Funktionale Frontend-Entwicklung

Jan Christiansen

28. Mai 2021

Inhaltsverzeichnis

1 Vorwort

In dieser Vorlesung wollen wir uns Programmiertechniken der funktionalen Programmierung am Beispiel Elm anschauen. Dabei stellt sich zunächst die Frage, was funktionale Programmierung bzw. eine funktionale Programmiersprache ist. Es gibt keine feste Definition des Begriffs funktionale Programmiersprache. Zumeist wird eine Programmiersprache als funktional bezeichnet, wenn die Methodik der Sprache vorgibt, dass man vorwiegend funktionale Entwicklungsmuster bei der Entwicklung einsetzt. Grundsätzlich kann man aber in jeder modernen Programmiersprache funktionale Muster anwenden. Daher ist das Studium der entsprechenden Konzepte auch für die Softwareentwicklung in anderen Paradigmen von Interesse. Insbesondere ist in der professionellen Softwareentwicklung in den letzten Jahren ein Trend zu den Prinzipien der funktionalen Programmierung zu erkennen. Erfahrene Softwareentickler entdecken häufig die Probleme beim Einsatz imperativer Konzepte und propagieren zunehmend den Einsatz funktionaler Techniken. Dies führte unter anderem zur Entwicklung von Bibliotheken wie Immutable.js für JavaScript, aber auch zur Entwicklung von Programmiersprachen wie Kotlin und Swift.

Die funktionale Programmierung zeichnet sich im Wesentlichen durch die folgenden zwei Hauptaspekte aus. Was diese Aspekte im Detail bedeuten werden wir im Laufe dieser Veranstaltung lernen.

- Funktionen sind *first-class citizens*. Das heißt, Funktionen können wie andere Werte als Argument oder Resultat einer Funktion genutzt werden. Außerdem können Funktionen wie andere Werte in Datenstrukturen abgelegt werden.

- Die Ausführung eines Programms entspricht der Auswertung eines Ausdrucks und nicht der Abarbeitung von Anweisungen wie es in der imperativen Programmierung der Fall ist.

2 Grundlagen

In diesem Kapitel führen wir die Grundlagen der Programmiersprache Elm ein. Am Ende des Kapitels werden wir in der Lage sein, eine einfache Frontend-Anwendung mit Elm zu programmieren.

2.1 Projekt-Setup

Zur Illustration der Beispiele verwenden wir das Kommando `elm repl`. Das Akronym *REPL* steht für *Read-Evaluate-Print-Loop* und beschreibt eine textuelle, interaktive Eingabe, in der man einfache Programme eingeben (*read*), die Ergebnisse des Programms ausrechnen (*evaluate*) und das Ergebnis auf der Konsole ausgeben (*print*) kann. Mit dem Begriff *Loop* wird dabei ausgedrückt, dass dieser Vorgang wiederholt werden kann. Wir werden die folgenden Programme immer in eine Datei mit der Endung `elm` schreiben. Um die Datei als Modul in der REPL importieren zu können, müssen wir den folgenden Kopf verwenden.

```
module Test exposing (..)
```

Die zwei Punkte in den Klammern beschreiben dabei, dass wir alle Definitionen im *Modul* Test zur Verfügung stellen wollen. Später werden wir in den Klammern explizit die Objekte auflisten, die unser Modul nach außen zur Verfügung stellen soll.

Um unser Modul in der REPL nutzen zu können, müssen wir zuerst ein Elm-Projekt anlegen. Zu diesem Zweck muss der Aufruf `elm init` ausgeführt werden. Das Kommando `elm init` legt unter anderem eine Datei `elm.json` an, die unser Paket beschreibt.

```
{
    "type": "application",
    "source-directories": [
        "src"
    ],
    "elm-version": "0.19.1",
    "dependencies": {
        "direct": {
            "elm/browser": "1.0.2",
            "elm/core": "1.0.5",
            "elm/html": "1.0.0"
        },
        "indirect": {
            "elm/json": "1.1.3",
            "elm/time": "1.0.0",
            "elm/url": "1.0.0",
```

```
            "elm/virtual-dom": "1.0.2"
        }
    },
    "test-dependencies": {
        "direct": {},
        "indirect": {}
    }
}
```

Dieser Aufruf installiert Basispakete, die bei der Arbeit mit Elm zur Verfügung stehen. Das Paket `elm/core` stellt zum Beispiel grundlegende Datenstrukturen wie Listen und Funktionen darauf zur Verfügung und `elm/html` stellt Kombinatoren zur Verfügung, um HTML-Seiten zu erzeugen. Unter `https://package.elm-lang.org` kann man die Dokumentationen zu diesen Paketen und noch vielen einsehen.

Wir legen die Datei mit unserem Modul im `src`-Verzeichnis ab, das `elm init` angelegt hat. Wir können dann das Modul laden, indem wir `import Test` in der REPL eingeben.

2.2 Sprach-Grundlagen

Der folgende Ausschnitt demonstriert, wie man in Elm Kommentare schreibt.

```
-- This is a line comment

{- This is a block comment -}
```

Durch die folgende Angabe kann man in Elm eine Variable definieren.

```
magicNumber : Int
magicNumber =
    42
```

Dabei gibt die erste Zeile den Typ der Variable an, in diesem Fall also ein Integer und die zweite Zeile ordnet der Variable einen Wert zu.

In einer funktionalen Programmiersprache sind Variablen nicht veränderbar wie in einer imperativen Sprache, sondern sind lediglich Abkürzungen für komplexere Ausdrücke. In diesem Fall wird die Variable sogar nicht als Abkürzung verwendet, sondern nur, um dem Wert einen konkreten Namen zu geben und diesen an verschiedenen Stellen verwenden zu können. Das heißt, wenn wir die Zeile

```
magicNumber =
    43
```

zu unserem Modul hinzufügen, erhalten wir einen Fehler, da wir die Variable nicht neu setzen können.

2.3 Grunddatentypen

Wir haben den Datentyp *Int* bereits kennengelernt. Daneben gibt es noch die folgenden Grunddatentypen.

```
f : Float
f =
    4.567


b1 : Bool
b1 =
    True


b2 : Bool
b2 =
    False


c1 : Char
c1 =
    'a'

c2 : Char
c2 =
    ' '


s : String
s =
    "Hello World!"
```

Das heißt, im Unterschied zu JavaScript unterscheidet Elm zwischen dem Typ *Int* und dem Typ *Float*.

Wenn wir eine der Definitionen aus dem Modul *Test* nutzen möchten, müssen wir das Modul zuerst mit dem Kommando `import` *Test* `exposing (..)` importieren. Danach können wir die Definitionen in *Test* verwenden. Dabei besagt das `exposing (..)`, dass wir alle Definitionen aus dem Modul *Test* importieren möchten.

2.3.1 Arithmetische Ausdrücke

Wir haben gesagt, dass in einer funktionalen Sprache und damit auch in Elm ein Programm ausgeführt wird, indem der Wert eines Ausdrucks berechnet wird. Dies lässt sich sehr schön mit Hilfe von arithmetischen und booleschen Ausdrücken illustrieren. Wir müssen für einen Ausdruck in Elm keinen Typ angeben, da der Compiler in der

Lage ist, den Typ selbst zu bestimmen. Man sagt, dass Elm den Typ *inferiert* und spricht von *Typinferenz*.

Die folgenden Definitionen zeigen einige Beispiele für arithmetische Ausdrücke.

```
ex1 =
    1 + 2


ex2 =
    19 - 25


ex3 =
    2.35 * 2.3


ex4 =
    2.5 / 23.2


ex5 =
    modBy 19 3
```

Elm erlaubt es nicht, Zahlen unterschiedlicher Art zu kombinieren. So liefert die folgende Definition zum Beispiel einen Fehler.

```
typeError = magicNumber + f
```

Wir können Zahlen nur mit `+` addieren, wenn sie den gleichen Typ haben. Daher müssen wir Zahlen ggf. explizit konvertieren.

Um einmal zu illustrieren, dass Elm sich sehr viel Mühe bei Fehlermeldungen gibt, wollen wir uns den Fehler anschauen, den die REPL liefert, wenn wir versuchen, zwei Zahlen, die unterschiedliche Typen haben, zu addieren.

```
-- TYPE MISMATCH ---------------------------------------------- src/Test.elm

I need both sides of (+) to be the exact same type. Both Int or both Float.

15|     magicNumber + f
        ^^^^^^^^^^^^^^

But I see an Int on the left and a Float on the right.

Use toFloat on the left (or round on the right) to make both sides match!

Note: Read <https://elm-lang.org/0.19.1/implicit-casts> to learn why Elm does
not implicitly convert Ints to Floats.
```

Wir wollen uns also an den Rat halten und die Funktion `toFloat` verwenden, um den Wert vom Typ *`Int`* in einen Wert vom Typ *`Float`* umzuwandeln. So können wir die obige Addition zum Beispiel wie folgt definieren.

```
convert = toFloat magicNumber + f
```

Im Unterschied zu anderen Sprachen führt der Operator `/` nur Divisionen von Fließkommazahlen durch. Das heißt, ein Ausdruck der Form `magigNumber / 10` liefert ebenfalls einen Typfehler. Um zwei ganze Zahlen zu dividieren, muss der Operator `//` verwendet werden, der eine ganzzahlige Division durchführt.

2.3.2 Boolesche Ausdrücke

Elm stellt die üblichen booleschen Operatoren für Konjunktion, Disjunktion und Negation zur Verfügung.

```
ex9 =
    False || True


ex10 =
    not (b1 && True)
```

Daneben gibt es die Vergleichsoperatoren `==` und `/=`, so wie `<`, `<=`, `>` und `>=`.

```
ex11 =
    'a' == 'a'


ex12 =
    16 /= 3


ex13 =
    (5 > 3) && ('p' <= 'q')


ex14 =
    "Elm" > "C++"
```

Um einen Ausdruck der Form `3 + 4 * 8` nicht klammern zu müssen, definiert Elm für Operatoren Präzedenzen (Bindungsstärken). Die Präzedenz eines Operators liegt zwischen `0` und `9`. Der Operator `+` hat zum Beispiel die Präzedenz 6 und `*` hat 7. Daher steht der Ausdruck `3 + 4 * 8` für den Ausdruck `3 + (4 * 8)`. Die Präzedenz einer Funktion ist 10, das heißt, eine Funktionsanwendung bindet immer stärker als jeder Infixoperator. Der Ausdruck `not True || False` steht daher zum Beispiel für `(not True) || False` und nicht etwa für `not (True || False)`.

Neben der Bindungsstärke wird bei Operatoren noch definiert, ob diese links- oder rechts-assoziativ sind. In Elm (wie in vielen anderen Sprachen) gibt es links- und rechtsassoziative Operatoren. Dies gibt an, wie ein Ausdruck der Form $x \circ y \circ z$ interpretiert wird. Falls der Operator $\circ$ linksassoziativ ist, gilt

$$x \circ y \circ z = (x \circ y) \circ z,$$

falls er rechts-assoziativ ist, gilt

$$x \circ y \circ z = x \circ (y \circ z).$$

2.4 Funktionsdefinition

In diesem Abschnitt wollen wir uns anschauen, wie man in Elm einfache Funktionen definieren kann. Funktionen sind in einer funktionalen Sprache das Gegenstück zu (statischen) Methoden in einer objektorientierten Sprache.

2.4.1 Konditional

Elm stellt einen `if`-Ausdruck der Form `if b then e1 else e2` zur Verfügung. Im Unterschied zu einer `if`-Anweisung wie er in objektorientierten Programmiersprachen zum Einsatz kommt, kann man bei einem `if`-Ausdruck den `else`-Zweig nicht weglassen. Beide Zweige des `if`-Ausdrucks müssen einen Wert liefern. Außerdem müssen beide Zweige Werte liefern, die den gleichen Typ besitzen. Um den `if`-Ausdruck einmal zu illustrieren, wollen wir eine Funktion `items` definieren. Die Funktion `items` wird zum Beispiel für den Warenkorb eines Shoppingsystems genutzt. Die Funktion erhält eine Zahl und liefert eine Lokalisierung für das Wort *Gegenstand*. Die Zahl gibt dabei an, um wie viele Gegenstände es sich handelt.

```
items : Int -> String
items quantity =
    if quantity == 0 then
        "Kein Gegenstand"

    else if quantity == 1 then
        "Ein Gegenstand"

    else
        String.fromInt quantity ++ " Gegenstände"
```

Die erste Zeile gibt den Typ der Funktion `items` an. Der Typ sagt aus, dass die Funktion `items` einen Wert vom Typ `Int` nimmt und einen Wert vom Typ `String` liefert. Der Parameter der Funktion `items` heißt `quantity` und die Funktion prüft, ob dieser Parameter gleich `0` ist, gleich `1` ist oder einen sonstigen Wert hat.

2.4.2 Fallunterscheidung

In Elm können Funktionen mittels `case`-Ausdruck (Fallunterscheidung) definiert werden. Ein `case`-Ausdruck ist ähnlich zu einem `switch case` in imperativen Sprachen. Wir können in einem `case`-Ausdruck zum Beispiel prüfen, ob ein Ausdruck eine konkrete Zahl als Wert hat. Als Beispiel definieren wir die Funktion `items` mittels `case`-Ausdruck. Mit dem Operator `++` hängt man zwei Zeichenketten hintereinander.

```
items : Int -> String
items quantity =
    case quantity of
        0 ->
            "Kein Gegenstand"

        1 ->
            "Ein Gegenstand"

        _ ->
            String.fromInt quantity ++ " Gegenstände"
```

Die Funktion `String.fromInt` wandelt eine ganze Zahl in eine Zeichenkette um. Man nennt diese Form eines Aufrufs einen qualifizierten Funktionsaufrufs. `String` ist dabei das Modul, in dem die Funktion `fromInt` definiert ist. Ein Modul ist vergleichbar mit einer Klasse mit statischen Methoden in einer objektorientierten Programmiersprache. Durch einen qualifizierten Funktionsaufruf können wir direkt am Aufruf sehen, in welchem Modul die Funktion definiert ist. Außerdem nutzen wir auf diese Weise den Namen des Moduls als Bestandteil des Funktionsnamens und können den Namen der Funktion so kürzer fassen. So kann es zum Beispiel mehrere Funktionen geben, die `fromInt` heißen und in verschiedenen Modulen definiert sind. Durch den qualifizierten Aufruf ist dann uns (und dem Compiler) klar, welche Funktion gemeint ist.

Die Fälle in einem `case`-Ausdruck werden von oben nach unten geprüft. Wenn wir zum Beispiel den Aufruf `items 0` auswerten, so passt die erste Regel und wir erhalten `"Kein Gegenstand"` als Ergebnis. Geben wir dagegen `items 3` ein, so passen die ersten beiden Regeln nicht. Die dritte Regel ist eine *Default*-Regel, die immer passt und daher nur als letzte Regel genutzt werden darf. Das heißt, wenn wir den Aufruf `items 3` auswerten, wird anschließend der Ausdruck `String.fromInt 3 ++ " Gegenstände"` ausgewertet. Die Auswertung dieses Ausdrucks liefert schließlich `"3 Gegenstände"` als Ergebnis.

Man bezeichnet das Prüfen eines konkreten Wertes gegen die Angabe auf der linken Seite einer `case`-Regel als *Pattern Matching*. Das heißt, wenn wir den Ausdruck `items 3` auswerten, führt die Funktion Pattern Matching durch, da überprüft wird, welcher der Regeln in der Funktion auf den Wert von `quantity` passen. Die Konstrukte auf der linken Seite der Regel, also in diesem Fall `0`, `1` und `_` bezeichnet man als *Pattern*, also als Muster.

2.4.3 Mehrstellige Funktionen

Bisher haben wir nur Funktionen kennengelernt, die ein einzelnes Argument erhalten. Um eine mehrstellige Funktion zu definieren, werden die Argumente der Funktion einfach durch Leerzeichen getrennt aufgelistet. Wir können zum Beispiel wie folgt eine Funktion definieren, die den Inhalt eines Online-Warenkorbs beschreibt.

```
cart : Int -> Float -> String
cart quantity price =
    "Summe (" ++ items quantity ++ "): " ++ String.fromFloat price
```

Dabei sieht der Typ der Funktion auf den ersten Blick etwas ungewöhnlich aus. Wir werden später sehen, was es mit diesem Typ auf sich hat. An dieser Stelle wollen wir nur festhalten, dass die Typen der Argumente bei mehrstelligen Funktionen durch einen Pfeil getrennt werden.

Um die Funktion `cart` anzuwenden, schreiben wir ebenfalls die Argumente durch Leerzeichen getrennt hinter den Namen der Funktion. Das heißt, der folgende Ausdruck wendet die Funktion `cart` auf die Argumente `3` und `23.42` an.

```
cart 3 23.42
```

Wenn eines der Argumente der Funktion `cart` das Ergebnis einer anderen Funktion sein soll, so muss diese Funktionsanwendung mit Klammern umschlossen werden. So wendet der folgende Ausdruck die Funktion `cart` auf die Summe von `1` und `2` und das Minimum von `1.23` und `3.14`

```
cart (1 + 2) (min 1.23 3.14)
```

Diese Schreibweise stellt für viele Nutzer*innen, die Programmiersprachen wie Java gewöhnt sind, häufig eine große Hürde dar. Im Grunde muss man sich bei dem Aufruf einer Funktion an Hand der Klammern und der Leerzeichen aber nur überlegen, wie viele Argumente man bei einem Funktionsaufruf an eine Funktion übergibt.

2.4.4 Lokale Definitionen

In Elm können Konstanten und Funktionen auch lokal definiert werden, das heißt, dass die entsprechende Konstante oder die Funktion nur innerhalb einer anderen Funktion sichtbar ist. Anders ausgedrückt ist der *Scope* einer *Top Level*-Definition das gesamte Modul. Im Kontrast dazu ist der *Scope* einer lokalen Definition auf einen bestimmten Ausdruck eingeschränkt.

Eine lokale Definition wird mit Hilfe eines **`let`**-Ausdrucks eingeführt.

```
quartic : Int -> Int
quartic x =
    let
        square =
            x * x
    in
    square * square
```

Ein **`let`**-Ausdruck startet mit dem Schlüsselwort **`let`**, definiert dann beliebig viele Konstanten und Funktionen und schließt schließlich mit dem Schlüsselwort **`in`** ab. Die Definitionen, die ein **`let`**-Ausdruck einführt, stehen nur in dem Ausdruck nach dem **`in`** zur Verfügung. Sie können wie im Beispiel `quartic` aber auf die Argumente der umschließenden Funktion zugreifen.

Man kann in einem **`let`**-Ausdruck auch Funktionen definieren, die dann auch nur in dem Ausdruck nach dem **`in`** sichtbar sind. Wir werden später Beispiele sehen, in denen dies sehr praktisch ist, zum Beispiel, wenn wir Listen verarbeiten. Dort wird häufig die Verarbeitung eines einzelnen Listenelementes als lokale Funktion definiert.

```
res : Int
res =
    let
        inc n =
            n + 1
    in
    inc 41
```

Wie andere Programmiersprachen, zum Beispiel Python, Elixir und Haskell, nutzt Elm eine *Off-side Rule*. Das heißt, die Einrückung eines Programms wird genutzt, um Klammerung auszudrücken und somit Klammern einzusparen. In objektorientierten Sprachen wie Java wird diese Klammerung durch geschweifte Klammern ausgedrückt. Dagegen muss die Liste der Definitionen in einem **let** zum Beispiel nicht geklammert werden, sondern wird durch ihre Einrückung dem **let**-Block zugeordnet.

Das Prinzip der *Off-side Rule* wurde durch Peter J. Landin[1] in seiner wegweisenden Veröffentlichung „The Next 700 Programming Languages“ (Landin, 1966) erfunden.

> *Any non-whitespace token to the left of the first such token on the previous line is taken to be the start of a new declaration.*

Um diese Aussage zu illustrieren, betrachten wir das folgende Beispielprogramm, das vom Compiler aufgrund der Einrückung nicht akzeptiert wird.

```
layout1 : Int
layout1 =
    let
    x =
        1
    in
    42
```

Das **let** definiert eine Spalte. Alle Definitionen im **let** müssen in einer Spalte rechts vom Schlüsselwort **let** starten. Die erste Definition, die in einer Spalte steht, die in der Spalte des **let** oder weiter rechts steht, beendet das **let**. Die Definition `layout1` wird nicht akzeptiert, da das **let** durch das `x` beendet wird, was aber keine valide Syntax ist, da das **let** mit dem Schlüsselwort **in** beendet werden muss.

Als weiteres Beispiel betrachten wir die folgende Definition, die ebenfalls aufgrund der Einrückung nicht akzeptiert wird.

```
layout2 : Int
layout2 =
    let
        x =
            1

         y =
```

[1] Peter J. Landin (`https://en.wikipedia.org/wiki/Peter_Landin`) war einer der Begründer der funktionalen Programmierung.

```
        2
in
42
```

Die erste Definition in einem `let`-Ausdruck, also hier das `x`, definiert ebenfalls eine Spalte. Alle Zeilen, die links von der ersten Definition starten, beenden die Liste der Definitionen. Alle Zeilen, die rechts von der ersten Definition starten, werden noch zur vorherigen Definition gezählt. Das heißt, in diesem Beispiel geht der Compiler davon aus, dass die Definition von `y` eine Fortsetzung der Definition von `x` ist.

2.5 Weitere Datentypen

In diesem Abschnitt wollen wir die Verwendung einiger einfacher Datentypen vorstellen, die wir zur Implementierung unserer ersten Anwendung benötigen.

2.5.1 Typ-Synonyme

In Elm kann ein neuer Typ eingeführt werden, indem ein neuer Name für einen bereits bestehenden Typ angegeben wird. Der folgende Code führt zum Beispiel den Namen *`Weight`* als Synonym für den Typ *`Int`* ein. Das heißt, an allen Stellen, an denen wir den Typ *`Int`* verwenden können, können wir auch den Typ *`Weight`* verwenden.

```
type alias Weight =
    Int
```

Ein Typsynonym wird verwendet, um einem Typ zu Dokumentationszwecken einen spezifischeren Namen zu geben. Außerdem wird ein Typsynonym verwendet, um Schreibarbeit zu sparen. Wir werden diesen Effekt später sehen, wenn wir komplexere Typen wie Recordtypen kennenlernen.

2.5.2 Aufzählungstypen

Wie andere Programmiersprachen stellt Elm Aufzählungstypen zur Verfügung. So kann man zum Beispiel wie folgt einen Datentyp definieren, der die Richtungstasten der Tastatur modelliert.

```
type Key
    = Left
    | Right
    | Up
    | Down
```

Wir können für den Datentyp *`Key`* Funktionen mit Hilfe von Pattern Matching definieren. Bei den einzelnen Werten des Typs spricht man auch von Konstruktoren. Das heißt, *`Left`* und *`Up`* sind zum Beispiel Konstruktoren des Datentyps *`Key`*.

Die folgende Funktion testet, ob es sich um eine der horizontalen Richtungstasten handelt.

```
isHorizontal : Key -> Bool
isHorizontal key =
    case key of
        Left ->
            True

        Right ->
            True

        _ ->
            False
```

Wir können diese Funktion auch definieren, indem wir im Pattern Matching alle Konstruktoren aufzählen und auf den Unterstrich verzichten. Der Unterstrich ist gleichbedeutend zu einer Variable, nur dass wir auf den in der Variable gespeicherten Wert nicht zugreifen können. Das heißt, der Fall mit dem Unterstrich passt für alle möglichen Fällen, die `k` noch annehmen kann. Im Fall der Funktion `isHorizontal` wird der Unterstrich-Fall zum Beispiel verwendet, wenn `k` den Wert *`Up`* oder den Wert *`Down`* hat. Die Verwendung des Unterstrichs ist zwar praktisch, sollte aber mit Bedacht eingesetzt werden. Wenn wir einen weiteren Konstruktor zum Datentyp *`Key`* hinzufügen, würde die Funktion `isHorizontal` zum Beispiel weiterhin funktionieren. Hätten wir `isHorizontal` definiert, indem wir alle Fälle auflisten, würde der Elm-Compiler einen Fehler liefern, da einer der Fälle nicht abgedeckt ist. Es ist besser, wenn der Compiler einen Fehler liefert, da sich sonst, ohne dass wir es bemerken, Fehler in der Anwendung einschleichen können, die schwer zu finden sind.

Die folgende Funktion macht vollständiges Pattern Matching und liefert zu einer Richtung eine entsprechende Zeichenkette zurück.

```
toString : Key -> String
toString key =
    case key of
        Left ->
            "Left"

        Right ->
            "Right"

        Up ->
            "Up"

        Down ->
            "Down"
```

2.5.3 Listen

Listen werden in Elm mit eckigen Klammern definiert und die Elemente der Liste werden durch Kommata getrennt.

```
list : List Int
list =
    [ 1, 2, 3, 4, 5 ]
```

Eine leere Liste stellt man einfach durch zwei eckige Klammern dar, also als `[]`. Der Infixoperator `(::)` hängt vorne an eine Liste ein zusätzliches Element an. Das heißt, der Ausdruck `1 :: [ 2, 3 ]` liefert die Liste `[ 1, 2, 3 ]`. Der Infixoperator `(++)` hängt zwei Listen hintereinander. Das heißt, der Ausdruck `[ 1, 2 ] ++ [ 3, 4 ]` liefert die List `[ 1, 2, 3, 4 ]`. Dabei ist immer zu beachten, dass in einer funktionalen Programmiersprache Datenstrukturen nicht verändert werden. Das heißt, der Operator `(::)` liefert eine neue Liste und verändert nicht etwa sein Argument.

3 Eine Erste Anwendung

In diesem Kapitel werden wir eine erste Frontend-Anwendung mit Elm entwickeln.

3.1 HTML-Kombinatoren

Wir wollen mit einem *Hallo Welt*-Beispiel starten. Zu diesem Zweck schreiben wir das folgende Programm.

```
module HelloWorld exposing (main)

import Html exposing (Html, text)

main : Html msg
main =
    text "Hallo Welt"
```

Das Modul `Html` stellt Funktionen zur Verfügung, um Html-Seiten zu erzeugen. Die Bedeutung des Typs `Html msg`, der in der Konstante `main` verwendet wird, werden wir uns später anschauen. Die Funktion `text` ist im Modul `Html` definiert und nimmt einen `String` und liefert einen Text-Knoten. Unter `https://package.elm-lang.org/packages/elm/html/latest/` findet sich eine Beschreibung des Moduls `Html`. Wenn wir eine Definition aus dem Modul `Html` verwenden wollen, müssen wir es in der Liste hinter `exposing` aufführen. Im obigen Beispiel importieren wir den Typ `Html` und die Funktion `text` aus dem Modul `Html`. Bei einem Datentyp kann man angeben, ob man nur den Typ oder auch die Konstruktoren importieren möchte. Wenn wir so wie oben nur den Namen des Typs angeben, importieren wir nur den Typ, dürfen die Konstruktoren aber nicht verwenden. Im Fall von `Html` importieren wir nur den Typ, da wir die Konstruktoren dieses Typs nie explizit verwenden. Wenn wir auch die Konstruktoren von `Html` verwenden möchten, müssen wir in der Liste nach `exposing` die Angabe `Html(..)` machen. Auf diese Weise importieren wir auch die Konstruktoren. Die gleichen Angaben, die wir beim Importieren eines Moduls machen, können wir auch verwenden, um Definitionen aus einem Modul zu exportieren. Dazu wird die `exposing`-Anweisung genutzt, die hinter dem Namen des Moduls steht. Hier exportiert das Modul `HelloWorld` zum Beispiel nur die Funktion `main`.

Wenn wir ein Modul importieren, können wir eine Definition immer auch qualifiziert verwenden, das heißt, wir können zum Beispiel `Html.text` schreiben, um die Funktion `text` aus dem Modul `Html` zu verwenden. Eigentlich ist es guter Stil, Definitionen qualifiziert zu verwenden, um explizit anzugeben, wo die Definition herkommt. Im Fall des Moduls `Html` verzichtet man aber häufig darauf, um Programme übersichtlich zu halten.

Unter `https://package.elm-lang.org/packages/elm/core/latest/` finden sich Module, die der Elm-Compiler direkt mitbringt. Diese Module werden von jedem Elm-Modul implizit wie folgt importiert.

```
import Basics exposing (..)
import List exposing ( List, (::) )
import Maybe exposing ( Maybe(..) )
import Result exposing ( Result(..) )
import String exposing ( String )
import Char exposing ( Char )
import Tuple

import Debug

import Platform exposing ( Program )
import Platform.Cmd as Cmd exposing ( Cmd )
import Platform.Sub as Sub exposing ( Sub )
```

Das heißt zum Beispiel, dass alle Definitionen aus dem Modul *Basics* direkt zur Verfügung stehen und wir sie unqualifiziert verwenden können. Aus dem Modul *String* wird nur der Typ *String* importiert. Das heißt, den Typ *String* können wir unqualifiziert verwenden. Wenn wir allerdings eine andere Definition aus dem Modul *String* verwenden möchten, müssen wir diese Definition qualifiziert nutzen. Zum Beispiel können wir *String*`.length` schreiben, um die Funktion zu nutzen, die die Länge einer Zeichenkette liefert.

Um unsere Anwendung zu testen, können wir den Befehl `elm reactor` verwenden, der einen lokalen Webserver startet. Unter der Adresse `localhost:8000` erhalten wir eine Auswahl aller Dateien, die sich in dem entsprechenden Verzeichnis befinden. Wenn wir unser *HelloWorld*-Beispiel auswählen, erhalten wir die entsprechende HTML-Seite. Wenn wir die Seite im Browser neu laden, wird der Elm-Code neu in JavaScript-Code übersetzt und wir erhalten die aktualisierte Version der Anwendung.

Das Modul *Html* stellt eine ganze Reihe von Funktionen zur Verfügung, mit deren Hilfe man HTML-Seiten definieren kann. Als weiteres Beispiel generieren wir einmal eine HTML-Seite mit einem div, das zwei Text-Knoten als Kinder hat.

```
main : Html msg
main =
    div [] [ text "Hallo Welt", text (String.fromInt 23) ]
```

Die Funktion `div` nimmt zwei Argumente. Das erste Argument ist eine Liste von Attributen, die das div-Element erhalten soll. Das zweite Argument ist eine Liste von Kind-Elementen. Wir könnten in der Liste der Kind-Elemente also auch wieder ein div-Element verwenden.

Um die Funktionsweise von Attributen zu illustrieren, geben wir unserem div-Element einmal eine Klasse und einen CSS-Stil. Die Funktion **`class`** nimmt einen String und liefert ein Attribut, das eine Klasse definiert. Die Funktion `style` kommt aus dem Modul *Html.Attributes* und nimmt zwei Strings, nämlich den Namen des Stils und

den Wert und liefert einen entsprechenden Stil. Das heißt, wir können die folgende Definition verwenden.

```
colorful : List (Attribute msg)
colorful =
    [ style "background-color" "red", style "height" "90px" ]


main : Html msg
main =
    div (class "text" :: colorful)
        [ text "Hallo Welt", text (String.fromInt 23) ]
```

3.2 Elm-Architektur

In diesem Kapitel wollen wir uns über die Architektur einer Elm-Anwendung unterhalten. Eine Elm-Anwendung besteht immer aus den folgenden klar getrennten Teilen.

- **Model**: der Zustand der Anwendung
- **View**: eine Umwandlung des Zustandes in eine HTML-Seite
- **Update**: eine Möglichkeit, den Zustand zu aktualisieren

Eine typische Elm-Anwendung hat immer die folgende Struktur.

```
module App exposing (main)

import Browser
import Html exposing (Html)


-- Model

type alias Model = ...


init : Model
init = ...


-- Update

type alias Msg = ...


update : Msg -> Model -> Model
```

```
update msg model = ...


-- View

view : Model -> Html Msg
view model = ...


main : Program () Model Msg
main =
    Browser.sandbox { init = init, view = view, update = update }
```

Wir haben einen Typ `Model`, der den internen Zustand unserer Anwendung repräsentiert. Außerdem haben wir einen Typ `Msg`, der Interaktionen mit der Anwendung modelliert. Diese Typen sind häufig einfach Synonyme für andere Typen, können aber auch direkt als Aufzählungstyp definiert sein. Die Konstante `init` gibt an, mit welchem Zustand die Anwendung startet. Die Funktion `update` nimmt eine Aktion und einen aktuellen Zustand und liefert einen neuen Zustand. Die Funktion `view` nimmt einen Zustand und liefert eine HTML-Seite. Außerdem stellt das Modul `Browser` eine Funktion `sandbox` zur Verfügung, deren Details wir auch erst später diskutieren werden. An dieser Stelle müssen wir nur wissen, dass wir der Funktion die Konstante `init` und die Funktionen `update` und `view`, wie oben angegeben, übergeben müssen. Wir geben hier auch den Typ der Funktion `main` an, werden ihn aber erst später diskutieren. Im Unterschied zur *HalloWelt*-Anwendung ist der Typ der Konstante `view` nun `Html Msg` und nicht mehr `Html msg`. Wir verweisen im `Html`-Typ also auf den Typ der Nachrichten, die wir an die Anwendung schicken können.

Wir wollen uns einmal ein sehr einfaches Beispiel für eine Anwendung ansehen. Wir implementieren einen einfachen Zähler, den der*die Nutzer*in hoch- und runterzählen kann.

```
module Counter exposing (main)

import Browser
import Html exposing (Html, text)


type alias Model =
    Int


init : Model
init =
    0


type Msg
```

```
    = Increase
    | Decrease


update : Msg -> Model -> Model
update msg model =
    case msg of
        Increase ->
            model + 1

        Decrease ->
            model - 1


view : Model -> Html Msg
view model =
    text (String.fromInt model)


main : Program () Model Msg
main =
    Browser.sandbox { init = init, view = view, update = update }
```

Da wir einen Zähler implementieren wollen, ist unser Zustand ein *Int*. Initial ist unser Zustand `0`. Um die Nachrichten darzustellen, die ein*e Nutzer*in auswählen kann, definieren wir einen Aufzählungstyp. Die Funktion `update` verarbeitet einen Zustand und eine Nachricht und liefert einen neuen Zustand. Die Funktion `view` liefert zu einem Zustand die HTML-Seite, die den Zustand repräsentiert.

Unserer Anwendung fehlt ein wichtiger Teil, nämlich die Möglichkeit, dass der*die Nutzer*in mit der Anwendung interagiert. Zu diesem Zweck müssen wir nur zwei Knöpfe zu unserer Seite hinzufügen, die die Nachrichten *Increase* und *Decrease* an die Anwendung schicken.

```
view : Model -> Html Msg
view model =
    div []
        [ text (String.fromInt model)
        , button [ onClick Increase ] [ text "+" ]
        , button [ onClick Decrease ] [ text "-" ]
        ]
```

Die Funktion `button` kommt aus dem Modul *Html* und erzeugt einen Knopf in der HTML-Struktur. Das Modul *Html.Events* stellt die Funktion `onClick` zur Verfügung. Wir übergeben der Funktion die Nachricht, die wir bei einem Klick an die Anwendung schicken wollen. Verwendet der*die Nutzer*in den Knopf zum Erhöhen des Zählers, wird die Funktion `update` mit der Nachricht *Increase* und dem aktuellen Zustand

aufgerufen. Nach der Aktualisierung wird die Funktion `view` aufgerufen und die entsprechende HTML-Seite angezeigt.

In diesem einfachen Beispiel können wir bereits den deklarativen Ansatz der Elm-Architektur erkennen. Die Funktion `view` beschreibt, wie ein Modell als HTML-Struktur dargestellt wird. Das heißt, wir beschreiben nur, was dargestellt werden soll, aber nicht wie die konkrete Darstellung durchgeführt wird. Im Kontrast dazu, würde eine sehr klassische JavaScript-Anwendung beschreiben, wie die HTML-Struktur geändert wird. Der entsprechende HTML-Knoten wird aus der HTML-Struktur herausgesucht und den Zähler durch den veränderten Wert ersetzt.

Abbildung 3.1 illustriert noch einmal, wie die Elm-Anwendung kommuniziert, wenn sie mittels *`Browser`*`.sandbox` gestartet wurde.

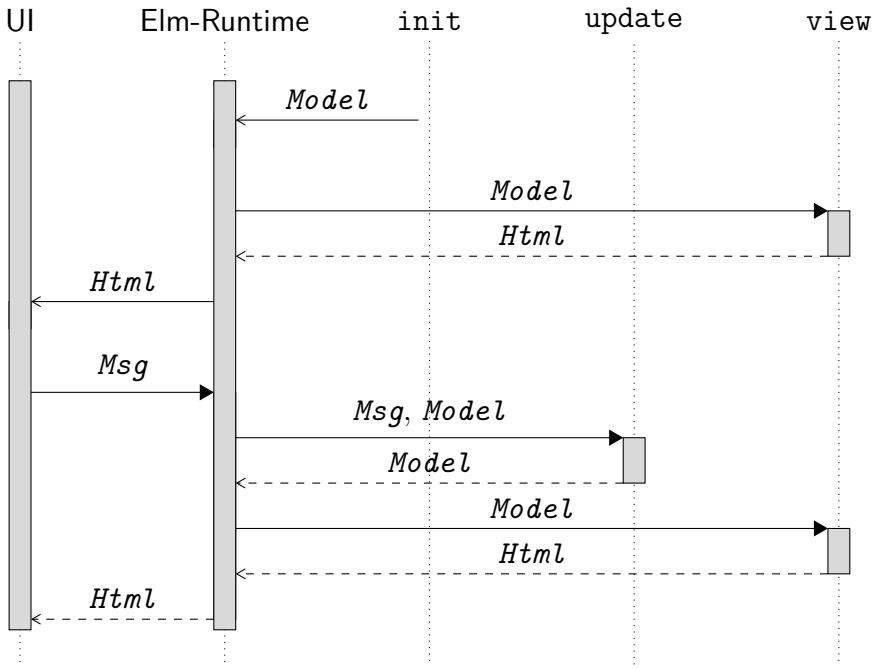


Abbildung 3.1: Kommunikation der Elm-Anwendung

4 Datentypen

In diesem Kapitel wollen wir die Grundlagen für die Definition von Datentypen in Elm einführen. Wir haben bereits eine Reihe von Datentypen kennengelernt, zum Beispiel Listen und Aufzählungstypen. In diesem Kapitel werden wir zum Beispiel lernen, wie man einen Datentyp für Listen in Elm definiert.

4.1 Algebraische Datentypen

An Stelle des Namen Aufzählungstyp verwendet man auch den Namen Summentyp. Dieser Name zeigt auch den Zusammenhang zum Namen algebraische Datentypen auf. Eine Algebra ist in der Mathematik eine Struktur, die eine Addition und eine Multiplikation zur Verfügung stellt. Neben der Addition (dem Summentyp) benötigen wir für einen algebraischen Datentyp also noch eine Multiplikation (den Produkttyp).

In Elm kann man sogenannte Produkttypen definieren, die benannten Paaren entsprechen. So kann man zum Beispiel auf die folgende Weise einen Datentyp für einen Punkt, zum Beispiel auf einer 2D-Zeichenfläche, definieren.

```
type Point
    = Point Float Float
```

Dieser Datentyp ist im Endeffekt nichts anderes als ein benanntes Paar. Auch im Fall von *Point* spricht man von einem Konstruktor. Das heißt, *Point* ist ein Konstruktor des Datentyps *Point*.

Hinter dem Namen des Konstruktors folgt ein Leerzeichen und anschließend folgen, durch Leerzeichen getrennt, die Typen der Argumente des Konstruktors. Im Gegensatz zu Funktionen und Variablen müssen Konstruktoren und Datentypen immer mit einem großen Anfangsbuchstaben beginnen. Das heißt, der Konstruktor *Point* erhält zwei Argumente, die beide den Typ *Float* haben. Um mit Hilfe eines Konstruktors einen Wert zu erzeugen, benutzt man den Konstruktor wie eine Funktion. Das heißt, man schreibt den Namen des Konstruktors und durch Leerzeichen getrennt die Argumente des Konstruktors. Wir können nun zum Beispiel wie folgt einen Punkt definieren.

```
examplePoint : Point
examplePoint =
    Point 2.3 4.2
```

Wie im Fall von Aufzählungstypen kann man auch auf Produkttypen *Pattern Matching* durchführen. Im Fall von Produkttypen kann man mit Hilfe des Pattern Matching nicht nur eine Fallunterscheidung durchführen, sondern auch auf die Inhalte des Konstruktors zugreifen. Die folgende Funktion verschiebt einen Punkt um einen Wert auf der x- und einen Wert auf der y-Achse.

```
translate : Point -> Float -> Float -> Point
translate point dx dy =
    case point of
        Point x y ->
            Point (x + dx) (y + dy)
```

Alternativ können wir zum Beispiel die folgende Funktion definieren, um einen *Point* in einen *String* umzuwandeln.

```
toString : Point -> String
toString point =
    case point of
        Point x y ->
            "(" ++ String.fromFloat x
                ++ ", "
                ++ String.fromFloat y
                ++ ")"
```

In der Definition eines Produkttyps können wir natürlich auch selbstdefinierte Datentypen verwenden. Wir betrachten zum Beispiel folgenden Datentyp, der einen Spieler in einem Spiel modelliert, der einen Namen und eine aktuelle Position hat.

```
type Player
    = Player String Point
```

Als Beispiel können wir nun einen *Player* definieren.

```
examplePlayer :: Player
examplePlayer =
    Player "Player A" (Point 10 100)
```

Wir können nun zum Beispiel eine Funktion definieren, die den Namen eines Spielers liefert.

```
name : Player -> String
name player =
    case player of
        Player n _ ->
            n
```

Der Unterstrich bedeutet, dass wir uns für das entprechende Argument des Konstruktors, hier also der *Point*, nicht interessieren. Wenn wir stattdessen, an die Stelle des Argumentes eines Konstruktors eine Variable schreiben, wird die Variable an den Wert gebunden, der an der ensprechenden Stelle im Konstruktor steht. Im Fall von `name` wird die Variable zum Beispiel an den Namen gebunden, der im *Player* steht.

Als weiteres Beispiel können wir auch eine Funktion zur Umwandlung eines Spielers in einen String schreiben.

```
toString : Player -> String
toString player =
    case player of
        Player pname point ->
            pname ++ " " ++ Point.toString point
```

Im Allgemeinen kann man Summen- und Produkttypen auch kombinieren. Die Kombination aus Summen- und Produkttypen wird als algebraischer Datentyp bezeichnet. Manchmal spricht man bei diesen Datentypen auch von einer *tagged union*. Man spricht von einer *union*, da der algebraische Datentyp eine Vereinigung von mehreren Typen darstellt. Man bezeichnet diese Vereinigung als *tagged*, da durch den Konstruktor markiert wird, um welchen Teil der Vereinigung es sich handelt. Wir können zum Beispiel wie folgt einen Datentyp definieren, der beschreibt, ob ein Spiel unentschieden ausgegangen ist oder ob ein Spieler das Spiel gewonnen hat. Der Konstruktor `Win` modelliert, dass einer der Spieler gewonnen hat. Wenn die Spielrunde unentschieden ausgegangen ist, liefert die Funktion als Ergebnis den Wert `Draw`. Da wir in diesem Fall keine zusätzlichen Informationen benötigen, hat der Konstruktor keine Argumente.

```
type GameResult
    = Win Player
    | Draw
```

4.2 Pattern Matching

Wir haben gesehen, dass man *Pattern Matching* nutzen kann, um Fallunterscheidungen über Zahlen zu treffen. Man kann *Pattern Matching* aber auch nutzen, um die verschiedenen Konstruktoren eines Datentyps zu unterscheiden. Wir können zum Beispiel wie folgt eine Funktion `isDraw` definieren, um zu überprüfen, ob ein Spiel unentschieden ausgegangen ist.

```
isDraw : GameResult -> Bool
isDraw result =
    case result of
        Draw ->
            True

        Win _ ->
            False
```

Diese Funktion liefert `True`, falls das `GameResult` gleich `Draw` ist und `False` andernfalls. Der Unterstrich besagt, dass uns egal ist, was an dieser Stelle in dem Wert steht. Das heißt, mit dem Muster `Win _` sagen wir, diese Regel soll genommen werden, wenn der Wert in `result` ein `Win`-Konstruktor mit einem beliebigen Argument ist. An Stelle des Unterstrichs können wir auch eine Variable verwenden, das heißt, statt `Win _` können wir auch `Win player` schreiben. Wir können zum Beispiel wie folgt eine Funktion definieren, die zu einem Spiel-Ergebnis eine Beschreibung in Form eines `String`s liefert.

```
description : GameResult -> String
description result =
    case result of
        Draw ->
            "Das Spiel ist unentschieden ausgegangen."

        Win player ->
            Player.name player ++ " hat das Spiel gewonnen."
```

In diesem Fall wird die Variable `player` an den Wert vom Typ *`Player`* gebunden, der im Konstruktor *`Win`* steht.

Pattern können auch geschachtelt werden. Das heißt, an Stelle einer Variable, können wir auch wieder ein *Pattern* verwenden. Die folgende Funktion verwendet zum Beispiel ein geschachteltes *Pattern*, um die x-Position eines Spielers zu bestimmen.

```
playerXCoord : Player -> Bool
playerXCoord player =
    case player of
        Player _ (Point x _) ->
            x
```

Als weiteres Beispiel für ein geschachteltes Pattern wollen wir eine Funktion definieren, die einen *`String`* liefert, der beschreibt, wie ein Spiel ausgegangen ist.

```
description : GameResult -> String
description result =
    case result of
        Draw ->
            "Das Spiel ist unentschieden ausgegangen."

        Win (Player name _) ->
            name ++ " hat das Spiel gewonnen."
```

Wenn wir zum Beispiel den Aufruf `description player` in der REPL auswerten[1], erhalten wir das folgende Ergebnis.

```
> description (Win player)
"Spieler A hat das Spiel gewonnen." : String
```

Das heißt, der Aufruf `description (Win player)` hat das Ergebnis

```
"Spieler A hat das Spiel gewonnen."
```

geliefert und dieses Ergebnis ist vom Typ *`String`*.

Ein **`case`**-Ausdruck wird für zwei Aufgaben genutzt. Zum einen führen wir eine Fallunterscheidung über die möglichen Konstruktoren eines Datentyps durch. Zum anderen zerlegen wir Konstruktoren in ihre Einzelteile. Bei Datentypen, die nur einen Konstruktor zur Verfügung stellen, wie etwa der Typ *`Point`*, müssen wir keine Fallunterscheidung

[1] Wobei `player` die oben definierte Konstante ist.

über die verschiedenen Konstuktoren durchführen. Daher kann man ein *Pattern* für Datentypen mit nur einem Konstuktor auch ohne einen **case**-Ausdruck verwenden. Die folgende Funktion liefert zum Beispeil die X-Koordinate eines Punktes.

```
xCoord : Point -> Float
xCoord (Point x _) =
    x
```

4.3 Rekursive Datentypen

Datentypen können auch rekursiv sein. Das heißt, wie eine rekursive Funktion kann ein Datentyp in seiner Definition wieder auf sich selbst verweisen. Wir können zum Beispiel wie folgt einen Datentypen definieren, der Listen mit Integern darstellt. In der funktionalen Programmierung haben sich die Namen *Nil* für eine leere Liste und *Cons* für eine nicht-leere Liste eingebürgert. Das Wort *Nil* ist eine Kurzform des lateinischen Wortes *nihil*, das „nichts“ bedeutet.

```
type IntList
    = Nil
    | Cons Int IntList
```

Wir wollen einmal eine Funktion definieren, die die Länge einer solchen Liste berechnet. Wir können diese Funktion ebenfalls rekursiv definieren, indem wir *Pattern Matching* verwenden.

```
length : IntList -> Int
length list =
    case list of
        Nil ->
            0

        Cons _ restlist ->
            1 + length restlist
```

Als weiteres Beispiel zeigt die folgende Funktion, wie wir die Zahlen in einer Liste aufaddieren können.

```
sum : IntList -> Int
sum list =
    case list of
        Nil ->
            0

        Cons int restlist ->
            int + sum restlist
```

Als nächstes wollen wir eine Funktion definieren, die zu einer Liste eine Liste berechnet, die jedes zweite Element der Originalliste enthält.

```
everySecond : IntList -> IntList
everySecond list =
    case list of
        Nil ->
            Nil

        Cons _ Nil ->
            Nil

        Cons _ (Cons int restlist) ->
            Cons int (everySecond restlist)
```

Als weiteres Beispiel eines rekursiven Datentyps wollen wir uns eine Baumstruktur anschauen. Der folgende Datentyp stellt zum Beispiel einen binären Baum mit ganzen Zahlen in den Knoten und Blättern dar.

```
type IntTree
    = Empty
    | Node IntTree Int IntTree
```

Die folgende Definition gibt einen Wert dieses Typs an.

```
tree : IntTree
tree =
    Node (Node Empty 3 (Node Empty 5 Empty)) 8 Empty
```

Wir können zum Beispiel wie folgt eine Funktion schreiben, die testet, ob ein Eintrag in einem Baum vorhanden ist.

```
find : Int -> IntTree -> Bool
find n tree =
    case tree of
        Empty ->
            False

        Node leftree int righttree ->
            n == int || find n leftree || find n righttree
```

4.4 Records

Da Elm als JavaScript-Ersatz gedacht ist, unterstützt es auch Record-Typen. Wir können zum Beispiel eine Funktion, die für einen Nutzer testet, ob er volljährig ist, wie folgt definieren.

```
hasFullAge : { firstName : String, lastName : String, age : Int } -> Bool
hasFullAge user =
    user.age >= 18
```

Der Ausdruck `user.age` ist eine Kurzform für `.age user`, das heißt, `.age` ist eine Funktion, die einen Wert vom Typ *`User`* nimmt und dessen Alter zurückliefert.

Es ist recht umständlich, den Typ des Nutzers in einem Programm jedes mal anzugeben. Um unser Beispiel leserlicher zu gestalten, können wir das folgende Typsynonym für unseren Record-Typ einführen.

```
type alias User =
    { firstName : String
    , lastName : String
    , age : Int
    }

hasFullAge : User -> Bool
hasFullAge user =
    user.age >= 18
```

Es gibt eine spezielle Syntax, um initial einen Record zu erzeugen.

```
exampleUser : User
exampleUser =
    { firstName = "Max", lastName = "Mustermann", age = 42 }
```

Wir können einen Record natürlich auch abändern. Zu diesem Zweck wird die folgende Schreibweise verwendet.

```
maturing : User -> User
maturing user =
    { user | age = 18 }
```

Da Elm eine rein funktionale Programmiersprache ist, wird hier der Record nicht wirklich abgeändert, sondern ein neuer Record mit anderen Werten erstellt. Wir können das Verändern eines Record-Eintrags und das Lesen eines Eintrags natürlich auch kombinieren. Wir können zum Beispiel die folgende Definition verwenden, um einen Benutzer altern zu lassen.

```
increaseAge : User -> User
increaseAge user =
    { user | age = user.age + 1 }
```

Es ist auch möglich, mehrere Felder auf einmal abzuändern, wie die folgende Funktion illustriert.

```
japanese : User -> User
japanese user =
    { user | firstName = user.lastName, lastName = user.firstName }
```

Zu guter Letzt können wir auch Pattern Matching verwenden, um auf die Felder eines Records zuzugreifen. Zu diesem Zweck müssen wir die Variablen im Pattern nennen wie die Felder des entsprechenden Record-Typs.

```
fullName : User -> String
fullName { firstName, lastName } =
    firstName ++ " " ++ lastName
```

Wenn wir für einen Record ein Typsynonym einführen, können wir auch die Syntax der algebraischen Datentypen nutzen, um einen Record zu erstellen. Das heißt, um einen Wert vom Typ `User` zu erstellen, können wir zum Beispiel auch `User "John" "Doe" 20` schreiben. Dabei gibt die Reihenfolge der Felder in der Definition des Records an, in welcher Reihenfolge die Argumente übergeben werden. Wir werden in Kapitel 6 sehen, dass diese Art der Konstruktion bei der Verwendung einer partiellen Applikation praktisch ist. Diese Verwendung hat allerdings den Nachteil, dass in der Definition des Records die Reihenfolge der Einträge nicht ohne Weiteres ändern können.

5 Polymorphismus

In diesem Kapitel wird das Konzept des parametrischen Polymorphismus vorgestellt. Dieses Konzept wird in anderen Programmiersprachen wie Java und C# auch als Generics bezeichnet. Wie in Programmiersprachen wie Java und C# kann man in Elm Datentypen definieren, die generisch über dem Typ der Elemente sind. Tatsächlich haben Programmiersprachen wie Java und C# dieses Konzept von den funktionalen Programmiersprachen übernommen. In diesem Kapitel wollen wir uns anschauen, wie das Konzept des parametrischen Polymorphismus in Elm umgesetzt ist.

5.1 Polymorphe Datentypen

Häufig möchte man einen Datentyp nicht nur mit einem konkreten Typ verwenden, sondern für verschiedene Typen. Ein Beispiel für einen solchen Datentyp ist der Datentyp *Maybe*. Dieser Datentyp wird genutzt, um anzuzeigen, dass eine Funktion möglicherweise kein Ergebnis liefert und stellt damit eine Art „Ersatz“ der Null-Referenz dar, die in objekt-orientierten Sprachen für diesen Zweck genutzt wird. Der Typ *Maybe* ist wie folgt definiert.

```
type Maybe a
    = Just a
    | Nothing
```

Maybe nimmt ein Argument, das a heißt, und auch als *Typparameter* oder *Typvariable* bezeichnet wird. Typvariablen werden in Elm im Gegensatz zu Typen klein geschrieben. Wenn wir den Datentyp *Maybe* verwenden, können wir für den Typparamenter einen konkreten Typ angeben. Ein Datentyp wie *Maybe*, der noch Typen als Argumente erhält, wird als *Typkonstruktor* bezeichnet. Die folgenden Beispiele definieren Werte von *Maybe*-Typen.

```
m1 : Maybe Int
m1 =
    Just 3


m2 : Maybe Int
m2 =
    Nothing


m3 : Maybe String
```

```
m3 =
    Just "a"
```

Als Beispiel für die Verwendung des Typs *Maybe* wollen wir die Funktion

```
toInt : String -> Maybe Int
```

aus dem Modul *String* betrachten. Falls das Argument der Funktion `toInt` keine ganze Zahl repräsentiert, liefert die Funktion den Wert *Nothing*. Andernfalls erhalten wir ein *Just*, das den Integer enthält, der aus dem String erzeugt wurde. Wir können zum Beispiel die folgende Funktion definieren, die eine Benutzereingabe für einen Monat überprüft.

```
parseMonth : String -> Maybe Int
parseMonth userInput =
    case String.toInt userInput of
        Just n ->
            toValidMonth n

        Nothing ->
            Nothing


toValidMonth : Int -> Maybe Int
toValidMonth month =
    if 1 <= month && month <= 12 then
        Just month

    else
        Nothing
```

Der Aufruf `parseMonth "a"` liefert *Nothing*, da `"a"` durch die Funktion *String*`.toInt` nicht in einen *Int* umgewandelt werden kann. Der Aufruf `parseMonth "-1"` liefert *Nothing*, da `"-1"` zwar eine Zahl darstellt, die Zahl aber nicht zwischen `1` und `12` liegt. Der Aufruf `parseMonth "1"` liefert schließlich als Ergebnis *Just* `1`.

Häufig möchten wir im Fehlerfall noch einen Grund für das Fehlschlagen der Operation zur Verfügung stellen. Für diesen Zweck wird der Datentyp *Result* genutzt. Der Datentyp *Result* ist dabei wie folgt definiert.

```
type Result error value
    = Ok value
    | Err error
```

Der Typkonstruktor *Result* erhält zwei Typparameter. Der erste Typparameter ist dabei der Typ, der im Konstruktor *Err* gespeichert wird. Der zweite Typparameter ist der Typ, der im Konstruktor *Ok* gespeichert wird.

Wir wollen den Datentyp *Result* nutzen, um in der Funktion `parseMonth` einen Grund zu liefern, warum die Konvertierung fehlgeschlagen ist. Hierbei bedeutet der

Typ *Result String Int*, dass wir im Erfolgsfall den Konstruktor *Ok* erhalten und sein Argument den Typ *Int* hat. Im Fehlerfall erhalten wir den Konstruktor *Err* und sein Argument ist vom Typ *String*.

```
parseMonth : String -> Result String Int
parseMonth userInput =
    case String.toInt userInput of
        Just n ->
            toValidMonth n

        Nothing ->
            Err ("Error parsing '" ++ userInput ++ "'")


toValidMonth : Int -> Result String Int
toValidMonth month =
    if 1 <= month && month <= 12 then
        Ok month

    else
        Err ("Invalid month " ++ String.fromInt month)
```

Der Aufruf `parseMonth "a"` liefert in dieser Implementierung

`Err "Error parsing 'a'"`.

Das heißt, wir erhalten nicht nur die Information, dass die Verarbeitung fehlgeschlagen ist, sondern auch, warum die Verarbeitung fehlgeschlagen ist.

Wir verwenden an dieser Stelle aus Einfachheit einen Wert vom Typ *String* für die Fehlermeldung. In einer Anwendung sollte man für die Modellierung von Fehlern immer auf einen strukturierten Datentyp wie den folgenden zurückgreifen. Andernfalls ist es zum Beispiel nicht sinnvoll möglich, eine Fallunterscheidung über den Fehler durchzuführen, der aufgetreten ist oder die Beschreibungen der Fehler zu internationalisieren.

```
type Error
    = ParseError String
    | InvalidMonth Int
```

Der *ParseError* beschreibt, dass der *String*, den wir eingelesen haben, bereits keine Zahl repräsentiert und der Konstruktor *InvalidMonth* beschreibt, dass die Zahl kein valider Monat war.

Da der Datentyp *Result* polymorph im Typ des Fehlers ist, können wir den Datentyp *Result* auch mit unserem Datentyp *Error* verwenden.

```
parseMonth : String -> Result Error Int
parseMonth userInput =
    case String.toInt userInput of
        Just n ->
```

```
            toValidMonth n

        Nothing ->
            Err (ParseError userInput)


toValidMonth : Int -> Result Error Int
toValidMonth month =
    if 1 <= month && month <= 12 then
        Ok month

    else
        Err (InvalidMonth month)
```

Der Aufruf `parseMonth "a"` liefert in dieser Implementierung `Err (ParseError "a")`.

Im Kontext von polymorphen Datentypen wollen wir uns auch noch Tupel anschauen. Neben den benannten Paaren, stellt Elm auch ganz klassische Paare zur Verfügung. Im Grunde handelt es sich dabei auch um algebraische Datentypen, nur dass die Paare so wie die Listen eine spezielle Syntax nutzen. Die Einträge eines Paares werden durch ein Komma getrennt und das Paar wird durch Klammern umschlossen. Das heißt, der Ausdruck `(1, False)` erzeugt zum Beispiel ein Paar, bei dem die erste Komponente den Wert `1` enthält und die zweite Komponente den booleschen Wert `False`. Der Typkonstruktor für Paare wird genau so geschrieben wie der Konstruktor für Paare und ist über zwei Typvariablen polymorph, nämlich dem Typ der ersten Komponente und dem Typ der zweiten Komponente. Das heißt, der Typ des Wertes `(1, False)` ist `(Int, Bool)`.

Wie bei jedem anderen algebraischen Datentyp kann man *Pattern Matching* auch für Paare verwenden. Als Beispiel betrachten wir die Funktion

```
uncons : String -> Maybe ( Char, String )
```

aus dem Modul `String`. Mit Hilfe dieser Funktion kann man einen `String` in das erste Zeichen und den Rest des *Strings* zerlegen. Die Funktion liefert `Nothing`, falls wir sie auf einen leeren `String` anwenden.

Mit Hilfe dieser Funktion können wir zum Beispiel wie folgt eine Funktion definieren, die alle Zeichen in einer Zeichenkette in Großbuchstaben verwandelt. Die Funktion `cons : Char -> String -> String` hängt ein Zeichen vorne an eine Zeichenkette.

```
toUpper : String -> String
toUpper str =
    case uncons str of
        Nothing ->
            ""

        Just ( char, rest ) ->
            cons (Char.toUpper char) (toUpper str)
```

Bei Datentypen wie einem Paar, die nur einen Konstruktor haben, kann man statt eines *Pattern Matchings* mit Hilfe eines **case**-Ausdrucks auch direkt in einer Funktionsdefinition *Pattern Matching* durchführen.

```
bar : ( Int, Int ) -> Int
bar ( x, y ) = x + y
```

Wir können diese Form des *Pattern Matching* wie folgt auch in **let**-Ausdrücken nutzen.

```
let
  ( x, y ) = pair
in
```

Neben Paaren bietet Elm auch Tupel anderer Stelligkeiten. Tupel kommen selten zum Einsatz und sollten nur von sehr allgemein verwendtbaren Bibliotheksfunktionen genutzt werden, da ein Tupel sehr wenig Dokumentationscharakter hat. Daher bietet sich als Alternative für ein Tupel fast immer ein algebraischer Datentyp oder ein Record an. Einen Sonderfall eines Tupels stellt das nullstellige Tupel `()` dar, dessen Typ man ebenfalls als `()` schreibt. Wir werden später Anwendungsfälle für diesen Datentyp kennenlernen.

Als weiteres Beispiel für einen polymorphen Datentyp wollen wir uns einen Listen-Datentyp anschauen, der nicht nur Zahlen enthalten kann, sondern Werte eines beliebigen Typs. Der folgende Datentyp definiert einen polymorphen Listendatentyp.

```
type List a
    = Nil
    | Cons a (List a)
```

Hierbei ist vor allem zu beachten, dass wir auch bei der rekursiven Verwendung den Typparameter `a` übergeben müssen, da *List* ein Typkonstruktor ist und somit ein Argument verlangt. Wir geben damit an, dass der Rest der Liste Elemente vom gleichen Typ wie die bisherige Liste enthält.

Mit Hilfe dieses Typs können wir die folgenden Werte definieren.

```
clist1 : List Int
clist1 =
    Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))


clist2 : List String
clist2 =
    Cons "a" (Cons "z" (Cons "y" Nil))


clist3 : List Bool
clist3 =
    Cons False (Cons True (Cons True (Cons False Nil)))
```

Der Listendatentyp ist in Elm genau definiert wie der Datentyp `List`, verwendet aber eine spezielle Syntax. Die folgende Definition liefert einen Syntaxfehler, illustriert aber den Listendatentyp, wie er in Elm definiert ist.

```
type List a
    = []
    | a :: List a
```

Das heißt, der Konstruktor für die leere Liste verwendet in Elm nicht den Namen `Nil`, sondern die Zeichenfolge `[]`. Der Konstruktor, um ein Element vorne an eine Liste anzuhängen, heißt außerdem nicht `Cons`, sondern `::` und wird infix verwendet. Das heißt, der Konstruktor `::` wird zwischen seine Argumente geschrieben. Alle anderen Konstruktoren werden vor ihre Argumente geschrieben. Bei Funktionen kennen wir auch beide Varianten, so wird eine selbstdefinierte Funktion vor ihre Argumente geschrieben, also zum Beispiel `f 1 2`. Die Funktion für die Addition wird aber zum Beispiel auch infix verwendet, also zum Beispiel `1 + 2`.

Mit Hilfe des vordefinierten Listendatentyps können wir wie folgt eine Liste definieren.

```
list1 : List Int
list1 =
    1 :: (2 :: (3 :: (4 :: [])))
```

Der Operator `::` ist rechts-assoziativ. Wir können die Klammern bei der Definition einer Liste also auch weglassen.

```
list2 : List String
list2 =
    "a" :: "z" :: "y" :: []
```

Wir haben in der Einleitung bereits eine Kurzschreibweise für konkrete Listen kennengelernt. Diese Kurzschreibweise stellt nur syntaktischen Zucker[1] für die obige Schreibweise dar. Daher kann man diese Kurzschreibweise auch in *Pattern* verwenden.

```
list3 : List Bool
list3 =
    [ False, True, True, False ]
```

So wie eine Liste definiert wird, die polymorph über dem Typ der Elemente ist, kann auch ein Baum-Datentyp definiert werden, der polymorph über dem Typ der Elemente ist.

```
type Tree a
    = Empty
    | Node (Tree a) a (Tree a)
```

Wir können dann wie folgt einen Baum mit ganzen Zahlen definieren.

[1] `https://de.wikipedia.org/wiki/Syntaktischer_Zucker`

```
tree1 : Tree Int
tree1 =
    Node (Node Empty 3 (Node Empty 5 Empty)) 8 Empty
```

Wir können den gleichen Datentyp aber auch nutzen, um einen Baum zu definieren, der Werte vom Typ `Maybe String` enthält.

```
tree2 : Tree (Maybe String)
tree2 =
    Node Empty (Just "a") (Node Empty Nothing Empty)
```

5.2 Polymorphe Funktionen

Wie wir polymorphe Datentypen definieren können, können wir auch polymorphe Funktionen definieren. Als einfachstes Beispiel wollen wir die Funktion `identity` aus dem Modul `Basics` betrachten.

```
identity : a -> a
identity x = x
```

Der Typ `a` in der Definition der Funktion `identity` wird als Typvariable bezeichnet. Alle Typvariablen in einem Funktionstyp sind implizit allquantifiziert. Das heißt, der Typ `a -> a` steht eigentlich für den Typ `forall a. a -> a`. Das heißt, die Funktion hat für alle möglichen Typen `tau`, den Typ `tau -> tau`. Wenn wir die Funktion `identity` verwenden, wählen wir implizit einen konkreten Typ, den wir für die Typvariable `a` einsetzen. Wenn wir zum Beispiel die Anwendung `identity "a"` betrachten, dann wählt der Compiler für die Typvariable `a` den Typ `String` und die konkrete Verwendung der Funktion `identity` erhält den Typ `String -> String`.

Als weiteres Beispiel wollen wir uns noch eine Funktion auf dem Datentyp `Result` anschauen. Die folgende Funktion kann genutzt werden, um einen *Default*-Wert für die Verwendung einer fehlgeschlagenen Berechnung anzugeben. Das heißt, falls die Berechnung erfolgreich war, verwenden wir den Wert, der im `Result`-Typ zur Verfügung steht und für den Fehlerfall geben wir einen *Default*-Wert an.

```
withDefault : a -> Result x a -> a
withDefault def result =
    case result of
        Ok a ->
            a

        Err _ ->
            def
```

Im Unterschied zur Funktion `identity`, ist die Funktion `withDefault` über zwei Typparameter parametrisiert. Wenn wir die Funktion `withDefault` anwenden, wählen wir implizit konkrete Typen für diese Typparameter.

Als Beispiel betrachten wir den Aufruf `withDefault 1 (parseMonth "a")`. Der Ausdruck `parseMonth "a"` hat den Typ `Result Error Int`. Das heißt, bei diesem Aufruf

wählen wir für die Typvariable `x` den Typ *`Error`* und für die Typvariable `a` den Typ *`Int`*. Dadurch muss das erste Argument von `withDefault` ebenfalls den Typ *`Int`* haben. Der Aufruf `withDefault 1 (parseMonth "a")` ist also typkorrekt. Da wir für die Typvariable `a` den Typ *`Int`* gewählt haben, wissen wir außerdem, dass der Aufruf einen *`Int`* als Ergebnis liefert.

Wenn wir den Aufruf `withDefault` *`False`* `(parseMonth "a")` in der REPL ausführen, erhalten wir einen Fehler.

```
-- TYPE MISMATCH ------------------------------------------------------- REPL

The 2nd argument to ‘withDefault‘ is not what I expect:

4|   withDefault False (parseMonth "a")
                       ^^^^^^^^^^^^^^

This ‘parseMonth‘ call produces:

    Result Error Int

But ‘withDefault‘ needs the 2nd argument to be:

    Result Error Bool

Hint: I always figure out the argument types from left to right. If an argument
is acceptable, I assume it is "correct" and move on. So the problem may actually
be in one of the previous arguments!

Hint: Elm does not have "truthiness" such that ints and strings and lists are
automatically converted to booleans. Do that conversion explicitly!
```

Wenn wir eine polymorphe Funktion verwenden, wählen wir für die Typvariablen konkrete Typen. Wir müssen aber für die gleiche Typvariable immer die gleiche Wahl treffen. Das heißt, die drei Vorkommen von `a` in der Signatur von `withDefault` müssen alle durch den gleichen konkreten Typ ersetzt werden. Wenn wir die Funktion `withDefault` auf die Argumente *`False`* und `parseMonth "a"` anwenden, würden wir das erste `a` aber durch *`Bool`* und das zweite `a` durch *`Int`* ersetzen. Dies ist nicht erlaubt und wir erhalten einen Fehler. Die Fehlermeldung schlägt vor, dass wir für beide Vorkommen den Typ *`Bool`* wählen und erwartet daher als zweites Argument einen Wert vom Typ *`Result Error Bool`*.

Da die vordefinierten Listen in Elm polymorph sind, können wir auch Funktionen definieren, die auf allen Arten von Listen arbeiten, unabhängig davon, welchen Typ die Elemente der Liste haben. Wir schauen uns einmal die Längenfunktion auf Listen an, die wie folgt definiert ist.

```
length : List a -> Int
length list =
    case l of
        [] ->
            0
```

```
_ :: restlist ->
    1 + length restlist
```

So wie wir den Konstruktor für eine nicht-leere Liste infix schreiben, so schreiben wir auch das *Pattern* für die nicht-leere Liste infix. Das heißt, das Muster `_ :: restlist` passt nur, wenn die Liste nicht leer ist. Außerdem wird die Variable `restlist` an den Rest der Liste gebunden. Das heißt, die Variable `restlist` enthält die Liste `list` aber ohne das erste Element der Liste.

Da das Argument den Typ *`List`* `a` hat, können wir diese Funktion mit jeder Art von Liste aufrufen. Wenn wir die Funktion mit einem Wert vom Typ *`List Bool`* aufrufen, wird die Typvariable `a` zum Beispiel durch den konkreten Typ *`Bool`* ersetzt. Wenn wir `length` mit einem Argument vom Typ *`List`* `(`*`Maybe String`*`)` aufrufen, wird die Typvariable `a` durch den konkreten Typ *`Maybe String`* ersetzt.

6 Funktionen höherer Ordnung

In diesem Kapitel wollen wir uns intensiver mit dem Thema Rekursion auseinandersetzen. Wie wir bereits gesehen haben, kann man mit Hilfe von Rekursion Funktionen in Elm definieren. Wenn man sich etwas länger mit rekursiven Funktionen beschäftigt, wird aber schnell klar, dass es unter diesen rekursiven Funktionen wiederkehrende Muster gibt. Wir wollen uns hier einige dieser Muster anschauen.

Auch in imperativen Sprachen verwendet man immer wieder gleiche Formen von Schleifen. In imperativen Sprachen werden diese wiederkehrenden Muster aber häufig nicht abstrahiert, da zum Beispiel die Programmiersprachenkonzepte nicht zur Verfügung stehen, um diese Abstraktion elegant zu verwenden.

6.1 Wiederkehrende rekursive Muster

Nehmen wir an, wir haben eine Liste von Zahlen und wollen alle Zahlen inkrementieren. Wir können wie folgt eine Funktion definieren, die diese Aufgabe übernimmt.

```
incList : List Int -> List Int
incList list =
    case list of
        [] ->
            []

        i :: is ->
            i + 1 :: incList is
```

Wir müssen den rekursiven Aufruf von `incList` an dieser Stelle nicht klammern, da, wie wir bereits gelernt haben, die Anwendung einer Funktion stärker bindet als ein Infix-Operator und damit `i + 1 :: incList is` implizit als `i + 1 :: (incList is)` geklammert ist. Wir müssen auch die Anwendungen von `+` und `::` nicht klammern, da der Operator `::` Präzedenz 5 und `+` Präzedenz 6 hat. Daher ist die rechte Seite der zweiten Regel implizit als `(i + 1) :: (incList is)` geklammert.

Nun nehmen wir an, wir möchten in einer Liste von Zahlen alle Zahlen quadrieren. Diese Aufgabe können wir wie folgt lösen.

```
squareList : List Int -> List Int
squareList list =
    case list of
        [] ->
            []
```

```
        i :: is ->
            i * i :: squareList is
```

Zu guter Letzt nehmen wir an, wir haben eine Liste von Zeichenketten und wollen von jedem *String* die Länge berechnen. Diese Aufgabe können wir wie folgt lösen.

```
lengthList : List String -> List Int
lengthList list =
    case list of
        [] ->
            []

        str :: strs ->
            String.length str :: lengthList strs
```

Diese drei Funktionen unterscheiden sich nur leicht voneinander. Ein Ziel funktionaler Programmierer ist es, solche Duplikation von Code zu vermeiden.

Die Funktionen `incList`, `squareList` und `lengthList` durchlaufen alle eine Liste von Elementen und unterscheiden sich nur in der Operation, die sie auf die Listenelemente anwenden. Wir wollen einmal diese unterschiedlichen Operationen als Funktionen definieren.

```
inc : Int -> Int
inc i = i + 1

square : Int -> Int
square i = i * i
```

Mit Hilfe dieser Funktionen werden die Gemeinsamkeiten der Funktionen `incList`, `squareList` und `lengthList` noch deutlicher.

```
incList : List Int -> List Int
incList list =
    case list of
        [] ->
            []

        i :: is ->
            inc i :: incList is


squareList : List Int -> List Int
squareList list =
    case list of
        [] ->
            []

        i :: is ->
```

```
            square i :: squareList is


lengthList : List String -> List Int
lengthList list =
    case list of
        [] ->
            []

        str :: strs ->
            String.length str :: lengthList strs
```

Das heißt, die drei Definitionen unterscheiden sich nur durch die Funktion, die jeweils verwendet wird. Allerdings unterscheiden sich auch die Typen der Funktionen, so hat die Funktion in den ersten beiden Beispielen den Typ `Int -> Int` und im letzten Beispiel `String -> Int`.

Wir können die Teile, die die drei Funktionen sich teilen, in eine Funktion extrahieren. Man nennt die Funktion, die wir dadurch erhalten, ein `map`. Diese Funktion erhält die Operation, die auf die Elemente der Liste angewendet wird, als Argument übergeben.

In Elm sind Funktionen *first class citizens*. Das heißt, Funktionen können wie andere Werte, etwa Zahlen oder Zeichenketten als Argumente und Ergebnisse in Funktionen verwendet werden. Außerdem können Funktionen in Datenstrukturen stecken. Die Funktion `map` hat die folgende Form.

```
map : (a -> b) -> List a -> List b
map func list =
    case list of
        [] ->
            []

        x :: xs ->
            func x :: map func xs
```

Mit Hilfe der Funktion `map` können wir die Funktionen `incList`, `squareList` und `lengthList` nun wie folgt definieren.

```
incList : List Int -> List Int
incList list =
    map inc list

squareList : List Int -> List Int
squareList list =
    map square list

lengthList : List Int -> List Int
lengthList list =
    map String.length list
```

Man nennt eine Funktion, die eine andere Funktion als Argument erhält, eine Funktion höherer Ordnung (*higher-order function*).

Neben dem Rekursionsmuster für `map`, wollen wir an dieser Stelle noch ein weiteres Rekursionsmuster vorstellen. Stellen wir uns vor, dass wir aus einer Liste von Zeichenketten die Liste aller Zeichenketten mit einer geraden Länge extrahieren möchten. Dazu können wir die folgende Funktion definieren.

```
keepEvenLength : List String -> List String
keepEvenLength list =
    case list of
        [] ->
            []

        str :: strs ->
            if modBy 2 (String.length str) == 0 then
                str :: keepEvenLength strs
            else
                keepEvenLength strs
```

Als nächstes nehmen wir an, wir wollen aus einer Liste mit allen Zahlen alle Zahlen raussuchen, die kleiner sind als `5`.

```
keepLessThan5 : List Int -> List Int
keepLessThan5 list =
    case list of
        [] ->
            []

        x :: xs ->
            if x < 5 then
                x :: keepLessThan5 xs
            else
                keepLessThan5 xs
```

Wir können diese beiden Funktionen wieder mit Hilfe einer Funktion höherer Ordnung definieren.

```
filter : (a -> Bool) -> List a -> List a
filter pred list =
    case list of
        [] ->
            []

        x :: xs ->
            if pred x then
                x :: filter pred xs
            else
                filter pred xs
```

Dieses Mal übergeben wir eine Funktion, die angibt, ob ein Element in die Ergebnisliste kommt oder nicht. Man bezeichnet eine solche Funktion, die einen booleschen Wert liefert auch als Prädikat.

Funktionen höherer Ordnung wie `map` und `filter` ermöglichen es, deklarativeren Code zu schreiben. Bei der Verwendung dieser Funktionen gibt der Entwickler nur an, was berechnet werden soll, aber nicht wie diese Berechnung durchgeführt wird. Wie die Berechnung durchgeführt wird, wird dabei einfach durch die Abstraktionen festgelegt. Diese Form der deklarativen Programmierung ist in jeder Programmiersprache möglich, die es erlaubt Funktionen als Argumente zu übergeben. Heutzutage bietet fast Programmiersprache dieses Sprachfeature. Daher haben Abstraktionen wie `map` und `filter` inzwischen auch Einzug in die meisten Programmiersprachen gehalten. Im Folgenden sind einige Programmiersprachen aufgelistet, die diese Abstraktionen ähnlich zu `map` und `filter` zur Verfügung stellen.

Java Das Interface `java.util.stream.Stream` stellt die folgenden beiden Methoden zur Verfügung.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
Stream<T> filter(Predicate<? super T> predicate)
```

C# LINQ (Language Integrated Query)[1] ist eine Technologie der .NET-Platform, um Anfragen elegant zu formulieren. Die folgenden beiden Methoden, die von LINQ zur Verfügung gestellt werden, entsprechen in etwa den Funktionen `map` und `filter`.

```
IEnumerable<TResult> Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)
IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

JavaScript Der Prototyp `Array` bietet Methoden `map` und `filter`, welche die Funktionalität von `map` und `filter` auf Arrays bieten.

Elm Elm stellt die Funktionen `map` und `filter` im Modul *`List`* zur Verfügung.

6.2 Anonyme Funktionen

Es ist recht umständlich extra die Funktionen `inc` und `square` zu definieren, nur, um sie in den Definitionen von `incList` und `squareList` zu verwenden. Stattdessen kann man anonyme Funktionen verwenden. Anonyme Funktionen sind einfach Funktionen, die keinen Namen erhalten. Die Funktion `incList` kann zum Beispiel wie folgt mit Hilfe einer anonymen Funktion definiert werden.

```
incList : List Int -> List Int
incList list =
    map (\x -> x + 1) list
```

Dabei stellt der Ausdruck `\x -> x + 1` die anonyme Funktion dar. Analog können wir die Funktion `squareList` mit Hilfe einer anonymen Funktion wie folgt definieren.

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/

```
squareList : List Int -> List Int
squareList list =
    map (\x -> x * x) list
```

Annonyme Funktionen, auch als Lambda-Ausdrücke bezeichnet, starten mit dem Zeichen `\` und listen dann eine Reihe von Argumenten auf, nach den Argumenten folgen die Zeichen `->` und schließlich die rechte Seite der Funktion. Das heißt, der Ausdruck `\x y -> x * y` definiert zum Beispiel eine Funktion, die ihre beiden Argumente multipliziert. Ein Lambda-Ausdruck der Form `\x y -> e` entspricht dabei der folgenden Funktionsdefinition.

```
f x y = e
```

Der einzige Unterschied ist, dass wir die Funktion nicht verwenden, indem wir ihren Namen schreiben, sondern indem wir den gesamten Lambda-Ausdruck angeben. Während wir `f` zum Beispiel auf Argumente anwenden, indem wir `f 1 2` schreiben, wenden wir den Lambda-Ausdruck an, indem wir `(\x y -> e) 1 2` schreiben.

Als weiteres Beispiel wollen wir eine Lambda-Funktion nutzen, um ein Prädikat zu definieren. Dazu betrachten wir noch einmal die Funktion `filter`. Wenn wir zum Beispiel aus einer Liste von Zeichenketten extrahieren möchten, deren Länge gerade ist, können wir diese Anwendung wie folgt definieren.

```
keepEvenLength : List String -> List String
keepEvenLength list =
    filter (\str -> modBy 2 (String.length str) == 0) list
```

6.3 Ge*curry*te Funktionen

Um Funktionen höherer Ordnung in vollem Umfang nutzen zu können, müssen wir uns eine grundlegende Eigenschaft von Funktionen in Elm anschauen, die wir bisher unter den Tisch gekehrt haben. Dazu schauen wir uns noch einmal die Definition von mehrstelligen Funktionen an, die wir in Abschnitt 2.4.3 eingeführt haben.

```
cart : Int -> Float -> String
cart quantity price =
    "Summe (" ++ items quantity ++ "): " ++ String.fromFloat price
```

Wir haben dabei gelernt, dass man zwischen zwei Argumente immer einen Pfeil schreiben muss, wir haben aber bisher nicht diskutiert warum. In einer Programmiersprache wie Java würden wir die Funktion eher wie folgt definieren.

```
cartP : ( Int, Float ) -> String
cartP ( quantity, price ) =
    "Summe (" ++ items quantity ++ "): " ++ String.fromFloat price
```

Die Funktion `cart` nennt man die ge*curry*te Variante und die Funktion `cartP` die unge*curry*te Variante. Die Funktion `cart` nimmt zwar auf den ersten Blick zwei Argumente, wir können den Typ der Funktion `add` aber auch anders angeben. Die Schreibweise

`Int -> Float -> String` steht eigentlich für den Typ `Int -> (Float -> String)`, das heißt, der Typkonstruktor `->` ist rechts-assoziativ. Das heißt, `cart` ist eine Funktion, die einen Wert vom Typ `Int` nimmt und eine Funktion vom Typ `Float -> String` liefert. Während der Funktionspfeil rechtsassoziativ ist, ist die Anwendung einer Funktion in Elm linksassoziativ. Das heißt, die Anwendung `(cart 4) 2.23` steht eigentlich für `(cart 4) 2.23`. Wir wenden also zuerst die Funktion `cart` auf das Argument `4` an. Wir erhalten dann eine Funktion, die noch einen `Float` als Argument erwartet. Diese Funktion wenden wir dann auf `2.23` an und erhalten schließlich einen `String`.

Die Idee, Funktionen mit mehreren Argumenten als Funktion zu repräsentieren, die ein Argument nimmt und eine Funktion liefert, wird als *Currying* bezeichnet. *Currying* ist nach dem amerikanischen Logiker Haskell Brooks Curry[2] benannt (1900–1982), nach dem auch die Programmiersprache Haskell benannt ist.

Die Definition von `cart` ist im Grunde nur eine vereinfachte Schreibweise der folgenden Definition.

```
cartL : Int -> Float -> String
cartL =
    \quantity ->
        \price ->
            "Summe ("
                ++ items quantity
                ++ "): "
                ++ String.fromFloat price
```

In dieser Form der Definition ist ganz explizit dargestellt, dass `cartL` eine Funktion ist, die ein Argument `quantity` nimmt und als Ergebnis wiederum eine Funktion liefert. Um Schreibarbeit zu reduzieren, entsprichen alle Definitionen, die wir in Elm angeben, im Endeffekt diesem Muster. Wir können die Funktionen aber mit der Kurzschreibweise von `cart`, die auf die Verwendung der Lambda-Funktionen verzichtet, definieren.

Mit Hilfe der Definition `cartL` können wir noch einmal illustrieren, dass die Funktionsanwendung linksassoziativ ist.

```
cartL 4 2.23
=
(cartL 4) 2.23
=
(\quantity ->
    \price ->
        "Summe (" ++ items quantity ++ "): " ++ String.fromFloat price
)
    4
    2.23
=
(\price ->
    "Summe (" ++ items 4 ++ "): " ++ String.fromFloat price
```

[2] https://en.wikipedia.org/wiki/Haskell_Curry

```
)
    2.23
=
"Summe (" ++ items 4 ++ "): " ++ String.fromFloat 2.23
```

6.4 Partielle Applikation

Mit der ge*curry*ten Definition von Funktionen gehen zwei wichtige Konzepte einher. Das erste Konzept wird partielle Applikation oder partielle Anwendung genannt. Funktionen in der ge*curry*ten Form lassen sich sehr leicht partiell applizieren. Applikation ist der Fachbegriff für das Anwenden einer Funktion auf konkrete Argumente. Eine partielle Applikation ist die Anwendung einer Funktion auf eine Anzahl von konkreten Argumenten, so dass der Anwendung noch weitere Argumente fehlen. Um zu illustrieren, was eine partielle Anwendung bedeutet, betrachtet wird die Anwendung von `cartL` auf das Argument `4`.

```
cartL 4
=
(\quantity ->
    \price ->
        "Summe (" ++ items quantity ++ "): " ++ String.fromFloat price
)
    4
=
\price ->
      "Summe (" ++ items 4 ++ "): " ++ String.fromFloat price
```

Das heißt, wenn wir die Funktion `cartL` partiell auf das Argument `4` anwenden, erhalten wir eine Funktion, die noch den Preis erwartet und einen Text liefert, der vier Gegenstände enthält. Wir können die Funktion `add` genau auf diese Weise partiell anwenden. Wir betrachten das folgende Beispiel.

```
items : List String
items =
    List.map (cart 4) [ 2.23, 1.99, 9.99 ]
```

Die partielle Applikation `cart 4` nimmt noch ein weiteres Argument, nämlich den Preis. Daher können wir sie mit Hilfe von `map` auf alle Elemente einer Liste anwenden. Wir erhalten dann die Beschreibungen von Einkaufswagen, die alle jeweils vier Elemente enthalten und unterschiedliche Preise haben.

6.5 Piping

Funktionen höherer Ordnung haben viele Verwendungen. Wir wollen uns hier noch eine Anwendung anschauen, die sich recht stark von Funktionen wie `map` und `filter` unterscheidet. Wir betrachten dazu folgendes Beispiel. Wir haben eine Liste von Zahlen

`list`, aus dieser wollen wir die geraden Zahlen filtern, dann wollen wir die verbleibenden Zahlen quadrieren und schließlich die Summe aller Zahlen bilden. Wir können diese Funktionalität wie folgt implementieren.

```
sumOfEvenSquares : List Int -> Int
sumOfEvenSquares list =
    List.sum (List.map square (List.filter (\x -> modBy 2 x == 0) list))
```

Die Verarbeitungsschritte müssen dabei in umgekehrter Reihenfolge angegeben werden. Das heißt, wir geben zuerst den letzten Verarbeitungsschritt an. Elm stellt einen Operator `(|>) : a -> (a -> b) -> b` zur Verfügung mit dessen Hilfe wir die Reihenfolge der Verarbeitungsschritte umkehren können. Wir können die Funktion mit Hilfe dieses Operators wie folgt definieren.

```
sumOfEvenSquares : List Int -> Int
sumOfEvenSquares list =
    list
        |> List.filter (\x -> modBy 2 x == 0)
        |> List.map square
        |> List.sum
```

Aus Gründen der Lesbarkeit wird eine solche Sequenz von Verarbeitungsschritten häufig wie oben aufgeführt eingerückt. Man spricht in diesem Zusammenhang auch von *piping*.

Hinter dem Operator `(|>)` steckt die folgende einfache Definition.

```
(|>) : a -> (a -> b) -> b
(|>) x f =
  f x
```

Das heißt, `(|>)` nimmt einfach das Argument und eine Funktion und wendet die Funktion auf das Argument an. Neben dieser Definition enthält die Elm-Implementierung noch die folgende Angabe.

```
infixl 0 |>
```

Das heißt, der Operator hat die Präzedenz `0` und ist links-assoziativ. Neben `(|>)` stellt Elm auch einen Operator `(<|) : (a -> b) -> a -> b` zur Verfügung.

6.6 Eta-Reduktion und -Expansion

Mit der ge*curry*ten Schreibweise geht noch ein weiteres wichtiges Konzept einher, die Eta-Reduktion bzw. die Eta-Expansion. Dies sind die wissenschaftlichen Namen für Umformungen eines Ausdrucks. Bei der Reduktion lässt man Argumente einer Funktion weg und bei der Expansion fügt man Argumente hinzu. In Abschnitt 6.1 haben wir die Funktion `map` mittels `map inc list` auf die Funktion `inc` und die Liste `list` angewendet. Wenn wir eine Lambda-Funktion verwenden, können wir den Aufruf aber auch als `map (\x -> inc x) list` definieren. Diese beiden Aufrufe verhalten sich exakt gleich. Den Wechsel von `\x -> inc x` zu `inc` bezeichnet man als Eta-Reduktion.

Den Wechsel von `inc` zu `\x -> inc x` bezeichnet man als Eta-Expansion. Ganz allgemein kann man durch die Anwendung der Eta-Reduktion einen Ausdruck der Form `\x -> f x` in `f` umwandeln, wenn `f` eine Funktion ist, die mindestens ein Argument nimmt. Durch die Eta-Expansion kann man einen Ausdruck der Form `f` in `\x -> f x` umwandeln, wenn `f` eine Funktion ist, die mindestens ein Argument nimmt.

Das Konzept der Eta-Reduktion und -Expansion lässt sich aber nicht nur auf Lambda-Funktionen sondern ganz allgemein auf die Definition von Funktionen anwenden. Als Beispiel betrachten wir noch einmal die folgende Definition aus Abschnitt 6.1.

```
incList : List Int -> List Int
incList list =
    map inc list
```

In Abschnitt 6.3 haben wir gelernt, dass diese Definition nur eine Kurzform für die folgende Definition ist.

```
incList : List Int -> List Int
incList =
    \list -> map inc list
```

Durch Eta-Reduktion können wir diese Definition jetzt zur folgenden Definition abändern.

```
incList : List Int -> List Int
incList =
    map inc
```

Das heißt, wenn wir eine Funktion definieren und diese Funktion ruft nur eine andere Funktion mit dem Argument auf, dann können wir dieses Argument durch die Anwendung von Eta-Reduktion auch weglassen.

Anders ausgedrückt stellen die beiden Varianten von `incList` einfach unterschiedliche Sichtweisen auf die Definition einer Funktion dar. In der Variante mit dem expliziten Argument `list` wird eine Funktion definiert, indem beschrieben wird, was die Funktion mit ihrem Argument macht. In der Variante ohne explizites Argument `list` wird eine Funktion definiert, indem eine Funktion als partielle Applikation einer anderen Funktion definiert wird.

7 Modellierung der Elm-Architektur

Nachdem wir uns ein paar Grundlagen erarbeitet haben, wollen wir ein paar Aspekte der Implementierung der Elm-Architektur näher betrachten. Zuerst einmal wollen wir den Typ der Funktion `sandbox` diskutieren, die wir verwendet haben, um eine Elm-Anwendung zu erstellen. Die Funktion hat die folgende Signatur.

```
sandbox :
    { init : model
    , view : model -> Html msg
    , update : msg -> model -> model
    }
    -> Program () model msg
```

Zuerst können wir feststellen, dass die Funktion einen Record als Argument erhält. Dieser Record hat drei Einträge, die `init`, `view` und `update` heißen. Die Funktion ist polymorph über zwei Typvariablen, nämlich `model` und `msg`. Der Eintrag `init` ist vom Typ `model`. Daher können wir die Funktion `sandbox` nicht nur mit einem festen Typ verwenden, sondern die Typen für das Modell und die Nachrichten wählen. Wir müssen dabei nur beachten, dass wir gleiche Typvariablen durch den gleichen Typ ersetzen.

Die Typen der Einträge `view` und `update` unterscheiden sich von den Typen, die wir bisher in Records verwendet haben, da es sich um Funktionstypen handelt. In Kapitel 6 haben wir bereits gesehen, dass wir in der Programmiersprache Elm Funktionen als Argumente übergeben können. In einer Sprache, in der Funktionen *first class citizens* sind, können wir Funktionen aber nicht nur als Argument übergeben, wir können sie auch in Datenstrukturen ablegen. Daher kann auch ein Record Funktionen enthalten, wie es im Argument von `sandbox` der Fall ist. Das heißt, `sandbox` ist eine *higher-order*-Funktion, die einen Record erhält. Der Record enthält einen Wert und zwei Funktionen.

Das Ergebnis der Funktion `sandbox` ist ein dreistelliger Typkonstruktor. Dieser erhält den Typ des Modells und den Typ der Nachrichten als Argumente. Der Typ `()` wird als *unit* bezeichnet und ist der Typ der nullstelligen Tupel. Dieser Typ hat nur einen nullstelligen Konstruktor, nämlich `()`. Der *unit*-Typ wird ähnlich verwendet wie der Typ `void` in Java. Das erste Argument von *`Program`* wird genutzt, wenn eine Anwendung mit Flags gestartet werden soll. In diesem Fall können der JavaScript-Anwendung, die aus dem Elm-Code erzeugt wird, initial Informationen übergeben werden. Das erste Argument von *`Program`* gibt den Typ dieser initialen Informationen an. Da diese Funktionalität bei einer einfachen Elm-Anwendung nicht benötigt wird, wird dem Typkonstruktor *`Program`* der Typ `()` übergeben. Das heißt, die Anwendung erhält beim Start ein Flag, das den Typ `()` hat. Die Anwendung erhält initial dann einfach den Wert `()`, der aber keinerlei Information enthält.

An der Typsignatur von `sandbox` erkennt man auch, dass *`Html`* ein Typkonstruktor ist. Man übergibt an den Typkonstruktor den Typ der Nachrichten. Das heißt, wenn

wir eine HTML-Struktur bauen, wissen wir, welchen Typ die Nachrichten haben, die in der Strukur verwendet werden. Hierdurch können wir dafür sorgen, dass zum Beispiel in den `onClick`-*Handlern* der Struktur nur Werte des Typs `msg` verwendet werden. Wir betrachten etwa das folgende Beispiel.

```
module Counter exposing (main)

import Browser
import Html exposing (Html, text)


type alias Model =
    Int


init : Model
init =
    0


type Msg
    = Increase
    | Decrease


update : Msg -> Model -> Model
update msg model =
    case msg of
        Increase ->
            model + 1

        Decrease ->
            model - 1


view : Model -> Html Msg
view model =
    div []
        [ text (String.fromInt model)
        , button [ onClick False ] [ text "+" ]
        , button [ onClick 23 ] [ text "-" ]
        ]


main : Program () Model Msg
main =
```

```
Browser.sandbox { init = init, view = view, update = update }
```

Dieses Beispiel ist eine leichte Abwandlung unseres initialen Beispiels. Dieses Programm kompiliert nicht, da die Funktion `view` eine HTML-Struktur vom Typ *`Html Msg`* erstellt, die `onClick`-*Handler*, die verwendet werden, aber Nachrichten vom Typ *`Bool`* und vom Typ *`Int`* versenden.

Um das Beispiel besser zu verstehen, werfen wir einen Blick auf die Signaturen der Funktionen `div` und `onClick`.

```
div : List (Attribute msg) -> List (Html msg) -> Html msg
```

```
onClick : msg -> Attribute msg
```

Wir sehen, dass der Typ der Attribute ebenfalls ein Typkonstruktor ist, der den Typ der Nachrichten als Argument erhält. Durch den Typ der Funktion `div` wird sichergestellt, dass die Attribute den gleichen Nachrichtentyp verwenden, wie die Kindelemente des `div`. Der Typ der Funktion `onClick` nimmt eine Nachricht und erzeugt ein Attribut, das Nachrichten vom gleichen Typ enthält.

Bei vielen Attributen und vielen HTML-Elementen spielt der Typ der Nachrichten keine Rolle. Wir betrachten zum Beispiel die Signatur des Attributs `style`.

```
style : String -> String -> Attribute msg
```

Diese Funktion ist polymorph über der Typvariable `msg` und die Variable wird nur ein einziges Mal verwendet. Daher kann man mit einem Aufruf der Funktion `style` ein Attribut mit einem beliebigen Nachrichtentyp erzeugen. Auf diese Weise ist es möglich, die `style`-Funktion für HTML-Strukturen mit beliebigen Nachrichtentypen zu verwenden. Das heißt, Funktionen, für die der Typ der Nachrichten irrelevant ist, verwenden die Typvariable `msg` kein zweites Mal in ihrer Typsignatur.

Durch diese Modellierung kann gewährleistet werden, dass der Typ der Nachrichten, die an die Anwendung mit Hilfe von `onClick`-*Handlern* geschickt werden, auch von der Funktion `update` verarbeitet werden kann. Das heißt, mit Hilfe des statischen Typsystems sorgen wir dafür, dass klar ist, welche Nachrichten unsere `update`-Funktion verarbeiten können muss.

8 Abonnement

In diesem Kapitel wollen wir uns die Funktionsweise der `subscriptions` anschauen. Wie der Name schon sagt, handelt es sich dabei um ein Abonnement, das heißt, wir teilen Elm damit mit, dass wir eigenständig von Elm informiert werden möchten. Um das Konzept des Abonnements zu illustrieren, werden wir eine einfache Stoppuhr implementieren, die nur einen Sekundenzeiger hat.

Elm ist eine rein funktionale Programmiersprache. Das heißt, wir können keine Seiteneffekte ausführen, wie zum Beispiel das Schreiben von Dateien oder das Verändern von Variablen. Man spricht in diesem Kontext auch von referenzieller Transpararenz. Ein Ausdruck ist referenziell transparent, wenn der Wert des Ausdrucks nur von den Werten seiner Teilausdrücke abhängt. Damit darf der Wert eines Ausdrucks zum Beispiel nicht vom Zeitpunkt abhängen, zu dem der Ausdruck ausgewertet wird. Ein Beispiel für einen Ausdruck dessen Wert vom Zeitpunkt seiner Auswertung abhängt, ist der aktuelle Zeitstempel. Wenn wir in Java eine Methode schreiben, welche die Methode `currentTimeMillis` aufruft, ist die Methode zum Beispiel mit hoher Wahrscheinlichkeit nicht referentiell transparent.

In Elm werden wir gezwungen, referentiell transparente Programme zu schreiben. In Programmiersprachen, die uns nicht dazu zwingen, solche Programme zu schreiben, ist es aber auch guter Stil, diese Eigenschaft an möglichst vielen Stellen zu gewährleisten. Man kann sich leicht vorstellen, dass es recht schwierig ist, Fehler zu finden, wenn wiederholte Aufrufe der gleichen Methoden immer wieder andere Ergebnisse liefern. Daher versucht man auch in anderen Programmiersprachen den Teil der Anwendung, der nicht referentiell transparent ist, möglichst von dem Teil zu trennen, der referentiell transparent ist.

8.1 Zeit

Um über die aktuelle Zeit informiert zu werden, müssen wir zuerst das entsprechende Elm-Paket zu unserem Projekt hinzufügen. Dazu führen wir das folgende Kommando aus.

```
elm install elm/time
```

Da wir unsere Uhr mit Hilfe eines SVG zeichnen wollen, fügen wir auch noch das SVG-Paket hinzu.

```
elm install elm/svg
```

Als wir die Elm-Architektur besprochen haben, haben wir das Programm mit Hilfe der Funktion `sandbox` erstellt.

```
sandbox :
    { init : model
    , view : model -> Html msg
    , update : msg -> model -> model
    }
    -> Program () model msg
```

Neben dieser Funktion gibt es auch eine Funktion `element`, die den folgenden Typ hat.

```
element :
    { init : flags -> ( model, Cmd msg )
    , view : model -> Html msg
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    }
    -> Program flags model msg
```

Diese Funktion nimmt als initialen Wert eine Funktion. Die Funktion, die das initiale Modell erzeugt, erhält als Argument einen Wert vom Typ `flags`. Es handelt sich dabei um Informationen, die das JavaScript-Programm, das die Elm-Anwendung startet, an die Anwendung übergeben kann. Das initiale Modell besteht im Vergleich zur Sandbox außerdem nicht nur aus einem Modell sondern noch aus einem Kommando in Form eines Wertes vom Typ *Cmd* `msg`. Die Funktion `update` liefert in diesem Fall auch nicht nur ein Modell als Ergebnis, sondern ein Modell und ein Kommando. Außerdem ist ein neues Feld hinzugekommen, das `subscriptions` heißt.

Den Typ *Cmd* `msg` benötigen wir in diesem Kapitel erst einmal nicht. Die Funktion `init` muss aber einen Wert von diesem Typ zurückgeben. Um diesen Wert zu erzeugen, können wir die Konstante *Cmd*`.none` nutzen. Das Paket `elm/time` stellt ein Modul mit dem Namen *Time* zur Verfügung. Dieses Modul stellt wiederum eine Funktion mit der Signatur

```
every : Float -> (Posix -> msg) -> Sub msg
```

zur Verfügung. Das erste Argument ist ein Intervall in Millisekunden, das angibt, wie häufig wir über die aktuelle Zeit informiert werden möchten. Wir müssen der Funktion dann noch ein funktionales Argument übergeben, das die aktuelle Zeit im Posix-Format erhält und daraus eine Nachricht unseres Nachrichtentyps macht.

Wir wollen nur die vergangenen Sekunden in unserer Uhr anzeigen, daher verwenden wir die folgenden Datentypen für unser Modell und die Nachrichten an die Anwendung.

```
type alias Seconds =
    Int


type alias Model =
    Seconds
```

```
type Msg =
    Tick
```

Mit Hilfe der Funktion `time` definieren wir die folgende `main`-Funktion. Die Implementierungen der Funktionen `init`, `view` und `update` werden wir im Folgenden diskutieren. Um die Definitionen zu vereinfachen, nutzen wir hier mehrere Lambda-Ausdrücke.

```
main : Program () Model Msg
main =
    Browser.element
        { init = \_ -> ( init, Cmd.none )
        , subscriptions = \_ -> Time.every 1000 (\_ -> Tick)
        , view = view
        , update = \msg model -> ( update msg model, Cmd.none )
        }
```

Unser initiales Modell setzt den Sekundenwert zu Anfang auf Null.

```
init : Model
init =
    0
```

Die Funktion `update` zählt lediglich unseren Sekundenzähler hoch.

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        Tick ->
            modBy 60 (model + 1)
```

Im Grunde könnten wir hier auch auf das Pattern Matching verzichten, da wir wissen, dass die einzigen Nachricht, die wir erhalten können, die Nachricht *Tick* ist. Durch das Pattern Matching gewährleisten wir aber, dass der Elm-Compiler sich über ein fehlendes *Pattern* beschwert, falls wir einen weiteren Konstruktor zum Typ *Msg* hinzufügen. Ohne das *Pattern Matching* würde die Anwendung sich einfach weiter verhalten wie zuvor und bei einer großen Anwendung ist ein Fehler dieser Art eventuell schwer zu finden. Da wir nur eine Sekundenanzeige umsetzen wollen, rechnen wir den Sekundenzähler noch modulo 60.

Als nächstes wollen wir die Uhr zeichnen. Dazu definieren wie uns zunächst eine Hilfsfunktion. Diese Funktion werden wir später nutzen, um den Wert des SVG-Attributes `transform` zu setzen. Dabei geben wird einen Winkel in Grad und einem Punkt an und rotieren dann ein Objekt um den Winkel um den angegebenen Punkt.

```
type alias Point =
    { x : Float, y : Float }


rotate : Float -> Point -> String
rotate angle point =
```

```
    "rotate("
        ++ String.fromFloat angle
        ++ ","
        ++ String.fromFloat point.x
        ++ ","
        ++ String.fromFloat point.y
        ++ ")"
```

Nun implementieren wir eine Funktion, die die aktuelle Sekundenzahl in Form einer Uhr anzeigt.

```
view : Model -> Html Msg
view model =
    let
        center =
            Point 200 200

        radius =
            100
    in
    svg
        [ width "500"
        , height "500"
        ]
        [ clockBack center radius
        , clockHand center radius model
        ]


clockBack : Point -> Float -> Svg msg
clockBack center radius =
    circle
        [ cx (String.fromFloat center.x)
        , cy (String.fromFloat center.y)
        , r (String.fromFloat radius)
        , fill "#aaddf9"
        ]
        []


clockHand : Point -> Float -> Seconds -> Svg msg
clockHand center radius seconds =
    line
        [ x1 (String.fromFloat center.x)
        , y1 (String.fromFloat center.y)
        , x2 (String.fromFloat center.x)
        , y2 (String.fromFloat (center.y - radius))
```

```
        , stroke "#2c2f88"
        , strokeWidth "2"
        , transform (rotate  (360 * toFloat seconds / 60) center)
        ]
        []
```

Um zu illustrieren, wie man Abonnements zeitweise aussetzt, wollen wir unsere Uhr um die Möglichkeit erweitern, sie anzuhalten, wieder zu starten und zurückzusetzen. Dazu erweitern wir erst einmal wie folgt unseren Datentyp *Msg*.

```
type Msg
    = Tick
    | Start
    | Stop
    | Reset
```

Außerdem fügen wir drei Knöpfe zu unserer Anwendung hinzu.

```
view : Model -> Html Msg
view model =
    let
        center =
            Point 200 200

        radius =
            100
    in
    div []
        [ svg
            [ width "500"
            , height "500"
            ]
            [ clockBack center radius
            , clockHand center radius model.seconds
            ]
        , button [ onClick Start ] [ text "Start" ]
        , button [ onClick Stop ] [ text "Stop" ]
        ]
```

Da unsere Uhr nun auch pausiert sein kann, müssen wir diese Information in unserem Zustand modellieren. Wir nutzen dazu einen Record, der zusätzlich den Zustand der Uhr hält.

```
type ClockState
    = Running
    | Paused
```

```
type alias Model =
   { seconds : Seconds, state : ClockState }
```

Als nächstes adaptieren wir die Funktion `update` wie folgt.

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        Tick ->
            case model.state of
                Running ->
                    { model | seconds = model.seconds + 1 }

                Paused ->
                    model

        Start ->
            { model | state = Running }

        Stop ->
            { model | state = Paused }

        Reset ->
            { model | seconds = 0 }
```

Wir müssen zu guter Letzt noch darauf achten, dass wir das initiale Modell anpassen. Statt eines einzelnen *`Int`* besteht unser initiales Modell jetzt aus einem Record, der die Uhr initial pausiert.

```
init : Model
init =
    { seconds = 0, state = Paused }
```

Wenn wir die *`Subscription`* nur ignorieren, kann es sein, dass die Uhr bis zu eine Sekunde benötigt, um nach einen Kopfdruck tatsächlich zu starten. Außerdem sollten wir die *`Subscription`* beenden, wenn wir sie gar nicht benötigen. Das Feld `subscriptions` des Programms ist eine Funktion, die ein Modell als Argument erhält und eine *`Subscription`* liefert. Die Funktion *`Sub`*`.none` liefert analog zu *`Cmd`*`.none` keine *`Subscription`*. Wir können wie folgt die *`Subscription`* beenden, wenn die Uhr im Zustand *`Paused`* ist.

```
subscriptions : Model -> Sub Msg
subscriptions model =
    case model.state of
        Running ->
            Time.every 1000 (\_ -> Tick)

        Paused ->
```

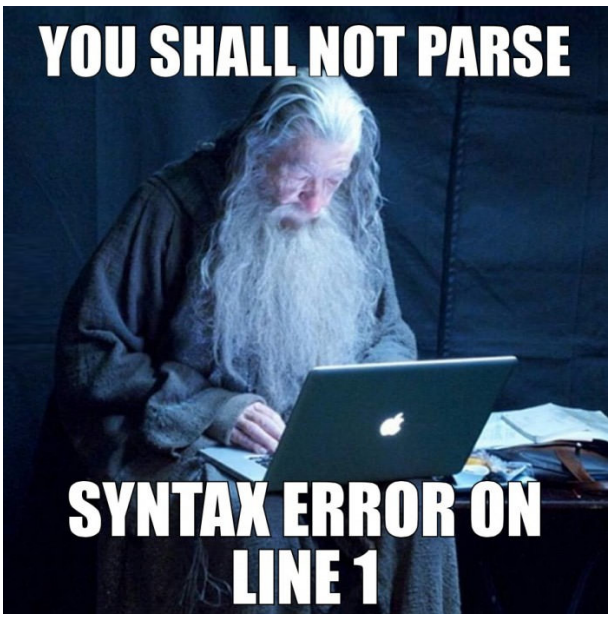
```
            Sub.none

main : Program () Model Msg
main =
    Browser.element
        { init = \_ -> ( init, Cmd.none )
        , subscriptions = subscriptions
        , view = view
        , update = \msg model -> ( update msg model, Cmd.none )
        }
```

Zum Abschluss soll hier noch die Funktion `batch : List (Sub msg) -> Sub msg` vorgestellt werden. Diese Funktion kann genutzt werden, um eine Liste von Abonnements zu einem Abonement zusammenzufassen. Auf diese Weise können wir in einer Anwendung über mehrere Ereignisse informiert werden. Im folgenden Abschnitt werden wir zum Beispiel lernen, wie man sich über Tastendrücke informieren lassen kann. Mit Hilfe der Funktion `batch` kann man dann zum Beispiel in einem festgelegten Interval oder wenn eine Taste gedrückt wird, informiert werden.

8.2 Decoder

Bei einigen Arten von Abonnements aber auch bei anderen Konzepten, die wir später kennenlernen werden, müssen Daten im JSON-Format in einen stärker strukturierten Elm-Datentyp umgewandelt werden. Um diese Aufgabe umzusetzen, werden in Elm *Decoder* verwendet.



Ein *Decoder* ist eine Elm-spezifische Variante des allgemeineren Konzeptes eines Parser-Kombinators. Parser-Kombinatoren sind eine leichtgewichtige Implementierung von Parsern. Ein Parser ist eine Anwendung, die einen String erhält und prüft, ob

der String einem vorgegebenen Format folgt. Jeder Compiler nutzt zum Beispiel einen Parser, um den Programmtext, den ein*e Programmierer*in eingibt, auf syntaktische Korrektheit zu prüfen. Neben der Überprüfung, ob der String einem Format entspricht, wird der String für gewöhnlich auch noch in ein strukturiertes Format umgewandelt. Im Fall von Elm, wird der String zum Beispiel in Werte von algebraischen Datentypen umgewandelt. In einer objektorientierten Programmiersprache würde ein Parser einen String erhalten und ein Objekt liefern, das eine strukturierte Beschreibung des Strings zur Verfügung stellt.

Um in einer Elm-Anwendung einen *Decoder* zu nutzen, müssen wir zuerst den folgenden Befehl ausführen.

```
elm install elm/json
```

Die Bibliothek `elm/json` ist eine eingebettete domänenspezische Sprache und stellt eine deklarative Technik dar, um *Decoder* zur Verarbeitung von JSON zu beschreiben.

Der Typkonstruktor *Decoder* ist im Modul *Json.Decode* definiert[1]. Das Modul stellt eine Funktion

```
decodeString : Decoder a -> String -> Result Error a
```

zur Verfügung. Mit Hilfe dieser Funktion kann ein *Decoder* auf eine Zeichenkette angewendet werden. Ein *Decoder* `a` liefert einen Wert vom Typ `a` als Ergebnis nach dem Parsen. Wenn wir einen *Decoder* `a` mit `decodeString` auf eine Zeichenkette anwenden, erhalten wir entweder einen Fehler und eine Fehlermeldung mit einer Fehlerbeschreibung oder wir erhalten einen Wert vom Typ `a`.

Das Modul *Json.Decode* stellt die folgenden primitiven *Decoder* zur Verfügung.

```
string : Decoder String
int : Decoder Int
float : Decoder Float
bool : Decoder Bool
```

Um zu illustrieren, wie diese *Decoder* funktionieren, schauen wir uns die folgenden Beispiele an. Der Aufruf *Decode*.`decodeString` *Decode*.`int "42"` liefert als Ergebnis *Ok* `42`. Das heißt, der *String* `"42"` wird erfolgreich mit dem *Decoder* `int` verarbeitet und liefert als Ergebnis den Wert `42`. Der Aufruf

```
Decode.decodeString Decode.int "a"
```

liefert dagegen als Ergebnis

```
Err (Failure ("This is not valid JSON! Unexpected token a in JSON at position 0"))
```

da der *String* `"a"` nicht der Spezifikation des Formates entspricht, da es sich nicht um einen *Int* handelt.

Die Idee der Funktion `map`, die wir für Listen kennengelernt haben, lässt sich auch auf andere Datenstrukturen anwenden. Genauer gesagt, kann man `map` für die meisten Typkonstruktoren definieren. Da *Decoder* ein Typkonstruktor ist, können wir `map` für *Decoder* definieren.

Elm stellt die folgende Funktion für *Decoder* zur Verfügung.

[1] Dieses Modul wird hier wie folgt importiert `import Json.Decode as Decode exposing (Decoder)`.

```
map : (a -> b) -> Decoder a -> Decoder b
```

Diese Funktion kann zum Beispiel genutzt werden, um das Ergebnis eines *Decoder* in einen Konstruktor einzupacken. Wir nehmen einmal an, dass wir den folgenden – zugegebenermaßen etwas artifiziellen – Datentyp in unserer Anwendung nutzen.

```
type alias User =
    { age : Int }
```

Wir können nun auf die folgende Weise einen *Decoder* definieren, der eine Zahl parst und als Ergebnis einen Wert vom Typ *User* zurückliefert.

```
decodeUser : Decoder User
decodeUser =
    Decode.map User Decode.int
```

Wir sind nun nur in der Lage einen JSON-Wert, der nur aus einer Zahl besteht, in einen Elm-Datentyp umzuwandeln. In den meisten Fällen werden wird das Alter des Nutzers auf JSON-Ebene nicht als einzelne Zahl dargestellt, sondern zum Beispiel durch folgendes JSON-Objekt.

```
{
  "age": 18
}
```

Auf diese Struktur können wir unseren *Decoder* nicht anwenden, da es sich nicht um eine reine Zahl handelt. Um dieses Problem zu lösen, nutzt man in Elm die folgende Funktion.

```
field : String -> Decoder a -> Decoder a
```

Mit dieser Funktion kann ein *Decoder* auf ein einzelnes Feld einer JSON-Struktur angewendet werden. Das heißt, der folgende *Decoder* ist in der Lage die oben gezeigte JSON-Struktur zu verarbeiten.

```
decodeUser : Decoder User
decodeUser =
    Decode.map User (Decode.field "age" Decode.int)
```

Der Aufruf

```
Decode.decodeString decodeUser "{ \"age\": 18 }"
```

liefert in diesem Fall als Ergebnis *Ok* `{ age = 18 }`. Das heißt, dieser Aufruf ist in der Lage, den *String*, der ein JSON-Objekt darstellt, in einen Elm-Record zu überführen. Ein Parser verarbeitet einen *String* häufig so, dass Leerzeichen für das Ergebnis keine Rolle spielen. So liefert der Aufruf

```
Decode.decodeString decodeUser "{\t   \"age\":\n     18}"
```

etwa das gleiche Ergebnis wie der Aufruf ohne die zusätzlichen Leerzeichen.

In den meisten Fällen hat die JSON-Struktur, die wir verarbeiten wollen nicht nur ein Feld, sondern mehrere. Für diesen Zweck stellt Elm die Funktion

```
map2 : (a -> b -> c) -> Decoder a -> Decoder b -> Decoder c
```

zur Verfügung, mit der wir zwei *Decoder* zu einem kombinieren können.

Wir erweitern unseren Datentyp wie folgt.

```
type alias User =
    { name : String, age : Int }
```

Nun definieren wir einen *Decoder* mit Hilfe von `map2` und kombinieren dabei einen *Decoder* für den *Int* mit einem *Decoder* für den *String*.

```
decodeUser : Decoder User
decodeUser =
    Decode.map2
        User
        (Decode.field "name" Decode.string)
        (Decode.field "age" Decode.int)
```

Der Aufruf

```
decodeString decodeUser "{ \"name\": \"Max Mustermann\", \"age\": 18}"
```

liefert in diesem Fall als Ergebnis *Ok* `{ age = 18, name = "Max Mustermann" }`.

Der *Decoder* `decodeUser` illustriert wieder gut die deklarative Vorgehensweise. Zur Implementierung des *Decoder*s beschreiben wir, welche Felder wir aus den JSON-Daten verarbeiten wollen und wie wir aus den Ergebnissen das Endergebnis formen. Wir beschreiben aber nicht, wie genau das Verarbeiten durchgeführt wird.

8.3 Tasten

Wir wollen nun zur Verarbeitung von Tasten zurückkommen und einen einfachen Zähler implementieren, bei dem es möglich ist, den Zähler mit Hilfe der Pfeiltasten zu erhöhen oder zu erniedrigen. Zu diesem Zweck müssen wir über alle Tastendrücke informiert werden. Die Funktion

```
onKeyDown : Decoder msg -> Sub msg
```

aus dem Modul *Browser.Events* erlaubt es uns, auf *KeyDown*-Ereignisse zu reagieren. Der *Decoder*, den wir übergeben, wandelt die JSON-Struktur, die bei einem Tastendruck geliefert wird, in einen Elm-Wert um. Wir definieren zum Beispiel den folgenden Decoder.

```
type Key
    = Up
    | Down


keyDecoder : Decoder (Maybe Key)
keyDecoder =
    let
        toKey string =
            case string of
                "ArrowUp" ->
                    Just Up

                "ArrowDown" ->
                    Just Down

                _ ->
                    Nothing
    in
    Decode.map toKey (Decode.field "key" Decode.string)
```

Dieser *Decoder* liefert abhängig davon, welchen *String* das Feld mit dem Namen `"key"`[2] enthält, einen Wert vom Typ *Maybe Key*. Diesen *Decoder* können wir nun nutzen, um in unserer Anwendung auf Tastendrücke zu lauschen. Hierzu nutzen wir die folgende Definition unserer Anwendung.

```
type alias Model =
    Int


main : Program () Model (Maybe Key)
main =
    program
        { init = \_ -> ( 0, Cmd.none )
        , subscriptions = \_ -> Browser.Events.onKeyDown keyDecoder
        , view = view
        , update = \msg model -> ( update msg model, Cmd.none )
        }
```

Der initiale Zustand der Anwendung besteht nun aus zwei Komponenten, dem Modell und dem Kommendo, das beim Start der Anwendung ausgeführt werden soll.

In dieser einfachen Anwendung nutzen wir einfach *Maybe Key* direkt als Nachrichtentyp.

```
update : Maybe Key -> Model -> Model
```

[2] Unter `https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key` lässt sich überprüfen, welchen Wert dieses Feld beim Druck einer bestimmten Taste annimmt.

```
update msg model =
    case msg of
        Just Up ->
            model + 1

        Just Down ->
            model - 1

        Nothing ->
            model
```

Daher verarbeitet `update` Nachrichten vom Typ *`Maybe Key`*. Die `view`-Funktion stellt einfach den Zähler als Text in einer HTML-Struktur an.

9 Kommandos

In diesem Kapitel wollen wir uns anschauen, wie man den Datentyp *Cmd* nutzt, den wir bisher ignoriert haben. Wir haben zuvor bereits gelernt, dass Elm eine rein funktionale Programmiersprache ist und man daher keine Seiteneffekte auführen kann. Einige Teile einer Frontend-Anwendung benötigen aber natürlich Seiteneffekte. Als ein Beispiel für einen solchen Seiteneffekt wollen wir uns das Würfeln einer Zufallszahl anschauen. Um Seiteneffekte in Elm ausführen zu können und dennoch eine referenziell transparente Anwendung zu behalten, wird die Durchführung von Seiteneffekten von der Elm-Runtime übernommen. Genauer gesagt, teilen wir Elm nur mit, dass wir einen Seiteneffekt durchführen möchten. Elm führt dann diesen Seiteneffekt durch und informiert uns über das Ergebnis. Auch die Kommandos sind wieder ein Beispiel für den deklarativen Ansatz, da man nur beschreibt, dass ein Seiteneffekt durchgeführt werden soll, man beschreibt aber nicht, wie dieser genau ausgeführt wird.

9.1 Zufall

Wir wollen eine Anwendung schreiben, mit der man einen Würfel werfen kann. Zuerst installieren wir das Paket `elm/random`. Als nächstes modellieren wir die möglichen Ergebnisse eines Würfels.

```
type Side
    = One
    | Two
    | Three
    | Four
    | Five
    | Six
```

Als nächstes definieren wir ein Modell für unsere Anwendung.

```
type alias Model =
    Side
```

Nun definieren wir einen initialen Zustand. Neben dem Modell, können wir auch ein Kommando angeben, das beim Start der Anwendung ausgeführt werden soll. Das Modul *Cmd* stellt eine Konstante `none` zur Verfügung, die angibt, dass kein Kommando durchgeführt werden soll.

```
init : Model
init =
    One
```

Als nächstes definieren wir die Nachrichten, die unsere Anwendung verarbeiten kann. Die Anwendung soll nur in der Lage sein einen Würfel zu würfeln, daher benötigen wir nur eine einzige Nachricht.

```
type Msg
    = Roll
```

Mit Hilfe eines Buttons können wir diese Nachricht an die Anwendung schicken.

```
view : Model -> Html Msg
view model =
    div []
        [ text (toString model)
        , button [ onClick Roll ] [ text "Roll" ]
        ]
```

Als nächstes benötigen wir die `update`-Funktion. Diese liefert neben dem neuen Modell auch ein Kommando, das als nächstes ausgeführt werden soll. Um dieses Kommando zu konstruieren, verwenden wir im Fall des Zufalls die vordefinierte Funktion

```
generate : (a -> msg) -> Generator a -> Cmd msg
```

aus dem Modul *Random*. Diese Funktion nimmt einen Zufallsgenerator, eine Funktion, die das Ergebnis des Zufallsgenerators in eine Nachricht verpackt und liefert ein Kommando. Wir benötigen also noch eine Nachricht und erweitern unseren Datentyp *Msg* wie folgt.

```
type Msg
    = Roll
    | Rolled Side
```

Außerdem benötigen wir einen Generator, der zufällig eine Seite liefert. Wir nutzen dafür die Funktion `uniform : a -> List a -> Generator a` aus dem Modul *Random*.

```
die : Random.Generator Side
die =
    Random.uniform One [ Two, Three, Four, Five, Six ]
```

Die Funktion `uniform` erhält einen Wert und eine Liste von Werten und liefert mit gleicher Verteilung den Wert oder eines der Elemente der Liste. An sich könnte die Funktion auch nur eine Liste erhalten. In diesem Fall könnten wir die Funktion aber mit einer leeren Liste aufrufen. Daher erhält `uniform` noch ein zusätzliches Argument, um zu gewährleisten, dass die Funktion immer mindestens ein Argument erhält. Alternativ könnte man die Funktion `uniform` als Argument auch einen Listendatentyp nehmen, bei dem die Liste nicht leer sein kann.

Mit Hilfe des Generators, der gleichverteilt Würfelseiten liefern kann, können nun die Funktion `update` wie folgt definieren.

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Roll ->
            ( model, Random.generate Rolled die )

        Rolled side ->
            ( side, Cmd.none )
```

Wenn der Benutzer auf den Button klickt, erhalten wir die Nachricht `Roll`. In diesem Fall lassen wir das Modell einfach wie es ist und fordern die Laufzeitumgebung auf, einen zufälligen Wert mit unserem Generator zu erzeugen. Wenn dieser Wert erzeugt wurde, wird die Funktion `update` wieder aufgerufen, dieses Mal aber mit der Nachricht `Rolled`. Der Konstruktor enthält die Seite, die gewürfelt wurde. Wenn wir diese Nachricht erhalten, ersetzen wir den alten Zustand durch unsere neue Würfelseite und geben an, dass wir kein Kommando ausführen wollen.

Zu guter Letzt müssen wir unsere Anwendung nur noch wie folgt zusammenbauen. Da wir zur Verwendung von Komandos die Funktion `Browser.element` verwenden müssen, müssen wir dem Record auch eine Feld `subscriptions` übergeben. Da wir in dieser Anwendung nicht über Ereignisse informiert werden möchten, nutzen wir die Konstante `Sub.none`, um zu signalisieren, dass wir keine Abonnements nutzen möchten.

```
main : Program () Model Msg
main =
    Browser.element
        { init = \_ -> ( init, Cmd.none )
        , subscriptions = \_ -> Sub.none
        , view = view
        , update = update
        }
```

Das Modul `Random` stellt ähnliche Funktionen zur Verfügung wie das Modul `Json.Decode` für die Definition von `Decoder`n. Zum Einen stellt das Modul `Random` die Funktion `map : (a -> b) -> Generator a -> Generator b` zur Verfügung. Mit Hilfe dieser Funktion können wir die Ergebnisse eines `Generator` abändern. Nehmen wir an, wir benötigen einen Zufallsgenerator, der Zahlen liefert an Stelle des Datentyps `Side`. In diesem Fall können wir wie folgt einen Generator definieren.

```
pips : Random.Generator Int
pips =
    let
        toPips side =
            case side of
                One ->
                    1

                Two ->
```

```
                2

            Three ->
                3

            Four ->
                4

            Five ->
                5

            Six ->
                6
    in
    Random.map toPips die
```

Das Modul *Random* stellt außerdem eine Funktion

```
map2 : (a -> b -> c) -> Generator a -> Generator b -> Generator c
```

zur Verfügung, mit der wir zwei Generatoren zu einem Generator kombinieren können. Wir können zum Beispiel wie folgt einen Generator definieren, der zufällig zwei Würfel würfelt und die Summe der Augenzahlen

```
dice : Random.Generator Int
dice =
    Random.map2 (+) pips pips
```

Die Schreibweise `(+)` ist im Endeffekt eine Kurzform von `\x y -> x + y`. Wenn man einen Infixoperator mit Klammern umschließt, kann man den eigentlich infix verwendeten Operator präfix schreiben. Zum Beispiel kann man statt `1 + 2` auch `(+) 1 2` schreiben.

9.2 HTTP-Anfragen

Als Abschluss dieses Kapitels wollen wir uns noch anschauen, wie man HTTP-Anfragen in Elm durchführen kann. Eine HTTP-Anfrage folgt dem gleichen Muster wie das Erzeugen eines zufälligen Wertes. Wir teilen dem System mit, welche Anfrage wir stellen möchten und das System ruft die Funktion `update` auf, wenn die Anfrage erfolgreich abgeschlossen ist. Im Unterschied zum Erzeugen eines Zufallswertes, kann in diesem Fall aber auch ein Fehler bei der Abarbeitung der Aufgabe auftreten. Um einen HTTP-Anfrage zu senden, müssen wir zunächst mit dem folgenden Kommando eine Bibliothek installieren.

```
elm install elm/http
```

Wir wollen eine einfache Anwendung entwickeln, mit der man sich XKCD-Comics anschauen kann. Die Route `https://api.isevenapi.xyz/api/iseven/{number}`[1] liefert für jede Zahl *number*, ob die Zahl gerade ist. Für die Zahl `3` erhalten wir als Ergebnis zum Beispiel das folgende JSON-Objekt.

```
{
  "iseven": false,
  "ad": "Buy isEvenCoin, the hottest new cryptocurrency!"
}
```

Wir modellieren diese Struktur erst einmal auf Elm-Ebene und definieren eine Funktion, um diese Informationen anzuzeigen.

```
type alias IsEven =
    { isEven : Bool
    , advertisement : String
    }


viewIsEven : IsEven -> Html msg
viewIsEven isEven =
    div []
        [ p []
            [ text
                ("This number is "
                    ++ (if isEven.isEven then
                            " even"

                        else
                            " not even"
                        )
                )
            ]
        , p [] [ text isEven.advertisement ]
        ]
```

Nachdem wir das Resultat eines Requests modelliert haben, wollen wir einen Request durchführen. Die Funktion

```
get  :
     { url : String
     , expect : Expect msg
     }
     -> Cmd msg
```

[1] `https://github.com/public-apis/public-apis`

aus dem Modul *Http* kann genutzt werden, um ein Kommando zu erzeugen, das eine *get*-Anfrage durchführt. Dazu wird eine URL und ein Wert vom Typ *Expect* `msg` angegeben, mit dem wir spezifizieren, welche Art von Ergebnis wir als Resultat von der Anfrage erwarten. Das Modul *Http* stellt zum Beispiel die Funktion

```
expectJson : (Result Error a -> msg) -> Decoder a -> Expect msg
```

zur Verfügung, um JSON zu verarbeiten, das von einer Anfrage zurückgeliefert wird. Dazu müssen wir zum einen einen *Decoder* angeben, der die JSON-Struktur in eine Elm-Datenstruktur umwandelt. Außerdem müssen wir eine Funktion angeben, die das Resultat des Decoders in eine Nachricht umwandeln kann. Hierbei ist allerdings zu beachten, dass die Anfrage auch fehlschlagen kann. Daher muss die Funktion auch in der Lage sein, einen möglichen Fehler zu verarbeiten.

Außer dem Ergebnis der Anfrage, wollen wir noch vor- und zurückblättern können. Wir definieren den folgenden Datentyp.

```
type Msg
    = Previous
    | Next
    | Response (Result Http.Error IsEven)
```

Wir definieren nun zuerst einen Decoer, um die JSON-Struktur, die wir vom Server erhalten, in den Record *IsEven* umzuwandeln.

```
isEvenDecoder : Decoder IsEven
isEvenDecoder =
    Decode.map2 IsEven
        (Decode.field "iseven" Decode.bool)
        (Decode.field "ad" Decode.string)
```

Mit Hilfe des Konstruktors *Request* können wir die folgende Funktion definieren, die eine Zahl erhält und ein Kommando liefert, das eine entsprechende Anfrage stellt.

```
getIsEven : Int -> Cmd Msg
getIsEven no =
    Http.get
        { url = "https://api.isevenapi.xyz/api/iseven/" ++ String.fromInt no
        , expect = Http.expectJson Request isEvenDecoder
        }
```

Wir nutzen zur Modellierung des internen Zustands unserer Anwendung die folgenden Datentypen.

```
type alias Model =
    { number : Int
    , response : Progress (Result Http.Error IsEven)
    }
```

```
type Progress a
    = Waiting
    | Finished a
```

Der einfachheitshalber nutzen wir hier im Modell den Datentyp *Http.Error*, den wir von der Anfrage zurückerhalten. In einer realen Anwendung würden wir diesen internen Fehler vermutlich nicht im Modell nutzen.

Die folgende Funktion aktualisiert das Modell, wenn die Anfrage ein Ergebnis geliefert hat. Wenn eine der Aktionen *Next* und *Previous* durchgeführt wird, wird eine neue Anfrage gestellt.

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Previous ->
            updateNumber (model.number - 1) model

        Next ->
            updateNumber (model.number + 1) model

        Response request ->
            ( { model | response = Finished request }, Cmd.none )


updateNumber : Int -> Model -> ( Model, Cmd Msg )
updateNumber number model =
    ( { model | number = number, response = Waiting }, getIsEven number )
```

Zu guter letzt müssen wir nur noch Funktionen schreiben, die abhängig vom aktuellen Zustand eine entsprechende Html-Seite anzeugen. Außerdem stellen wir Knöpfe für die verschiedenen Aktionen zur Verfügung.

```
view : Model -> Html Msg
view model =
    div []
        [ div []
            [ button [ onClick Previous ] [ text "Previous" ]
            , text (String.fromInt model.number)
            , button [ onClick Next ] [ text "Next" ]
            ]
        , viewResponse model.response
        ]


viewResponse : Progress (Result Http.Error IsEven) -> Html msg
viewResponse progress =
    case progress of
```

```
        Waiting ->
            text "Loading ..."

        Finished (Ok result) ->
            viewIsEven result

        Finished (Err error) ->
            text ("The following error occured " ++ Debug.toString mes)


main : Program () Model Msg
main =
    Browser.element
        { init = \_ -> ( { number = 0, response = Waiting }, getIsEven 0 )
        , subscriptions = \_ -> Sub.none
        , view = view
        , update = update
        }
```

Die Funktion `viewError` nutzt einfachheitshalber hier eine Funktion, die nur zum Debugging einer Anwendung gedacht ist und in der Lage ist, einen beliebigen Elm-Wert in einen *String* umzuwandeln.

10 Faltungen

Nachdem wir uns die meisten Konzepte der Elm-Architektur angeschaut haben, wollen wir uns noch eine Abstraktion anschauen, die in Elm (und der funktionalen Programmierung im Allgemeinen), sehr häufig zum Einsatz kommt. Wir haben die Funktionen `map` und `filter` kennengelernt, die wiederkehrende rekursive Muster abstrahieren. In diesem Kapitel wollen wir uns mit der Idee der Faltung beschäftigen. Bei einer Faltung handelt es sich um eine Funktion, die ebenfalls ein wiederkehrendes rekursives Muster bei der Verarbeitung von Listen abstrahiert. Mit Hilfe einer Faltung können wir zum Beispiel die folgenden Funktionen definieren.

- Berechnung der Summe einer Liste von Zahlen
- Berechnung des Produktes einer Liste von Zahlen
- Berechnung der Länge einer Liste beliebiger Elemente
- die Funktionen `map` und `filter`

Tatsächlich kann man mit einer Faltung aber sogar alle Funktionen definieren, die eine Liste auf eine bestimmte Art und Weise verarbeiten.

10.1 Rechtsfaltung für Listen

Zunächst definieren wir die ersten drei Beispiele mit expliziter Rekursion, also ohne die Verwendung einer Abstraktion.

```
sum : List Int -> Int
sum list =
    case list of
        [] ->
            0

        x :: xs ->
            x + sum xs


product : List Int -> Int
product list =
    case list of
        [] ->
            1
```

```
        x :: xs ->
            x * product xs


filter : (a -> Bool) -> List a -> List a
filter pred list =
    case list of
        [] ->
            []

        x :: xs ->
            if pred x then
                x :: filter pred xs

            else
                filter pred xs
```

Wir wollen einmal schauen, was diese drei Funktionen gemeinsam haben. Alle drei Funktionen führen eine Fallunterscheidung über die Liste durch. Außerdem verwenden alle drei Funktionen einen einfachen Rückgabewert im Fall der leeren Liste. Wenn wir die Funktionen als `g` bezeichnen, haben alle drei Funktionen die folgende Form.

```
g list =
    case list of
        [] ->
            nil

        ...
```

Alle drei Funktionen zerlegen eine nicht-leere Liste in ihren Kopf und den Rest. Der Kopf der Liste wird dabei mit dem Ergebnis des rekursiven Aufrufs der Funktion auf die Restliste kombiniert. Im Fall der Funktion `filter` ist es nicht direkt offensichtlich, dass der Kopf der Liste mit dem Ergebnis des rekursiven Aufrufs kombiniert wird. Wir können die Funktion `filter` aber zum Beispiel wie folgt umdefinieren, um diesen Punkt deutlicher zu machen.

```
filter : (a -> Bool) -> List a -> List a
filter pred list =
    let
        conditionalCons element result =
            if pred element then
                element :: result

            else
                result
    in
```

```
    case list of
        [] ->
            []

        x :: xs ->
            conditionalCons x (filter pred xs)
```

Das heißt, alle drei Funktionen haben die folgende Form.

```
g list =
    case list of
        [] ->
            nil

        x :: xs ->
            cons x (g xs)
```

Daraus ergibt sich die folgende Definition der Funktion `foldr`. Wir werden später sehen, was das `r` in `foldr` bedeutet.

```
foldr : (a -> b -> b) -> b -> List a -> b
foldr cons nil list =
    case list of
        [] ->
            nil

        x :: xs ->
            cons x (foldr cons nil xs)
```

Wir können jetzt die Funktionen von oben wie folgt definieren.

```
sum : List Int -> Int
sum list =
    foldr (\element result -> element + result) 0 list

product : List Int -> Int
product list =
    foldr (\element restult -> element * result) 1 list

filter : (a -> Bool) -> List a -> List a
filter pred list =
    let
        conditionalCons element result =
            if pred element then
                element :: result

            else
                result
```

```
    in
    foldr conditionalCons [] list
```

Diese Definitionen können wir mithilfe von Eta-Reduktion noch etwas vereinfachen. Wir erhalten die folgenden Definitionen.

```
sum : List Int -> Int
sum =
    foldr (+) 0

product : List Int -> Int
product =
    foldr (*) 1

filter : (a -> Bool) -> List a -> List a
filter pred =
    foldr
        (\element restult ->
            if pred element then
                element :: result

            else
                result
        )
        []
```

Eine mögliche Sichtweise der Funktion `foldr` ist das Ersetzen aller Konstruktoren einer Liste durch Funktionsaufrufe bzw. Konstanten. Genauer gesagt werden bei einem Aufruf `foldr cons nil list` in einer Liste `list` alle Vorkommen des Konstruktors `::` durch die Funktion `cons` und alle Vorkommen des Konstruktors `[]` durch den Wert `nil` ersetzt. Im Folgenden werden wir den Konstruktor `::` als `(::)` vor seine Argumente schreiben, um diesen Punkt zu illustrieren. We betrachten eine Liste `[ a, b, c ]` und erhalten durch den Aufruf `foldr cons nil [ a, b, c ]` das folgende Ergebnis.

```
foldr cons nil [ a, b, c ]
=
foldr cons nil (a :: (b :: (c :: [])))
=
foldr cons nil ((::) a ((::) b ((::) c [])))
=
cons a (cons b (cons c nil))
```

Hier sieht man, dass alle Vorkommen von `::` durch Aufrufe von `cons` ersetzt werden und das Vorkommen von `[]` durch `nil`.

10.2 Linksfaltung für Listen

Neben der Funktion `foldr` stellt Elm auch eine Funktion `foldl` zur Verfügung. Die Funktion `foldl` ist eine endrekursive Variante der Faltung. Endrekursiv bedeutet da-

bei, dass die letzte Aktion der Funktion der rekursive Aufruf ist. Die letzte Aktion der Funktion `foldr` ist zum Beispiel die Anwendung von `f` auf die Argumente `x` und `foldr f e xs`. Endrekursive Funktionen sind wichtig, da sie sich effizienter in Maschinencode übersetzen lassen als nicht endrekursive Funktionen. Bei der Ausführung eines Funktionsaufrufs muss auf dem Stack gespeichert werden, wie nach Beendung des Aufrufs fortgefahren wird. Wenn eine Funktion endrekursiv ist, kann diese Information einfach durch die neue Information ersetzt werden. Endrekursive Funktionen können daher auch ohne Verbrauch von Speicher auf dem Laufzeitstack übersetzt werden. Zu diesem Zweck muss der Compiler aber eine entsprechende Optimierung, die Endrekursions-Optimierung (*Tail Call Optimization*) implementieren. Viele Programmiersprachen, die nicht für Rekursion gedacht sind, reservieren standardmäßig einen vergleichsweise kleinen Bereich für den Stack. Daher kann es in solchen Sprachen bei der Verwendung von Rekursion schnell zu einem *Stack Overflow* kommen, das heißt, dass der Stack-Speicher komplett aufgebraucht ist. Wenn endrekursive Funktionen verwendet werden und der Compiler eine *Tail Cail Optimization* macht, kann es nicht zu einem *Stack Overflow* kommen. Diese Argumentation gilt nicht nur für Faltungen sondern für alle rekursiven Funktionen. Es kann sich also unter Umständen lohnen, eine Funktion endrekursiv zu definieren, um den Stack-Verbrauch gering zu halten.

Um zu verstehen, wie die Linksfaltung funktioniert, implementieren wir endrekursive Varianten von `sum`, `product` und `length`.

```
sum : List Int -> Int
sum =
    let
        sumIter acc list =
            case list of
                [] ->
                    acc

                x :: xs ->
                    sumIter (x + acc) xs
    in
    sumIter 0


product : List Int -> Int
product =
    let
        productIter acc list =
            case list of
                [] ->
                    acc

                x :: xs ->
                    productIter (x * acc) xs
    in
```

```
    productIter 1
```

Wir können aus dem Muster der Hilfsfunktionen wie im Fall von `foldr` die folgende abstrakten Variante definieren.

```
foldl : (a -> b -> b) -> b -> List a -> b
foldl func acc list =
    case list of
        [] ->
            acc

        x :: xs ->
            foldl func (func x acc) xs
```

Die Funktion überprüft, ob die Liste leer ist oder nicht. Falls die Liste nicht leer ist, berechnet die Funktion `func x acc` und ruft sich anschließend rekursiv auf. Das heißt, die Funktion `foldl` führt als letzte Aktion den rekursiven Aufruf durch.

Wie die Funktion `foldr` ersetzt auch die Funktion `foldl` alle Vorkommen des Konstruktors `::` durch eine Funktion `cons` und den Konstruktor `[]` durch `nil`, hierbei wird allerdings auch noch die Reihenfolge der Elemente in der Liste umgekehrt.

```
foldl cons nil [ a, b, c ]
=
foldl cons nil (a :: (b :: (c :: [])))
=
foldl cons nil ((::) a ((::) b ((::) c [])))
=
cons c (cons b (cons a nil))
```

Das Umkehren der Liste macht bei den bisherigen Funktionen keinen Unterschied, so liefern die Aufrufe `foldl (+) 0` und `foldl (*) 1` die gleichen Resultate wie die analogen Aufrufe mit `foldr`. In diesen beiden Fällen liegt das daran, dass `(+)` und `(*)` kommutativ und assoziativ sind. Im Allgemeinen liefern die beiden Funktionen aber nicht die gleichen Ergebnisse, wenn wir die gleichen Argumente verwenden. Der Aufruf `foldr (::) []` liefert zum Beispiel eine Funktion, die eine Liste ab und genau so wieder aufbaut. Im Gegensatz dazu liefert `foldl (::) []` eine Funktion, die eine Liste in umgekehrter Reihenfolge zurückliefert.

10.3 Faltungen auf anderen Datentypen

Die Idee der Funktion `foldr` lässt sich auch auf andere Datentypen übertragen. Der folgende Algorithmus liefert uns zu einem Datentyp `tau` den Typ der Faltung für diesen Datentyp.

1. Schreiben Sie die Typen aller Konstruktoren von `tau` auf.

2. Ersetzen Sie in den Typen der Konstruktoren den Datentyp `tau` durch eine noch nicht verwendete Typvariable.

3. Schreiben Sie die Signatur für `fold`, indem sie all diese Typen als Argumente übergeben, dann den Typ `tau` und die neugewählt Typvariable als Ergebnis.

Wir wollen diesen Algorithmus einmal am Beispiel des Listendatentyps durchführen. Das heißt, wie betrachten `tau = List a`. Die Typen der Konstruktoren von `tau` sind `[] : List a` und `(::) : a -> List a -> List a`. In diesen Typen ersetzen wir den Typ `List a` jetzt durch `b` und erhalten `b` und `a -> b -> b`. Die Signatur für die Faltung für Listen lautet dann also

```
fold : b -> (a -> b -> b) -> List a -> b
```

Wir wollen uns den Algorithmus einmal an Hand des folgenden polymorphen Baum-Datentyps anschauen.

```
data Tree a
  =  Leaf a
  |  Node (Tree a) (Tree a)
```

Die Typen der Konstruktoren sind `Leaf : a -> Tree a` und `Node : Tree a -> Tree a -> Tree a` und es gilt `tau = Tree a`. Wir ersetzen jetzt alle Vorkommen von `tau` durch `b` und erhalten `a -> b` und `b -> b -> b`. Daraus ergibt sich die folgende Signatur für die Faltung von `tau`.

```
fold : (a -> b) -> (b -> b -> b) -> Tree a -> b
```

Wir können diese Funktion nun wie folgt definieren.

```
fold : (a -> b) -> (b -> b -> b) -> Tree a -> b
fold leaf node tree =
    case tree of
        Leaf x ->
            leaf x

        Node lefttree righttree ->
            node (fold leaf node lefttree) (fold leaf node righttree)
```

Als Beispiel für die Verwendung von `fold` für den Datentyp `Tree` wollen wir eine Funktion definieren, die die Anzahl der Blätter in einem Baum zählt. Dazu definieren wir die Funktion erst einmal mit Hilfe expliziter Rekursion wie folgt.

```
leaves : Tree a -> Int
leaves tree =
    case tree of
        Leaf _ ->
            1

        Node lefttree righttree ->
            leaves lefttree + leaves righttree
```

Mit Hilfe der Funktion `foldTree` können wir diese Funktion wie folgt definieren.

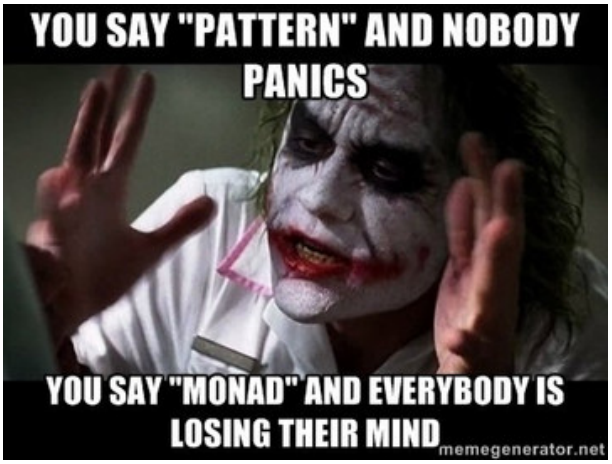
```
leaves : Tree a -> Int
leaves =
    fold (always 1) (+)
```

Als weiteres Beispiel können wir wie folgt eine Funktion definieren, die alle Blätter eines Baumes in einer Liste gesammelt zurückgibt.

```
flatten : Tree a -> List a
flatten =
    fold List.singleton (++)
```

11 Abstraktionen

Wir haben in verschiedenen Kontexten immer wieder die gleichen Funktionen kennengelernt. In diesem Kapitel wollen wir uns ein wenig den Hintergrund hinter diesen Funktionen anschauen. Die Konzepte die wir in diesem Kapitel lernen, sind vergleichbar mit *Pattern* in objektorientierten Sprachen. Das heißt, man identifiziert Funktionen, die man für verschiedene Datenstrukturen definieren kann und beschreibt, welche Eigenschaften diese Funktionen haben sollten.



11.1 Funktoren

Wir haben die Funktion `map` kennengelernt, die auf vielen verschiedenen Datentypen definiert werden konnte. Wir haben zum Beispiel die folgenden Funktionen kennengelernt.

```
map : (a -> b) -> List      a -> List      b
map : (a -> b) -> Decoder   a -> Decoder   b
map : (a -> b) -> Generator a -> Generator b
```

Diese Signaturen unterschieden sich nur in dem Typkonstruktor, für den sie definiert sind. Das heißt, es gibt eine Definition von `map` für den Typkonstruktor *`List`*, eine Definition für den Typkonstruktor *`Decoder`* und eine Definition für den Typkonstruktor *`Generator`*. Das heißt, die Funktion `map` hat immer die Form

```
map : (a -> b) -> f a -> f b,
```

wobei `f` ein Typkonstruktor ist. Man bezeichnet einen Typkonstruktor `f`, für den es eine Funktion `map` gibt, als Funktor. Es gibt noch viele weitere Typkonstruktoren, für die wir eine Funktion `map` definieren können. Neben den Implementierungen von `map`, die wir kennengelernt haben, gibt es zum Beispiel noch die folgenden Funktionen.

```
map : (a -> msg) -> Cmd  a -> Cmd  msg
map : (a -> msg) -> Sub  a -> Sub  msg
map : (a -> msg) -> Html a -> Html msg
```

Zur Illustration wollen wir eine weitere Variante der Funktion `map` definieren, diesesmal für den Typkonstruktor *Maybe*.

```
map : (a -> b) -> Maybe a -> Maybe b
map func maybev =
    case maybev of
        Nothing ->
            Nothing

        Just v ->
            Just (func v)
```

Leider können wir auch eine Funktion vom Typ `(a -> b) -> Maybe a -> Maybe b` definieren, die keine „sinnvolle“ Implementierung darstellt.

```
mapWeird : (a -> b) -> Maybe a -> Maybe b
mapWeird _ _ =
    Nothing
```

Um solche Implementierungen zu vermeiden, sollte die Implementierung der Funktion `map` für jeden Typkonstruktor bestimmte Gesetze erfüllen. Das heißt, die Funktion muss den angegebenen Typ haben und sich auf gewisse Weise verhalten. Die Funktion `map` muss für alle möglichen Werte für `fx`, `f` und `g` die folgenden beiden Gesetze erfüllen.

```
map (\x -> x) fx = fx
map (\x -> g (f x)) fx = map g (map f fx)
```

Die Funktion `mapWeird` erfüllt zum Beispiel das erste Gesetz nicht, da für `fx = Just 42` die erste Gleichung nicht erfüllt ist.

```
mapWeird (\x -> x) (Just 42)
=
Nothing
/=
Just 42
```

11.2 Applikative Funktoren

Wir haben die folgende Funktion kennengelernt, um aus „einfachen“ Decodern einen komplexeren zusammenzubauen.

```
apply : Decoder a -> Decoder (a -> b) -> Decoder b
apply =
    Decode.map2 (|>)
```

Auch die Funktion `apply` kann für verschiedene Typkonstruktoren definiert werden. So können wir in Elm zum Beispiel die folgenden Funktionen definieren.

```
apply : List      a -> List      (a -> b) -> List      b
apply : Decoder   a -> Decoder   (a -> b) -> Decoder   b
apply : Generator a -> Generator (a -> b) -> Generator b
```

Damit ein Typkonstruktor `f` ein applikativer Funktor ist, muss es eine Funktion

```
apply : f a -> f (a -> b) -> f b
```

geben (Mcbride und Paterson, 2008). Damit `f` ein applikativer Funktor ist, muss es auch noch eine Funktion `pure : a -> f a` geben. Es gibt eine solche Funktion für alle drei Typkonstruktoren, sie heißt nur immer anders. Im Fall von *`List`* ist die Funktion `pure` zum Beispiel `singleton`, im Fall von *`Decoder`* ist es `succeed`.

Zur Illustration wollen wir die Funktion `map` für den Typkonstruktor *`Maybe`* einmal definieren.

```
apply : Maybe a -> Maybe (a -> b) -> Maybe b
apply maybev maybef =
    case maybev of
        Nothing ->
            Nothing

        Just v ->
            case maybef of
                Nothing ->
                    Nothing

                Just f ->
                    Just (f v)
```

Wir definieren außerdem die Funktion `pure : a -> f a` für *`Maybe`*.

```
pure : a -> Maybe a
pure =
    Just
```

Im Gegensatz zu `map` können wir mit `apply` zwei Strukturen kombinieren. Im Fall des Typkonstruktors *`Decoder`* haben wir zum Beispiel gesehen, dass wir mithilfe der Funktion `apply` aus zwei einfachen Decodern einen komplexeren Decoder bauen können. Im Fall von *`Maybe`* können wir `apply` auch nutzen, um zwei Werte zu kombinieren. Wir betrachten das folgende Beispiel. Wir wollen vom Benutzer zwei Zahlen einlesen und diese addieren. Wir nutzen dazu die Funktion *`String`*`.toInt : String -> Maybe Int`. Da das Parsen von beiden Eingaben möglicherweise fehlschlagen kann, müssen wir zwei Werte vom Typ *`Maybe Int`* kombinieren. Dazu können wir die Funktion `apply` nutzen.

```
add : String -> String -> Maybe Int
add userInput1 userInput2 =
```

```
    pure (+)
        |> apply (String.toInt userInput1)
        |> apply (String.toInt userInput2)
```

Damit ein Typkonstruktor ein applikativer Funktor ist, müssen die Funktionen `pure` und `apply` ebenfalls Gesetze erfüllen. Auf diese Gesetze wollen wir hier aber nicht eingehen. Es sei an dieser Stelle aber noch kurz erwähnt, dass jeder applikative Funktor auch ein Funktor ist. Wir können die Funktion `map` nämlich mit Hilfe von `pure` und `apply` definieren.

```
map : (a -> b) -> Maybe a -> Maybe b
map func maybe =
    apply maybe (pure func)
```

11.3 Monaden

In der funktionalen Programmierung gibt es eine ganze Reihe von Abstraktionen wie Funktor und applikativer Funktor. Wir wollen uns an dieser Stelle noch eine dieser Abstraktionen anschauen, die Monade heißt und vergleichsweise legendär auch außerhalb der funktionalen Programmierung ist.

Es gibt einige Funktionen, die sich mit Hilfe eines applikativen Funktors nicht ausdrücken lassen. Wir betrachten dazu das Beispiel

`apply : Decoder a -> Decoder (a -> b) -> Decoder b`.

Für unser Beispiel gehen wir davon aus, dass die JSON-Struktur, die wir verarbeiten wollen ein Feld mit der Version der Schnittstelle hat. Abhängig von der Version wollen wir jetzt dei einen oder anderen *Decoder* verwenden. Wir definieren dazu erst einmal einen *Decoder*, der die Version liefert.

```
version : Decoder Int
version =
    Decode.field "version" Decode.int
```

Außerdem haben wir die folgenden beiden *Decoder* bei denen sich der Name des Feldes geändert hat.

```
decoderVersion1 : Decoder Bool
decoderVersion1 =
    Decode.field "bool" Decode.bool


decoderVersion2 : Decoder Bool
decoderVersion2 =
    Decode.field "boolean" Decode.bool
```

Wir möchten jetzt gern einen *Decoder* definieren, der abhängig von der Version `decoderVersion1` oder `decoderVersion2` verwendet. Diese Art von *Decoder* können wir mit Hilfe von `apply` aber nicht definieren. Das Problem besteht darin, dass wir abhängig von einem Wert den *Decoder* bestimmen möchten. Das Argument *Decoder* `(a -> b)` erlaubt es aber nicht, den *Decoder* danach zu wählen, welchen Wert wir als `a` übergeben bekommen.

Zu diesem Zweck können wir die folgende Funktion verwenden.

```
andThen : (a -> Decoder b) -> Decoder a -> Decoder b
```

Hier haben wir statt eines Arguments *Decoder* `(a -> b)` jetzt ein `a -> Decoder b`. Das heißt, wir können vom konkreten Wert, der vom Typ `a` übergeben wird, den *Decoder* wählen, den wir anschließend verwenden. Wir können den folgenden *Decoder* definieren.

```
decoder : Decoder Bool
decoder =
    let
        chooseVersion v =
            case version of
                1 ->
                    decoderVersion1

                2 ->
                    decoderVersion2

                _ ->
                    Decode.fail
                        ("Version "
                            ++ String.fromInt v
                            ++ " not supported"
                        )
    in
    version
        |> Decode.andThen chooseVersion
```

Neben der Funktion `andThen` muss ein Typkonstruktor `f`, der eine Monade ist, noch eine Funktion `return : a -> f a` zur Verfügung stellen. Im Fall von *Decoder* ist `return` wie folgt definiert.

```
return : a -> Decoder a
return =
    Decode.succceed
```

Wie beim Funktor und beim applikativen Funktor müssen die Funktionen einer Monade auch Gesetze erfüllen.

```
andThen f (return x) = f x
andThen return fx = fx
andThen (\x -> andThen g (f x)) fx = andThen g (andThen f fx)
```

Wenn ein Typkonstruktor eine Monade ist, dann ist er auch ein applikativer Funktor. Wir können nämlich wie folgt die Funktionen eines applikativen Funktors definieren, indem wir die Funktionen der Monade verwenden.

```
pure : a -> Decoder a
pure =
    return


apply : Decoder a -> Decoder (a -> b) -> Decoder b
apply dx df =
    Decode.andThen (\x -> Decode.andThen (\f -> return (f x)) df) dx
```

Die Typeclassopedia bietet noch weitere Informationen zu Abstraktionen in der funktionalen Programmierung.

12 Weitere Themen

In diesem Kapitel wollen wir uns noch ein paar abschließende Themen anschauen, die bei der Programmierung mit Elm relevant sein können.

12.1 Spezielle Typvariablen

Einige Funktionen wie zum Beispiel die Funktion `(<)` lassen sich auf verschiedene Typen anwenden. Wir können zum Beispiel den Aufruf `3 < 4`, aber auch `3.4 < 4.3` sowie `"Schmidt" < "Smith"` auswerten. Mit den bisher bekannten Sprachkonstrukten könnten wir der Funktion `(<)` nur den Typ `a -> a -> Bool` geben. Dies würde aber bedeuten, dass wir die Funktion `(<)` für alle möglichen Typen verwenden können. So müsste Elm etwa den Aufruf `(+) < (*)`, das heißt, den Vergleich von zwei Funktionen, akzeptieren. Elm unterstützt für dieses Problem leider bisher nur eine Ad-hoc-Lösung. Es ist schon seit längerer Zeit eine alternative Lösung für dieses Problem geplant, bisher ist aber keine Entscheidung für eine der möglichen Alternativen gefallen.

Es gibt spezielle Namen für Typvariablen, die ausdrücken, dass der Typ nicht komplett polymorph ist, sondern nur bestimmte Typen für die Typvariable eingesetzt werden können. Der Typ der Funktion `(<)` ist zum Beispiel

`comparable -> comparable -> Bool`.

Das heißt, wir können für die Typvariable `comparable` nur Typen einsetzen, die vergleichbar sind. Wenn wir die Funktion `(<)` zur Definition einer anderen Funktion nutzen, erhält auch diese aufrufende Funktion diese spezielle Form von Typvariable. Der Typ der Funktion `List.maximum` ist zum Beispiel

`List comparable -> Maybe comparable`.

Das heißt, wir können nur zu einer Liste von vergleichbaren Elementen das Maximum bestimmen. Vergleichbar sind in Elm die Typen *String*, *Char*, *Int*, *Float*, *Time*, sowie Listen und Tupel von vergleichbaren Typen.

Wenn wir versuchen, den Ausdruck `(+) < (*)` in Elm zu verwenden, erhalten wir den folgenden Fehler.

```
-- TYPE MISMATCH ------------------------------------------------------------ REPL

I cannot do a comparison with this value:

3|   (+) < (*)
     ^^^
This ‘+‘ value is a:

    number -> number -> number
```

```
But (<) only works on Int, Float, Char, and String values. It can work on lists
and tuples of comparable values as well, but it is usually better to find a
different path.

Hint: I only know how to compare ints, floats, chars, strings, lists of
comparable values, and tuples of comparable values.
```

Das heißt, wir erhalten einen Fehler, wenn wir das Programm übersetzen.

Neben `comparable` gibt es noch die Typvariable `number`, die für die Typen *`Int`* und *`Float`* genutzt werden kann. Die Konstante `1` hat zum Beispiel den Typ `number` und die Funktion `(+)` hat den Typ `number -> number -> number`. Außerdem gibt es noch die Typvariable `appendable`, die Typen repräsentiert, die sich mit `(++)` konkatenieren lassen, das sind die Typen *`String`* und *`List`*.

Der Hauptkritikpunkt an dieser Art von Lösung besteht darin, dass der Nutzer keine weiteren Typen hinzufügen kann. Das heißt, alle Typen, die nicht von Haus aus zu den vergleichbaren Typen gehören, können mit Hilfe von `(<)` nicht verglichen werden. Eine Lösung für dieses Problem stellen zum Beispiel Typklassen dar, die in der funktionalen Programmiersprache Haskell genutzt werden, um Funktionen überladen zu können. In diesem Fall kann der Nutzer auch selbst Instanzen für eine Funktion hinzufügen. Eine Erweiterung von Elm um Typklassen oder ein vergleichbares Feature ist geplannt, hat aber keine hohe Priorität[1].

12.2 Interop mit JavaScript

Um in Elm mit JavaScript-Code zu kommunizieren, kann man Ports verwenden. Ein Port besteht dabei aus zwei Komponenten, einer Komponente, die Informationen an den JavaScript-Code schickt und einer Komponente, die informiert wird, wenn der JavaScript-Code ein Ergebnis produziert hat. Um Informationen an den JavaScript-Code zu senden, wird ein Kommando genutzt und um über ein Ergebnis informiert zu werden, nutzt man ein Abonnement.

Bisher haben wir Elm-Anwendungen ausgeführt, indem wir `elm reactor` genutzt haben. Um einen Port zu verwenden, müssen wir aber Zugriff auf den JavaScript-Code haben, der ausgeführt wird. Um dies zu erreichen, können wir `elm make Snake.elm` aufrufen, wobei `Snake.elm` den Elm-Code enthält. Dieser Aufruf erzeugt eine HTML-Datei, in die der gesamte erzeugte JavaScript-Code eingebettet ist[2]. Im erzeugten JavaScript-Code wird eine Zeile der folgenden Art genutzt, um die Elm-Anwendung zu starten.

```
var app = Elm.Snake.init({ node: document.getElementById("elm") });
```

Wir wollen uns jetzt zuerst anschauen, wie wir eine JavaScript-Funktion aus dem Elm-Code heraus aufrufen können. Ein Modul, das Ports verwendet, muss mit den Schlüsselwörtern `port module` starten. Als Beispiel fügen wir die folgende Zeile in unser Elm-Programm ein.

[1] `https://github.com/elm-lang/elm-compiler/issues/38`

[2] Alternativ kann man mit Hilfe des Parameters `-output` auch dafür sorgen, dass der JavaScript-Code in eine JavaScript-Datei geschrieben wird.

```
port callFunction : String -> Cmd msg
```

Hier definieren wir, dass wir einen *String* an eine JavaScript-Funktion übergeben möchten. Um diese Aktion auszuführen, nutzen wir das gewohnte Konzept eines Kommandos. Auf JavaScript-Ebene können wir mit dem folgenden Code einen *Callback* registrieren, der aufgerufen wird, wenn wir in unserer Elm-Anwendung das Kommando ausführen, das wir von `callFunction` erhalten.

```
app.ports.callFunction.subscribe(function(str) {
  ...
});
```

Anstelle des `...` können wir JavaScript-Code ausführen, der den übergebenen *String* in der Variable `str` nutzt.

Um informiert zu werden, wenn dieser *Callback* seine Ausführung beendet hat, nutzen wir ein Abonnement. Wir definieren dazu zuerst den folgenden Port in unserer Elm-Anwendung.

```
port returnResult : (String -> msg) -> Sub msg
```

Wir modellieren hier eine Funktion, die ebenfalls einen *String* als Ergebnis liefert. Mit Hilfe dieser *Subscription* können wir uns in der Elm-Anwendung informieren lassen, wenn der JavaScript-Code ein Ergebnis liefert. In der JavaScript-Anwendung rufen wir an einer beliebigen Stelle den folgenden Code auf.

```
app.ports.returnResult.send(...);
```

Das `...` ist dabei der *String*, den wir an die Elm-Anwendung geben möchten. Wenn im JavaScript-Code diese Zeile aufgerufen wird, wird die Elm-Anwendung über das entsprechende Abonnement darüber informiert.

Die Seite JavaScript Interop gibt noch mal eine etwas ausführlichere Einführung in die Verwendung von Ports.

12.3 Umsetzung einer größeren Anwendung

Die Seite Web Apps des Elm-Guide bietet noch weitere Themen, die bei der Umsetzung einer größeren Anwendung relevant sein können. Dazu gehört zum Beispiel die Umsetzung einer Anwendung mit Routing, also eine Anwendung, die aus mehreren Seiten besteht. Die Seite enthält außerdem Tipps, was man bei der Umsetzung einer Anwendung mit mehreren Modulen beachten sollte.

Die Seite Optimization bietet Informationen zur Performance einer Webanwendung. Dort findet sich zum Beispiel eine Erklärung des *virtual DOM*, der dafür sorgt, dass das Rendern von HTML im Browser effizient durchgeführt wird, obwohl die Funktion `view` immer die gesamte HTML-Struktur als Ergebnis liefert. Das Kapitel stellt außerdem die Funktion `lazy : (a -> Html msg) -> a -> Html msg` vor, die Caching von Funktionsaufrufen implementiert. Das heißt, wenn man eine Funktion hat, die eine HTML-Struktur liefert, kann man mit Hilfe von `lazy` dafür sorgen, dass diese Funktion nur ausgeführt wird, wenn sich die Argumente der Funktion im Vergleich zum vorherigen Aufruf geändert haben.

Literatur

Landin, P. J. (März 1966). „The next 700 Programming Languages“. In: *Communications of the ACM* 9.3, S. 157–166. ISSN: 0001-0782. DOI: 10.1145/365230.365257.

Mcbride, Conor und Ross Paterson (Jan. 2008). „Applicative Programming with Effects“. In: *Journal of Functional Programming* 18.01. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796807006326.