und ein Tibble `vars` entgegen nimmt und einen `logical`-Vektor (`append=FALSE`) oder eine Tibble (`append=TRUE`) ausgibt. Die Spaltennamen von `vars` decken hierbei alle in `prop` vorkommenden Variablensymbole ab. Jede Zeile ist eine **Interpretation** $I\colon \{A,\dots,G\} \to \{0,1\}$, dh eine Belegung der Variablensymbole mit `0` oder `1`. Wir schreiben $I \models \varphi$, um auszudrücken, dass eine Interpretation $I$ ein **Modell** ist für eine Proposition $\varphi$ (dh unter der Interpretation $I$ ist $\varphi$ wahr), ansonsten schreiben wir $I \not\models \varphi$ (dh unter der Interpretation $I$ ist $\varphi$ falsch). Diese **Modellrelation** ist wie folgt definiert.

- $I \not\models 0$, $I \models 1$
- $I \models X$ genau dann, wenn $I(X) = 1$ für $X \in \{A,\dots,G\}$
- $I \models \neg\varphi$ genau dann, wenn $I \not\models \varphi$
- $I \models (\varphi \wedge \psi)$ genau dann, wenn $I \models \varphi$ und $I \models \psi$
- $I \models (\varphi \vee \psi)$ genau dann, wenn $I \models \varphi$ oder $I \models \psi$
- $I \models (\varphi \to \psi)$ genau dann, wenn $I \not\models \varphi$ oder $I \models \psi$
- $I \models (\varphi \leftarrow \psi)$ genau dann, wenn $I \models \varphi$ oder $I \not\models \psi$
- $I \models (\varphi \leftrightarrow \psi)$ genau dann, wenn $I \models \varphi \to \psi$ und $I \models \varphi \leftarrow \psi$

Hinweis: Das klingt möglicherweise kompliziert. Bis auf $\to$ und $\leftarrow$ ist dies jedoch genau das Verhalten der R-Operatoren, die zur Repräsentation der Propositionen benutzt werden.

Die Ausgabe gibt an, ob die Interpretation der jeweiligen Zeile von `vars` ein Modell für `prop` ist. Ist `append=TRUE`, wird das Ergebnis als zusätzliche Spalte an `vars` angehängt und mit dem Text des Ausdrucks als Spaltennamen versehen.

Hinweis: `expr_text(unclass(prop))`

```
tb <- tibble(
  A = rep(c(0, 1), each=2),
  B = rep(c(0, 1), times=2))
interprete(Prop(A & B), tb, append=TRUE)
## # A tibble: 4 x 3
##       A     B `A & B`
##   <dbl> <dbl> <lgl>
## 1     0     0 FALSE
## 2     0     1 FALSE
## 3     1     0 FALSE
## 4     1     1 TRUE
interprete(Prop(A >= B), tb, append=TRUE)
## # A tibble: 4 x 3
##       A     B `A >= B`
##   <dbl> <dbl> <lgl>
## 1     0     0 TRUE
## 2     0     1 TRUE
```

```
## 3     1     0 FALSE
## 4     1     1 TRUE
interprete(Prop(A == A), tb, append=TRUE)
## # A tibble: 4 x 3
##       A     B `A == A`
##   <dbl> <dbl> <lgl>
## 1     0     0 TRUE
## 2     0     1 TRUE
## 3     1     0 TRUE
## 4     1     1 TRUE
interprete(Prop((0 & A) | (B <= A)), tb, append=TRUE)
## # A tibble: 4 x 3
##       A     B `(0 & A) | (B <= A)`
##   <dbl> <dbl> <lgl>
## 1     0     0 TRUE
## 2     0     1 TRUE
## 3     1     0 FALSE
## 4     1     1 TRUE
interprete(
  Prop((A >= B) >= C),
  tibble(A = 0, B = 1, C = 1))
## [1] TRUE
```

**e)** Schreibe eine Funktion `is_tautology(prop)`, die `TRUE` oder `FALSE` ausgibt je nachdem, ob die Proposition `prop` eine Tautologie ist, dh jede beliebige Interpretation ein Modell ist.

Hinwies: Nutze Brute-Force, dh teste alle möglichen Interpretationen aus.

```
is_tautology(Prop(A == A))
## [1] TRUE
is_tautology(Prop(A >= A))
## [1] TRUE
is_tautology(Prop((A & (A >= B)) >= B))
## [1] TRUE
is_tautology(Prop((A >= B) >= A))
## [1] FALSE
```

# 3 Tidy-Eval-Framework (24 Punkte)

Wir können ein häufiges Muster im Tidy-Eval-Framework bestehend aus `q <- enquo(arg)` am Anfang eines Funktionskörpers gefolgt von Nutzung von `!!q` mittels `{{arg}}` abkürzen. Auch die Benutzung des Ausdruckstexts von `arg` in einem String kann vereinfacht werden. Dies wird in folgendem Beispiel gezeigt.

```
df <- tibble(
  g1 = c(1, 1, 2, 2, 2),
  g2 = c(1, 2, 1, 2, 1),
  a = sample(5),
  b = sample(5)
)
grouped_summarize2 <- function(df, group, col) {
  q_group <- enquo(group)
  q_col <- enquo(col)
  nm <- str_c("mean_", quo_name(q_col))
  df %>%
    group_by(!!q_group) %>%
```

```
    summarise(!!nm := mean(!!q_col))
}
df %>% grouped_summarize2(g1, a)
## # A tibble: 2 x 2
##      g1 mean_a
##   <dbl>  <dbl>
## 1     1   4
## 2     2   2.33

grouped_summarize <- function(df, group, col) {
  df %>%
    group_by({{group}}) %>%
    summarise('{{col}}_mean' := mean({{col}}))
}
df %>% grouped_summarize(g1, a)
## # A tibble: 2 x 2
##      g1 a_mean
##   <dbl>  <dbl>
## 1     1   4
## 2     2   2.33
```

Im Folgenden kann die verkürzte Schreibweise mit `{{` oder auch die längere Schreibweise aus der Vorlesung genutzt werden.

**a)** Schreibe eine Funktion `rate(df, expr, name)`, die eine Spalte `expr` von `df` (oder einen Ausdruck, der Spaltennamen enthält) normiert zu `df` als neue Spalte `name` hinzufügt. "Normiert" bedeutet hier, dass die Summe der neu erzeugten Spalte 1 ergibt.

```
set.seed(2)
df <- tibble(
  a = sample(5),
  b = sample(5))
df
## # A tibble: 5 x 2
##       a     b
##   <int> <int>
## 1     5     1
## 2     3     5
## 3     2     4
## 4     4     2
## 5     1     3
df %>% rate(a, 'rate_a')
## # A tibble: 5 x 3
##       a     b rate_a
##   <int> <int>  <dbl>
## 1     5     1 0.333
## 2     3     5 0.2
## 3     2     4 0.133
## 4     4     2 0.267
## 5     1     3 0.0667
df %>% rate(a+b, 'rate_sum_ab')
## # A tibble: 5 x 3
##       a     b rate_sum_ab
##   <int> <int>       <dbl>
## 1     5     1       0.2
```

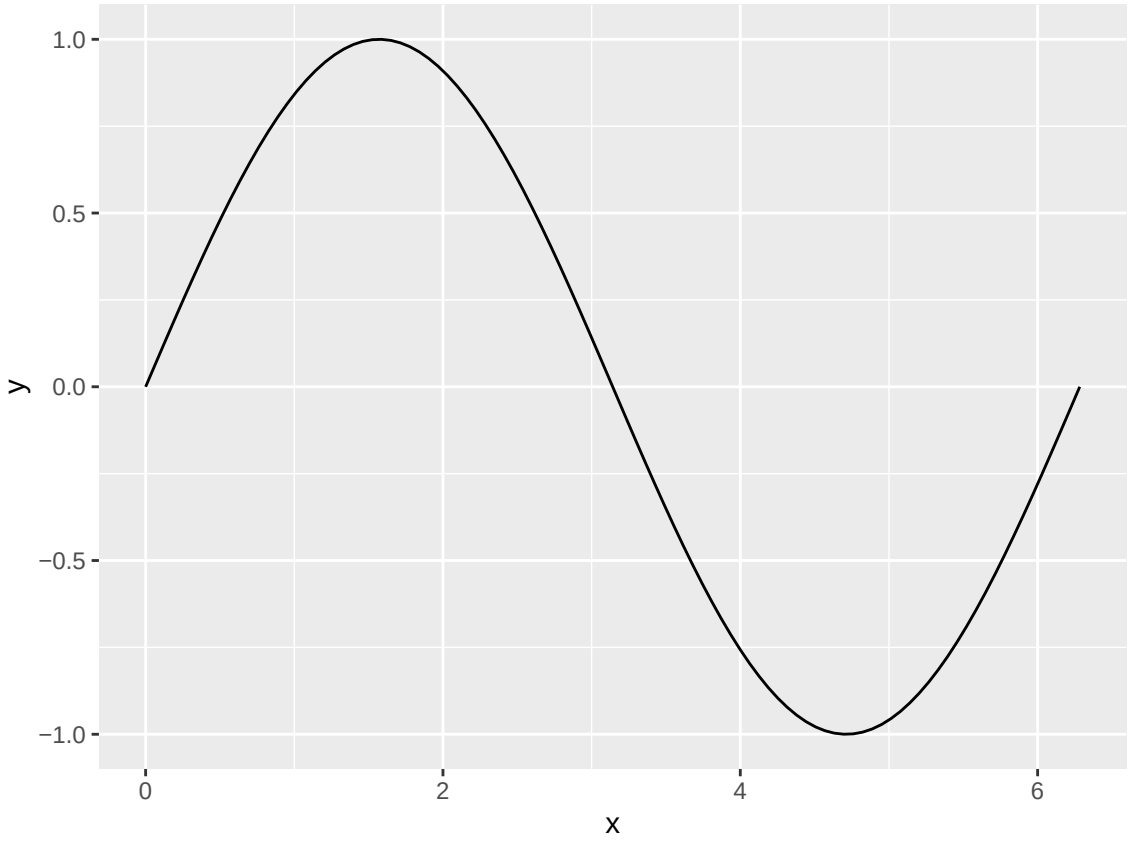
```
## 2     3     5       0.267
## 3     2     4       0.2
## 4     4     2       0.2
## 5     1     3       0.133
```

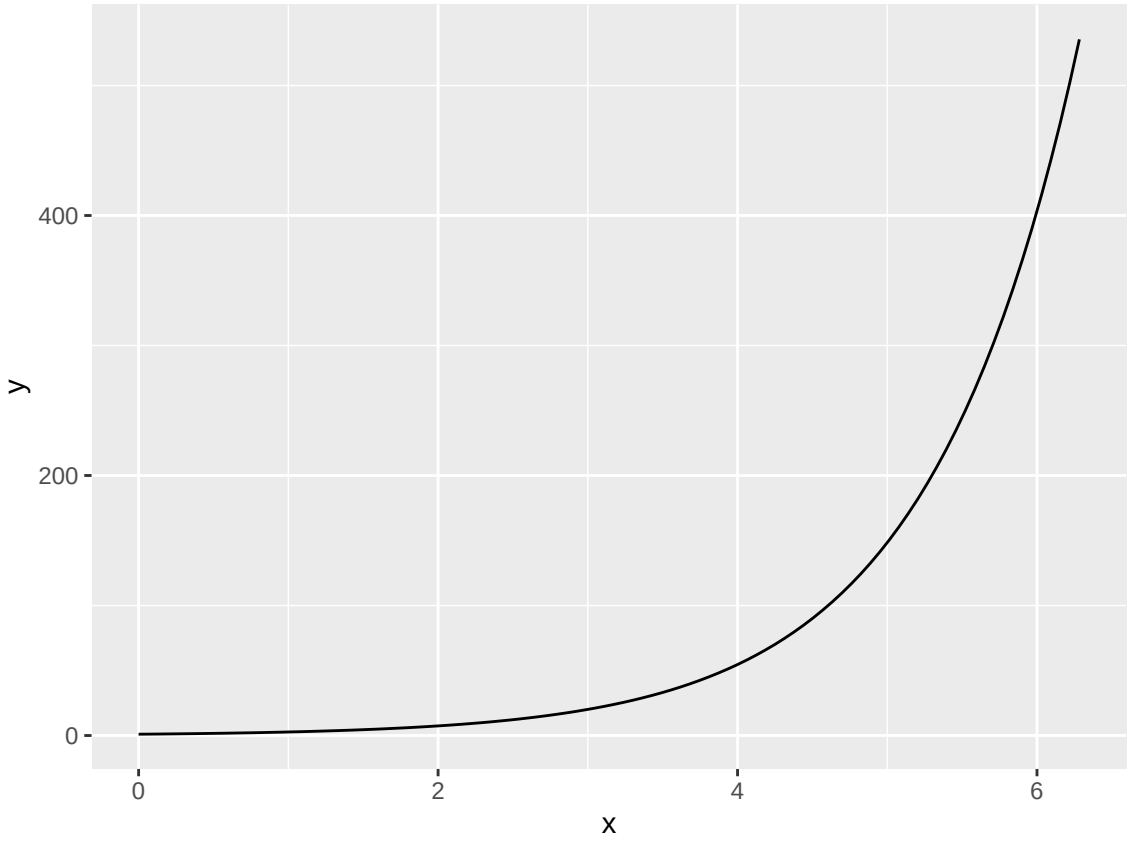
**b)** Schreibe eine Funktion `show_na(df, expr)`, die ein Tibble mit den Zeilen von `df` ausgibt, bei denen `expr` zu `NA` evaluiert. In `expr` sollen Spaltennamen wie Variablen benutzt werden können.

```
df <- tibble(
  a = c(0, NA, 0, NA, 0),
  b = c(1, 1, 0, 0, 2))
df %>% show_na(a)
## # A tibble: 2 x 2
##       a     b
##   <dbl> <dbl>
## 1    NA     1
## 2    NA     0
df %>% show_na(0/b)
## # A tibble: 2 x 2
##       a     b
##   <dbl> <dbl>
## 1     0     0
## 2    NA     0
```

**c)** Schreibe eine Funktion `ggplot_line(df, expr_x, expr_y)`, die für ein Tibble `df` einen ggplot-Objekt erzeugt, das `expr_x` gegen `expr_y` als Linie anzeigt. Wie zuvor sollen `expr_x` und `expr_y` das Tidy-Eval-Framework unterstützen.

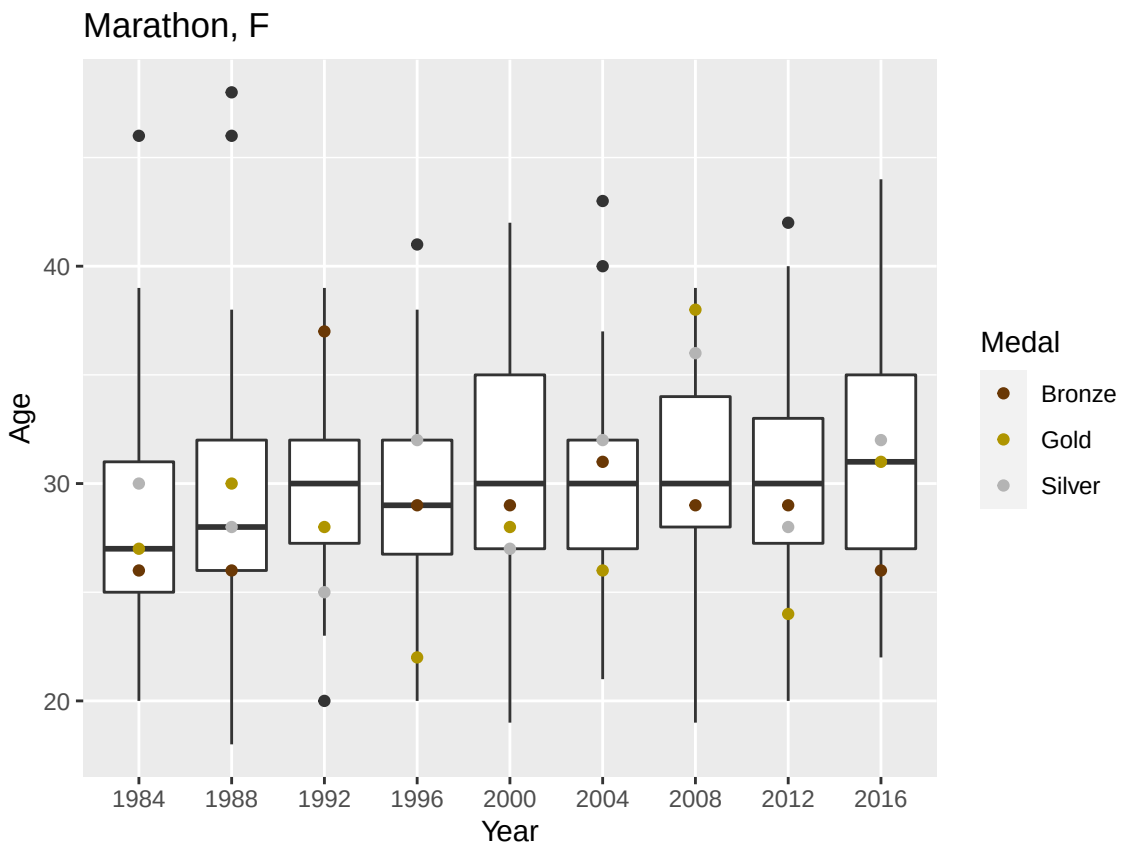
```
x <- seq(0, 2*pi, len=100)
df <- tibble(
  x = x,
  sin = sin(x))
df %>% ggplot_line(x, sin)
df %>% ggplot_line(x, exp(x))
```


1.0
0.5
0.0
−0.5
−1.0
y
0
2
4
6
x



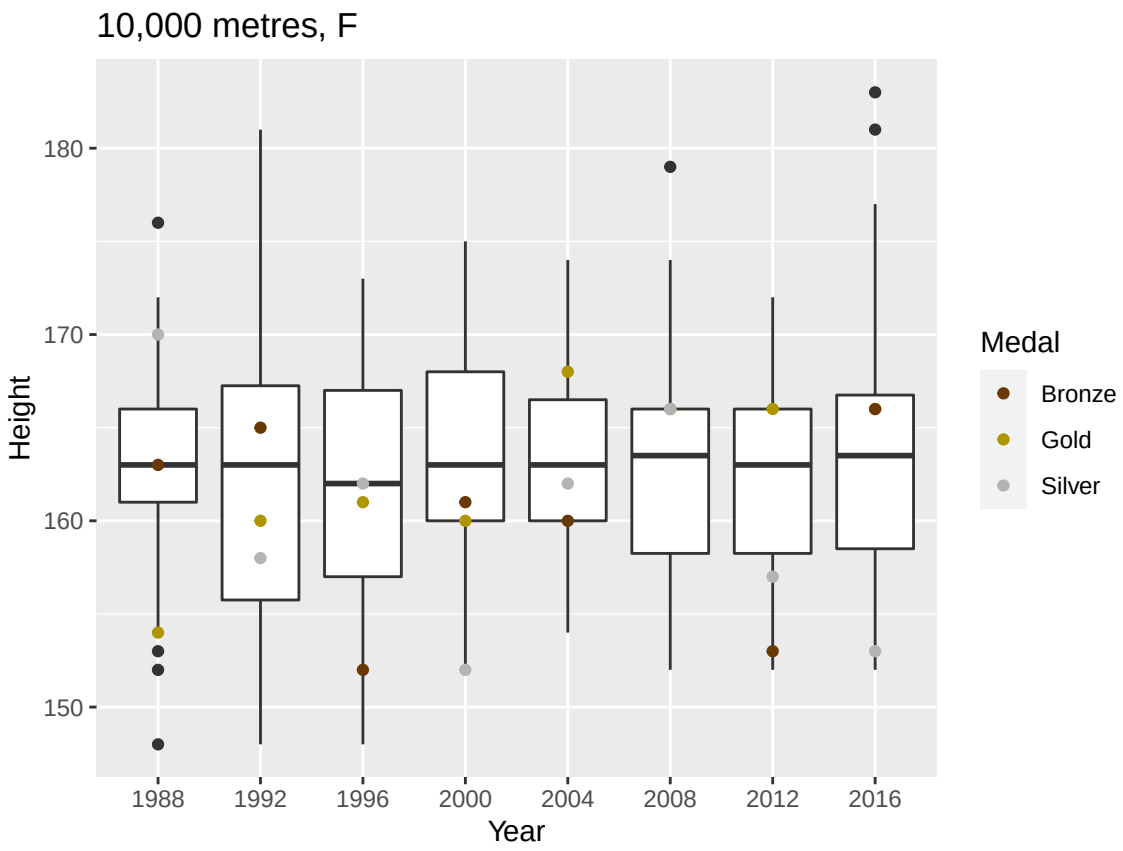

**d)** Schreibe eine Funktion `plot_olympia(event, sex, metric)`, die Plots der folgenden Art erzeugt.

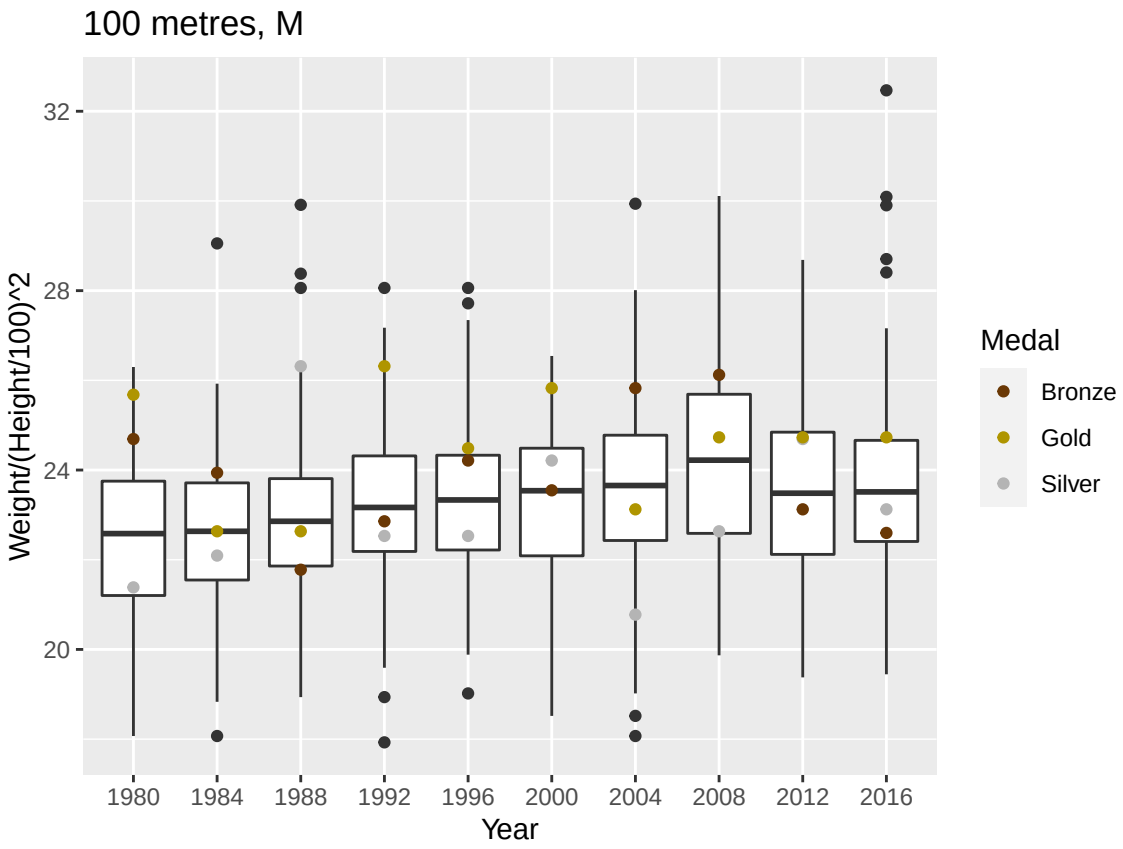
```
athletes <- read_csv("athletes.csv")
##
## -- Column specification --------------------------------------------------------
## cols(
##   Name = col_character(),
##   Sex = col_character(),
##   Age = col_double(),
##   Height = col_double(),
##   Weight = col_double(),
##   Year = col_double(),
##   Event = col_character(),
##   Medal = col_character()
## )
medal_color <- c(Bronze = "#6A3805", Silver = "#B4B4B4", Gold = "#AF9500")
  athletes %>%
    filter(Event == 'Marathon', Sex == 'F') %>%
    mutate(Year = as.factor(Year)) ->
    d
  ggplot(d, aes(x = Year, y = Age)) +
    geom_boxplot(na.rm = TRUE) +
    geom_point(data = drop_na(d), aes(color = Medal)) +
    scale_color_manual(values = medal_color) +
    ggtitle(str_c("Marathon, F"))
```




Dabei kann *Event*, *Sex* und die angezeigte Variable übergeben werden. Für letztere wird das Tidy-Eval-Framework genutzt.

```
plot_olympia("10,000 metres", "F", Height)
plot_olympia("100 metres", "M", Weight/(Height/100)^2) # BMI
```


10,000 metres, F
Height
180
170
160
150
1988
1992
1996
2000
2004
2008
2012
2016
Year
Medal
Bronze
Gold
Silver


100 metres, M
Weight/(Height/100)^2
32
28
24
20
1980
1984
1988
1992
1996
2000
2004
2008
2012
2016
Year
Medal
Bronze
Gold
Silver