Lambda Kalkül

Der λ-Kalkül, oder im Englischen Lambda Calculus genannt, geht auf Alonzo Church (1903–1995) zurück und bildet die theoretische Grundlage der funktionalen Programmierung. Es ist ein in den 1930er-Jahren geschaffener Formalismus zur Darstellung der berechenbaren Funktionen.

Was ist eine Funktion?

Dies ist dir sicher noch aus dem Mathematikunterricht in Erinnerung. Aber keine Sorge, es wird jetzt nicht wirklich mathematisch, versprochen :)

Eine Funktion im mathematischen Sinne ist eine Zuordnung von zwei Werten. Das heißt, für jeden möglichen Eingabewert wird immer genau ein Rückgabewert geliefert.
Die elementarste Funktion ist die **Identitätsfunktion**:

```
f(x) = x
```

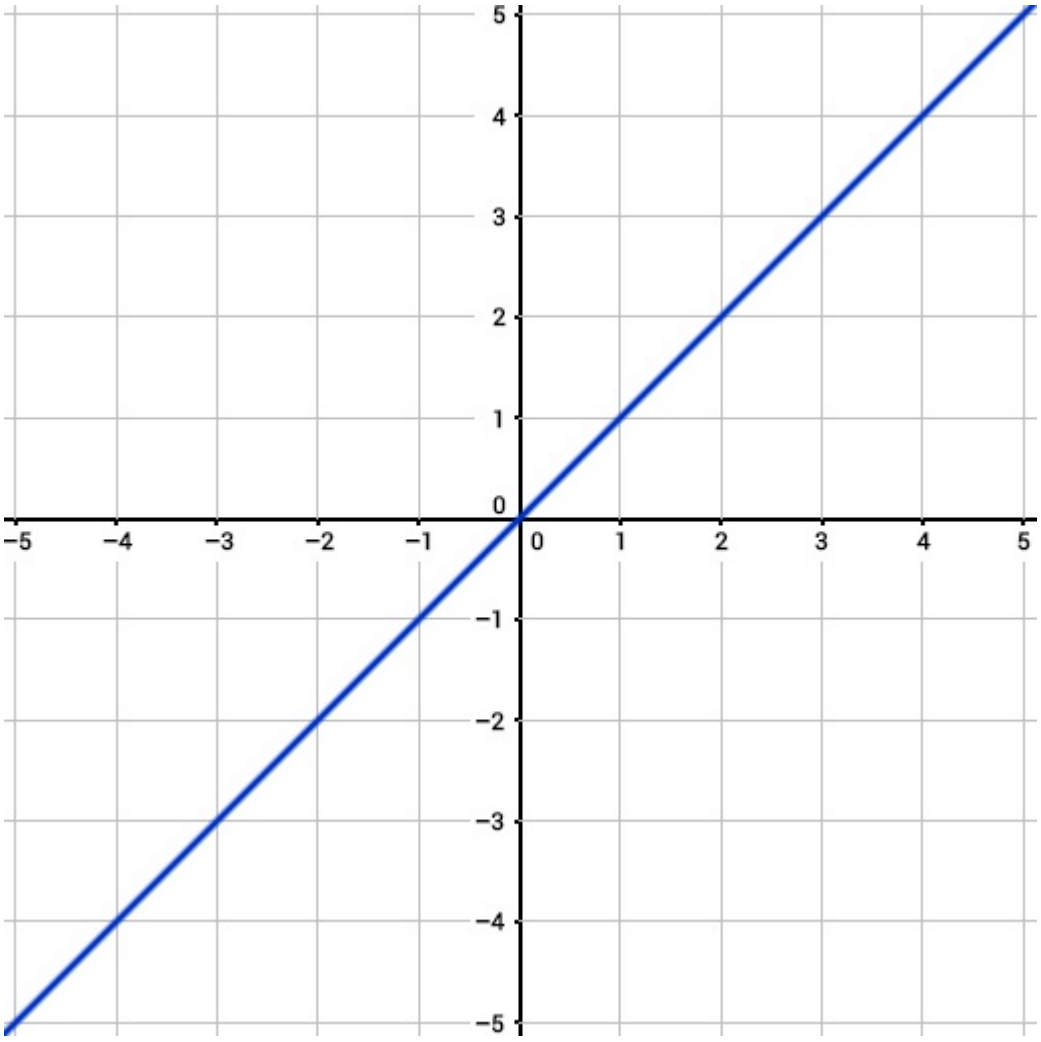
Vielleicht kennst du auch noch die Schreibweise, die dasselbe bedeutet:

```
f: x ↦ x
```

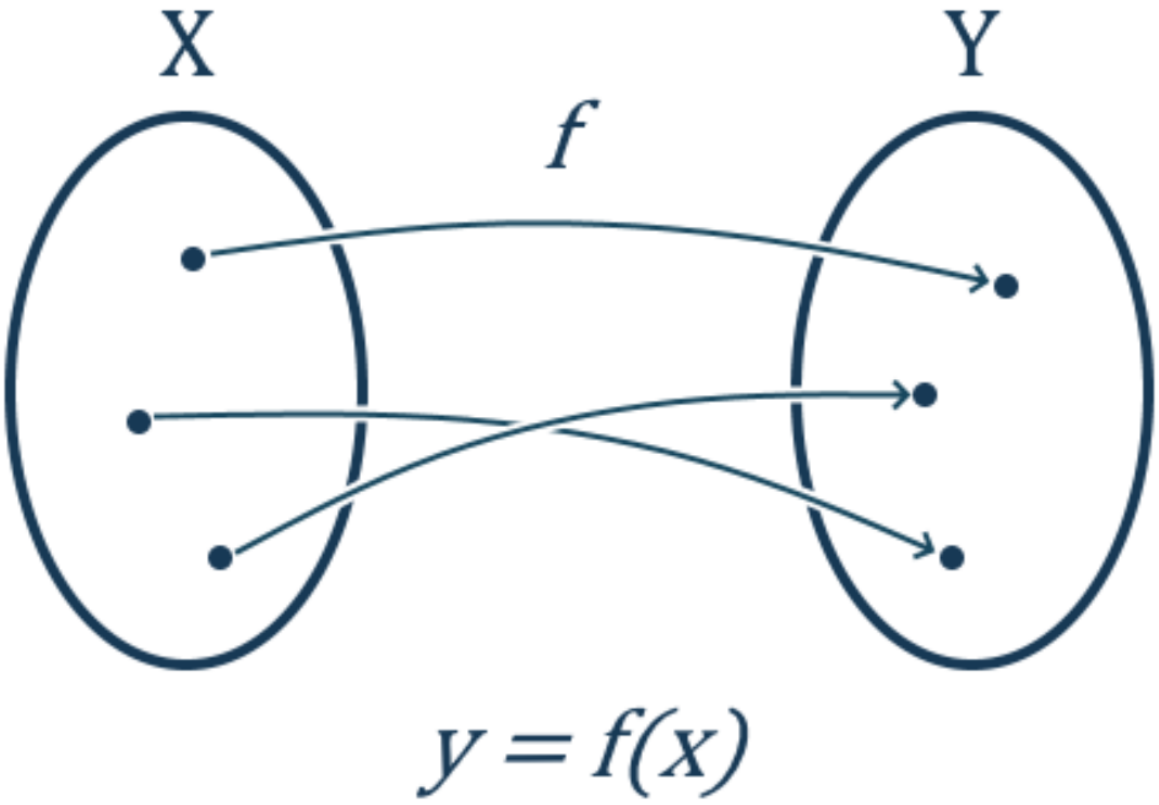
Für x kann nun eine beliebig Zahl eingesetzt werden, z.b. ist hier die Eingabe 2 und die Ausgabe ebenso 2:

```
f(2) = 2
```

Sehr anschaulich können Funktionen auch im Koordinatensystem dargestellt werden. Die Identitätsfunktion schaut so aus:



Des Weiteren, können Funktionen auch als Beziehungen von Mengen dargestellt werden, was dir vom Mathematikunterricht sicher auch noch bekannt vorkommen wird:



Andere Beispiele für Funktionen, die in der Mathematik öfters verwendet werden, sind Sinus-Funktion, Kosinus-Funktion, Wurzelfunktion, usw.

Aufgabe 1: Mathematische Funktionen

Denke dir andere Funktionen aus. Wende dann auf diese Funktionen verschiedene Eingaben an, und stelle die Funktion im Koordinatensystem dar.

Eine Funktion in einer funktionalen Programmiersprache ist der mathematischen Funktion zur Gänze angelehnt. Hier wird immer ein bestimmter Wert oder eine bestimmte Ausgabe für eine gegebene Eingabe berechnet. Eine Funktion hat in der Regel eine oder mehrere Eingabeparameter und einen Rückgabewert. Eine Funktion wird aufgerufen, indem ihr der/die entsprechenden Argumente übergeben werden.

Das λ-Kalkül arbeitet mit sogenannten λ-Termen. Diese sind aus 3 Bausteinen zusammengesetzt und wie folgt definiert:

T ist ein λ-Term, wenn eine der folgenden Bedingungen erfüllt ist:

1. **Variablen**: T ist von der Form a, wobei a ein Variablenname ist. Der Wert von T ist dann die Belegung der Variablen a.
   Beispiel: meist einzelne Kleinbuchstaben wie etwa x,y,z, ...

2. **Funktionsanwendung (Applikation)**: T ist von der Form (T_1 T_2), wobei T_1 und T_2 λ-Terme sind.
   Der Wert von T ist das Ergebnis der Anwendung der Funktion T_1 auf den Wert des Terms T_2.

Eine Applikation geschieht dann, wenn die Funktion auf ein konkretes Argument angewendet wird. Zum Beispiel wird hier die 3 auf die Identitätsfunktion angewendet, die wir oben kennengelernt haben. Dabei bildet (λx.x) die Funktion mit einem Parameter x, und die Zahl 3 bildet das Argument.

```
(λx.x) 3
```

Jetzt wird der Funktionskörper, also alles was nach dem Punkt kommt, ersetzt mit der 3 und das Lambda wird dadurch aufgelöst. Übrig bleibt dann als Ergebnis nur noch die 3.

```
3
```

Diese Applikation ist auch unter dem Begriff ***Beta Reduktion*** bekannt, die wir anschließend gleich in Aufgabe 2 üben werden.

3. **Abstraktion**: T ist von der Form λa.T_1 wobei T_1 wieder ein λ-Term und a eine Variable ist. Der Wert von T ist eine einstellige Funktion mit der Funktionsvorschrift T_1 die nach dem Parameter a abstrahiert ist.

Beispiel: λx.x (=Identitätsfunktion)

Mit diesen 3 aufgelisteten Bestandteilen, ist das Lambda Kalkül vollständig. Es gibt also keine Zahlen, Funktionsnamen, arithmetische Funktionen oder Wahrheitswerte.

Aufgabe 2: Applikation

Wie oben dargestellt, wird in der Applikation jedes Vorkommen der Variablen im "Körper" der Funktion mit dem eingesetzten Term ersetzt.

Beispiel:

```
(λf.f 1) (λx.x)
```

ersetze *f* mit *λx.x*:

```
= (λx.x) 1
```

ersetze *x* mit *1*:

```
= 1
```

Versuche analog dazu, folgende Applikation durchzuführen:

- (λx.x+1): mit dem Argument x=5
- (λx.λy.xy): mit den Argumenten x=2 and y=2
- (λx. 2x+3): mit dem Argument x=10
- (λx.λy.x - y): mit den Argumenten x=10 and y=5
- (λx.x > 5): mit dem Argument 0 (Hinweis: das Resultat ist entweder TRUE oder FALSE)
- (λx.x y): mit dem Argument (λy.y x)

Funktionsanwendungen können auch **verkettet** werden, um komplexere Ausdrücke zu erstellen. Zum Beispiel ist eine Funktion h gegeben mit

```
h = λx.x^2
```

und eine Funktion g

```
g = λx.λy.x*y
```

die verkettete Funktionsanwendung h(g(2,3)) wäre folgendermaßen geschrieben:

```
h.(g.2.3)
```

bzw.

```
h.((λx.λy.x*y).2.3)
```

Das Resultat wäre dann 36.

In der folgenden Tabelle sehen wir die Gegenüberstellung der mathematische Schreibweise und der Lamba-Notation:

mathematische Schreibweise	λ-Term
$f(x) = x^2$	$\lambda x.x^2$
$f(x,y) = x^2 + y^2$	$\lambda x.\lambda y.(x^2+y^2)$

Aufgabe 3: Funktionen in den verschiedenen Darstellungsarten

Denke dir andere Funktionen aus (mit 1,2 und 3 verschiedenen Inputparametern), und gib sie sowohl in der konventionellen Form, als auch als λ Form an.

Lambda Ausdrücke in Scala

In den meisten Programmiersprachen, so auch in Scala, wird der Lambda Ausdruck mit einem Pfeil dargestellt. So wird beispielweise der Lambda Ausdruck λx.x+1 dargestellt:

```
x -> x+1
```

Hier sehen wir, dass x der Parameter ist, und x+1 der Funktionskörper bzw. der Rückgabewert der Funktion ist.
Im Programmierjargon wird dies auch eine anonyme Funktion genannt, oder einfach nur ein Lambda. Der Name *anonym* erklärt sich dadurch, dass die Funktionen selbst keinen Namen haben. Bei den Programmierübungen wirst du dann selbst Lambdas schreiben, jedoch werden später bei der Programmierung die Funktionen meist mit einem treffenden und beschreibenden Namen benannt sein.

Diese anonyme Funktion kann auch einer Variable zugewiesen werden. So wird hier z.B. die quadratische Funktion in der Variable square gespeichert.

```
val square = (x: Int) => x * x
```

Diese Funktion kann einer Variablen zugewiesen und dann wie eine normale Funktion aufgerufen werden:

```
val result = square(3)
```

Anonyme Funktionen sind selbstverständlich auch mit mehreren Parametern möglich, wie z.b. die Addierfunktion:

```
val add = (x: Int, y: Int) => x + y
```

Zuweisung an die Variable result:

```
val result = add(3, 4)
```

Aufgabe 4: Anonyme Funktionen

Aufgabe: Versuche auch hier, dir eigene anonyme Funktionen auszudenken und schreibe sie in der Scala Form an. Vergiss auch nicht, diese Funktionen einer Variable zuzuweisen.