Abschnitt i1indu

17. Juni 2000

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme des folgenden sind vorbehalten:

Studenten dürfen für ihre persönlichen Lernzwecke dieses Dokument ausdrucken und auf ihren Rechnern Kopien der entsprechenden Datei vorhalten.

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken und Dateikopien; insbesondere darf dieses Skriptum nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Datei auf irgendwelchen „Skriptenservern“ angelegt werden. Gegen die Anlage von Verweisen („hypertext links“) auf diese Datei ist selbstverständlich nichts einzuwenden.

Copyright © Herbert Klaeren, 1999.

Inhaltsverzeichnis

1 Einführung **9**
1.1 Was ist Informatik? . . . 9
1.2 Geschichte der Programmierung . . . 17

2 Induktive Definitionen **21**
2.1 Natürliche Zahlen . . . 21
2.2 Wortmengen . . . 28
2.3 Syntaktische Beschreibungsmittel . . . 32
2.4 Terme . . . 40
2.5 Aufgaben . . . 47

3 Algorithmen und Programme **49**
3.1 Algorithmus . . . 49
3.2 Programme . . . 54
3.2.1 Ausdrücke, Namen . . . 55
3.2.2 Fallunterscheidungen . . . 59
3.2.3 Rekursive Definitionen . . . 60
3.3 Aufgaben . . . 62

4 Abstrakte Datentypen **93**
4.1 Einführung . . . 93
4.2 Boolesche Werte . . . 95
4.3 Zähler . . . 98
4.4 Listen . . . 101
4.5 Bäume . . . 108
4.6 Aufgaben . . . 116

5 Logische Kalküle **117**
5.1 Ein Kalkül für die Aussagenlogik . . . 118
5.2 Ein Kalkül für die Prädikatenlogik . . . 122
5.3 Der Reduktionskalkül RC_1 . . . 125
5.4 Der λ-Kalkül . . . 127

A Mathematische Grundlagen **119**
A.1 Mengen, Relationen, Abbildungen . . . 119

A.2 Formale Logik 124
A.2.1 Aussagenlogik 124
A.2.2 Prädikatenlogik 126
A.3 Halbordnungen 128
A.4 Aufgaben 129

Kapitel 2

Induktive Definitionen

Wir stehen immer wieder vor dem Problem, mit endlichen Mitteln unendliche Mengen zu beschreiben. Die Beschreibung einer Programmiersprache beispielsweise muß endlich sein, aber die Menge der Programme, die in dieser Sprache geschrieben werden können, wird normalerweise unendlich sein. Die Mathematik hat eine jahrhundertealte Tradition in der endlichen Beschreibung unendlicher Objekte. Vielfach sind solche Beschreibungen *axiomatisch*, d. h. sie beschreiben eine Menge durch Eigenschaften, die man von ihren Elementen verlangt. Dabei muß nicht in jedem Fall immer klar sein, daß es eine Menge mit solchen Eigenschaften überhaupt gibt. In der Informatik, wo wir immer mit dem tatsächlich machbaren zu tun haben, bevorzugt man deshalb sogenannte *induktive Definitionen*. Wir führen diese zunächst an zwei Beispielen vor, bevor wir uns allgemein damit beschäftigen.

2.1 Natürliche Zahlen

Natürliche Zahlen werden in allerlei unterschiedlichen *Darstellungen* verwendet, z.B.

Dezimaldarstellung: 1 2 3 4 5 6 7 8 9 10 11 ...

Römische Zahlen: I II III IV V VI VII VIII IX X XI ...

Strichdarstellung: | || ||| |||| ||||| |||||| ||||||| ...

Diese Darstellungen eignen sich sämtlich nicht für eine *Definition* der natürlichen Zahlen. Einerseits ist in keinem Fall klar, wofür die Punkte stehen sollen, andererseits legen die unterschiedlichen Darstellungen auch unterschiedliche Rechenverfahren nahe, obwohl das abstrakte Konzept der Zahl, die hinter einer Darstellung steht, in allen Fällen gleich ist. So läßt sich

z. B. das in der Schule erlernte Verfahren zur Multiplikation von Dezimalzahlen auf Zahlen in römischer Schreibweise nicht anwenden.

Die folgende Definition der natürlichen Zahlen ist in der Mathematik durchaus üblich:

Definition 2.1 [Mes71] Die Menge der natürlichen Zahlen ist eine total geordnete Menge $(\mathbb{N}; \leq)$ mit den Eigenschaften:

1. $(\mathbb{N}; \leq)$ hat kein größtes Element.
2. $(\mathbb{N}; \leq)$ ist wohlgeordnet. Das kleinste Element von $\mathbb{N}$ wird mit 0 bezeichnet.
3. Jedes Element von $\mathbb{N}$ außer der 0 hat genau einen *Vorgänger*, d.h.

$$(\forall n \in \mathbb{N} \setminus \{0\})\ (\exists n_1 \in \mathbb{N})\ n_1 \leq n \land$$
$$(\forall n_2 \in \mathbb{N})(n_2 \leq n \land n_2 \neq n \ \Rightarrow\ n_2 \leq n_1)$$

Durch diese Axiome (von E. SCHMIDT) können wir kaum einen Hinweis bekommen, wie die Menge $\mathbb{N}$ aussehen könnte; im Gegenteil, es müßte zuerst bewiesen werden, daß es überhaupt ein *Modell*, d. h. eine von $\emptyset$ verschiedene Menge $(\mathbb{N}; \leq)$ mit den angegebenen Eigenschaften gibt. Als nächstes wäre dann noch zu zeigen, daß durch das Axiomensystem eine Menge bis auf Isomorphie eindeutig bestimmt ist; siehe dazu [Mes71]. In der Informatik sind wir fast nur an *konstruktiven Definitionen* wie der folgenden interessiert, die von PEANO (1889) stammt (siehe auch die Diskussion in [Bau89]):

Definition 2.2 (Peano-Axiome) Die Menge $\mathbb{N}$ der natürlichen Zahlen ist gegeben durch folgende Eigenschaften:

1. Es gibt eine natürliche Zahl $0 \in \mathbb{N}$.
2. Zu jeder Zahl $n \in \mathbb{N}$ gibt es eine Zahl $n' \in \mathbb{N}$, die man *Nachfolger von n* nennt.
3. Für alle $n \in \mathbb{N}$ ist $n' \neq 0$.
4. Aus $n' = m'$ folgt $n = m$.

5. Eine Menge M von natürlichen Zahlen, welche die 0 enthält und mit jeder Zahl $m \in M$ auch deren Nachfolger m', ist mit $\mathbb{N}$ identisch.

Das besondere an dieser Art von Definitionen ist, daß sie keinen expliziten Beweis brauchen, daß es die solchermaßen definierten mathematischen Objekte wirklich gibt, weil die Definition selbst ein Konstruktionsverfahren nahelegt:

1. Wir gehen von einem *Anfangspunkt* („Verankerung") aus, indem wir in (1) zunächst einmal ein Element vorgeben.
2. Ausgehend von dieser Verankerung wird in (1)–(4) ein *Erzeugungsverfahren* beschrieben, wie wir weitere Elemente konstruieren können.
3. Schließlich wird in (5) ein *induktiver Abschluß* gebildet, so daß außer den solchermaßen erzeugten Elementen keine weiteren existieren. Axiom 5 wird auch *Induktionsaxiom* genannt; es bestärkt das sogenannte *Erzeugungsprinzip*.

Bei jeder induktiven Definition finden wir eine solches Induktionsaxiom; häufig verbirgt es sich unter unscheinbaren Formulierungen wie (am Beispiel der natürlichen Zahlen):

- Die Menge $\mathbb{N}$ der natürlichen Zahlen ist die *kleinste* Menge mit den Eigenschaften 1.–4. aus 2.2.
- Die Menge $\mathbb{N}$ der natürlichen Zahlen ist bestimmt durch

  1.–4. wie in 2.2

  5. Durch 1.–4. werden alle natürlichen Zahlen erzeugt.

Überzeugen wir uns nun, daß sich $\mathbb{N}$ aus den Peano-Axiomen schrittweise konstruieren läßt! Aus 1. und 2. folgt, daß es natürliche Zahlen

$$0, 0', 0'', 0''', 0'''', \ldots$$

gibt. Lassen wir hierbei die 0 am Anfang weg, so entsteht die zuvor erwähnte Strichdarstellung für Zahlen. Die Axiome 3 und 4 besagen, daß das von

der Dezimaldarstellung gewöhnte Phänomen, daß es verschiedene Darstellungen für eine bestimmte Zahl gibt (z.B. 7, 07, 007 etc.), bei der Strichdarstellung nicht auftritt: jedes Anfügen eines Striches „ ′ “ erzeugt eine völlig neue Zahl.

Normalerweise verwenden wir statt „n'“ die Bezeichnung „$n+1$“. Natürlich läßt sich zeigen, daß die mit Hilfe der Peano-Axiome spezifizierte Menge $\mathbb{N}$ die Axiome von SCHMIDT erfüllt.

Aus dem Induktionsaxiom folgt ein wichtiges Beweisprinzip, das Prinzip der *vollständigen Induktion*:

Lemma 2.3 (Vollständige Induktion) Zum Beweis der Tatsache, daß ein bestimmtes Prädikat P für alle natürlichen Zahlen gilt, genügt es, die folgenden Beweise zu führen:

1. $P(0)$, d.h. das Prädikat gilt für die Null („Induktionsverankerung“)

2. $P(n) \Rightarrow P(n+1)$, d.h. aus der Annahme, daß P für irgendein $n \in \mathbb{N}$ gilt („Induktionsannahme“), können wir ableiten, daß $P(n+1)$ gilt („Induktionsschluß“).

Auf induktiv definierten Mengen lassen sich auf besonders einleuchtende Weise Abbildungen f durch *Fallunterscheidung und Rekursion* definieren: Ein Element von $n \in \mathbb{N}$ ist nach der induktiven Definition entweder $n = 0$ oder der Nachfolger eines anderen Elements, also $n = m + 1$. Für $f(0)$ müssen wir offensichtlich einen Wert vorgeben. Bei der Definition von $f(m+1)$ dagegen können wir voraussetzen, daß wir das Abbild $f(m)$ bereits kennen und mit seiner Hilfe das Abbild für m definieren. Mit anderen Worten: Für die Definition von $f(n+1)$ *rekurrieren* (lat. *recurrere* = zurücklaufen) wir auf den Wert von $f(n)$. Deshalb wird diese Vorgehensweise bei der Definition *Rekursion* genannt.

Rekursion und Induktion gehören unauflöslich zusammen: Eine induktive Definition eines Datenbereichs legt es nahe, Abbildungen darüber durch Rekursion zu definieren; umgekehrt erhalten durch Rekursion („rekursiv“) definierte Abbildungen immer dort eine vernünftige Bedeutung, wo sie sich auf induktiv erzeugte Daten beziehen. Anderenfalls besteht die Gefahr, daß es sich um *zirkuläre* Definitionen handelt, die gar nichts wirklich definieren.

Der nachfolgende Satz über die sog. *primitive Rekursion* läßt sich aus den Peano-Axiomen beweisen; wir verzichten allerdings auf diesen Beweis.

Satz 2.4 (Primitive Rekursion) *Seien* $f : \mathbb{N}^n \to \mathbb{N}$ *und* $g : \mathbb{N}^{n+2} \to \mathbb{N}$ *für ein* $n \in \mathbb{N}$ *gegebene Abbildungen. Dann gibt es genau eine Abbildung*

$$h : \mathbb{N}^{n+1} \to \mathbb{N} ,$$

so daß für alle $x_1, \dots, x_n, y \in \mathbb{N}$ *gilt:*

$$h(x_1, \dots, x_n, 0) = f(x_1, \dots, x_n) \quad (2.1)$$
$$h(x_1, \dots, x_n, y+1) = g(x_1, \dots, x_n, y, h(x_1, \dots, x_n, y)) \quad (2.2)$$

Definition 2.5 Unter den Bedingungen von Satz 2.4 heißt h durch primitive Rekursion aus f und g definiert. Die beiden Gleichungen in 2.4 heißen *Schema der primitiven Rekursion*.

Beispiel 2.6 Wir definieren die Addition durch primitive Rekursion:

$$\begin{aligned} \mathrm{add}(x, 0) &= x \\ \mathrm{add}(x, y+1) &= \mathrm{add}(x, y) + 1 \end{aligned}$$

In der Terminologie von 2.4 ist hier $n = 1$, $f : \mathbb{N} \to \mathbb{N}$ mit $f(x) = x$ und $g : \mathbb{N}^3 \to \mathbb{N}$ mit $g(x, y, z) = z + 1$. Beachte, daß hier „$z + 1$“ keine Addition darstellt, sondern nur eine andere Schreibweise für die Nachfolgerfunktion „z'“ ist. Die primitiv-rekursive Definition führt, wie man sich leicht überlegt, zu einem effektiven Rechenverfahren. Die Addition „$2 + 3$“ z.B. läßt sich demnach folgendermaßen berechnen:

$$\begin{aligned} \mathrm{add}(2, 3) &= \mathrm{add}(2, 2+1) \\ &= \mathrm{add}(2, 2) + 1 \\ &= \mathrm{add}(2, 1+1) + 1 \\ &= (\mathrm{add}(2, 1) + 1) + 1 \\ &= (\mathrm{add}(2, 0+1) + 1) + 1 \\ &= ((\mathrm{add}(2, 0) + 1) + 1) + 1 \\ &= ((2+1) + 1) + 1) \\ &= 5 . \end{aligned}$$

Wir sehen, daß sich hier zunächst rekursive Aufrufe von *add* bilden, die einen immer größeren „Kontext“ von Rechnungen ansammeln, die nach Beendigung der Rekursionen ausgeführt werden müssen. Man hätte dies vermeiden können durch eine alternative Definition von *add*, die der Additionsmethode kleiner Kinder durch Abzählen an den Fingern entspricht:

$$\begin{aligned} \mathrm{add}(x,0) &= x \\ \mathrm{add}(x,y+1) &= \mathrm{add}(x+1,y) \end{aligned}$$

Diese Definition ist allerdings im technischen Sinne unserer Definition zunächst nicht primitiv-rekursiv, weil dazu der „Parameter“ x unverändert an den rekursiven Aufruf weitergegeben werden müßte. Die Beispielrechnung sieht mit der veränderten (*„endrekursiven“*) Definition jedoch wesentlich besser aus:

$$\begin{aligned} \mathrm{add}(2,3) &= \mathrm{add}(2,2+1) \\ &= \mathrm{add}(3,2) \\ &= \mathrm{add}(3,1+1)+1 \\ &= \mathrm{add}(4,1) \\ &= \mathrm{add}(4,0+1) \\ &= \mathrm{add}(5,0) \\ &= 5\,. \end{aligned}$$

Beispiel 2.7 Betrachten wir noch drei Beispiele primitiv-rekursiver Definitionen:

$$\begin{aligned} \mathrm{mult}(x,0) &= 0 \\ \mathrm{mult}(x,y+1) &= \mathrm{add}(x,\mathrm{mult}(x,y)) \\ \\ \exp(x,0) &= 1 \\ \exp(x,y+1) &= \mathrm{mult}(x,\exp(x,y)) \\ \\ \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(y+1) &= y \end{aligned}$$

Wie man am Beispiel der Multiplikation $\mathrm{mult}(x, y) = x \cdot y$ und der Exponentialfunktion $\exp(x, y) = x^y$ sieht, kann man immer kompliziertere Funktionen durch sukzessive primitive Rekursionen gewinnen. Die *Vorgängerfunktion* zeigt, daß das Schema der primitiven Rekursion aus 2.4 auch dazu benutzt werden kann, um ohne Rekursion durch reine Fallunterscheidung neue Funktionen zu definieren.

Definition 2.8 Für ein $n \in \mathbb{N}$ definieren wir:

$$\mathrm{Ops}^{(n)}(\mathbb{N}) \stackrel{\mathrm{def}}{=} \{f \mid f : \mathbb{N}^n \to \mathbb{N}\}$$

und weiter

$$\mathrm{Ops}(\mathbb{N}) \stackrel{\mathrm{def}}{=} \bigcup_{n \in \mathbb{N}} \mathrm{Ops}^{(n)}(\mathbb{N}) .$$

$\mathrm{Ops}(\mathbb{N})$ heißt die Klasse der *arithmetischen Funktionen*.

Besonderes Augenmerk verdienen in dieser Definition die *nullstelligen* Funktionen $f \in \mathrm{Ops}^{(0)}(\mathbb{N})$. Da $\mathbb{N}^0$ nach Definition A.1 nur aus genau einem Element „()“ besteht, kann auch der Graph von f nur ein einziges Paar $((), n)$ enthalten. Eine nullstellige Funktion $f \in \mathrm{Ops}^{(0)}(\mathbb{N})$ ist daher gleichbedeutend mit einer *Konstanten* $n \in \mathbb{N}$. Dieser Trick, Konstanten und Funktionen zu einem einheitlichen Konzept zu verbinden, wird ziemlich häufig angewendet.

Die folgende Definition der *primitiv-rekursiven Funktionen* zeigt, wie man auch Funktionenklassen induktiv definieren kann.

Definition 2.9 Die Klasse der *primitiv-rekursiven (arithmetischen) Funktionen* ist die kleinste Klasse $\mathrm{Prim}(\mathbb{N})$ von Funktionen $f \in \mathrm{Ops}(\mathbb{N})$ mit den Eigenschaften:

1. Die nullstellige Funktion $0^{(0)} : \{()\} \to \mathbb{N}$ mit $0^{(0)}() = 0$ liegt in $\mathrm{Prim}(\mathbb{N})$.

2. Die einstellige Funktion $' : \mathbb{N} \to \mathbb{N}$ mit $'(n) = n' = n + 1$ liegt in $\mathrm{Prim}(\mathbb{N})$.

3. Die n-stelligen Funktionen $\pi_i^{(n)} : \mathbb{N}^n \to \mathbb{N}$, $i \in \{1, \ldots, n\}$ (n-stellige Projektionen auf die i-te Komponente) mit $\pi_i^{(n)}(x_1, \ldots, x_n) = x_i$ sind in $\mathrm{Prim}(\mathbb{N})$.

4. Definiere $\mathrm{Prim}^{(n)}(\mathbb{N}) \stackrel{\text{def}}{=} \mathrm{Prim}(\mathbb{N}) \cap \mathrm{Ops}^{(n)}(\mathbb{N})$. Ist $f \in \mathrm{Prim}^{(n)}(\mathbb{N})$, $g_1, \ldots, g_n \in \mathrm{Prim}^{(m)}(\mathbb{N})$, so ist

$$f \circ [g_1; \ldots; g_n] \; : \; \mathbb{N}^m \to \mathbb{N} \, ,$$

definiert durch

$$f \circ [g_1; \ldots; g_n](a_1, \ldots, a_m) \stackrel{\text{def}}{=} f(g_1(a_1, \ldots, a_m), \ldots, g_n(a_1, \ldots, a_m))$$

in Prim($\mathbb{N}$).

5. Sind $f, g \in \mathrm{Prim}(\mathbb{N})$ und ist h nach 2.4 durch primitive Rekursion aus f und g definiert, so ist $h \in \mathrm{Prim}(\mathbb{N})$.

Primitiv-rekursive Definitionen sind ein so natürliches Ausdrucksmittel, daß man lange Zeit glaubte, alle berechenbaren totalen Funktionen wären primitiv-rekursiv. Erst 1928 stellte ACKERMANN eine Funktion vor, die offensichtlich berechenbar, aber nicht primitiv-rekursiv ist. Wir wollen die Eigenschaften der primitiv-rekursiven Funktionen an dieser Stelle nicht weiter studieren.

2.2 Wortmengen

Im Gegensatz zu der Frühzeit der Computer haben wir es heute wesentlich seltener mit Rechnungen auf Zahlen zu tun; viel häufiger werden *Texte*, also Zeichenreihen verarbeitet. Auch Programme sind Texte.

Definition 2.10 Sei $\Sigma = \{a_1, \ldots, a_k\}$ eine endliche Menge, genannt *Alphabet*. Die Elemente von Σ nennen wir auch *Symbole*. Die Menge Σ^* der *Wörter über* Σ ist die kleinste Menge mit den folgenden Eigenschaften:

1. Es gibt ein *leeres Wort* $\epsilon \in \Sigma^*$.

2. Wenn $w \in \Sigma^*$ und $a \in \Sigma$, so ist $wa \in \Sigma^*$.

Wörter über $\Sigma = \{a, b, c\}$ sind also etwa

$$\epsilon, \epsilon a, \epsilon b, \epsilon c, \epsilon aa, \epsilon ab, \epsilon ac, \ldots, \epsilon abc, \ldots \; .$$

Wörter entstehen also dadurch, daß man an ein bestehendes Wort hinten noch einen Buchstaben anhängt. Normalerweise lassen wir das ϵ am Anfang von nicht-leeren Wörtern weg und schreiben einfach

$$\epsilon, a, b, c, aa, ab, ac, \ldots, abc, \ldots$$

Das Analogon zur vollständigen Induktion nennt man bei den Wortmengen *Wortinduktion*. Der „Schluß von n auf $n+1$" muß dabei im Prinzip k mal ausgeführt werden: für jeden Buchstaben a_i des Alphabets schließt man von w auf wa_i. Meistens wird man es jedoch so gestalten können, daß es auf diesen Buchstaben gar nicht genau ankommt; dann haben wir wiederum nur *einen* Induktionsschluß (s. Beispiel 2.12).

Auch auf Wörtern lassen sich Funktionen primitiv-rekursiv definieren; die sogenannten *primitiv-rekursiven Wortfunktionen* wurden von G. ASSER eingehend studiert.

Satz 2.11 *Sei* $\Sigma = \{a_1, \ldots, a_k\}$ *ein Alphabet und seien* $f : (\Sigma^*)^n \to \Sigma^*$ *und* $g_1, \ldots, g_k : (\Sigma^*)^{n+2} \to \Sigma^*$ *gegebene Abbildungen. Dann gibt es genau eine Abbildung*

$$h : (\Sigma^*)^{n+1} \to \Sigma^* ,$$

so daß für alle $x_1, \ldots, x_n, w \in \Sigma^*$, $i \in \{1, \ldots, k\}$ *gilt:*

$$h(x_1, \ldots, x_n, \epsilon) = f(x_1, \ldots, x_n) \tag{2.3}$$

$$h(x_1, \ldots, x_n, wa_i) = g_i(x_1, \ldots, x_n, w, h(x_1, \ldots, x_n, w)) \tag{2.4}$$

Beachte, daß die „Gleichung" 2.4 in Wirklichkeit für k Gleichungen steht. Während wir bei der primitiven arithmetischen Rekursion nur zwei Fälle „0" und „$n+1$" unterscheiden, unterscheiden wir bei der primitiven Wortrekursion $k+1$ Fälle: leeres Wort und alle k möglichen Endbuchstaben. Im übrigen beachte man die große Ähnlichkeit zu Definition 2.4.

Wie in 2.9 kann man nun die Klasse der primitiv-rekursiven Wortfunktionen definieren; dabei dienen die nullstellige ϵ-Funktion und die einstelligen „Nachschreibefunktionen" als Ausgangsfunktionen.

Wir betrachten an einem Beispiel, wie man Eigenschaften primitiv-rekursiver Wortfunktionen durch Wortinduktion beweisen kann:

Beispiel 2.12 Definiere

$$\mathsf{cat}(v, \epsilon) = v \tag{2.5}$$
$$\mathsf{cat}(v, wa) = \mathsf{cat}(v, w)a \tag{2.6}$$

Behauptung: cat ist assoziativ, d.h. für alle $u, v, w \in \Sigma^*$ gilt

$$\mathsf{cat}(\mathsf{cat}(u, v), w) = \mathsf{cat}(u, \mathsf{cat}(v, w)) \ , \tag{2.7}$$

Beweis: Wortinduktion über w.

1. $w = \epsilon$: Hier gilt

$$\begin{aligned} \mathsf{cat}(\mathsf{cat}(u, v), \epsilon) &= \mathsf{cat}(u, v) \\ \mathsf{cat}(u, \mathsf{cat}(v, \epsilon)) &= \mathsf{cat}(u, v) \end{aligned}$$

   nach Gleichung 2.5.

2. Gelte die Behauptung bereits für w und sei a_i beliebig. Dann gilt

   linke Seite:

$$\begin{aligned} \mathsf{cat}(\mathsf{cat}(u, v), wa_i) &= \\ &= \mathsf{cat}(\mathsf{cat}(u, v), w)a_i \text{ nach Gl. 2.6} \\ &= \mathsf{cat}(u, \mathsf{cat}(v, w))a_i \text{ nach Ind.Vor.} \end{aligned}$$

   rechte Seite:

$$\begin{aligned} \mathsf{cat}(u, \mathsf{cat}(v, wa_i)) &= \\ &= \mathsf{cat}(u, \mathsf{cat}(v, w)a_i) \text{ nach Gl. 2.6} \\ &= \mathsf{cat}(u, \mathsf{cat}(v, w))a_i \text{ nach Gl. 2.6} \end{aligned}$$

❒

Bisher sind wir bei den primitiv-rekursiven Funktionen stets innerhalb eines einheitlichen Bereichs geblieben, also innerhalb der natürlichen Zahlen bzw. der Wörter. Das ist jedoch keine zwingende Voraussetzung; Satz 2.11 kann auch so formuliert werden:

Satz 2.13 *Sei* $\Sigma = \{a_1, \ldots, a_k\}$ *ein Alphabet,* A *eine Menge,* $f : (\Sigma^*)^n \to A$ *und* $g_1, \ldots, g_k : (\Sigma^*)^{n+1} \times A \to A$ *gegebene Abbildungen. Dann gibt es genau eine Abbildung*

$$h : (\Sigma^*)^{n+1} \to A \ ,$$

so daß für alle $x_1, \ldots, x_n, w \in \Sigma^*$, $i \in \{1, \ldots, k\}$ *gilt:*

$$h(x_1, \ldots, x_n, \epsilon) = f(x_1, \ldots, x_n) \quad (2.8)$$
$$h(x_1, \ldots, x_n, wa_i) = g_i(x_1, \ldots, x_n, w, h(x_1, \ldots, x_n, w)) \quad (2.9)$$

Als Anwendung dieses Satzes definieren wir die Länge von Wörtern:

Beispiel 2.14

$$\begin{aligned} \mathbf{len} : \Sigma^* &\to \mathbb{N} \\ \mathbf{len}(\epsilon) &= 0 \\ \mathbf{len}(wa_i) &= \mathbf{len}(w) + 1 \end{aligned}$$

Als weiteres Beispiel, bei dem auch die Fälle der einzelnen Buchstaben wirklich unterschieden werden, definieren wir den Wert von Dezimalzahlen, die man als Wörter über dem Alphabet $\Sigma = \{\mathbf{1,2,3,4,5,6,7,8,9,0}\}$ verstehen kann:

Beispiel 2.15

$$\mathbf{val} : \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}, \mathbf{0}\}^* \to \mathbb{N}$$

$$\begin{aligned} \mathbf{val}(\epsilon) &= 0 \\ \mathbf{val}(w\mathbf{0}) &= 10 \cdot \mathbf{val}(w) \\ \mathbf{val}(w\mathbf{1}) &= 10 \cdot \mathbf{val}(w) + 1 \\ \mathbf{val}(w\mathbf{2}) &= 10 \cdot \mathbf{val}(w) + 2 \\ \mathbf{val}(w\mathbf{3}) &= 10 \cdot \mathbf{val}(w) + 3 \\ \mathbf{val}(w\mathbf{4}) &= 10 \cdot \mathbf{val}(w) + 4 \\ \mathbf{val}(w\mathbf{5}) &= 10 \cdot \mathbf{val}(w) + 5 \\ \mathbf{val}(w\mathbf{6}) &= 10 \cdot \mathbf{val}(w) + 6 \\ \mathbf{val}(w\mathbf{7}) &= 10 \cdot \mathbf{val}(w) + 7 \\ \mathbf{val}(w\mathbf{8}) &= 10 \cdot \mathbf{val}(w) + 8 \\ \mathbf{val}(w\mathbf{9}) &= 10 \cdot \mathbf{val}(w) + 9 \end{aligned}$$

Die Abbildung **val** weist uns auf die gerade in der Informatik zentrale Unterscheidung von *Syntax* und *Semantik* hin: *Syntax* beschreibt uns die

Zeichen(ketten), die wir zur *Bezeichnung* irgendwelcher Objekte verwenden; *Semantik* spricht über die hinter der Bezeichnung stehenden Objekte, also die Bedeutung der Zeichen. Dezimalzahlen sind in diesem Beispiel rein syntaktische Objekte, d. h. Zeichenreihen über einem bestimmten Alphabet. Die Regeln des kleinen Einmaleins oder die bereits erwähnten, in der Grundschule erlernte Algorithmen zur Addition, Multiplikation etc. von Dezimalzahlen lassen sich rein syntaktisch beschreiben; der Begriff des *Werts* einer Dezimalzahl ist hierzu nicht erforderlich. Trotzdem verbinden wir mit Dezimalzahlen stets einen *abstrakten Zahlbegriff*, d. h. wir ordnen solchen Dezimalzahlen eine *Semantik* zu. Die Abbildung **val** ordnet jeder Zeichenfolge über dem Alphabet $\{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}, \mathbf{0}\}$ eine abstrakte Zahl aus dem semantischen Bereich $\mathbb{N}$ zu; sie ist ein besonders einfaches Beispiel für eine sog. *Semantikfunktion*.

2.3 Syntaktische Beschreibungsmittel

Die bislang betrachteten Methoden der induktiven Definition von Mengen sind noch nicht geeignet, auf vernünftige Weise die Syntax von Programmiersprachen zu definieren, obwohl dies in aller Regel auch induktiv geschieht. Wir brauchen flexiblere Ausdrucksmittel zur induktiven Beschreibung von Programmiersprachen.

Während die allerersten Programmiersprachen eine Syntaxbeschreibung durch Beispiele und Gegenbeispiele verwendeten, wurde 1958 von JOHN BACKUS eine *formale* Beschreibung der Syntax der Sprache ALGOL vorgelegt. Das von ihm verwendete Beschreibungsmittel heißt *Backus-Normalform (BNF)*; da später PETER NAUR kleinere Verbesserungen an der Schreibweise vornahm und außerdem wesentlich zur Redaktion des ALGOL-Berichts [Bau68] beitrug, spricht man auch von der *Backus-Naur-Form*.

Jede Art von Sprachdefinition ruft ein grundlegendes Problem hervor: Da die Definition der Sprache selbst sprachliche Mittel verwendet, müssen wir zwischen der zu definierenden Sprache *(„Objektsprache")* und der zur Beschreibung verwendeten Sprache *(„Metasprache")* unterscheiden. Dementsprechend spricht man dann auch von *„Objektzeichen"* und *„Metazeichen"*. Es ist klar, daß die Metazeichen in der Objektsprache nicht vorkommen können, sofern man nicht eigens Mechanismen dazu ersinnt.

Die Objektzeichen heißen auch *Terminalzeichen* oder *Terminalsymbole* (lat.

„terminalis" = „endgültig") oder *Grundzeichen*. Unter den Metazeichen gibt es in der Regel auch irgendeine Form von *Variablen*, die man *syntaktische Variablen, Nichtterminalsymbole* oder *Nonterminalsymbole* nennt. Durch diese Zweistufigkeit von Metasprache und Objektsprache kommen semantische Aspekte ins Spiel: Die Objektsprache ergibt sich als Semantik einer metasprachlichen Spezifikation.

Allgemeine Definitionsmechanismen für die Syntax von Programmiersprachen gehen auf Arbeiten des Linguisten NOAM CHOMSKY zurück, der die syntaktisch korrekten Sätze der englischen Sprache mathematisch präzise beschreiben wollte. Zu diesem Zweck führte er *Grammatiken* als mathematische Objekte ein, die wir heute als *Chomsky-Grammatiken* bezeichnen. Chomsky definierte mehrere *Typen* solcher Grammatiken mit unterschiedlicher Leistungsfähigkeit. Von besonderer Bedeutung für die Informatik sind die Sprachen des Chomsky-Typs 2; sie heißen *kontextfreie Sprachen* und bilden die Grundlage für alle Programmiersprachen. Die oben erwähnte BNF ist formal äquivalent zu einer kontextfreien Chomsky-Grammatik. Dies scheint Backus und Naur jedoch nicht klar gewesen zu sein; sie haben die Notation aus rein pragmatischen Gesichtspunkten so entworfen. Die Theorie der formalen Sprachen zeigt, daß dies ein glücklicher Einfall war, denn die kontextfreien Sprachen lassen sich mit vertretbarem Aufwand maschinell analysieren.

Es sei jedoch an dieser Stelle bemerkt, daß zu der kontextfreien Definition einer Programmiersprache in der Regel noch eine gewisse Menge von zusätzlichen einschränkenden Bedingungen formuliert werden, die sich im Prinzip durch eine Grammatik des Chomsky-Typs 1 („kontextsensitive Grammatik") beschreiben ließen. Man spricht daher in diesen Fällen von *kontextsensitiven Nebenbedingungen*.

Der Vollständigkeit halber wollen wir noch erwähnen, daß die Chomsky-Grammatiken vom Typ 3 *reguläre Grammatiken* heißen, die zugehörigen Sprachen dementsprechend *reguläre Sprachen*.

Definition 2.16 Eine beliebige Menge L von Wörtern über Σ nennen wir eine *formale Sprache* über Σ.

Definition 2.17 (Kontextfreie Grammatik) Eine *kontextfreie Grammatik* ist ein 4-Tupel $\mathcal{G} = \langle N, \Sigma, P, S \rangle$, wobei

1. N ein Alphabet von sogenannten *Nonterminalsymbolen* ist,

2. Σ ein Alphabet von sogenannten *Terminalsymbolen* mit $N \cap \Sigma = \emptyset$,

3. P eine endliche Menge von sogenannten *Produktionen* oder *Regeln*, $P \subset N \times (N \cup \Sigma)^*$ und

4. $S \in N$ ein Startsymbol.

Statt $(A, \alpha) \in P$ schreiben wir normalerweise $A \to \alpha$. Die Menge $\mathcal{S}(\mathcal{G})$ der *Satzformen* von $\mathcal{G}$ ist die kleinste Teilmenge von $(N \cup \Sigma)^*$ mit den folgenden Eigenschaften:

1. $S \in \mathcal{S}(\mathcal{G})$.

2. Wenn $\alpha A \beta \in \mathcal{S}(\mathcal{G})$ für ein Nonterminalsymbol $A \in N$ und Zeichenfolgen $\alpha, \beta \in (N \cup \Sigma)^*$ und wenn $(A, \gamma) \in P$ eine Regel ist, so gilt auch $\alpha\gamma\beta \in \mathcal{S}(\mathcal{G})$. („Regelanwendung")

Die *durch $\mathcal{G}$ definierte Sprache* (sozusagen die *Semantik von $\mathcal{G}$*) ist definiert als $\mathcal{L}(\mathcal{G}) \stackrel{\text{def}}{=} \mathcal{S}(\mathcal{G}) \cap \Sigma^*$.

Für ein $A \in N$ definieren wir die von A erzeugte Sprache $\mathcal{L}(A)$ als die Sprache, die sich mit derselben Regelmenge und A als Startsymbol ergibt:

$$\mathcal{L}(A) \stackrel{\text{def}}{=} \mathcal{L}(\mathcal{G}') \text{ für } \mathcal{G}' \stackrel{\text{def}}{=} \langle N, \Sigma, P, A \rangle$$

Die Bezeichnung „kontextfrei" leitet sich übrigens von der Tatsache ab, daß die Nonterminalsymbole bei der Anwendung einer Regel durch rechte Regelseiten ersetzt werden können unabhängig von dem Zusammenhang (Kontext), in dem sie in der Satzform vorkommen.

Definition 2.17 liefert eine einfache Methode, Elemente der durch $\mathcal{G}$ definierten Sprache zu erzeugen: Man beginnt mit dem Startsymbol und ersetzt dann nach und nach jeweils eine beliebige syntaktische Variable (Nonterminalsymbol) durch eine rechte Seite einer Regel, bei der dieses Nonterminalsymbol auf der linken Seite vorkommt. (Beachte, daß es mehrere solche Regeln geben kann.) Der Prozeß bricht ab, wenn keine syntaktischen Variablen in der laufenden Satzform mehr vorhanden sind.

Der Unterschied zwischen kontextfreien Grammatiken und BNF-Definitionen besteht nun, abgesehen von syntaktischem Zucker zur Notation der Regeln, in der Festlegung gewisser Konventionen, so daß wir auf die

Spezifikation der Mengen der Nonterminal- und Terminalsymbole und des Startsymbols verzichten können. Darüber hinaus werden Regeln mit gleicher linker Seite zu einer einzigen Regel zusammengefaßt:

Definition 2.18 (BNF) Die *Metazeichen* der BNF sind

- das *Definitionszeichen* ::=
- das *Alternativzeichen* |
- die *Nonterminalklammern* ⟨ ⟩

Die Nonterminalklammern verwandeln eine beliebige Folge „string" von Buchstaben, Ziffern und Leerzeichen in eine syntaktische Variable. Alle Symbole, die weder Metazeichen noch Nonterminalsymbole sind, gelten als Terminalsymbole.

Eine *BNF-Definition* $\mathcal{B}$ besteht aus einer endlichen Menge von *BNF-Regeln* der Form

$$\langle \text{string} \rangle \quad ::=$$

oder

$$\langle \text{string} \rangle \quad ::= \quad \alpha$$

oder

$$\langle \text{string} \rangle \quad ::= \quad \alpha_1 | \dots | \alpha_n \text{ für } n \geq 2,$$

wobei $\alpha, \alpha_1, \dots, \alpha_n$ Folgen von Nonterminal- und Terminalsymbolen sind. Dabei darf es für jede syntaktische Variable $\langle A \rangle$ höchstens eine Regel mit linker Seite $\langle A \rangle$ geben.

Eine BNF-Definition heißt *vollständig*, wenn es für jede syntaktische Variable $\langle A \rangle$, die auf einer rechten Regelseite auftaucht, auch eine Regel mit linker Seite $\langle A \rangle$ gibt.

Im folgenden werden wir stets davon ausgehen, daß eine BNF-Definition vollständig ist.

Definition 2.19 (Semantik der BNF) Die Semantik einer BNF-Definition $\mathcal{B}$ wird mit Hilfe der Semantik 2.17 einer kontextfreien Grammatik definiert. Formal geht man so vor, daß man zunächst aus einer BNF-Definition die Mengen der vorkommenden Nonterminal- und Terminalsymbole aufsammelt und entsprechende Mengen N und Σ definiert. Das Nonterminalsymbol auf der linken Seite der ersten BNF-Regel gilt dann als Startsymbol S.

Zur Bildung der Regelmenge P haben wir dann noch eventuell vorkommende Alternativen in einzelne Regeln aufzulösen. Bei BNF-Regeln mit leerer rechter Seite wird in der Regel der Grammatik folgerichtig das leere Wort ϵ eingetragen.

Beispiel 2.20 An einigen einfachen BNF-Definitionen möchten wir die davon definierten Satzformen und Sprachen vorstellen.

1. $\langle S \rangle ::= ab \mid a\langle S \rangle b$

   Hieraus ergeben sich die Satzformen $\langle S \rangle$, ab, $a\langle S \rangle b$, $aabb$, $aa\langle S \rangle bb$ usf. Es fällt daher nicht schwer, einzusehen, daß die von dieser BNF-Definition erzeugte Sprache sich auch als $\{a^n b^n \mid n \geq 1\}$ beschreiben läßt.

2. $\langle S \rangle ::= a\langle S \rangle b$

   Dieses Beispiel zeigt, daß die von einer BNF-Definition definierte Sprache auch leer sein kann; es ist nämlich möglich, daß alle nach Definition 2.19 erzeugbaren Satzformen wie hier noch syntaktische Variablen enthalten. Die Frage, ob die durch eine BNF-Definition erzeugte Sprache leer ist, kann man relativ leicht (algorithmisch) entscheiden. Mit dieser und ähnlichen Fragen beschäftigt sich die Theorie der *Formalen Sprachen*.

3. Dieser Sachverhalt muß nicht unbedingt an der Definition auf den ersten Blick erkenntlich sein, da es auch *überflüssige Nonterminalsymbole* gibt:

   $\langle S \rangle ::= a\langle A \rangle$
   $\langle A \rangle ::= a\langle A \rangle b$
   $\langle B \rangle ::= ab$

   Hier gibt es zwar eine Regel, die auf der rechten Seite keine syntaktische Variable mehr enthält, aber das entsprechende Nonterminalsymbol B kann in keiner Satzform auftreten. B ist ein *unerreichbares Nonterminalsymbol*.

Ein Vorteil der BNF besteht darin, daß wir für die Nonterminalsymbole aussagekräftige Bezeichnungen wählen können:

Beispiel 2.21 Ein Software-Büro möchte für seine Programme eindrucksvolle Beschreibungen generieren und verwendet dazu die folgende BNF-Definition $\mathcal{B}$:

⟨Beschreibung⟩ ::= Das Programm arbeitet nach dem
⟨Prinzip⟩ der ⟨Adjektiv⟩ ⟨1. Hälfte⟩⟨2. Hälfte⟩.
⟨Prinzip⟩ ::= Prinzip|Verfahren|Algorithmus|System
⟨Adjektiv⟩ ::= iterierten|rezidivierten|substantivierten
⟨1. Hälfte⟩ ::= Rekursions|Iterations|Varianz|Diversifikations
⟨2. Hälfte⟩ ::= analyse|elimination|substitution|integration

Zu der von $\mathcal{B}$ erzeugten Sprache gehören Sätze wie: *„Das Programm arbeitet nach dem Prinzip der rezidivierten Diversifikationsintegration.“*.

Beispiel 2.22 Als ein Beispiel mit mehr Wirklichkeitsbezug führen wir ein (leicht vereinfachtes) Stück aus der ALGOL 60-Syntax vor, und zwar die Definition der sogenannten „Boolean Expressions“. Dies sind im wesentlichen die hier in Anhang A.2 eingeführten aussagenlogischen Ausdrücke, wobei allerdings für die Implikation das Zeichen $\supset$ und für die Äquivalenz das Zeichen $\equiv$ verwendet wird:

⟨Boolean expression⟩ ::= ⟨implication⟩|
⟨Boolean expression⟩ ≡ ⟨implication⟩
⟨implication⟩ ::= ⟨Boolean term⟩|
⟨implication⟩ ⊃ ⟨Boolean term⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩|
⟨Boolean term⟩ ∨ ⟨Boolean factor⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩|
⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩|
¬⟨Boolean primary⟩
⟨Boolean primary⟩ ::= ⟨logical value⟩|⟨variable⟩|
(⟨Boolean expression⟩)
⟨logical value⟩ ::= **true**|**false**

Diese BNF-Definition ist nicht vollständig; es fehlt die Definition von ⟨variable⟩. Üblicherweise steht ⟨variable⟩ für eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt.

Worin liegt nun der Wert einer solchen formalen Syntaxbeschreibung? Zum einen kann man sich relativ leicht davon überzeugen, ob eine bestimmte Formulierung, die man gerne verwenden möchte, syntaktisch korrekt ist, indem man nämlich versucht, diese Formulierung durch die angegebenen Regeln aus dem Startsymbol abzuleiten *(„Erzeugungsprozeß“)*. Andererseits lassen sich die Regeln aber auch in umgekehrter Richtung interpretieren, so daß es möglich wird, maschinell zu entscheiden, ob (und ggf. auf welche Weise) eine vorgelegte Zeichenkette mit einer solchen BNF-Definition erzeugt worden sein kann *(„Erkennungsprozeß“)*.

Je nach Art der BNF-Definition kann dieser Erkennungsprozeß, der bei jeder Behandlung einer Programmiersprache durch einen Compiler, d.h. also ein Übersetzungsprogramm, erfolgen muß, mehr oder weniger leicht zu bewerkstelligen sein. Wir wollen dies hier nicht weiter diskutieren. Am Beispiel wollen wir jedoch zeigen, daß die Umkehrung der Erzeugung in Form einer Erkennung durchaus möglich ist.

Beispiel 2.23 Sei $\mathcal{B}$ die BNF-Definition aus Beispiel 2.22 und sei folgende Zeichenkette gegeben:

$$\neg x \vee b \wedge x \supset \textbf{true} \equiv c.$$

Wir wollen feststellen, ob dies eine $\langle$Boolean expression$\rangle$ ist. Dazu können wir wie folgt argumentieren: Da die Zeichenkette das Zeichen $\equiv$ enthält, kommt nur die zweite Alternative der ersten Regel in Frage. Es müßte daher nachgewiesen werden, daß c eine $\langle$implication$\rangle$ ist und $\neg x \vee b \wedge x \supset \textbf{true}$ eine $\langle$Boolean expression$\rangle$. Das erste Ziel läßt sich über die Kette $\langle$implication$\rangle \to \ldots \to \langle$variable$\rangle$ leicht erfüllen. Betrachten wir das zweite Ziel. Die in Frage kommende Zeichenkette enthält $\equiv$ nicht mehr, also muß es einfach eine $\langle$implication$\rangle$ sein. Da sie tatsächlich das Zeichen $\supset$ enthält, kommt nur die zweite Alternative der zweiten Regel in Frage. Es müßte also **true** ein $\langle$Boolean term$\rangle$ sein und $\neg x \vee b \wedge x$ eine $\langle$implication$\rangle$. Wiederum ist das erste Ziel leicht erfüllt und wir stehen vor der Aufgabe, nachzuweisen, daß $\neg x \vee b \wedge x$ ein $\langle$Boolean term$\rangle$ ist. Da das Zeichen $\vee$ hierin vorkommt, kommt auch von der dritten Regel wiederum die zweite Alternative in Frage; es muß daher $b \wedge x$ ein $\langle$Boolean factor$\rangle$ sein und $\neg x$ ein $\langle$Boolean term$\rangle$. Zur Erfüllung des ersten Ziels weist man entsprechend der zweiten Alternative der vierten Regel nach, daß b ein $\langle$Boolean factor$\rangle$ ist und x ein $\langle$Boolean secondary$\rangle$. Das zweite Ziel ergibt sich dann noch nach der zweiten Alternative der fünften Regel, da x ein $\langle$Boolean primary$\rangle$ ist. Der eingegebene Ausdruck ist also syntaktisch korrekt.

Zu beachten ist dabei übrigens, daß wir bei diesem Erkennungsprozeß unmerklich durch die Bildung von Teilzielen (syntaktischen Unterstrukturen) den Term folgendermaßen geklammert haben:

$$(((\neg x) \lor (b \land x)) \supset \mathbf{true}) \equiv c,$$

d.h. die in A.2.1 formulierten Prioritätsregeln der Aussagenlogik sind von Backus und Naur unauffällig in die BNF-Syntax eingearbeitet worden.

Compiler haben es mit der Erkennung von Programmen schwerer als wir in diesem Beispiel, da man von ihnen erwartet, daß sie einen Eingabetext zeichenweise möglichst nur einmal von links nach rechts lesen und dabei erkennen. Wir sind in der Eingabezeichenkette mehrmals hin und her gelaufen.

Bei dieser Gelegenheit wollen wir gleich das Problem der *Zwischenräume* (engl. *white space*) diskutieren. In der Theorie der formalen Sprachen gibt es Zwischenräume nicht, wenn sie nicht ausdrücklich in Form bestimmter Terminalsymbole spezifiziert werden. Sobald wir jedoch mit Programmiersprachen zu tun haben, wie es bei EBNF-Definitionen in der Regel der Fall ist, kommen sofort durch die technische Darstellung der Programmtexte im Rechner, z.B. im sogenannten *ASCII-Code*, Terminalzeichen ins Spiel, die im eigentlichen Sinne keine Zeichen sind, weil man sie nicht sehen bzw. nicht unterscheiden kann. Hierzu gehören der *Leerschritt* (ASCII-Symbol Nummer 32, SP = *space*), das *Tabulatorsymbol* (Symbol Nummer 9, HT = *horizontal tabulator*) und gewisse *Zeilenendsymbole* (Symbol Nummer 10, LF = *line feed* und Nummer 13, CR = *carriage return*). Üblicherweise spezifiziert man das Auftreten von Zwischenräumen *nicht* in der EBNF-Definition, sondern vereinbart, daß solche Zwischenräume beliebiger Art und Anzahl überall *zwischen* den Terminalsymbolen vorkommen dürfen. Umgekehrt formuliert, sind Zwischenräume nur *im inneren* von Terminalsymbolen (wie etwa **true**) verboten.

Die BNF macht es ihrem Benutzer nicht leicht, *Wiederholungen* von Strukturen zu beschreiben. Bei Programmiersprachen kommt es jedoch so häufig vor, daß ein bestimmtes Konstrukt gar nicht oder beliebig oft oder auch einmal bis beliebig oft vorkommen kann, daß man dafür in Abweichung von Backus bzw. Naur eigene Abkürzungen eingeführt hat:

Definition 2.24 In einer BNF-Definition bedeutet

$$\langle A \rangle^*$$

das Fehlen oder das beliebig häufige Vorkommen von A und

$$\langle A \rangle^+$$

das einfache oder beliebig häufige Vorkommen von A.

Technisch gesprochen, müßte man die Metazeichen der BNF um $*$ und $+$ erweitern und die Semantikdefinition entsprechend erweitern. Da wir die Semantik der BNF hier mit Hilfe kontextfreier Grammatiken definiert haben, die solche Mechanismen nicht kennen, müßte man hier weiter ausgreifen; wir wollen darauf jedoch verzichten, da dieses Erweiterungskonzept intuitiv doch leicht verständlich ist.

2.4 Terme

Im vorangegangenen Abschnitt haben wir eine indirekte Art der Definition unendlicher Mengen gesehen: durch Grammatiken bzw. BNF wird ein Erzeugungsprozeß beschrieben, der wie bei den zuvor dargestellten induktiven Definitionen eine Menge generiert. Allerdings ist eine so einfache Fallunterscheidung wie bei den natürlichen Zahlen und den Wortmengen in diesem Rahmen nicht möglich und deshalb ist es nicht so einfach, Abbildungen auf der erzeugten Menge zu definieren. Es muß vielmehr zuerst ein *Analyseprozeß* gestartet werden, der zur Grammatik paßt.

In diesem Abschnitt wollen wir nun wieder zu einer direkteren Methode der induktiven Beschreibung beliebiger Mengen übergehen.

Wörter sind Zeichenreihen ohne innere Struktur, sieht man einmal von der Zerlegung eines Wortes in Buchstaben ab. Vielfach ist man jedoch an Zeichenreihen mit innerer Struktur interessiert, z.B.

$$(3x + 1) \cdot (5 \cdot (y + z))^2$$

oder

$$(a \land (b \lor c)) \Rightarrow (a \lor b)\ .$$

Solche Zeichenreihen bezeichnen wir üblicherweise als *Terme*; wir zerlegen Terme in *Teilterme* und benutzen ggf. Klammern, um Teilterme auszuzeichnen. In diesem Buch werden die Begriffe „Term“ und „Ausdruck“ immer völlig synonym verwendet. Die Notwendigkeit von Klammern in den oben

aufgeführten Termen ergibt sich dadurch, daß wir die *Operationssymbole* zwischen ihre Argumente schreiben. Der polnische Logiker ŁUKASIEWICZ zeigte (1925), daß auch eine eindeutige klammerlose Schreibweise möglich ist; die sog. *polnische Notation* liegt der Definition 2.27 zugrunde.

Definition 2.25 Ein *Operationsalphabet* ist ein Alphabet $\Sigma = \{F_1, \dots, F_m\}$ von *Operationssymbolen* zusammen mit einer Abbildung

$$\sigma : \Sigma \to \mathbb{N} .$$

$\sigma(F)$ heißt die *Stelligkeit* oder auch der *Rang* von F. Es ist auch der Begriff *Rangalphabet* statt Operationsalphabet gebräuchlich. Für $n \in \mathbb{N}$ definieren wir

$$\Sigma^{(n)} \overset{\text{def}}{=} \{F \in \Sigma \mid \sigma(F) = n\} .$$

$\Sigma^{(n)}$ heißt Menge der n-stelligen Operationssymbole. Statt $F \in \Sigma^{(n)}$ schreiben wir gelegentlich auch $F^{(n)} \in \Sigma$.

Beispiel 2.26 Das Operationsalphabet für die ganzen Zahlen mit Nachfolger, Vorgänger und den 4 Grundrechenarten läßt sich beschreiben als $(\Omega; \sigma)$ mit

$$\begin{aligned} \Omega^{(0)} &= \{0\} \\ \Omega^{(1)} &= \{\text{succ}, \text{pred}\} \\ \Omega^{(2)} &= \{+, -, *, \text{div}, \text{mod}\} \end{aligned}$$

und das für aussagenlogische Ausdrücke als

$$\begin{aligned} \Gamma^{(0)} &= \{W, F\} \\ \Gamma^{(1)} &= \{\neg\} \\ \Gamma^{(2)} &= \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \end{aligned}$$

Definition 2.27 Sei X eine Menge; die Elemente von X heißen *Variablen*. Sei ferner $(\Sigma; \sigma)$ ein Operationsalphabet. Es gelte $\Sigma \cap X = \emptyset$. Die Menge $T_\Sigma(X)$ der Σ-*Terme über X* ist die kleinste Teilmenge von $(\Sigma \cup X)^*$ mit den Eigenschaften:

1. $X \subseteq T_\Sigma(X)$
2. Falls $t_1, \dots, t_n \in T_\Sigma(X)$ und $F \in \Sigma^{(n)}$, so ist $Ft_1 \dots t_n \in T_\Sigma(X)$

Beachte, daß aus Bedingung 2 für $n = 0$ folgt, daß $\Sigma^{(0)} \subseteq T_\Sigma(X)$.

Eine wichtige Eigenschaft der Termmenge $T_\Sigma(X)$ ist, daß es (trotz des Fehlens von Klammern!) zu jedem Term eine *eindeutige Zerlegung* gibt, d.h. für jeden Term sind sowohl das Operationssymbol an der Termspitze als auch die Folge der Teilterme eindeutig bestimmt. Dies wird in dem folgenden Satz formalisiert:

Satz 2.28 (Eindeutige Termzerlegung) *Jeder Term* $t \in T_\Sigma(X)$ *ist entweder*

atomar, *d.h.* $t = x \in X$ *oder* $t \in \Sigma^{(0)}$ *oder*

zusammengesetzt, *d.h.* $t = Ft_1 \dots t_n$ *mit* $n \geq 1$.

Bei zusammengesetzten Termen sind F, n *und* $t_1, \dots, t_n$ *eindeutig bestimmt.*

Beweis: Offensichtlich gilt t atomar $\Leftrightarrow$ $\mathbf{len}(t) = 1$. (Zu **len** siehe 2.14.) Es bleibt die Eindeutigkeit zu zeigen. Wir zeigen eine etwas stärkere Behauptung:

Sind $t_1, \dots, t_n, t'_1, \dots, t'_n \in T_\Sigma(X)$, so gilt

$$t_1 \dots t_n = t'_1 \dots t'_n \;\Rightarrow\; (\forall\, 1 \leq i \leq n)\; t_i = t'_i\,.$$

Dies beweisen wir durch vollständige Induktion über $p \stackrel{\text{def}}{=} \mathbf{len}(t_1 \dots t_n)$.

1. $p = 1$. Es folgt $n = 1$ und t_1 atomar, somit auch $t_1 = t'_1$.
2. Gelte die Behauptung bereits für m und sei $t_1 \dots t_n$ eine Folge der Länge $m + 1$. Wir unterscheiden die folgenden beiden Fälle:
   (a) t_1 atomar: Es folgt, daß t'_1 mit einem atomaren Term $a = t_1$ beginnt. Das ist nur möglich für $t_1 = a = t'_1$. $t_2 \dots t_n$ hat die Länge m, so daß wir die Induktionsvoraussetzung anwenden können.
   (b) $t_1 = Fu_1 \dots u_r$ mit $r \geq 1$ und $u_i \in T_\Sigma(X)$. Daraus folgt, daß t'_1 ebenfalls mit F beginnt und, da die Operationssymbole eine feste Stelligkeit besitzen, $t'_1 = Fu'_1 \dots u'_r$. Nach Induktionsannahme folgt für
   $$u_1 \dots u_r t_2 \dots t_n = u'_1 \dots u'_r t'_2 \dots t'_n$$
   sofort $u_i = u'_i$ und $t_j = t'_j$ für alle in Frage kommenden i, j.

Wir haben in diesem Beweis eine vollständige Induktion über die Länge von Termen angewendet. Natürlich verfügen Termmengen wie natürliche Zahlen und Wörter auch über ein eigenes Induktionsprinzip, das wir *Terminduktion*, *strukturelle Induktion* oder auch *algebraische Induktion* nennen. Die Berechtigung für diese Bezeichnung findet sich in Definition 2.30.

Die polnische Notation, auch klammerlose Präfixnotation genannt, ist nicht die einzige gebräuchliche Termnotation. Wir können alle üblichen Termnotationen durch entsprechende Änderungen an Definition 2.27 präzise beschreiben, z.T. durch eine Erweiterung des Alphabets, jedenfalls aber durch eine Änderung der Erzeugungsklausel 2. Wir wollen am Beispiel der aussagenlogischen Terme einige der üblichen Termnotationen vorstellen.

Beispiel 2.29 Sei $\Sigma = \{\wedge, \vee, \neg\}$ mit $\sigma(\wedge) = \sigma(\vee) = 2$ und $\sigma(\neg) = 1$. Sei $X \stackrel{\text{def}}{=} \{a, b, c\}$ ein Variablenalphabet. $T_\Sigma(X)$ ist die Menge der aussagenlogischen Terme mit Variablen a, b und c. Wir betrachten den Term $\neg(a \wedge (b \vee c)) \vee (a \vee b)$.

1. **Polnische Notation:** $\vee\neg \wedge a \vee bc \vee ab$.

2. **Präfixnotation mit Klammern:** Wegen der festen Stelligkeit der Operationssymbole ist eine Klammerung im Prinzip nicht nötig. Für den Menschen kann eine Klammerung jedoch in der Regel das Verständnis erleichtern. Dann sieht der Beispielterm so aus:

$$\vee(\neg(\wedge(a, \vee(b, c))), \vee(a, b)) \ .$$

   Dies ist die in der Mathematik für Funktionen übliche Schreibweise.

3. **Alternative Präfixnotation mit Klammern:** Unsere Programmiersprache Scheme verwendet ein etwas abweichendes Schema für das Setzen von Klammern:

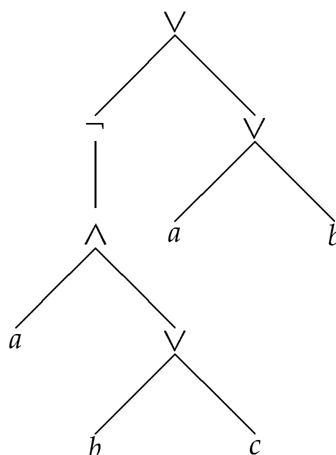
$$(\vee\ (\neg\ (\wedge\ a\ (\vee\ b\ c)))\ (\vee\ a\ b)) \ .$$

4. **Infixnotation:** Für zweistellige Operationssymbole ist vielfach eine *Infixnotation* gebräuchlich, bei der das Operationssymbol zwischen seine Argumente geschrieben wird. In der Regel sind dabei Klammern für eine eindeutige Zerlegung unerläßlich: $\neg(a{\wedge}(b{\vee}c)){\vee}(a{\vee}b)$.

Man mache sich klar, daß der Term $\neg \mathtt{a} \wedge \mathtt{b} \vee \mathtt{c} \vee \mathtt{a} \vee \mathtt{b}$ sich nicht eindeutig interpretieren läßt, wenn man keine Prioritätsregeln voraussetzt.

5. **Postfixnotation:** Für die unmittelbare Auswertung von Termen auf einer sogenannten *Stapel-Maschine* eignet sich die Umkehrung der polnischen Notation (UPN = umgekehrte polnische Notation); in der Tat müssen bei einem bekannten Taschenrechner-Fabrikat arithmetische Terme in der UPN eingegeben werden. In UPN sieht der Beispielterm so aus: $\mathtt{abc} \vee \wedge \neg \mathtt{ab} \vee \vee$. Beachte, daß die Argumente der Operationssymbole in der ursprünglichen Reihenfolge erhalten bleiben. UPN entsteht also nicht durch symbolweise Spiegelung der polnischen Notation!

6. **Bäume:** Die übersichtlichste Darstellung von Termen erhält man durch *Bäume*:



Wir werden Bäume später noch (4.10, 4.15) formal einführen.

Am gebräuchlichsten sind die *geklammerte Präfixnotation* und die *Infixnotation*, weil sie dem menschlichen Leser entgegenkommen. Manchmal werden dabei Klammern weggelassen, wenn kein Zweifel über die Zuordnung besteht, z.B. bei $\sin \alpha$ oder $\cos \frac{\pi}{2}$. Für ein rein formales Operieren auf und mit Termen, z.B. für Induktionsbeweise für irgendwelche Eigenschaften ist die *polnische Notation* am besten geeignet. Für eine Auswertung eines Terms

nach einem möglichst einfachen Verfahren wählt man am besten die *umgekehrte polnische Notation*. Für eine bestmögliche Übersicht über die Teilterm-struktur wiederum sind die *Bäume* ein ausgezeichnetes Mittel, sofern man den Aufwand für die zweidimensionale Darstellung nicht scheut.

Man nimmt sich häufig die Freiheit heraus, Terme stets in der Notation zu schreiben, die für den jeweiligen Zweck am geeignetsten erscheint. Dabei muß man sich jedoch im klaren darüber sein, daß es in jedem Fall *algorithmisch*, also rein mechanisch möglich ist, Terme aus der einen in die andere Notation zu übersetzen.

Definition 2.30 Sei $(\Sigma; \sigma)$ ein Operationsalphabet, A eine Menge. Für jedes n sei jedem Operationssymbol $F \in \Sigma^{(n)}$ eine Funktion

$$F_A : A^n \longrightarrow A$$

zugeordnet. Dann nennt man A eine Σ-*Algebra*. Die Funktionen F_A heißen *Operationen* der Algebra. Wenn wir betonen wollen, daß es sich tatsächlich um eine Algebra handelt, verwenden wir das Skript-Symbol $\mathcal{A}$ und nennen A die *Trägermenge* von $\mathcal{A}$.

Die Funktionen F_A nennen wir auch *Operationen* der Algebra; wenn man so will, sind die Operationen die semantischen Objekte, die den (syntaktischen) Operationssymbolen zugeordnet sind. Eine wichtige Eigenschaft von Termen, die mit der eindeutigen Termzerlegung (Satz 2.28) zusammenhängt, ist, daß jeder Term einen eindeutigen Wert hat, nachdem man für die Variablen Werte vorgegeben hat:

Satz 2.31 *Sei X eine Variablenmenge, $\mathcal{A}$ eine Σ-Algebra. Sei ferner $f : X \to A$ eine Funktion („Variablenbelegung"); dann wird durch die Vorschriften*

$$\begin{aligned} \hat{f}(x) &= f(x) \quad \text{für } x \in X \\ \hat{f}(Ft_1 \ldots t_n) &= F_A(\hat{f}(t_1), \ldots, \hat{f}(t_n)) \end{aligned}$$

eine Abbildung $\hat{f} : T_\Sigma(X) \to A$ eindeutig bestimmt.

(Eine Abbildung mit diesen Eigenschaften nennt man in der Algebra einen *Homomorphismus*.)

Beweis: Gemäß Definition A.4 konstruieren wir den Graphen der Abbildung $\hat{f}$ induktiv, wobei wir uns an die Vorschriften des Satzes halten [End72]:

$$\begin{aligned} \rho_0 &\stackrel{\text{def}}{=} \{(x, f(x)) \mid x \in X\} \cup \{(F, F_A) \mid F \in \Sigma^{(0)}\} \\ \rho_{i+1} &\stackrel{\text{def}}{=} \rho_i \cup \\ &\quad \{(Ft_1 \dots t_n, F_A(a_1, \dots, a_n)) \mid F \in \Sigma^{(n)}, (t_j, a_j) \in \rho_i,\ 1 \le j \le n\} \\ \rho_{\hat{f}} &\stackrel{\text{def}}{=} \bigcup_{i=0}^{\infty} \rho_i \end{aligned}$$

Wir haben zwei Dinge zu beweisen: daß $\hat{f}$ mit diesem Graphen eine (totale) Abbildung ist und daß es eindeutig bestimmt ist. Die erste Behauptung folgt aus der eindeutigen Termzerlegung (2.28). Zeigen wir also die zweite:

Seien $\hat{f}, g$ zwei Funktionen mit den obigen Eigenschaften. Wir wollen zeigen, daß für alle $t \in T_\Sigma(X)$ gilt $\hat{f}(t) = g(t)$. Dies tun wir durch eine Induktion über die Länge von Termen.

Die einzigen Terme der Länge 1 sind die Variablen und die konstanten Operationssymbole aus $\Sigma^{(0)}$. Für Variable gilt

$$\hat{f}(x) = f(x) = g(x)\ ,$$

für konstante Operationen $\hat{f}(F) = F_A = g(F)$. Gelte die Behauptung nun für alle Terme der Länge bis zu n. Sei $t = Ft_1 \dots t_m$ ein Term der Länge $n + 1$. Dann gilt

$$\begin{aligned} \hat{f}(Ft_1 \dots t_m) &= F_A(\hat{f}(t_1), \dots, \hat{f}(t_m)) \\ &= F_A(g(t_1), \dots, g(t_m)) \\ &= g(Ft_1 \dots t_m). \end{aligned}$$

Hierbei haben wir im zweiten Schritt die Induktionsannahme ausnutzen können, weil die Terme $t_1, \dots, t_m$ zusammen höchstens die Länge n haben. ❒

Das Definitionsschema für die Abbildung $\hat{f}$ erinnert bereits an das Schema der primitiven Rekursion, auch wenn es deutlich einfacher ist. In der

Tat läßt sich mit Hilfe des vorangehenden Satzes tatsächlich die Gültigkeit einer „primitiven Termrekursion" beweisen. Weil die Rekursion hier über die Struktur der Terme verläuft, nennen wir sie kurz *strukturelle Rekursion* statt primitive Termrekursion.

Damit haben wir übrigens auch eine Semantik von Termen: Satz 2.31 garantiert für jede Algebra $\mathcal{A}$ eine eindeutige Funktion $\hat{f} : T_\Sigma \to A$. Wir nennen sie *Darstellungsfunktion* (representation function); sie bildet jeden Term t auf seine Semantik t_a von A ab. Wir sagen dann auch: t_a wird durch t *bezeichnet* oder t_a ist eine *Darstellung* für t. Die folgenden beiden Sonderfälle können bei der Darstellungsfunktion auftreten:

1. Ein Element von A ist Darstellung verschiedener Terme, d. h. die Darstellungsfunktion ist nicht injektiv. Das ist eigentlich fast immer der Fall.
2. Ein Element von A wird durch gar keinen Term bezeichnet, d. h. die Darstellungsfunktion ist nicht surjektiv. Diesen Fall mag man durchaus als unerwünscht betrachten.

Schränkt man sich auf die Teilmenge $A' \subseteq A$ der Elemente von A ein, die durch einen Term bezeichnet werden, so gibt es eine Abbildung

$$a : A' \longrightarrow T_\Sigma$$

mit der Eigenschaft $\hat{f}(a(x)) = x$ für alle $x \in A'$. Eine solche Abbildung heißt *Abstraktionsfunktion*; sie abstrahiert von den Eigenarten der Darstellung eines bestimmten Elements.

Wir werden Terme im folgenden zur Spezifikation und Darstellung abstrakter Datentypen verwenden.

2.5 Aufgaben

Aufgabe 2.1 Beweisen Sie unter Benutzung der Definitionen von '+' und '·' durch 'add' (2.6) und 'mult', daß die folgenden Gleichungen gelten

$$(a + b) + c = a + (b + c) \quad (2.10)$$

$$a + b = b + a \quad (2.11)$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \tag{2.12}$$
$$a + c = b + c \Rightarrow a = b \tag{2.13}$$

Aufgabe 2.2 Beweisen Sie durch Wortinduktion:

$$\mathrm{cat}(v, w) = \mathrm{cat}(z, w) \Rightarrow v = z \tag{2.14}$$
$$\mathrm{cat}(v, w) = \mathrm{cat}(v, z) \Rightarrow w = z \tag{2.15}$$

Aufgabe 2.3 Formulieren Sie das Prinzip der Terminduktion („strukturelle Induktion“) ausführlich wie in 2.2!

Aufgabe 2.4 Eine Menge einfacher arithmetischer Ausdrücke in Infix-Notation sei in EBNF wie folgt gegeben:

Expression	=	Term {"+" Term} .
Term	=	Factor {"∗" Factor} .
Factor	=	"0"|"1"|...|"9"| "(" Expression ")" .

Schreiben Sie ein Programm, welches solche Terme einliest und in umgekehrter polnischer Notation (UPN) ausgibt.

Aufgabe 2.5 Für das Operationsalphabet der aussagenlogischen Ausdrücke, $(\Sigma; \sigma) = \{\wedge, \vee, \neg, W, F\}$ mit $\sigma(\wedge) = \sigma(\vee) = 2$, $\sigma(\neg) = 1$ und $\sigma(W) = \sigma(F) = 0$ geben Sie eine Alternative zur Definition der Terme an, so daß die übliche Infix-Notation entsteht.

Aufgabe 2.6 Schreiben Sie ein Programm, welches einen in polnischer Notation gegebenen arithmetischen Term in die Infixnotation mit Klammern überführt. Nehmen Sie dabei an, daß die Eingabe außer den Operatoren $+, -, *$ und $/$ nur einstellige Zahlen und i, j, k, l, m, n als Variablen enthält, also z.B. $+5 * i7$.

Aufgabe 2.7 Welche Möglichkeiten für „Syntaxfehler“ gibt es bei den Eingabetermen aus Aufgabe 2.6? Wie kann man diese algorithmisch feststellen?

Literaturverzeichnis

[Bau68] BAUMANN, RICHARD: *Algol-Manual der Alcor-Gruppe*. R. Oldenbourg, München/Wien, 3. Auflage, 1968.

[Bau89] BAUER, F. L.: *100 Jahre Peano-Zahlen*. Informatik-Spektrum, 12:340–341, 1989.

[End72] ENDERTON, H. B.: *A Mathematical Introduction to Logic*. Academic Press, 1972.

[GKP89] GRAHAM, R. L., D. E. KNUTH und O. PATASHNIK: *Concrete Mathematics*. Addison-Wesley, 1989.

[Hal69] HALMOS, P: *Naive Mengenlehre*. Vandenhoeck & Ruprecht, Göttingen, 1969.

[KCR98] KELSEY, RICHARD, WILLIAM CLINGER und JONATHAN REES: *Revised^5 Report on the Algorithmic Language Scheme*. SIGPLAN Notices, 33(9):26–76, 1998.

[Kla83] KLAEREN, HERBERT: *Algebraische Spezifikation — Eine Einführung*. Springer Verlag, Berlin-Heidelberg-New York, 1983.

[Knu73] KNUTH, D. E.: *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973. Sorting and Searching.

[Mes71] MESCHKOWSKI, H.: *Einführung in die moderne Mathematik*, Band 75/75a der Reihe *Hochschultaschenbücher*. BI, 1971.

[MHR80] METROPOLIS, N., J. HOWLETT, and GIAN-CARLO ROTA (editors): *A History of Computing in the Twentieth Century*. Academic Press, 1980.

[Rie91] RIESE, ADAM: *Rechnung auff der Linihen und Federn /Auff allerley handthirung gemacht / durch Adam Risen (Faksimile 2. Aufl. Erfurt 1532)*. Magistrat der Stadt Erfurt, 1991.

[Wex81] WEXELBLAT, RICHARD L. (editor): *History of Programming Languages*, New York, 1981. Academic Press.