Was im Seminar nicht besprochen wurde

Inhaltsverzeichnis

Kompliziertere Funktionen

Funktionen als Werte. In Typst ist es möglich, Funktionen als Wert einer Variable zu betrachten. Hier ein Beispiel:

```
let caller(fn) = {
  [Jetzt wird sie ausgeführt:]
  fn()
}

let f() = {
  lorem(20)
}

caller(f)
```

Jetzt wird sie ausgeführt: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

`f` ist also eine Funktion, die einen Blindtext aus 20 Wörtern generiert. `caller` ist eine Funktion, die ein Argument `fn` übernimmt, `[Jetzt wird die Funktion ausgeführt: ]` zurückgibt und dann noch das Ergebnis `fn()`. Also führt es den Code aus, der in `fn` liegt.

Lambda-Ausdrücke. Mithilfe sogenannter „Lambda"-Ausdrücke können wir den Code kürzen. Damit kann eine Funktion wie folgt definiert werden: `(k) => k+1`. Diese Funktion nimmt ein Argument `k` und gibt den Wert von `k+1` zurück. (Im Lambda-Ausdruck können, wenn die Funktion nur ein Argument übernimmt, die Klammern auch weggelassen werden. Mehr dazu folgt.) Dieser Lambda-Ausdruck hat keinen Namen und ist daher ein einfacher Wert, keine Variable, und kann damit z.B. in eine Variable gespeichert werden oder als Argument übergeben werden. Hier ein Beispiel:

```
let caller(fn) = {
  [Jetzt wird sie ausgeführt:]
  fn()
}
```

Jetzt wird sie ausgeführt:Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

```
caller(() => lorem(20))
```

Anstatt also `f` zu definieren, können wir einfach den Lambda-Ausdruck `() => lorem(20)` als Funktion übergeben. Die leeren Klammern vor dem `=>` Pfeil heißen einfach, dass der Lambda-Ausdruck keine Argumente übernimmt.

Warum Lambda-Ausdrücke? Weil Lambda-Ausdrücke mehr Flexibilität bieten und etwa in den folgenden Abschnitten zu States und Countern wichtig sind. Typst verbietet das Verändern von globalen Variablen, Funktionen können sich keine Werte merken, sondern produzieren mit den gleichen Parametern **immer** die gleiche Ausgabe. Das kann man hiermit teilweise umgehen.

Erlaubte Notationen. Typst ist beim Syntax ganz schön variabel. Hier ein paar Beispiele, was alles erlaubt ist (unter anderem mehrzeilige Ausdrücke):

```
c => c + 1

(e) => { e + 1 }

(e,) => { e + 1 }

(e, f, g) => {
  e + f + g
}

a => {
  lorem(a)    // Das funktioniert natürlich nur, wenn a eine Zahl ist,
  lorem(a*2)  // sonst gibt's einen Fehler.
}

let f = c => c * 7 // f(3) = 21
```

Erweiterte Show-Regeln

Es ist möglich in `show`-Regeln Inhalte mehr anzupassen, als im Seminar gezeigt. Dazu werden Ausdrücke verwendet, die ein Argument entgegennehmen. Hier ein Beispiel, in dem alle Überschriften mit „Überschrift:“ beginnen sollen:

```
#show heading: it => [Überschrift: ] + it.body

=== Wichtiger Text
```

Wird zu:

Überschrift: Wichtiger Text

Ein wichtiger Hinweis muss angebracht werden: es ist möglich rekursive Regeln zu erzeugen, die den Compiler zum Absturz bringen. Hier ein Beispiel:

```
#show heading: it => heading([Überschrift: ] + it.body)
```

Erzeugt man in der `show`-Regel das Element, das durch diese Regel beeinflusst wird, dann stürzt der Compiler ab, weil er diese Regel versucht rekursiv ohne Abbruchbedingung anzuwenden.

Counter

Pure Functions. Ein imperativer (und in Typst nicht-funktionsfähiger) Ansatz zu zählen wäre Folgender:

```
#let counter = 0
#let step() = {
  counter += 1;
}

Hallo #counter!
#step()
Das zweite Hallo: #counter!
```

Schreibt man diesen Code in Typst, erhält man den folgenden Fehler: `error: variables from outside the function are read-only and cannot be modified`.

Das klappt also nicht. Es verstößt auch gegen den Grundsatz von „Pure Functions“ in Typst. Eine Funktion kann sich keine Dinge merken und keine Variablen von außen verändern. Mit den gleichen Argumenten gibt eine Funktion immer den gleichen Rückgabewert zurück. *(Fun-Fact: Typst ist wegen genau dieses Prinzips auch so unglaublich schnell. Es kann sich nämlich den Wert der Funktionen merken und muss ihn nicht immer neu berechnen.)* `step()` übernimmt gar keine Argumente, also muss sie immer den gleichen Rückgabewert haben. Für eine imperative Denkweise ist das problematisch. Wie lösen wir das Problem?

Lasst uns zählen! Es geht also trotzdem. Aber wie? Die Antwort: States. Eine spezifische Art dieser States sind Counter. Ohne viel zu reden, hier ein Beispiel:

```
#let my_counter = counter("my_counter")
#my_counter.display() \
#my_counter.step()
#my_counter.display() \
#my_counter.update(c => c * 3)
#my_counter.display() \
```

0
1
3

Und zack! Da sind unsere Counter und unsere Lambda-Ausdrücke sind plötzlich nützlich geworden. Doch was passiert hier eigentlich? In der ersten Zeile wird der Counter erstellt, dort hat er bereits den Wert `0`:

```
#let my_counter = counter("my_counter")
```

Anstatt aber einfach mit `#my_counter` auf den Wert des Counters zugreifen zu können, benötigen wir die Methode `.display()`. Damit kann Typst es in der richtigen Reihenfolge für sich selbst auswerten. Und damit können wir nicht direkt auf den Wert zugreifen und ihn verändern, damit das „Pure Function“-Konzept erhalten bleibt.

```
#my_counter.display()
```

Mit der Methode `.step()` wird der Counter erhöht. Alternativ kann man den Wert auch mit `.update(1)` auf einen spezigischen Wert (hier 1) setzen.

```
#my_counter.step()
#my_counter.display()
#my_counter.update(1) // Beispiel für Update
#my_counter.display()
```

Die `.update()`-Funktion ist sehr mächtig. Man kann ihr statt einem Wert auch eine Lambda-Funktion übergeben. Diese Lambda-Funktion übernimmt den aktuellen Wert, führt den Ausdruck aus und setzt den Wert des Counters auf den Rückgabewert der Funktion. Hier wird der aktuelle Counter-Wert einfach nur verdreifacht, zurückgegeben und in den Counter gespeichert:

```
#my_counter.update(c => c * 3)
#my_counter.display()
```

Typische Counter. Es gibt einige Counter, die standardmäßig definiert sind, wie etwa:

```
#set heading(numbering: "1.")
#let page_counter = counter(page)
#let heading_counter =
counter(heading)

  = Überschrift
  == Noch eine

  Ich bin auf Seite
#page_counter.display() und das
ist die Überschrift Nummer
#heading_counter.display()
```

1. Überschrift

1.1. Noch eine

Ich bin auf Seite 4 und das ist die Überschrift Nummer 1.1.

States

Wenn man Counter verstanden hat, werden States genauso verständlich. Anstatt Zahlen wie in Countern, kann man in States *alles* speichern. Hier ein Beispiel, indem jeder Aufruf einer Funktion einen State verändert. Ein Dictionary mit `list` und einem Zähler `ctr` wird angelegt. Immer, wenn `example` aufgerufen wird, soll `example` die Liste um das Element `ctr+1` erweitern, und dann `ctr` um 1 erhöhen und das Dictionary anzeigen.

```
#let dict_state = state("dict")
#state("dict").update((ctr: 0,
list: ()))

#let example() = {
```

```
(ctr: 1, list: (1,))
(ctr: 2, list: (1, 2))
(ctr: 3, list: (1, 2, 3))
FINALLY: (ctr: 3, list: (1, 2, 3))
```

```
  state("dict").update(dict => {

dict.at("list").push(dict.ctr+1)
    dict.at("ctr") += 1
    dict
  })

  state("dict").display()
}

#example() \
#example() \
#example() \
FINALLY: #state("dict").display()
```

Locate

Du willst also unbedingt auf den Wert eines Counters oder States zugreifen? Das geht. Aber nur in einem abgesicherten Bereich. Und ändern darfst du den Wert dadrin nicht! Dieser abgesicherte Bereich ist die `locate`-Funktion:

```
#let my_state = state("my_state")
#my_state.update(17)

#locate(loc => {
  my_state.at(loc) + 1
})
```

18

Und wieder ein Lambda-Ausdruck! `locate` nimmt einen Lambda-Ausdruck mit einem Argument. Damit erhält die Funktion die Information, wo sich der Typst-Compiler gerade befindet und kann den richtigen Wert des States raussuchen. Immer, wenn ein `.update()` gemacht wird, wird der Wert gespeichert. Am Ende kennt Typst alle Zustände, die der State hatte. Mit `.at()` kann man auf einen dieser Werte zugreifen. Damit erhält man dann auch tatsächlich den Wert an sich. Aus dem `locate()` bekommt man ihn aber nicht heraus, denn `locate()` gibt Content zurück, hier die `18` als Text formatiert. Soetwas wie …

```
#let my_state = state("my_state")
#my_state.update(17)

// wir wollen 18, die +1 ist heir außerhalb des locates(...)
#let added = locate(loc => { my_state.at(loc) }) + 1 // FEHLER!
#added // 18 soll ausgegeben werden
})
```

… ist also nicht möglich. Es gibt den Fehler: `cannot add content and integer`, weil `locate(...) + 1` keine gültige Addition ist, da `locate(...)` Content zurückgibt, was nicht mit Zahlen addiert werden kann.

Zeitreisen. Mit States und Countern ist es auch möglich durch die Zeit zu reisen. Mit `.final()` bekommt man den letzten Wert des States am Ende des Dokuments:

```
#let s = state("timetraveler")

#let increase_s() = s.update(k => {
  k.push(k.len())
  return k
})

Am Ende ist `#s` = #locate(loc =>
s.final(loc))

Vorher:
#s.update((0,1))
#s.display()

#increase_s()
#s.display()

#increase_s()
#s.display()
```

Am Ende ist `#s` = `(0, 1, 2, 3)`
Vorher: `(0, 1)`
`(0, 1, 2)`
`(0, 1, 2, 3)`

Zu kompliziert?

Das Konzept von „Pure Functions“, „Lambda Ausdrücken“ und „States“ in Typst ist nicht einfach. Es beinhaltet viele Ansätze von nicht-imperativen Programmiersprachen. Wie geht man damit am besten um? Die Dokumentation ist natürlich ein guter Anfang, der Discord hilft immer und in allen anderen Fällen: Probieren geht über studieren. Man kann auch mit einem System warm werden, indem man einfach seine Limitationen ausreizt und probiert, was geht und was nicht. Vor den Fehlermeldungen muss man im Gegensatz zu denen bei LaTeX ja keine Angst haben.