2 Über den Umgang mit mathematischen Notationen

Viele Notationen sind historisch gewachsen, und es sind zum Teil dabei auch unterschiedliche, untereinander konkurrierende Notationen entstanden. Man kann bei der Angabe einer Menge einen Doppelpunkt oder einen senkrechten Strich verwenden; man kann e^x oder auch $\exp(x)$ schreiben, etc. So schreibt man in einigen Fällen das Funktions- oder Relationszeichen vor, hinter oder auch zwischen die Argumente. Es soll in diesem Kapitel um solche notationellen Besonderheiten gehen und wie man mit diesen umgehen sollte. Wie drückt man beispielsweise aus, dass man mit $f(x)$ das eine Mal einen konkreten Funktionswert meint, das andere Mal die Funktion als Ganzes? Beim Notieren mathematischer Texte – sollte man da eher die umgangssprachliche Notation „genau dann wenn" verwenden oder das Symbol $\Leftrightarrow$? In diesem Kapitel geht es um solche Fragestellungen, bis hin zu dem Punkt, dass eine notationelle Maschinerie, die man mal aufgebaut hat, einem, bei unbedachtem Umgang, in Form von Paradoxien „um die Ohren fliegen" kann.

2.1 Infix, Präfix, Postfix

Zusätzlich zur Zeichenfestlegung von Operatoren, z.B. „+" für die Addition, ist die Reihenfolge von Funktionen und Operanden ein wichtiger Aspekt der Notation in Mathematik, Logik und Informatik.
Beispiele hierfür sind:

Name	Beschreibung	Beispiele	Programmiersprachen
Präfixnotation	Funktion vor Operanden	$\sin x$, $-x$	Lisp: (+ 3 4)
Postfixnotation	Funktion nach Operanden	$x!$	PostScript: 3 4 add
Infixnotation	Funktion zwischen Operanden	$x + y$, $x \wedge y$	C, Java: 3 + 4

Bei der gebräuchlichen Infixnotation in der Arithmetik wird neben der Reihenfolge von Funktion und Operanden auch die Rechenreihenfolge durch die Wertigkeit der Opera-

tionen („Punkt vor Strich") bestimmt. Durch das Setzen von Klammern kann man Teilausdrücke festlegen, die abweichend von der Punkt- vor Strich-Regel zuerst berechnet werden müssen.
Beispiel: $(3 + 4 + 5) \cdot 6 \cdot 7 + 8$. Die Operatoren stehen hierbei zwischen den einzelnen Werten, es handelt sich hierbei also um eine Infixnotation. Die Klammern erzwingen, dass die Addition von 3, 4 und 5 vor der nachfolgenden Multiplikation mit 6 zu geschehen hat, entgegen der Punkt-vor-Strich-Regel, die dann anzuwenden wäre, wenn keine Klammern vorhanden sind.
Die Präfixnotation wird auch **Polnische Notation** genannt[1]. Wenn die Stelligkeit der Funktion f bekannt ist (also die Anzahl der Argumente von f), so benötigt man eigentlich keine Klammern, um eine solche Formel eindeutig lesen zu können. So findet sich die klammerlose Schreibweise

$$f\ x_1\ x_2\ \ldots\ x_n$$

meist in der formalen Logik, und die Schreibweise

$$f(x_1, x_2, \ldots, x_n)$$

in der gängigen Mathematik. Die Schreibweise mit äußeren Klammern

$$(\ f\ x_1\ x_2\ \ldots\ x_n\ )$$

wird in der Programmiersprache Lisp verwendet.
Bei der Postfixnotation schreibt man den Operator *nach* den zu verknüpfenden Argumenten; sie wird daher auch **Umgekehrte Polnische Notation** (UPN) genannt. In der Informatik ist die UPN deshalb von Interesse, weil sie eine stapelbasierte Abarbeitung ermöglicht: Operanden werden beim Lesen auf den Stapel (Stack) gelegt, ein Operator holt sich die Anzahl an Operanden vom Stapel, die seiner Stelligkeit entspricht und legt das Ergebnis der Operation wieder auf dem Stapel ab. Am Ende liegt dann das Ergebnis des Terms oben auf dem Stapel. Deshalb bildet die UPN die Grundlage für stapelbasierte Programmiersprachen wie Forth oder PostScript.
Weiterhin übernahm die Firma Hewlett-Packard (seit den 60er Jahren) die UPN für ihre Taschenrechner. Hier hat die UPN auch ganz praktische Bedeutung, da hierbei die Berechnung eines Ausdrucks im Allgemeinen mit weniger Tastendrücken auskommt.

[1] Nach dem polnischen Mathematiker Jan Łukasiewicz (1878–1956).

Beispielsweise benötigt die Berechnung von $(4 + 5) * (8 + 9)$ auf einem Taschenrechner mit algebraischer Eingabelogik 12 Tastendrücke, nämlich „(4 + 5) * (8 + 9) = “, während nur 9 Tastendrücke, d.h. „4 ENTER 5 + 8 ENTER 9 + * “, auf einem UPN Taschenrechner benötigt werden. Zusätzlich werden noch die jeweiligen Zwischenergebnisse angezeigt.

2.2 Funktionswert vs. Funktion, λ-Notation

Nehmen wir an, es wurde eine Funktion f definiert (z.B. auf den natürlichen Zahlen). Was für ein Objekt ist dann $f(x)$? Es gibt zwei Möglichkeiten, die auch in der mathematischen Alltagsnotation mal so oder so gemeint sein können. Wenn wir $x \in \mathbb{N}$ als eine feste Zahl ansehen, so ist ebenfalls $f(x) \in \mathbb{N}$. Wenn wir x als Variable, als formalen Platzhalter für das Argument von f ansehen, so meint $f(x)$ dasselbe wie f, also die Funktion als Ganzes. Dann ist $f(x)$ also ein Element von $\mathbb{N}^{\mathbb{N}}$, der Menge aller Funktionen von $\mathbb{N}$ nach $\mathbb{N}$.

Gelegentlich sieht man in der Literatur die Bezeichnungsweise $f(\,\cdot\,)$, um einerseits zum Ausdruck zu bringen, dass die Funktion f ein Argument hat, dieses Argument aber namentlich nicht benennt, um die oben angesprochene Mehrdeutigkeit zu vermeiden. Durch diese Notation ist dann die Funktion f als Ganzes gemeint.

Um diese Mehrdeutigkeit auszuschließen, wurde der λ-Kalkül bzw. die λ-Notation eingeführt. Man unterscheidet hier klar zwischen der Anwendung einer Funktion auf ein bestimmtes Argument, was den Funktionswert ergibt, und der Funktion als Ganzes. Die folgende Notation geht auf Church[2] zurück. Es bezeichnet $f(\alpha)$ – meist klammerfrei geschrieben, also $f\,\alpha$ – die Anwendung der Funktion f auf das Argument α (welches selber ein komplexer λ-Ausdruck sein kann). Dahingegen bedeutet $\lambda x.T$ eine Funktion mit einem Argument x; diese Funktion wird über den Ausdruck T definiert. Auf diese Weise können Funktionen „anonym” eingeführt werden; wir könnten ihr jedoch auch einen Namen geben, indem wir $f = \lambda x.T$ schreiben. In diesem Sinne ist λ ein „Funktionsbildungs-Operator“.

Es folgen einige Beispiele:

$$\lambda x.\,(x + 3)^2$$

[2] Alonzo Church (1903–1995), amerikanischer Mathematiker und Logiker.

Dies bezeichnet die (unbenannte oder anonyme) Funktion mit einem Argument x, welche ihr um 3 erhöhtes Argument quadriert. Dagegen bedeutet

$$\lambda x.\,(x+3)^2\ \ 5$$

dass die soeben beschriebene Funktion auf das Argument 5 angewandt wird.
Man hat einen Kalkül entwickelt, um mit diesen λ-Ausdrücken umgehen und sie vereinfachen zu können. Die wichtigste Operation ist hierbei die λ-**Konversion**, die das Einsetzen des Arguments in den Parameter vornimmt. Beispiel:

$$\lambda x.\,(x+3)^2\ \ 5 \quad \overset{\lambda\text{-Konversion}}{\longmapsto} \quad (5+3)^2$$

Der Lambda-Kalkül hat die Entwicklung funktionaler Programmiersprachen wie **Lisp**[3] und **Haskell**[4] und deren Semantik und Auswertungsprinzipien wesentlich beeinflusst. In funktionalen Sprachen besteht ein Programm aus Funktionsdefinitionen, Funktionsanwendungen und Funktionskompositionen (siehe Abschnitt über Funktionen). So schreibt man obigen λ-Ausdruck in Lisp wie folgt

```
(lambda (x) (* (+ x 3) (+ x 3)))
```

bzw. in Haskell

```
(\x -> (x + 3) * (x + 3))
```

Beispielsweise kann man in Lisp sehr einfach zwei Listen elementweise addieren:

```
(mapcar (lambda (x y) (+ x y)) '(0 1 2) '(10 11 12))
```

Ergebnis ist dann die Liste: `(10 12 14)`
Mit Hilfe des λ-Kalküls lassen sich auch Auswertungsstrategien studieren wie zum Beispiel **lazy evaluation**. Das heißt, man verzögert die Auswertung eines Ausdrucks solange, bis der Wert des Ausdrucks wirklich benötigt wird – was gelegentlich die Auswertung eines Ausdrucks überflüssig macht. (Wenn man z.B. die Und-Verknüpfung zweier Boole'scher Formeln F und G berechnen möchte (vgl. Abschnitt über logische Operationen) und man hat F bereits zu **falsch** ausgewertet, dann kann man sich das Auswerten von G ersparen, da das Ergebnis der Und-Verknüpfung in jedem Fall **falsch** sein wird.)

[3] Erfinder von Lisp ist John McCarthy (1927–2011), der gesagt hat: alle Programmiersprachen sind im Kern nichts anderes als λ-Kalkül; der Rest ist ein bisschen syntaktischer Zuckerguss.

[4] Benannt nach Haskell Brooks Curry (1900–1982).

Die Möglichkeiten des λ-Kalküls gehen noch weit über diese Beispiele hinaus, wenn man realisiert, dass man mit Hilfe der λ-Notation auch Funktionale oder Funktionen „höherer Ordnung“ erfassen kann. Dies sind Funktionen, die Funktionen als Argumente haben und ggf. Funktionen als Funktionswerte ergeben. Beispielsweise ist $\lambda x.\,(x\ x)$ diejenige Funktion mit einem Argument, die ihr Argument als Funktion interpretiert und diese Funktion wiederum auf sich selbst anwendet. Dass solche Interpretationen in der Welt der Programmiersprachen sinnvoll sein können, wird kurz im Abschnitt über Objekt- und Metasprache angesprochen.

Funktionale kennt man aus der Mathematik zum Beispiel in Form des Ableitungsoperators $\frac{d}{dx}$. Angewandt auf eine Funktion f liefert dieser als Ergebnis die nach x abgeleitete Funktion f'.

2.3 Syntax und Semantik, Metasprache und Objektsprache

Bisher haben wir die mathematischen Notationen als Abkürzungen verwendet, die jeweils eine fest vorgegebene Bedeutung haben. Zum Beispiel bedeutet das Symbol $\sqrt{\ }$ das Quadratwurzelziehen. Allenfalls die Angabe der betreffenden Grundmenge kann noch einen Unterschied ausmachen: Während $\sqrt{-5}$ auf der Grundmenge der reellen Zahlen undefiniert ist, hat diese Formel auf der Menge der komplexen Zahlen eine wohldefinierte Bedeutung. Aber in jedem Fall ist immer klar, dass mit „$\sqrt{\ }$“ eine Zahl gemeint ist, die mit sich selbst multipliziert die Zahl unter dem Wurzelzeichen ergibt. Also mathematische Zeichen sind von vornherein fest mit entsprechenden mathematischen Interpretationen oder Berechnungsvorschriften assoziiert.

In der **formalen Logik** oder **Metamathematik** ist das nicht mehr so. Nun sind die mathematischen Formeln selber das Objekt der mathematischen Betrachtung. Jetzt ist $(\sqrt{-5} + x - y^2)^3$ zunächst nur ein Gebilde, das aus Klammern und einigen seltsamen Zeichen aufgebaut ist. Um keinesfalls irgendeine inhaltliche Interpretation vorwegzunehmen, liest man die Symbole $\wedge$, $\vee$, $\neg$ nun nicht mehr als UND, ODER, NICHT, sondern zum Beispiel als „Dach“, „umgestürztes Dach“ und „Haken“. Man gibt Regeln an, wie man korrekt entsprechende Formeln aufbaut (siehe z.B. im Abschnitt über induktive Definitionen). Dies alles regelt die so genannte **Syntax** von Formeln.

Im nächsten Schritt werden mit solchen Formeln Bedeutungen, Interpretationen assoziiert. Hierzu gehört zum Beispiel auch die Angabe einer Grundmenge (reelle Zahlen oder komplexe Zahlen). Dann kann man darangehen festzustellen, ob eine solche Formel (immer) wahr ist, oder eine Lösung für die vorkommende Variable (oder Variablen) besitzt, und ähnliches. Nun befinden wir uns im Bereich der **Semantik**, also der inhaltlichen Interpretation einer (syntaktisch korrekt aufgebauten) Formel.

Solche semantischen Interpretationen müssen keineswegs eindeutig sein und von vornherein durch die verwendete Symbolik feststehen. Zum Beispiel hatten wir im Abschnitt über logische Operationen angegeben, dass man diese auch arithmetisch interpretieren kann und zum Beispiel auch die Fourier-Repräsentation wählen kann. Dies ist nichts anderes als eine andere Semantik für logische Formeln. Noch klarer ist das vielleicht bei Programmiersprachen. Bei der einen Programmiersprache könnte n++ vielleicht bedeuten, dass der Wert der Variablen n um 1 erhöht wird, was man bei dieser Symbolik auf den ersten Blick vielleicht nicht assoziieren würde; in einer anderen Programmiersprache könnte dies vielleicht etwas ganz anderes bedeuten. Insbesondere bedeutet Semantik im Kontext von Programmiersprachen die Art und Weise, wie bestimmte syntaktische Konstrukte real auf einem Computer ausgeführt werden sollen.

In der Mathematik werden Formeln gewissermaßen mittels ihrer eigenen Symbolik unmittelbar interpretiert, sie machen insofern direkt Aussagen über Zahlen oder andere mathematische Objekte. In der Metamathematik reden wir *über* Formeln und machen über diese Aussagen. Zum Beispiel könnte man in der Metamathematik sagen: die logische Formel $(x \wedge y)$ impliziert die Formel $(x \vee y)$. Was wir nun aber nicht dürfen, diese metamathematische Aussage durch

$$(x \wedge y) \rightarrow (x \vee y)$$

auszudrücken. Durch Verwenden des objektsprachlichen Symbols $\rightarrow$ haben wir Metasprache und Objektsprache miteinander vermischt. Wir dürfen aber sagen: die logische Formel $(x \wedge y)$ impliziert die Formel $(x \vee y)$, also ist die neue Formel

$$(x \wedge y) \rightarrow (x \vee y)$$

– vielleicht aufgrund eines zuvor aufgestellten Kalküls – daraus ableitbar.

Auch innerhalb von Beweisen ist es so, dass wir eine metamathematische Sichtweise einnehmen. Wir argumentieren *über* Formeln und warum diese wahr sind, oder warum

diese eine Lösung besitzen etc. Wenn wir im Rahmen eines Beweises zeigen, dass sich eine Formel F äquivalent umformen lässt in G, und sich G wiederum äquivalent umschreiben lässt in H, so halten wir es stilistisch nicht für gut, diesen beweistechnischen Sachverhalt durch

$$\begin{aligned} F &\Leftrightarrow G \\ &\Leftrightarrow H \end{aligned}$$

zu beschreiben (wegen des Verwendens des objektsprachlichen Doppelpfeils $\Leftrightarrow$). Hier hat man zwischen die Formeln F, G und H das Formelzeichen $\Leftrightarrow$ geschrieben und so in der Objektsprache eine Monster-Formel $F \Leftrightarrow G \Leftrightarrow H$ hergestellt. Wir halten es für besser angebracht, diese Äquivalenz von Formeln in der Metasprache mittels „genau dann wenn“ auszudrücken, also:

$$\begin{aligned} F &\text{ gdw. } G \\ &\text{ gdw. } H \end{aligned}$$

Manche Autoren verstehen nur den einfachen Doppelpfeil $\leftrightarrow$ als objektsprachliches Element, dagegen den doppelten Doppelpfeil $\Leftrightarrow$ als metasprachliche Abkürzung von „genau dann wenn“. In diesem Fall wäre obige Notation wieder OK. Wir wollen betonen, dass wir hier nicht eine bestimmte Notation vorschreiben wollen, sondern es geht uns nur darum bewusst zu machen, dass man sich seiner verwendeten Symbolik in einem mathematischen Text klar sein sollte, diese dann auch konsequent in der jeweiligen Interpretation einsetzen und Metasprache und Objektsprache nicht vermischen sollte.

2.4 Paradoxien, Gödel und Russell

Eine unzulässige Vermischung von Metasprache und Objektsprache findet auch bei dem folgenden Satz statt, der gewissermaßen aus der Objektsprache (dem Satz selbst) heraus in der Metasprache über sich selbst redet:

Dieser Satz ist falsch.

Ist er nun wahr oder falsch? Wenn er wahr wäre, müsste er – wie er selber sagt – falsch sein; wenn er falsch wäre, müsste er wahr sein.
Es gibt noch viele Beispiele dieser Art: Der Barbier sagt: „Ich rasiere alle Leute im Dorf, die sich nicht selbst rasieren“. Er wohnt selber im Dorf. Soll er sich nun rasieren oder nicht?

Eine bekannte Regel sagt: *Keine Regel ohne Ausnahme*. Hat diese Regel eine Ausnahme?

Diese Beispiele zeigen, dass die Vermischung von Metasprache und Objektsprache zu unerwünschten logischen Paradoxien führen kann. Es ist aber (leider oder auch nicht leider – je nach Standpunkt) so, dass Computerprogramme einerseits selber nichts anderes als Wörter über einem geeigneten Alphabet (Menge der ASCII-Symbole) sind, andererseits ebensolche ASCII-Wörter als Eingabe entgegennehmen und verarbeiten und hierzu eine Ausgabe (ebenfalls ein ASCII-Wort) berechnen. Das heißt, Programme verarbeiten und „reden“ sozusagen über Objekte derselben Art, wie sie selber welche sind. Dies ist die perfekte Vermischung von Objekt- und Metasprache. Dass Programme andere Programme als Eingabe haben können, ist von Interpretern und Compilern bekannt. Man könnte einem solchen Programm auch seinen eigenen Programmtext als Eingabe zuführen. Indem man diese Idee weiter verfolgt, sieht man (mit einem indirekten Beweisargument, siehe Extra-Abschnitt), dass es kein Programm geben kann, das systematisch und immer korrekt seine Eingabe, ein Programm, daraufhin untersucht, ob dieses immer terminiert (und selber dabei immer terminiert). Das heißt: das Halteproblem ist unentscheidbar.

Einen ähnlichen Effekt hat Gödel im Kontext der Formeln mit Quantoren festgestellt. Diese Formeln enthalten als Grundoperationen $+$ und $*$, die semantisch als Addition bzw. Multiplikation über der Grundmenge $\mathbb{N}$ interpretiert werden. Derartige Formeln „reden“ eigentlich zunächst über natürliche Zahlen, wie zum Beispiel die Formel

$$F(x) \;=\; \forall y \forall z \,((x = y * z) \;\rightarrow\; (y = 1) \vee (z = 1))$$

die ausdrückt, dass x eine Primzahl ist, genauer, die genau dann den Wahrheitswert **wahr** annimmt, wenn für x eine Primzahl (oder die Zahl 1) eingesetzt wird. Indem man nun derartige Formeln systematisch durchnummeriert, kann man Formeln mit natürlichen Zahlen identifizieren. Beispielsweise könnte man einer öffnenden Klammer die Zahl 1, einer schließenden Klammer die Zahl 2 zuordnen, usw. Die folgende Formel $F(x)$ drückt aus, dass x eine gerade Zahl ist.

$$\begin{array}{cccccccccc} \exists & y & (& y & + & y & = & x &) \\ 3 & 4 & 1 & 4 & 5 & 4 & 6 & 7 & 2 \end{array}$$

Wie angedeutet, könnte man nun, nachdem man einzelnen Zeichen Zahlen zugeordnet hat, der gesamten Formel die Zahl 341454672 zuordnen. Nun könnte man diese einer

Formel zugeordnete natürliche Zahl in die Formel selbst einsetzen: $F(341454672)$. Da 341454672 eine gerade Zahl ist, ist diese Formel **wahr**. Andere Formeln ergeben dagegen den Wahrheitswert **falsch**, wenn man die eigene Formel-Zahl für die freie Variable einsetzt; also symbolisch $F(\#F)$ betrachtet. Über diesen Trick, Formeln als Zahlen zu betrachten (auch als **Gödelisierung** bekannt), findet wieder die oben angesprochene Vermischung von Metasprache (Formeln) und Objektsprache (Zahlen) statt.
Nun hat Gödel in seinem berühmten Unvollständigkeitssatz eine Formel $G(x)$ konstruiert, die ihre eigene Nicht-Beweisbarkeit konstatiert, sofern man die zu G gehörige natürliche Zahl für x einsetzt.
Auf diesem Weg weist Gödel nach, dass für jeden „Vorschlag“ für einen Beweiskalkül für die Theorie der natürlichen Zahlen (mit Addition und Multiplikation als Basisoperationen) sich immer eine Formel konstruieren lässt, die zwar wahr, aber in dem betreffenden Kalkül nicht beweisbar ist. (Kalküle werden nochmals im nächsten Abschnitt diskutiert.)

Eine ähnliche Problematik wurde schon vor Gödel von Russell[5] beobachtet. Cantor beschreibt den von ihm eingeführten Mengenbegriff wie folgt:

> *Unter einer Menge verstehen wir jede Zusammenfassung von bestimmten wohlunterschiedenen Objekten unserer Anschauung oder unseres Denkens zu einem Ganzen.*

Wenn man den Mengenbegriff allzu unbedacht oder naiv anwendet, wie es in dieser „Definition“ zum Ausdruck kommt, so könnte man auch – nach Russell – die Menge aller Mengen bilden, die sich nicht selbst als Element enthalten. Formaler beschreiben wir diese Russell’sche Menge $\mathcal{M}$ so:

$$\mathcal{M} = \{ M \mid M \notin M \}$$

Gibt es diese Menge $\mathcal{M}$? Genau eine der folgenden beiden Aussagen müsste dann wahr sein, die andere falsch:

$$\text{entweder } \mathcal{M} \in \mathcal{M} \text{ oder } \mathcal{M} \notin \mathcal{M}$$

[5] Bertrand Arthur William Russell (1872–1970), englischer Philosoph, Logiker, Mathematiker, Sozialwissenschaftler und Politiker. Erhielt 1950 den Nobelpreis für Literatur.

Aus dem Ersteren folgt, nach Definition von $\mathcal{M}$, dass $\mathcal{M}$ *nicht* Element von $\mathcal{M}$ sein dürfte. Aus Letzterem folgt gerade, dass $\mathcal{M}$ Element von $\mathcal{M}$ ist. Also haben wir einen logischen Widerspruch, der nur so zu erklären bzw. zu beseitigen ist, dass diese Art der unbedachten Mengenbildung als unzulässig anzusehen ist.

Im mathematischen Alltagsleben trifft man auf diese Paradoxien (auch Antinomien genannt) eher nicht. Sie zeigen uns aber gewisse grundsätzliche Grenzen der Kalkülisierbarkeit auf. Nicht die gesamte Mathematik lässt sich algorithmisieren (was man durchaus auch als positiv empfinden kann).