6 Formelsatz

Nach einer allgemeinen Einführung in die Problematik werden wir die Erzeugung von Formeln im *FrameMaker* und in LaTeX besprechen. Danach werden wir den Algorithmus erläutern, der LaTeX-Formeln in eine graphische Darstellung umsetzt.

6.1 Einführung

Sowohl Autoren als auch Dokumentverarbeitungssysteme stoßen auf Probleme, wenn in einem Dokument mathematische Formeln vorkommen. Das Problem der Autoren ist, die Formeln erst einmal einzugeben, und das Problem der Dokumentsysteme ist, eine ansprechende graphische Darstellung für die Formeln zu finden. Das ist aus folgenden Gründen schwieriger als bei reinem Text:

- Mathematische Formeln sind graphisch gesehen zweidimensionale Objekte. Bei Brüchen wie $\frac{1}{2}$ steht der Zähler über dem Nenner, und Exponenten werden hochgestellt wie in x^2.
- Aus traditionellen Gründen enthalten Formeln eigenartige Sonderzeichen. Bekannt sind z. B. die Zahl π, das Symbol ∞ für unendlich oder als besonders schwieriges Beispiel das Wurzelzeichen, das sich in Höhe und Länge seinem Argument anpaßt: $\sqrt{x}$, $\sqrt{2}$, $\sqrt{a+b}$.
- Formelteile variieren je nach ihrem Kontext in der Größe. Das haben wir gerade an dem Wurzelzeichen gesehen, aber es gilt auch für ganz gewöhnliche Buchstaben, wenn sie als Exponent oder in einem Bruch auftauchen: x^x, $x+\frac{1}{x}$.
- Um einen guten visuellen Eindruck hervorzurufen, sind viele kleine Details zu beachten. Die Positionen von Formelteilen sollten sich nach ihrer eigenen Größe und der ihrer Nachbarn richten, wobei gewisse Mindestabstände nicht unterschritten werden sollten. Vergleichen Sie die Lage der Indizes in
  $\overline{x^2}$, x^2, x_2^2, x_2, $\overline{P^2}$, P^2, P_2^2, P_2.

Wie sollte nun eine Formel eingegeben werden? Sicherlich wird niemand eine Formel wie x_i durch exakte graphische Anweisungen wie etwa „(x 11 pt groß) (1 mm nach rechts) (1 mm nach unten) (i 9 pt groß)“ beschreiben wollen. Als Ausweg bietet sich eine logische, d. h. an der Bedeutung orientierte Formelbeschreibung an. Auf den ersten Blick scheint es, als ob Systeme wie *troff* und LaTeX eine solche logische Formelbeschreibung ermöglichen. Unser Beispiel x_i kann in *troff* als `x sub i` spezifiziert werden und in LaTeX als `x_i`. Bei genauerem Hinsehen stellt sich aber heraus, daß die Formelbeschreibungsmöglichkeiten dieser Systeme gar nicht logisch orientiert sind, sondern bestenfalls ein Mittelding zwischen graphischer und logischer

Beschreibung darstellen. Der Grund dafür liegt in prinzipiellen Schwierigkeiten, die wir im folgenden darstellen werden.

Obwohl die Mathematik gemeinhin als äußerst exakte Wissenschaft gilt, sind mathematische Formeln im allgemeinen kein Musterbeispiel an exakter Information. Für Nicht-Mathematiker ohnehin fast unverständlich, sind sie oft auch für Mathematiker anderer Spezialgebiete schwer interpretierbar. Eine Formel erhält eine bestimmte Bedeutung nur abhängig von einem Kontext. Im engeren Sinne ist dieser Kontext das Dokument, in dem die Formel vorkommt, und in weiterem Sinne das gemeinsame Wissen von Autoren und Lesern, die in demselben Gebiet der Mathematik arbeiten. So kommt es oft vor, daß dieselbe Notation (graphische Erscheinungsform) je nach Kontext ganz verschiedene Bedeutungen hat, während umgekehrt dieselbe Bedeutung (logische Struktur) je nach der Tradition oder dem Geschmack der Schreiber auf verschiedene Weise graphisch dargestellt werden kann. Ein paar Beispiele sollen das verdeutlichen.

In Dokumenten, die nicht dem Bereich der Analysis angehören, ist f' einfach der Name irgendeiner anderen Funktion als f. In der Analysis dagegen ist f' die Ableitung von f, die aber genausogut durch Notationen wie $\frac{df}{dx}$ oder Varianten davon dargestellt werden kann.

Der Buchstabe *i* bezeichnet meist einen (beliebigen) Index wie in x_i. Wenn es um komplexe Zahlen geht, ist *i* allerdings der Eigenname der imaginären Einheit $\sqrt{-1}$. Manche bezeichnen die imaginäre Einheit allerdings als *j*; der *FrameMaker* benutzt dagegen den griechischen Buchstaben ι dafür.

Das alles zeigt noch nicht, daß eine logische Beschreibung völlig unmöglich ist, sondern nur, daß sie äußerst umständlich wäre. Es müßte jedes einzelne Symbol wie der Strich in f' oder der Buchstabe *i* genau nach seiner Bedeutung klassifiziert werden, was sich letztlich genauso verbietet wie die Eingabe exakter Positionen.

Was die rein logische Formelbeschreibung völlig utopisch macht, ist die Tatsache, daß ständig neue mathematische Konzepte erfunden und in Formeln dargestellt werden, d. h. die Menge der möglichen Formelbedeutungen ist potentiell unendlich. Das gilt jedoch nicht für die Menge der Notationen, d. h. der möglichen graphischen Darstellungen. Meist wird ein neues mathematisches Konzept durch irgendeine bereits existierende Notation oder eine neue Kombination bekannter Schreibweisen dargestellt, die den Erfindern des Konzepts irgendwie günstig erscheint. Nur äußerst selten werden ganz neue Symbole oder Schreibweisen entwickelt. Daher ist es keine große Beschränkung, daß existierende Dokumentsysteme die Vielfalt möglicher Formelnotationen auf einen endlichen Satz von Grundnotationen zurückführen, die dann beliebig kombiniert werden können. Mit der Verbreitung des Computersatzes von Formeln wird das Erfinden völlig neuer Symbole und Schreibweisen wohl bald ganz zum Erliegen kommen.

Die in LaTeX und *FrameMaker* konkret eingeschlagenen Wege zur Formelbeschreibung unterscheiden sich deutlich voneinander. In LaTeX wird eine im wesentlichen abstrakt graphische Beschreibung benutzt, die von Positionierungs- und Größendetails absieht. Diese werden vom System automatisch bestimmt, wie wir später sehen werden. Die Formel f' wird als `f'` beschrieben, egal ob sie irgendeine Funktion oder die Ableitung von f bedeutet. Die Schreibweise $\frac{df}{dx}$ für die Ableitung wird wie ein gewöhnlicher Bruch beschrieben, obwohl sie mathematisch etwas anderes darstellt. Das LaTeX-System ist graphisch gesehen ziemlich vollständig, d. h. es können beliebig aussehende Formeln erzeugt werden, auch völlig unsinnige wie z. B. $(x()7+(.$ Das System ist erweiterbar, da Symbole über Namen angesprochen werden, z. B. `\infty` für ∞.

Die LaTeX-Formelbeschreibung hat auch gewisse logische Aspekte, etwa wenn zwischen dem griechichen Buchstaben Sigma als Variable (Σ, eingegeben als `\Sigma`) und als Summenzeichen ($\sum$, beschrieben durch `\sum`) unterschieden wird. Diese Tendenz kann von den Autoren noch durch Definition und Benutzung von Makros, d. h. symbolischen Namen für Formelteile, verstärkt werden. So ist es z. B. möglich, den Namen `\im` für die imaginäre Einheit einzuführen, der als Makro für den Buchstaben i steht. Die Formel $i \cdot i = -1$ kann dann wahlweise graphisch als `i \cdot i = -1` oder eher logisch als `\im \cdot \im = -1` eingegeben werden. Die logische Darstellung ist (in diesem Fall) umständlicher, bietet aber den Vorteil, daß die Darstellung der imaginären Einheit durch eine Änderung der Makrodefinition von `\im` sehr leicht von i auf j umgestellt werden kann; es muß nicht jede einzelne Formel im Dokument umgeschrieben werden.

Die Formeldarstellung im *FrameMaker* ist logischer orientiert. Es wird zwischen sozusagen gewöhnlichen Formeln, die etwa dem Bereich der Schulmathematik entsprechen, und ungewöhnlichen unterschieden. Die gewöhnlichen Formeln werden logisch orientiert eingegeben. Dabei muß zwischen gleich oder ähnlich aussehenden Dingen sorgfältig unterschieden werden: das binäre Minus wie in $x - y$ ist logisch verschieden von dem unären wie in -1, ein Exponent wie in x^2 von einem oberen Index wie in x^2 und die Zeichenfolge „sin“ von der Funktion „sin“. Die logische Beschreibung bedingt eine gewisse Starrheit; ein unäres „+“ ist z. B. nicht vorgesehen, und mit jeder öffnenden Klammer entsteht normalerweise gleich eine schließende. Daher können Formeln, die im Aussehen etwas vom Standard abweichen, nur schwer erzeugt werden. Es ist immerhin möglich, da neue logische Operatoren hinzudefiniert werden können.

Die eher logische Beschreibung im *FrameMaker* bietet noch einen Vorteil: während LaTeX eine Formel nur so weit „versteht“, wie es zu einem einigermaßen gelungenen Satz nötig ist, beherrscht *FrameMaker* die wichtigsten der Schulrechenregeln. Es ist daher möglich, Standardformeln automatisch algebraisch umformen und Terme auswerten zu lassen. Im folgenden werden die Fähigkeiten des *FrameMaker* genauer beschrieben; danach wird LaTeX behandelt.

6.2 Formeln im *FrameMaker*

Im *FrameMaker* können Formeln in drei verschiedenen Kontexten auftreten: im Inneren einer Textzeile, als eigenständiges Gleitobjekt und als Bestandteil eines Gleitobjekts, das noch anderes Material enthält.

Die Eingabe einer Formel beginnt damit, daß eine leere Formel in einem dieser Kontexte erzeugt wird. Diese leere Formel wird dann nach und nach mit Inhalt aufgefüllt. Mögliche Inhalte können entweder über die Tastatur eingegeben oder mit der Maus aus einem Katalog mathematischer Objekte ausgewählt werden. Dieser Katalog ist in acht Seiten eingeteilt, die jeweils verwandte Operatoren enthalten. Im folgenden wird erst die Tastatureingabe besprochen, dann werden die acht Seiten des mathematischen Katalogs behandelt.

Tastatureingabe von Formelteilen

Zahlen und Buchstaben *müssen* über die Tastatur eingegeben werden, da sie nicht im mathematischen Katalog enthalten sind. Nahezu alle der Symbole, die zufällig auf der Tastatur vorkommen wie z. B. „+“, „=“ und „<“, können ebenfalls mit einem Tastendruck eingegeben werden. Sie sind allerdings auch im mathematischen Katalog enthalten. Auch diejenigen einfachen Symbole, die im Katalog vorkommen, aber keine Tasten besitzen, können über die Tastatur eingegeben werden. Es gibt dafür zwei Möglichkeiten: Tastaturcodes (eigenartige Zeichenfolgen) sowie *symbolische Namen*, die mit \ eingeleitet und mit „Return“ beendet werden. Der symbolische Name für das Unendlichkeitszeichen ∞ ist z. B. `\infty`. Das ist interessanterweise (bis auf das abschließende „Return“) identisch mit dem Kommando, das das Zeichen in LaTeX erzeugt.

Ein neu eingegebenes Zeichen wird vom *FrameMaker* in die bereits bestehende Formelstruktur logisch eingefügt. Das Resultat dieses Einfügeprozesses hängt davon ab, ob es zum Zeitpunkt der Einfügung einen Einfügepunkt gegeben hat und wo dieser lag, oder ob ein Formelteil selektiert war und welcher das war. In den Beispielen in Abbildung 6.1 wird der Einfügepunkt durch einen senkrechten Strich angedeutet und ein selektierter Formelteil durch einen Kasten. Die Beispiele beziehen sich auf die Eingabe eines *x*, das logisch gesehen mit einem Formelteil multipliziert wird. Beachten Sie, daß in dem mittleren Beispiel ein Klammerpaar von *FrameMaker* automatisch eingefügt wird.

Einige Tasten haben Kommandofunktion, bewirken also nicht, daß das betreffende Zeichen in die Formel eingefügt wird. Eine solche Taste ist \, die symbolische Namen einleitet. Die Tasten ′ und " bezeichnen den Anfang eines in der Formel vorkommenden Wortes. Das Wort wird durch die „Return“-Taste beendet.

Einfache Symbole

Die erste Seite des mathematischen Katalogs trägt die Überschrift „Symbols“. Sie enthält griechische Buchstaben wie α und Γ, andere Symbole wie ∞, diakritische

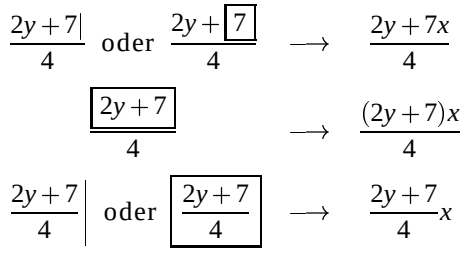

Abbildung 6.1: Verhalten beim Einfügen des Buchstabens x

Zeichen, die sich an Symbole anhängen lassen, wie z. B. bei x', $\overline{x}$, $\hat{x}$ und $\vec{x}$, und zwei Knöpfe „Start String“ und „End String“, mit denen in Formeln vorkommende Wörter begonnen und beendet werden.

Alle diese Symbole können mit der Maus angewählt werden. Sie werden dann in die Formel eingefügt. Logisch gesehen wirken sie wie der Buchstabe *x* in den Beispielen oben als ein multiplikativer Faktor. Der griechische Buchstabe Σ sollte vom Summenzeichen $\sum$, das auf einer anderen Seite steht, unterschieden werden.

Kleine Operatoren

Die Seite mit der Überschrift „Operators“ enthält die grundlegenden unären und binären Operatoren. Die Argumentpositionen werden durch ein Fragezeichen dargestellt. So findet man unter anderem die binären $?+?$ und $?-?$ sowie die unären $-?$ und $\pm?$. Unäres $+?$ und binäres $?\pm?$ sind nicht vorgesehen. Es wird also schwerfallen, Formeln zu erzeugen, in denen diese Operatoren vorkommen.

Neben vielen anderen gibt es vier Indexoperatoren $?^{?}$, $?_{?}$, ${}^{?}?$ und ${}_{?}?$ sowie vier Potenzoperatoren $\sqrt{?}$, $\sqrt[?]{?}$, $?\times 10^{?}$ und $?^{?}$. Der Fall des rechten oberen Index und der der allgemeinen Potenz sehen also gleich aus. Benutzer des *FrameMaker* sollten sich anhand der Stellung auf der Seite merken, was was ist. Der Unterschied wird bedeutsam, wenn Formeln manipuliert oder ausgewertet werden.

Die hier besprochenen Operatoren können mit der Maus ausgewählt werden. Sie werden dann in die bestehende Formelstruktur eingefügt, wobei wieder entscheidend ist, wo der Einfügepunkt liegt bzw. was selektiert ist. Beim Einfügen des binären Plus-Operators wird z. B. aus $2x|$ die Formel $2x+?$, während sich aus $2\boxed{x}$ $2\,(x+?)$ ergibt. Im zweiten Fall werden die Klammern vom *FrameMaker* automatisch eingefügt.

Große Operatoren

Die Seite mit der Überschrift „Large“ enthält „große“ Operatoren für Summen, Produkte, Integrale, Durchschnitte und Vereinigungen. Der Summenoperator kommt

in drei Versionen vor: als $\sum ?$, $\sum_{?}^{?}$ und als $\sum\limits_{?}^{?}$. Für den Integraloperator gibt es sogar fünf Versionen: $\int ?$, $\int_{?}^{?}$, $\int\limits_{?}^{?}$, $\int_{?}^{?}$ und $\int_{?}^{?}$. Die drei letzten Varianten des Integrals sind bloße graphische Variationen desselben logischen Symbols. Es gibt Kommandos, um zwischen diesen drei Variationen hin- und herzuwechseln, ohne daß große Teile der Formel neu eingegeben werden müssen.

Beachten Sie, daß in *FrameMaker* die beiden Grenzen des Integrals genau übereinander stehen. In LaTeX dagegen ist, wenn keine besonderen Tricks angewandt werden, die obere Grenze etwas nach rechts verschoben, um der Neigung des Integralzeichens zu folgen ($\int\limits_{?}^{?}$ und $\int_{?}^{?}$).

Klammern

Auf der Seite „Delimiters“ werden verschiedene Klammeroperatoren angeboten. Für runde Klammern gibt es vier Versionen:

$(?)$ $(?$ $?)$ $(?,?)$

Während bei der Tastatureingabe einer öffnenden Klammer immer gleich die entsprechende schließende miterscheint, gibt es hier also auch Einzelklammern ohne Partner.

Dieselben vier Varianten gibt es mit eckigen Klammern [] sowie mit geschweiften {}. Zwischen diesen drei Klammerformen kann auch nachträglich hin- und hergeschaltet werden.

Wenn der Operator $(?)$ mit der Maus angewählt wird, hängt es wie immer vom Einfügepunkt bzw. der Selektion ab, was geschieht. So wird $2x|$ zu $2x(?)$, während $2\boxed{x}$ zu $2\,(x)$ wird.

Relationen

Die Seite „Relations“ enthält eine Sammlung binärer Relationssymbole, unter anderem $? \approx ?$ und $? = ?$. Diese verhalten sich merkwürdig beim Einfügen in bereits bestehende Formeln. Das Gleichheitszeichen wird nämlich völlig anders behandelt als alle anderen Relationen (siehe Tabelle 6.1).

Der Rest des Katalogs

Die Analysis-Seite „Calculus“ enthält noch einmal das Integralzeichen in fünf Varianten. Dazu kommen Symbole für totale und partielle Ableitungen sowie Grenzwerte.

Vor der Eingabe	Nach Einfügen von „≈“	Nach Einfügen von „=“
$abc\vert$	$ab\,(c \approx ?)$	$abc = ?$
$a\boxed{bc}$	$a\,(bc \approx ?)$	$abc = ?$
$ab\vert c$	$ab \approx c$	$abc = ?$

Tabelle 6.1: Verhalten beim Einfügen eines Relationssymbols

Die Seite „Matrices“ enthält Matrizen, die wahlweise von eckigen Klammern eingeschlossen sind oder ohne Klammern geschrieben werden.

Die Seite „Functions“ bietet Standardfunktionen wie sin ? und log ? an. Dazu kommt eine allgemeine Funktionsanwendung $?(?)$.

Erweitern des Katalogs

Der Katalog mathematischer Symbole kann erweitert und abgewandelt werden. Dazu muß man wissen, daß jedes Symbol einen *Namen*, einen *Typ* und eine graphische *Erscheinungsform* hat. Der Name des binären Additionssymbols z. B. ist $? + ?$ (so erscheint es im Katalog), sein Typ ist „Infix“ und seine graphische Erscheinungsform ist $+$ (ohne die Fragezeichen). Die Definition eines Symbols kann verändert werden, und neue Symbole können hinzudefiniert werden.

Die wichtigsten Typen sind „Atom“ wie z. B. ∞, „Infix“ wie z. B. $? + ?$, „Prefix“ wie z. B. $-?$, „Postfix“ wie z. B. die Fakultät $?!$ und „Delimiter“ wie z. B. $(?)$. Einige vordefinierte Symbole haben Typen, die neudefinierte Symbole nicht haben können. Dazu gehört z. B. der Typ „Roots“ für $\sqrt{?}$.

Positionierung

Die Position von Teilformeln wird automatisch durch den *FrameMaker* festgelegt. Es gibt eine globale und eine lokale Methode, die Positionen zu verändern.

Bei der globalen Methode wird die graphische Erscheinungsform eines Symbols verändert. Zur Erscheinungsform eines Symbols gehört nämlich nicht nur das sichtbare Symbol, sondern auch leerer Raum darum herum. Wenn beim binären „+“ dieser Raum vergrößert wird, dann rücken in allen Formeln alle Summanden weiter vom Plus-Zeichen weg.

Lokal kann die Position jeder Teilformel beliebig verändert werden. Diese Veränderung erfolgt mit der Maus, aber nicht durch direktes Verschieben. Es gibt ein Feld von acht Pfeilen in acht verschiedene Richtungen. Ein Mausklick auf einen solchen Pfeil verschiebt den selektierten Formelteil um einen festen Betrag in die durch den Pfeil angedeutete Richtung.

Angewandte Regel	Vorher	Nachher
Add Fractions	$\frac{a}{4}+\frac{2a}{3}$	$\frac{11a}{12}$
Factor	$a^2+2\boxed{a}^3+1$	$a(a+2a^2+\frac{1}{a})$
Multiply Out	$(a-3)(2a+4)$	$2a^2-2a+12$
Evaluate	5!	120
Evaluate	a^3	aaa
Simplify	$a^x a^{2x+3}$	a^{3x+3}
Expand All Terms	$\prod_{x=2}^{5} x^3-2$	$6\,(25)\,62\,(123)$

Tabelle 6.2: Beispiele für Formeltransformationen

Edieren von Formeln

Das Edieren von Formeln erfolgt nach denselben Prinzipien wie das von Text. Eine selektierte Teilformel kann durch Druck auf die „Delete"-Taste gelöscht werden, an eine andere Stelle verschoben oder kopiert werden. Gewisse Besonderheiten sind aber zu beachten.

Das Löschen selektierter Teilformeln erfolgt in mehreren Stufen. Aus $2x+\boxed{3y}$ z. B. wird durch Betätigen der „Delete"-Taste $2x+\boxed{?}$. Eine erneute Betätigung liefert $2x|$.

Nur logisch vollständige Teilformeln können selektiert werden. In der Formel b^2-4ac z. B. sind b, b^2 und b^2-4ac selektierbar, aber nicht b^2-4 oder b^2-4a. Nur selektierbare Teilformeln können gelöscht, verschoben oder kopiert werden. Angesichts dieser Tatsache erscheint es z. B. schwierig, eine Formel der Form $A+B$ mit großen Summanden in $A-B$ zu verwandeln, da das Plus-Zeichen alleine nicht selektierbar ist. Jeder Versuch, es zu selektieren, selektiert gleich die ganze Formel.

Transformieren und Auswerten von Formeln

Der *FrameMaker* beherrscht die wichtigsten Regeln der Schulmathematik, die auf selektierte Teilformeln angewandt werden können. Dazu gehören algebraische Umformungen und die Ableitungsregeln. Die Fähigkeit zu integrieren beschränkt sich auf Polynome. Als Spezialfall der Umformungsregeln können arithmetische Ausdrücke ausgewertet werden. Da all das über das Gebiet der Erstellung von Dokumenten hinausgeht, betrachten wir hier nur wenige Beispiele, die in Tabelle 6.2 gezeigt sind. Zum letzten Beispiel ist zu sagen, daß das Produkt sich nicht nur auf x^3, sondern auf x^3-2 bezieht. Das Ergebnis ist die *FrameMaker*-Schreibweise für $6\cdot 25\cdot 62\cdot 123$. *FrameMaker* verwendet nämlich nie Malpunkte, sondern schreibt die Faktoren einfach nebeneinander, auch wenn es Zahlen sind. Die Klammern hindern die Zahlen daran, ineinander zu fließen.

6.3 Formeln in LaTeX

In LaTeX gibt es Formeln im laufenden Text und in eigenen Zeilen (displayed equations). Formeln in eigenen Zeilen werden etwas anders gesetzt als Formeln im laufenden Text. Im folgenden geben wir einige Beispiele mit Kommentaren. In einer dreigeteilten Zeile befindet sich immer links die Eingabe für eine Formel, im Innern der Zeile die Formel gesetzt im Textstil und rechts die Formel gesetzt im Display-Stil. Danach folgen Kommentare zu dieser Formel. (In diesen Beispielen treten die Formeln im „Display"-Stil nicht in eigenen Zeilen auf, wie man leicht sehen kann. Sie sehen aber trotzdem so aus als ob, denn LaTeX erlaubt es, durch besondere Kommandos den Formeln einen falschen Kontext vorzugaukeln.)

`x_{1}` oder `x_1` x_1 x_1

Der Operator „_" bezeichnet Indizierung. Die Klammern `{ }` dienen zur Zusammenfassung von logischen Formelteilen. Sie erscheinen nicht direkt in der gesetzten Formel. In diesem Beispiel sind die Klammern überflüssig, weil `1` atomar ist.

`x_{12}` x_{12} x_{12}

Hier sind die Gruppierungsklammern notwendig.

`x_12` x_12 x_12

Wenn sie fehlen, nimmt LaTeX an, daß sich der Indizierungsoperator nur auf `1` bezieht.

`x^2` oder `x^{2}` x^2 x^2

Der Operator `^` bezeichnet Exponenten oder hochgestellte Indizes. Beachten Sie, daß der Exponent im Display-Stil etwas höher sitzt als im Textstil.

`x^{uv}, {x^u}^v, x^{u^v}` x^{uv}, x^{uv}, x^{u^v} x^{uv}, x^{uv}, x^{u^v}

Bei mehreren Exponenten muß durch eine Klammerung klargemacht werden, was gemeint ist. Entsprechend verschieden sehen die Ergebnisse aus. Die Eingabe `x^u^v` ist verboten und führt zu einem Fehler.

`A^1_1 - x_2^2 + P^3_3` $A^1_1 - x^2_2 + P^3_3$ $A^1_1 - x^2_2 + P^3_3$

Es kann auch gleichzeitig ein unterer und ein oberer Index angefügt werden. Die Reihenfolge in der Spezifikation ist beliebig. Die vertikale Position der Indizes hängt von der Größe des indizierten Zeichens ab. Horizontal passen sich die Indizes der Form des Zeichens an. Beim „*P*" befindet sich z. B. der untere Index etwas weiter links als der obere.

`\overline{x}+\overline{x^2}-x^2` $\overline{x} + \overline{x^2} - x^2$ $\overline{x} + \overline{x^2} - x^2$

Teilformeln können überstrichen werden. Der Strich ist so lang wie die entsprechende Teilformel. Seine Lage hängt von der Höhe der überstrichenen Teilformel ab. Diese wiederum wird von der Existenz des Überstrichs beeinflußt; die Exponenten sitzen im überstrichenen Fall etwas tiefer als sonst.

```
\underline{x}+\underline{x_2}-x_2
```
$\underline{x}+\underline{x_2}-x_2$ $\underline{x}+\underline{x_2}-x_2$

Teilformeln können auch unterstrichen werden. Die Lage des Unterstrichs hängt von der Gestalt der Teilformel ab. Im Gegensatz zur Überstreichung beeinflußt die Unterstreichung die Lage des Index nicht.

```
\frac{x^2+y^2}{x^2+\frac{x}{y+1}}
```
$\frac{x^2+y^2}{x^2+\frac{x}{y+1}}$ $\frac{x^2+y^2}{x^2+\frac{x}{y+1}}$

Mit `\frac` werden Brüche bezeichnet. Zähler und Nenner folgen als Operanden von `\frac`. Die Klammern `{ }` werden zur Abgrenzung dieser Operanden benötigt. Brüche werden im Display-Stil viel größer gesetzt als im Textstil. Im Nenner liegen Exponenten etwas tiefer als im Zähler.

```
\sqrt{x + y} \cdot \sqrt[n]{-1}
```
$\sqrt{x+y}\cdot\sqrt[n]{-1}$ $\sqrt{x+y}\cdot\sqrt[n]{-1}$

Der Operator `\sqrt` bezeichnet das Wurzelzeichen. Er hat ein obligatorisches Argument, das in `{ }` eingeschlossen wird, und ein optionales Argument in `[ ]`. Das Wort `\cdot` bezeichnet den Malpunkt.

```
\sum_{i=1}^n i^2
```
$\sum_{i=1}^n i^2$ $\sum_{i=1}^{n} i^2$

Das Summenzeichen wird durch `\sum` dargestellt. Die Grenzen werden mit Hilfe der schon vorgestellten Index- und Exponentenoperatoren angefügt. Im Display-Stil ist das Summenzeichen größer und die Grenzen rücken direkt darunter bzw. darüber.

```
\sum_2^2 \int_2^2
```
$\sum_2^2 \int_2^2$ $\sum_{2}^{2} \int_{2}^{2}$

Auch das Integralzeichen ist im Display-Stil größer als im Textstil. Bei ihm sind aber die Grenzen immer rechts am Zeichen, wenn keine besondere Vorkehrungen getroffen werden.

```
\sum\limits_2^2 \sum\nolimits_2^2
```
$\sum\limits_2^2 \sum\nolimits_2^2$ $\sum_{2}^{2} \sum_{2}^{2}$

```
\int\limits_2^2 \int\nolimits_2^2
```
$\int\limits_2^2 \int\nolimits_2^2$ $\int_{2}^{2} \int_{2}^{2}$

Die Position der Grenzen kann durch die Befehle `\limits` und `\nolimits` unabhängig vom Stil festgelegt werden. Die Beispiele sollen auch verdeutlichen, daß sich die horizontale Position der beiden Grenzen nach der Form des Zeichens richtet. Bei dem gerade stehenden Summenzeichen stehen die Grenzen genau übereinander, während bei dem schrägen Integralzeichen die untere Grenze etwas weiter links steht als die obere.

```
-u = v - w oder -u=v-w
```
$-u = v - w$ oder $-u=v-w$ $-u = v - w$

Das erste „-“ ist unär; es steht direkt vor seinem Argument ohne Zwischenraum.

Das zweite „-“ ist binär; LaTeX fügt etwas Zwischenraum vor und nach ihm ein. Außerdem unterscheidet LaTeX zwischen binären Operatoren und Relationssymbolen; vor und nach „=“ wird etwas mehr Platz gelassen als vor und nach dem binären „-“, was allerdings praktisch kaum sichtbar ist. Die Zwischenräume in der Formeleingabe sind dagegen völlig unerheblich und werden von LaTeX nicht beachtet.

```
\int f d\mu
```
$$\int f d\mu$$
$$\int f d\mu$$

Griechische Buchstaben werden durch ihre (englischen) Namen `\alpha` usw. bezeichnet.

```
\int\! f\, {\sf d} \mu
```
$$\int f\, \mathsf{d}\mu$$
$$\int f\, \mathsf{d}\mu$$

Manchmal positioniert LaTeX nicht optimal. Zur Abhilfe werden den Autoren einige Korrekturoperatoren zur Verfügung gestellt. Der Operator `\!` bezeichnet einen kleinen negativen Zwischenraum, der bewirkt, daß f näher an das Integralzeichen heranrückt. Der Operator `\,` bezeichnet dagegen einen kleinen positiven Zwischenraum.

Der Buchstabe *d* ist keine Variable, sondern gehört zum Integral fest dazu. Das kann durch eine besondere Schriftart, hier `\sf`, hervorgehoben werden.

```
fac 5
```
$$fac5$$
$$fac5$$

LaTeX kennt keine mehrbuchstabigen Funktionsnamen. Es vermutet, daß es sich bei dieser Formel um ein Produkt der vier Faktoren f, a, c und 5 handelt.

```
{\it fac}\; 5
```
$$\mathit{fac}\; 5$$
$$\mathit{fac}\; 5$$

Zur Abhilfe sollte man „fac“ gruppieren und durch einen expliziten Schriftartselektor wie `\it` (oder auch `\rm` oder `\sf`) als Text, nicht als Formelteil auszeichnen. Ferner ist ein expliziter Abstand zum logischen Argument `5` nötig. Der Operator `\;` ähnelt dem `\,`-Operator, bewirkt aber einen größeren Abstand.

Natürlich erhebt die obige Beispielsammlung keinen Anspruch auf Vollständigkeit. Es gibt in LaTeX noch viele andere mögliche Operatoren in Formeln.

Formelstile

Wir haben bereits gesagt, daß eine Gesamtformel entweder im *Display-Stil D* oder im *Textstil T* gesetzt wird. Für Teilformeln gibt es zwei weitere Stile: den *Script-Stil S*, der für obere und untere Indizes (englisch „superscripts“ und „subscripts“) benutzt wird, und den *Script-Script-Stil SS* für Indizes von Indizes. Zeichengröße, Positionen und Abstände hängen vom Stil ab. Generell werden sie in der Reihenfolge *D*, *T*, *S*, *SS* kleiner.

Der Stil einer Gesamtformel ist *D* oder *T*, je nachdem, ob sie aus dem Text hervorgehoben wird oder im Text vorkommt. Der Stil einer Teilformel ergibt sich normalerweise automatisch aus dem Stil ihres Kontextes, d. h. der nächstgrößeren Teilformel. Er stimmt meist mit dem Stil des Kontexts überein. Die einzigen Ausnahmen

sind die Fälle, wo die Teilformel ein (oberer oder unterer) Index ist oder ein Zähler oder Nenner eines Bruchs. Für diese Fälle ergibt sich der Stil der Teilformel gemäß der folgenden Tabelle:

Kontext	*D*	*T*	*S*	*SS*
Indizes	*S*	*S*	*SS*	*SS*
Zähler und Nenner	*T*	*S*	*SS*	*SS*

Es gibt auch explizite Stilwechselkommandos `\displaystyle`, `\textstyle`, `\scriptstyle` und `\scriptscriptstyle`, die den Stil von Teilformeln unabhängig vom Kontext festlegen. Sie werden wie die Fontwechselkommandos `\bf` und `\it` benutzt, wirken also von der Position ihres Auftretens nach rechts bis zum Ende der aktuellen Gruppe. Beachten Sie, daß Gruppen durch `{ }` abgegrenzt werden, die in der gesetzten Formel unsichtbar sind. Die in der Mathematik üblichen Gruppierungsklammern `( )` haben keinen Einfluß auf die LaTeX-interne Gruppenbildung.

Beispiele:

```
a {b \scriptstyle c (d} + e)
x_i + y_{\textstyle i}
```

$$ab^{c(d}+e)$$
$$x_i+y_{i}$$

Gestauchte Teilformeln

Zusätzlich zu der Unterscheidung zwischen den vier oben genannten Formelstilen gibt es noch die zwischen *gestaucht* (englisch *cramped*) und normal. Eine Teilformel ist gestaucht, wenn sie unter einem anderen Objekt liegt. Beispiele sind die Nenner von Brüchen, überstrichene Teilformeln und solche unter einem Wurzelzeichen. In einer gestauchten Teilformel werden obere Indizes etwas niedriger positioniert als normal; vergleichen Sie z. B. x^2 mit $\overline{x^2}$. Im Gegensatz zu den vier Hauptstilen gibt es keine Möglichkeit, Gestauchtheit durch ein Kommando auszulösen oder zu verhindern.

Horizontale Zwischenräume

Abstände mit positiver oder negativer Größe können in Formeln genauso wie im Text erzeugt werden, nämlich durch das Kommando `\hspace{`*Länge*`}`. Als Längeneinheiten kommen wie im Text absolute wie `cm` oder fontabhängige wie `em` in Frage. Bei den fontabhängigen ist zu beachten, daß der Font vom Stil abhängt; im Script-Stil ist 1 `em` deutlich kleiner als im Display-Stil.

Zusätzlich zu den gewöhnlichen Einheiten gibt es noch eine sogenannte *mathematische Einheit* `mu`. Die Größe dieser Einheit hängt vom Font und damit vom Stil ab, da 1 `mu` immer $\frac{1}{18}$ `em` groß ist.

Die Zwischenraumerzeuger, die wir in den Formelbeispielen kennengelernt haben, sind in mathematischen Einheiten spezifiziert. Der negative Zwischenraum `\!` ist -3`mu` groß. Die drei Arten positiver Zwischenräume `\,` `\:` `\;` sind 3, 4 und 5 mathematische Einheiten lang. Genau genommen erzeugen diese Befehle Kleber (siehe Abschnitt 4.3.2), können also etwas schrumpfen oder wachsen.

Formelklassen

Wir haben bereits erwähnt, daß um ein binäres „+“ herum ein kleiner Zwischenraum eingefügt wird und um ein Gleichheitszeichen ein etwas größerer. Hier wollen wir nun die genauen Regeln für diese Zwischenräume vorstellen.

Jedes Symbol und jede Teilformel besitzt eine *Klasse*. Zwischen zwei horizontal benachbarten Symbolen oder Teilformeln wird ein Zwischenraum eingefügt, dessen Größe von der Klasse der beiden Nachbarn sowie dem Stil abhängt. Die Größe wird in mathematischen Einheiten angegeben, deren Umsetzung in absolute Einheiten wiederum vom Stil abhängt.

Jedem Symbol, sei es auf der Tastatur vorhanden wie „+“ oder über einen symbolischen Namen erzeugbar wie `\infty` für ∞, sowie jeder Teilformelart ist eine feste Klasse zugeordnet (bis auf die spezielle Behandlung von Binär-Operatoren). Im folgenden werden die einzelnen Klassen mit ihren typischen Vertretern vorgestellt.

Ord Dies ist die Klasse für in sich abgeschlossene Teilformeln. Typische Vertreter sind Buchstaben wie x, Ziffern wie 1, atomare Symbole wie ∞ und gruppierte Teilformeln, wie sie mit `\overline{ }`, `\underline{ }` oder Gruppierungsklammern `{ }` erzeugt werden.

Op ist die Klasse für „große“ Operatoren wie $\sum$, $\prod$ oder $\int$.

Bin Zu dieser Klasse gehören binäre Operatoren wie „+“ und „−“. Wenn allerdings ein solcher Operator in einem nicht-binären Kontext vorkommt wie in den Formeln -1, f^+ oder $(+)$, dann ist seine Klasse **Ord** statt **Bin**. Das ist die einzige Ausnahme von der Regel, daß die Klasse eines Formelteils nur von dessen Innenleben abhängt.

Rel ist die Klasse der Relationssymbole wie „=“, „<“ oder „$\leq$“. Diese Symbole haben immer die Klasse **Rel**, egal ob sie in einem binären Kontext vorkommen oder nicht.

Open Öffnende Klammern wie „(“ oder „[“.

Close Schließende Klammern wie „)“ oder „]“.

Punct Trennsymbole wie „,“, „;“ oder „?“ .

Inner ist die Klasse, die für Brüche benutzt wird. Kein einfaches Symbol hat diese Klasse.

Die Größe des zwischen zwei benachbarten Formelstücken einzufügenden Zwischenraums ist aus Tabelle 6.3 abzulesen. Die angegebenen Zahlenwerte sind ma-

	Ord	**Op**	**Bin**	**Rel**	**Open**	**Close**	**Punct**	**Inner**
Ord	0	3	(4)	(5)	0	0	0	(3)
Op	3	3	—	(5)	0	0	0	(3)
Bin	(4)	(4)	—	—	(4)	—	—	(4)
Rel	(5)	(5)	—	0	(5)	0	0	(5)
Open	0	0	—	0	0	0	0	0
Close	0	3	(4)	(5)	0	0	0	(3)
Punct	(3)	(3)	—	(3)	(3)	(3)	(3)	(3)
Inner	(3)	3	(4)	(5)	(3)	0	(3)	(3)

Tabelle 6.3: Abstand zwischen zwei Formelstücken

thematische Einheiten (mu). Zahlen in Klammern gelten nur für Display-Stil und Textstil; im Script- und Script-Script-Stil sind diese Abstände 0. Die mit „—“ gekennzeichneten Tabellenfelder entsprechen Kombinationen, die unmöglich sind, weil in ihnen ein **Bin**-Formelstück in nichtbinärem Kontext stehen würde.

Wie wird nun festgestellt, ob ein **Bin**-Objekt in einem nicht-binären Kontext liegt? In den folgenden Regeln soll ein Objekt alles außer Zwischenräumen oder Stilkommandos sein. Bei der Ermittlung des linken oder rechten Nachbarobjekts werden diese übersprungen.

1. Wenn ein **Bin**-Objekt das erste Objekt seiner Gruppe ist oder sein linkes Nachbarobjekt eine der Klassen **Bin**, **Op**, **Rel**, **Open** oder **Punct** hat, wird es in ein **Ord**-Objekt verwandelt.
2. Wenn ein **Bin**-Objekt das letzte Objekt seiner Gruppe ist oder sein rechtes Nachbarobjekt eine der Klassen **Rel**, **Close** oder **Punct** hat, wird es in ein **Ord**-Objekt verwandelt.

Diese beiden Regeln werden auf die Objekte in jeder Gruppe in einem Durchlauf von links nach rechts angewandt. Eine Gruppe der Form **Bin-Bin-Bin-Bin** verwandelt sich also in **Ord-Bin-Ord-Ord**, weil auf das erste **Bin** Regel 1 angewendet werden kann, *danach* aber nicht mehr auf das zweite **Bin** (dessen linker Nachbar bereits ein **Ord** geworden ist), usw. Betrachten Sie dazu die folgenden Beispiele mit dem **Bin**-Symbol „∗“:

$$* \qquad ** \qquad *** \qquad **** \qquad ***** \qquad ******$$

Alle hier erkennbaren Zwischenräume wurden automatisch erzeugt.

Die Klasse eines Formelteils kann explizit durch die Kommandos `\mathord{ }`, `\mathop{ }` usw. geändert werden, die gleichzeitig gruppierende Wirkung haben. Die Sonderregel, daß **Bin**-Objekte in nicht-binärem Kontext zu **Ord**-Objekten

werden, ist mächtiger als das Kommando `\mathbin{ }`, wie folgende Beispiele zeigen:

`x+y`	`x+?`	`x\mathbin{+}?`	`x+\mathord{?}`	`x=?`
$x+y$	$x+?$	$x+?$	$x+?$	$x=?$

Im ersten Beispiel ist das Plus-Zeichen von den **Ord**-Objekten *x* und *y* umgeben, behält also die Klasse **Bin**. Weil das Fragezeichen die Klasse **Punct** hat, erhält das Plus-Zeichen im zweiten Beipiel die Klasse **Ord**, und die Abstände fallen auf beiden Seiten weg. Daran ändert auch das `\mathbin`-Kommando im dritten Beispiel nichts. Wenn dagegen das Fragezeichen durch `\mathord` die Klasse **Ord** erhält, dann behält das Plus die Klasse **Bin** und die Abstände sind wie im ersten Beispiel. Das letzte Beispiel zeigt, daß die Klasse **Rel** grundsätzlich anders behandelt wird als **Bin**. Das Gleichheitszeichen bleibt **Rel** auch vor dem Fragezeichen. Gemäß Tabelle 6.3 wird links von ihm ein Abstand gelassen, rechts dagegen nicht.

6.4 Der Formelsatz-Algorithmus von TEX

Im folgenden wollen wir den Formelsatz-Algorithmus von TEX vorstellen, der auch dem Formelsatz von LATEX zugrunde liegt. Der Formelsatz-Algorithmus wurde genau wie der Rest von TEX von Donald E. Knuth entworfen. In Anhang G des TEX-Buchs [Knu86b] wird der Algorithmus ausführlich, aber informal beschrieben. Eine exakte Beschreibung als kommentierter Pascal-Programmtext ist in [Knu86a] enthalten. Wie die oben genannten Beispiele ahnen lassen, ist der Algorithmus recht kompliziert. Dazu kommt noch, daß Knuths Programm aufgrund der gewählten Sprache ziemlich schwer zu verstehen ist. Daher haben wir in [HW96] versucht, eine bessere Beschreibung zu entwickeln, die trotzdem noch dieselben Berechnungen ausführt. Hier lehnen wir uns an diese Beschreibung an, vereinfachen sie aber in vieler Hinsicht durch Weglassen von Einzelheiten.

Der Formelsatz geschieht in mehreren Schritten:

- *Einlesen:* Die Formelbeschreibung, eine lineare Zeichenfolge, wird eingelesen und in eine baumartige Internform verwandelt. Die Struktur dieser Internform wird in Unterabschnitt 6.4.1 vorgestellt. Den eigentlichen Einleseprozeß und den Aufbau der Internform werden wir hier nicht betrachten. Es sei nur so viel gesagt, daß während des Einlesens Makros expandiert werden und daß als Einzelzeichen wie „*x*“ oder „+“ oder als Namen wie `\infty` gegebene mathematische Symbole in eine Internform verwandelt werden. In TEX werden diese Symbole durch eine Zahl beschrieben, in die verschiedene Informationen hineinkodiert sind.
- *Setzen:* Die Internform der Formel, die noch keine Positions- oder Größenangaben enthält, wird in eine graphische Beschreibung umgesetzt, die die Position, Größe und Art jedes Zeichens genau festlegt. Die Umsetzung erfordert einige zusätzliche Stilparameter, die in Unterabschnitt 6.4.2 eingeführt werden. Die

entstandene graphische Beschreibung ist von ihrer Natur her ein *Kastenterm*, der aus vielen ineinandergeschachtelten rechteckigen Kästen besteht. Solche Kästen werden von TEX nicht nur beim Formelsatz benutzt, sondern auch bei allen anderen Formatierungsaufgaben.

Die möglichen Kastenterme werden in Unterabschnitt 6.4.3 vorgestellt. Die Umsetzung der Internform in Kastenterme stellt den eigentlichen Formelsatz dar. Sie wird ausführlich in Unterabschnitt 6.4.4 besprochen.

- *Einpassen:* Die gesetzte Formel, die durch einen Kastenterm gegeben ist, wird in ihre Umgebung eingepaßt. Eine Display-Formel steht zentriert in einer eigenen Zeile mit Zwischenraum darüber und darunter, während eine Textformel in eine Textzeile gesetzt wird. Textformeln sind durch Zeilenumbrüche trennbar, d. h. sie werden genau genommen nicht durch einen einzelnen Kasten, sondern durch eine Folge von Inhaltskästen, Kleberelementen und Strafelementen dargestellt, wie wir es in Abschnitt 4.3 kennengelernt haben.

  Wir werden die Nachbehandlung von Formeln in diesem Buch nicht betrachten. Daher werden wir zur Vereinfachung auch nichts über Kleber und Strafelemente in Formeln aussagen.

- *Ausgeben:* Der letzte Schritt hat eigentlich nichts mehr mit dem Formelsatz zu tun. Der Kastenterm für eine ganze Seite wird in die Seitenbeschreibungssprache *dvi* übersetzt und ausgegeben. Wir werden diese Übersetzung in Abschnitt 9.4 besprechen, nachdem wir die Sprache *dvi* eingeführt haben.

Wegen der Fülle der in Formeln möglichen Operatoren werden wir uns bei der Definition der Internform und ihrer Umsetzung in Kastenterme auf einige wichtige Fälle beschränken, nämlich Zwischenräume, Symbole, Über- und Unterstreichungen, Brüche und Indizes oben oder unten oder beide zusammen, aber immer nur rechts von der Grundformel angeordnet. Nicht besprochen werden aus Platzgründen Klassenkommandos, Stilkommandos, große Operatoren und Indizes in `\limits`-Position, die Befehle `\limits` und `\nolimits`, Gruppierungen, Wurzeln, Textfonts in Formeln wie `{\it fac}` sowie einige Dinge, die nicht in den Beispielen vorkamen, wie diakritische Zeichen, größenmäßig angepaßte Klammersymbole und verallgemeinerte Brüche. Manche dieser Dinge sind sehr einfach zu behandeln, während andere relativ aufwendig sind.

6.4.1 Die Interndarstellung von Formeln

Wie bereits erwähnt, wird eine in der Eingabe vorkommende Formelbeschreibung vom System eingelesen und in eine baumartige Interndarstellung umgewandelt, die als Grundlage für den zweidimensionalen Formelsatz dient.

Wir werden zunächst die wirkliche Internform von TEX darstellen, wie sie von Knuth beschrieben wird. Diese Form hat einige Schwächen, die wesentlich dazu beitragen, Knuths Beschreibung des Formelsatzes unverständlich zu machen. Wir werden daher im Anschluß einen eigenen Entwurf für die Internform vorstellen, der besser geeignet ist, den Formelsatz zu beschreiben.

Die Original-Internform

Gemäß Knuths Beschreibung werden Formeln intern als *mathematische Listen* dargestellt. Jede mathematische Liste ist eine Folge von *mathematischen Elementen*. Ein Element ist ein *Atom*, ein *horizontaler Zwischenraum*, ein *Stilkommando* (z. B. `\textstyle`) oder ein *verallgemeinerter Bruch*. Es gibt noch einige andere Fälle, die wir hier nicht betrachten wollen.

Atome bestehen aus (mindestens) drei Teilen: einem *Kern* (englisch *nucleus*), einem *oberen Index* (englisch *superscript*) und einem *unteren Index* (englisch *subscript*). Jeder dieser Teile kann leer, ein Symbol oder eine mathematische Liste sein. Es gibt dreizehn Arten von Atomen, von denen manche mehr als die obengenannten drei Teile haben. Acht Atomarten entsprechen den von uns bereits eingeführten Klassen **Ord**, **Op** usw., die die Abstände innerhalb einer Formel kontrollieren. Die übrigen fünf Arten haben eine ernsthaftere Bedeutung. Atome der Art „overline“ z. B. entsprechen überstrichenen Teilformeln.

Die von Knuth vorgeschlagene Internform hat einige Schwächen. Die Felder für die oberen und unteren Indizes sind meistens leer. Sie sollten nur bei Bedarf angelegt werden. Die dreizehn Atomarten sind aus einer Mischung völlig verschiedener Konzepte hervorgegangen, nämlich einer Klassifikation, die zum Einfügen von Zwischenräumen benötigt wird, und der Benutzung von Operatoren, die etwas über die gewünschte Gestalt der Teilformel aussagen. Der von Knuth beschriebene Formelsatz wird durch diese Vermischung von Konzepten unnötig kompliziert, wie wir an zwei Beispielen zeigen werden.

„Overline“-Atome werden während eines ersten Durchgangs durch die Formel behandelt. Der Überstrich wird zu der betreffenden Teilformel hinzugefügt. Anschließend wird das ganze in ein **Ord**-Atom verwandelt, da die Zwischenräume um überstrichene Teilformeln genauso groß sein sollen wie die um **Ord**-Atome. Die Zwischenräume selbst werden erst in einem zweiten Durchlauf durch die Formel eingefügt.

Brüche sind mathematische Elemente, die keine Atome sind. Ihr Aussehen wird während des ersten Durchgangs durch die Formel berechnet. Danach werden sie in **Inner**-Atome verwandelt. Die Art **Inner** definiert die Größe der Zwischenräume, die im zweiten Durchgang durch die Formel eingefügt werden.

Die Vermischung von Konzepten führt also dazu, daß die Internform der Formel während der Übersetzung in einen Kastenterm mehrfach abgeändert werden muß. Das ist einer der Gründe dafür, daß Knuths Beschreibung des Formelsatzes schwer zu verstehen ist.

Eine alternative Internform

Um die oben angesprochenen Probleme zu vermeiden, haben wir die Definition der Internform von Formeln völlig überarbeitet.

Wie bei Knuths Internform unterscheiden wir zwischen mathematischen Listen und mathematischen Elementen. Wir nennen die Menge der Listen *MList* und die Menge der Elemente *M*. Jede Liste ist eine endliche Folge von Elementen, die auch leer sein kann, d. h. jedes Element aus *MList* kann in der Form $[m_1, \ldots, m_n]$ geschrieben werden mit $m_1, \ldots, m_n$ aus *M*. Es gibt mehrere Arten von Elementen, die sich im Aufbau unterscheiden. Als Arten werden nur solche benutzt, die etwas über die Gestalt der entsprechenden Teilformel aussagen. Wir unterscheiden dabei nicht zwischen Atomen und Nicht-Atomen.

Es folgt eine Liste der von uns betrachteten Elementarten. Sie enthält nur solche Arten, für die wir die Übersetzung in Kastenterme definieren werden. Für eine volle Behandlung von TEX-Formeln wären noch weitere Elementarten erforderlich. Viele der angegebenen Arten sind zudem noch vereinfacht.

- **Sym** (s)
  Diese Art Element bezeichnet ein einzelnes Symbol *s* aus der Menge *Symbol* der Symbole. Wir werden gleich mehr über diese Menge sagen.
- **MathSpace** (n)
  Ein Element dieser Art repräsentiert einen horizontalen Zwischenraum der Größe *n*, gemessen in mathematischen Einheiten (`mu`). Die Kommandos `\!`, `\,` usw. werden beim Einlesen in **MathSpace**-Elemente verwandelt. Eigentlich stehen diese Kommandos für Kleber, aber wir ignorieren hier ihre Dehnbarkeit und Schrumpfbarkeit.
- **Over** (l)
  Eine überstrichene Teilformel, die als mathematische Liste *l* gegeben ist.
- **Under** (l)
  Eine unterstrichene Teilformel.
- **Frac** (l_1, l_2)
  Ein Bruch mit Zähler l_1 und Nenner l_2. Beides sind mathematische Listen aus *MList*. Brüche in TEX sind eigentlich viel allgemeiner und erfordern daher zusätzliche Komponenten, die wir hier wegfallen lassen.
- **Sup** (l, l')
  Eine mathematische Liste *l* aus *MList* mit einem oberen Index l', ebenfalls aus *MList*.
- **Sub** (l, l')
  Eine mathematische Liste *l* mit einem unteren Index l'.
- **SupSub** (l, l_1, l_2)
  Eine mathematische Liste *l* mit einem oberen Index l_1 und einem unteren Index l_2, alle drei aus *MList*.

Unter einem Element kann man sich entweder einen Term vorstellen, also eine durch Klammern eindeutig gegliederte Zeichenfolge, oder einen Baum, wobei die fettgedruckte Elementart eine Wurzelmarkierung darstellt und die in Klammern angegebenen Komponenten den Teilbäumen entsprechen.

Ein Beispiel

In unserer Internform kann die Formel `(x_i+y)^{\overline{n+1}}` wie folgt als Liste von fünf Elementen dargestellt werden:

[**Sym** („(“),
  Sub ([**Sym** („*x*“)], [**Sym** („*i*“)]),
  Sym („+“),
  Sym („*y*“),
  Sup ([**Sym** („)“)], [**Over** ([**Sym** („*n*“), **Sym** („+“), **Sym** („1“)])])
]

Beachten Sie, daß die runden Klammern keine Bedeutung bei der Bestimmung von Teilformeln haben; der erste Operand von ^ ist nur die schließende Klammer. Das ändert sich, wenn geschweifte Klammern zur Gruppierung benutzt werden; die Formel `{(x_i+y)}^{\overline{n+1}}` hat als Internform

[**Sup** ([**Sym** („(“),
          Sub ([**Sym** („*x*“)], [**Sym** („*i*“)]),
          Sym („+“),
          Sym („*y*“),
          Sym („)“)],
         [**Over** ([**Sym** („*n*“), **Sym** („+“), **Sym** („1“)])])
]

Symbole

In der Internform sind die Argumente des **Sym**-Konstruktors Elemente der Menge *Symbol* der mathematischen Symbole. In dem Beispiel haben wir die Symbole als Zeichen geschrieben wie „*x*“ oder „+“. In Knuths originaler Beschreibung sind Symbole Zahlen, in die verschiedene Informationen kodiert sind. Hier werden wir nicht genauer sagen, was Symbole sind, sondern einfach annehmen, daß es die Menge *Symbol* der Symbole gibt und daß auf dieser Menge gewisse Funktionen definiert sind, die die in ein Symbol kodierten Informationen liefern. Wir werden diese Funktionen später bei Bedarf kennenlernen.

6.4.2 Zusatzparameter für den Formelsatz

Der eigentliche Formelsatz übersetzt eine Formel, dargestellt als mathematische Liste, d. h. als Element von *MList*, in einen Kastenterm. Die Übersetzung hängt noch von verschiedenen Parametern ab, die wir jetzt einführen werden. Es handelt sich einmal um den *Stil*, in dem eine Formel gesetzt wird, und zum zweiten um gewisse *Stilparameter*, die angeben, wie dick ein Bruchstrich sein soll oder wie hoch ein oberer Index gesetzt wird.

Formelstile

Wir haben die vier Stile Display *D*, Text *T*, Script *S* und Script-Script *SS* bereits eingeführt und erläutert. Das TEX-Buch kennt jedoch acht Stile; außer den vier eben genannten noch gestauchte (englisch „cramped“) Versionen D', T', S' und SS', die unter Bruch- oder Überstrichen auftreten. In einem gestauchten Stil sitzen obere Indizes etwas tiefer als sonst.

Eine genaue Analyse der Art, wie Stile festgelegt und benutzt werden, zeigt jedoch, daß es besser ist, die Stilinformation von der Gestauchtheitsinformation getrennt zu halten. Wir definieren daher als Menge der Stile

$$\mathit{Style} = \{D, T, S, SS\}.$$

Für die Gestauchtheit nehmen wir die Menge *Bool* der Wahrheitswerte *true* und *false*.

In der Tabelle auf Seite 148 haben wir gezeigt, wie der Stil eines Index bzw. eines Zählers oder Nenners von dem Stil des Kontexts abhängt. Wir wollen diese Abhängigkeit durch die zwei Funktionen

$$\mathit{script}, \mathit{fract} : \mathit{Style} \to \mathit{Style}$$

formalisieren, d. h. es soll gelten $\mathit{script}(D) = S$, $\mathit{fract}(D) = T$ usw.

Stilparameter

Die exakte graphische Struktur einer Formel hängt von den *Stilparametern* ab. Sie kontrollieren die Lage von Indizes, den Abstand zwischen Zähler und Bruchstrich, die Dicke des Bruchstrichs usw.

In TEX sind die Stilparameter Zusatzinformationen zu den Fonts, die beim Formelsatz benutzt werden. Da die Wahl des Fonts vom Formelstil abhängt, modellieren wir die Stilparameter als Funktionen des Typs $\mathit{Style} \to \mathit{Dim}$, wobei *Dim* die Menge der möglichen Längenangaben sein soll.

An dieser Stelle wollen wir nur die wichtigsten Stilparameter kurz erläutern. Die anderen werden vorgestellt, wenn sie gebraucht werden.

xHeight	Höhe des Zeichens „*x*“ im aktuellen Font
Quad	Längeneinheit 1 `em` im aktuellen Font
RuleThickness	Dicke von Strichen
AxisHeight	Abstand von der Achse zur Grundlinie

Die *Achse* ist die Linie, auf der Bruchstriche sitzen. Betrachten Sie z. B. $x + \frac{y}{z}$. Die Grundlinie befindet sich am Unterrand von „Sie“ und „*x*“. Der Bruchstrich liegt etwas über der Grundlinie. Viele mathematische Symbole wie z. B. das Pluszeichen in dem Beispiel sind symmetrisch zur Achse.

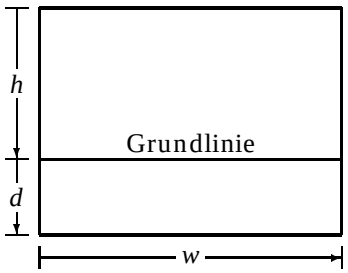

Abbildung 6.2: Kasten mit Höhe h, Tiefe d und Breite w

Außer den stilabhängigen Parametern gibt es noch einige Konstanten, die vom Stil unabhängig sind. Ein Beispiel ist die Konstante *scriptSpace* aus *Dim*, die angibt, wieviel Extraplatz nach einem Index gelassen werden soll, üblicherweise 0,5 pt.

6.4.3 Die Zieldarstellung: Kastenterme

Kästen

Eine mathematische Liste oder ein mathematisches Element wird während des eigentlichen Formelsatzes in eine Liste von *Kästen* (englisch *boxes*) übersetzt. Ein Kasten kann aus ineinandergeschachtelten Teilkästen bestehen, die letztlich Zeichen enthalten, ganz weiß oder ganz schwarz sind. Jeder Kasten hat eine horizontale *Grundlinie*, an deren linkem Ende sich der *Referenzpunkt* des Kastens befindet. Jeder Kasten hat drei Größenattribute: die *Höhe* (englisch *height*) *h*, die *Tiefe* (englisch *depth*) *d* und die *Breite* (englisch *width*) *w*. Die Höhe bezeichnet den Abstand vom Oberrand des Kastens zur Grundlinie, die Tiefe den von der Grundlinie zum Unterrand und die Breite den Abstand vom linken zum rechten Rand. Abbildung 6.2 zeigt einen Kasten mit Höhe *h*, Tiefe *d* und Breite *w*.

Alle drei Abmessungen eines Kastens können negativ sein. Um zu verstehen, was das bedeutet, müssen wir genauer definieren, was „Abstand von *A* zu *B*“ heißt. Jede Seite ist mit einem Koordinatensystem versehen, dessen *x*-Achse wie üblich nach rechts zeigt, während die *y*-Achse nach unten weist. Mit Hilfe dieser Koordinaten kann man also definieren:

$$\begin{aligned} \textit{Horizontaler Abstand} \text{ von } A \text{ zu } B &= x\text{-Koordinate von } B - x\text{-Koordinate von } A \\ \textit{Vertikaler Abstand} \text{ von } A \text{ zu } B &= y\text{-Koordinate von } B - y\text{-Koordinate von } A \end{aligned}$$

Die meisten Kästen haben eine positive Breite, das heißt, ihr linker Rand ist links vom rechten Rand. Kästen negativer Breite können z. B. aus negativen Zwischenräumen hervorgehen. Sie haben die Eigenschaft, daß ihr linker Rand, der den Referenzpunkt trägt, rechts von ihrem rechten Rand liegt. Das klingt unlogisch, ent-

spricht aber den unüblichen physikalischen Eigenschaften dieser Kästen: sie haben eine negative Breite, was für einen konkreten physikalischen Körper unmöglich ist.

Höhe und Tiefe eines Kastens sind positiv, wenn die Grundlinie im Inneren des Kastens liegt, d. h. unter dem Oberrand und über dem Unterrand. Die Höhe ist negativ, wenn die Grundlinie über dem Oberrand liegt, und die Tiefe, wenn sie unter dem Unterrand liegt.

Ein Kasten für sich alleine hat noch keine feste Position auf der Seite. Seine Lage ist nur bezüglich des Referenzpunkts eines umfassenden Kastens bestimmt, nachdem ein solcher aufgebaut worden ist. Dementsprechend legt jeder Kasten die Positionen seiner Unterkästen relativ zu seinem Referenzpunkt fest.

Kastenterme und ihre Attribute

Im folgenden werden wir die Menge *Box* der Kastenterme einführen, die einfache und zusammengesetzte Kästen beschreiben. Im Vergleich zu der Situation in TEX und auch in unserem Artikel [HW96] sind die Kästen etwas vereinfacht und gleichförmiger gemacht.

Jeder Kasten hat die Form **Box**(h, d, w, c), wobei *h*, *d* und *w* die Höhe, Tiefe und Breite bezeichnen, während *c* den Inhalt des Kastens darstellt. Die Menge *Content* der Kasteninhalte hat die folgenden Arten von Elementen:

- **White**
  Kein Inhalt; solche Kästen dienen als Zwischenräume und Abstandshalter.
- **Black**
  Der ganze Kasten ist schwarz. Wir werden solche Kästen benutzen, um Bruchstriche, Über- und Unterstriche zu erzeugen. Bei diesen Strichen ist die Summe aus Höhe und Tiefe keineswegs 0, sonst wären sie nämlich unsichtbar.
- **Chr** (ch)
  Dies ist ein Kasten, der ein Ausgabezeichen *ch* enthält. Ausgabezeichen sind etwas anderes als Symbole; in der Formeleingabe `x^x` z. B. kommt zweimal dasselbe Symbol „*x*“ vor, während in ihrer graphischen Umsetzung x^x zwei verschiedene Ausgabezeichen vorkommen; eins etwas größer als das andere. Die Umwandlung von Symbolen in Ausgabezeichen ist stilabhängig.
- **HList** (hl)
  Nachdem wir bisher nur atomare Kästen gesehen haben, ist dies die erste Art zusammengesetzter Kästen. Das Argument *hl* ist nämlich – in erster Näherung – eine Liste von Kästen. Diese Liste kann auch leer sein. Im einfachsten Fall werden diese Kästen horizontal nebeneinandergelegt, so daß ihre Referenzpunkte auf gleicher Höhe liegen. Der Referenzpunkt des Gesamtkastens ist der des ersten Teilkastens. In Abbildung 6.3 zeigen wir einen **HList**-Kasten aus drei Teilkästen.

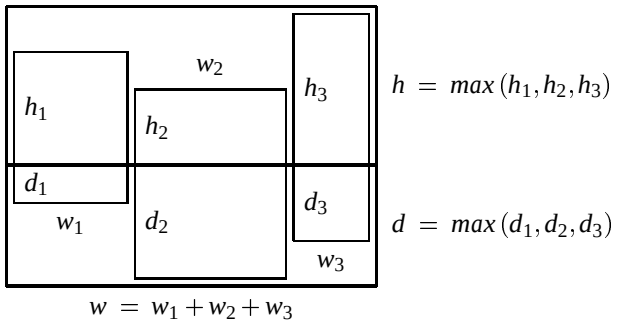

Abbildung 6.3: Ein **HList***-Kasten aus drei Teilkästen*

Die im Bild zu sehenden kleinen Abstände zwischen den drei nebeneinandergestellten Teilkästen und zwischen ihnen und dem umfassenden Gesamtkasten dienen nur zur besseren Veranschaulichung. In Wirklichkeit sitzen die drei Teilkästen im Ergebnis direkt nebeneinander und der umfassende Kasten liegt direkt an ihnen an. Gewöhnlich ist also die Breite des Gesamtkastens gleich der Summe der Breiten der Einzelkästen.

Abweichend von dem oben beschriebenen kann jeder einzelne Kasten in der horizontalen Liste um einen gewissen Betrag nach unten verschoben sein. Allgemein hat daher die Liste *hl* die Form $[s_1 \triangleright b_1, \ldots, s_n \triangleright b_n]$, wobei s_i die Verschiebung des Kastens b_i ist. Die Verschiebung darf auch negativ sein, d. h. nach oben gerichtet. Wir nennen die Menge dieser Listen *HList*.

- **VList** (*vl*)
  Neben der horizontalen Aneinanderreihung von Teilkästen gibt es auch eine vertikale. Das Argument *vl* ist wieder eine Liste von Kästen mit Verschiebungen.

  Wenn alle Verschiebungen 0 sind, werden die Kästen so übereinander gelegt, daß ihre linken Ränder auf einer Linie liegen und der Unterrand jedes Kastens den Oberrand des nächsten berührt. Im Gegensatz zur horizontalen Liste gibt es keine natürliche Wahl für den Referenzpunkt des umfassenden Kastens. Wir werden Beispiele treffen, wo der Referenzpunkt der des obersten oder untersten Kastens oder eines Kastens dazwischen ist. In Abbildung 6.4 werden zwei Beispiele veranschaulicht. Ähnlich wie im vorhergehenden Bild dienen die kleinen Abstände zwischen den Kästen nur der besseren Veranschaulichung.

  Die Verschiebung geht diesmal nach rechts statt nach unten. Sie darf auch negativ, d. h. nach links gerichtet sein. Wegen der unterschiedlichen Bedeutung der Verschiebung geben wir der Menge der als Parameter von **VList** möglichen Listen einen neuen Namen, *VList*.

Die meisten Elemente von horizontalen und vertikalen Listen haben die Verschiebung 0. Diese werden wir nicht explizit angeben, d. h. wir schreiben einfach *b* statt $0 \triangleright b$.

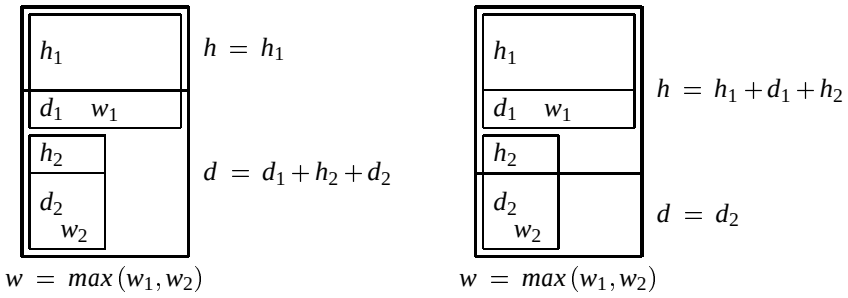

Abbildung 6.4: Zwei Beispiele vertikaler Aneinanderreihung

Funktionen zum Bau atomarer Kästen

Im folgenden führen wir Funktionen ein, die die Konstruktion atomarer Kästen erleichtern. Dabei stellen wir von den meisten Funktionen zuerst eine allgemeinere Version vor, die durch einen Strich gekennzeichnet ist, und dann eine Funktion für einen wichtigen Spezialfall ohne Strich.

Die Funktion $rule' : Dim \times Dim \times Dim \to Box$ erzeugt einen schwarzen Kasten mit gegebener Höhe, Tiefe und Breite. Sie ist definiert durch

$$rule'(h,d,w) = \mathbf{Box}(h,d,w,\mathbf{Black}).$$

Oft ist die Tiefe 0. Für diesen Spezialfall definieren wir eine eigene Funktion *rule*:

$$rule(h,w) = rule'(h,0,w).$$

Die Funktion $hSpace : Dim \to Box$ erzeugt einen Leerraum gegebener Breite zur Verwendung in einer horizontalen Liste. Höhe und Tiefe sind 0. Die Funktion ist definiert durch

$$hSpace(w) = \mathbf{Box}(0,0,w,\mathbf{White}).$$

Die Funktion $vSpace' : Dim \times Dim \to Box$ erzeugt einen Leerraum gegebener Höhe und Tiefe zur Verwendung in einer vertikalen Liste. In diesem Fall ist die Breite 0. Die Funktion ist definiert durch

$$vSpace'(h,d) = \mathbf{Box}(h,d,0,\mathbf{White}).$$

Ähnlich wie bei den schwarzen Kästen gibt es einen häufigen Spezialfall mit Tiefe 0, für den wir eine eigene Funktion *vSpace* einführen:

$$vSpace(h) = vSpace'(h,0).$$

Horizontale Kombination

Desweiteren benötigen wir eine Funktion $hBox : HList \to Box$, die eine horizontale Kastenliste mit Verschiebungen in einen umfassenden Kasten packt. Ihre Definition hat die Form

$$hBox\,(hl) \;=\; \mathbf{Box}\,(h,d,w,\mathbf{HList}\,(hl)),$$

wobei die Ausmaße h, d und w des Ergebniskastens aus dem Argument hl bestimmt werden müssen. Dazu nehmen wir an, daß hl die Form $[s_1 \triangleright b_1, \ldots, s_n \triangleright b_n]$ hat, wobei $b_i = \mathbf{Box}\,(h_i,d_i,w_i,c_i)$ ist. Dann ist

$$\begin{aligned} h &= max\,(0, h_1 - s_1, \ldots, h_n - s_n) \\ d &= max\,(0, d_1 + s_1, \ldots, d_n + s_n) \\ w &= w_1 + \cdots + w_n \end{aligned}$$

Vertikale Kombination

Um eine vertikale Kastenliste mit Verschiebungen in einen umfassenden Kasten zu packen, benutzen wir eine Funktion $vBox : VList' \to Box$, die als Argument eine Liste verschobener Kästen mit einem ausgezeichneten (hier unterstrichenen) Element nimmt; die Menge aller solchen Listen soll $VList'$ heißen. Der Sinn des ausgezeichneten Elements ist es, den Referenzpunkt des umfassenden Kastens festzulegen.

Die Definition von $vBox$ hat die Form

$$vBox\,(vl') \;=\; \mathbf{Box}\,(h,d,w,\mathbf{VList}\,(vl)).$$

Zur Bestimmung von h, d, w und vl schreiben wir vl' als

$$[s_1 \triangleright b_1, \ldots, s_m \triangleright b_m, \underline{s \triangleright b}, s'_1 \triangleright b'_1, \ldots, s'_n \triangleright b'_n],$$

wobei m und n auch 0 sein dürfen. Dann ist

$$\begin{aligned} h &= (h_1 + d_1) + \cdots + (h_m + d_m) + h \\ d &= d + (h'_1 + d'_1) + \cdots + (h'_n + d'_n) \\ w &= max\,(0, w_1 + s_1, \ldots, w_m + s_m, w + s, w'_1 + s'_1, \ldots, w'_n + s'_n) \end{aligned}$$

wenn die Ausmaße der Teilkästen auf natürliche Weise benannt werden. Die Liste vl im Ergebnis entspricht der im Argument bis auf den Wegfall der Auszeichnung:

$$vl \;=\; [s_1 \triangleright b_1, \ldots, s_m \triangleright b_m, s \triangleright b, s'_1 \triangleright b'_1, \ldots, s'_n \triangleright b'_n].$$

6.4.4 Der eigentliche Formelsatz

Der eigentliche Formelsatz besteht aus einer Übersetzung der Formelinternform in einen Kastenterm. Wir werden diese Übersetzung im folgenden durch die Definition von Funktionen beschreiben, die jeweils gewisse Arten von Formeln übersetzen.

Das Setzen von Gesamtformeln

Für die Übersetzung einer vollständigen Formel in einen Kastenterm benutzen wir die zwei Funktionen *displayFormula* und *inlineFormula*. Die erste dient zur Übersetzung von Formeln in Displays, d. h. abgesetzt vom Text in eigenen Zeilen. Die zweite behandelt Formeln in Textzeilen. Beide Funktionen haben den Typ $MList \to HList$, weil eine Gesamtformel intern als mathematische Liste (Menge *MList*) dargestellt wird. Das Ergebnis ist eine horizontale Liste, kein Kasten, da eine Inline-Formel umbrochen werden kann.

Die beiden Funktionen werden unter Benutzung einer gemeinsamen Hilfsfunktion

$$MListToHList : Style \times Bool \times MList \to HList$$

realisiert, die noch zwei andere Argumente hat: einen Formelstil (Menge *Style*) und einen Wahrheitswert (Menge *Bool*), der Auskunft über die Gestauchtheit gibt.

$$\begin{aligned} displayFormula(ml) &= MListToHList(D, false, ml) \\ inlineFormula(ml) &= MListToHList(T, false, ml) \end{aligned}$$

Der Stil ist einmal D und einmal T. Gesamtformeln sind nie gestaucht.

Eigentlich ist der Unterschied zwischen den beiden Funktionen größer. In TEX werden Display-Formeln in einer eigenen Zeile zentriert und auf Wunsch auch numeriert. Formeln in Textzeilen werden dagegen dem Zeilenumbruch unterworfen, d. h. es müssen potentielle Umbruchstellen gefunden und Strafelemente eingefügt werden.

Das weitere Vorgehen

Die Funktion *MListToHList* werden wir erst später definieren. Um die einzelnen Elemente der mathematischen Liste zu übersetzen, benutzt *MListToHList* eine Hilfsfunktion

$$ElemToHList : Style \times Bool \times M \to HList,$$

deren erste zwei Argumente wieder Stil und Gestauchtheit angeben.

Im Zuge ihrer Arbeit ruft die Funktion *ElemToHList* wiederum *MListToHList* auf Teilformeln auf. Alle diese rekursiven Aufrufe erfolgen über die Hilfsfunktion

$$MListToBox : Style \times Bool \times MList \to Box,$$

die definiert ist durch

$$MListToBox(st, cr, ml) \;=\; hBox(MListToHList(st, cr, ml)).$$

Im folgenden werden wir *ElemToHList* für die verschiedenen Arten von mathematischen Elementen definieren, die wir hier in diesem Buch eingeführt haben.

Horizontale Zwischenräume

Zwischenräume in einer Formel werden in mathematischen Einheiten (mu) angegeben, deren Größe $\frac{1}{18}$ em beträgt. Die Längeneinheit em ist durch den Stilparameter *Quad* gegeben. Damit ergibt sich:

$$ElemToHList(st, cr, \mathbf{MathSpace}(n)) \;=\; [hSpace(n \cdot Quad(st)/18)]$$

Das Setzen von Symbolen

Die nächste Art von mathematischen Elementen, die wir betrachten, hat die Struktur $\mathbf{Sym}(s)$, wobei s ein Mitglied der Menge *Symbol* ist. Zunächst muß das Symbol in ein Ausgabezeichen umgewandelt werden. Diese Umwandlung hängt vom Formelstil ab; in x^x z. B. tritt zweimal dasselbe Symbol auf, das in zwei verschiedene Ausgabezeichen umgewandelt wird. Die Umwandlung wird von der Funktion $setSym : Style \times Symbol \to Char$ erledigt, die wir hier nicht näher spezifizieren.

Die Menge *Char* der Ausgabezeichen wird hier ebenfalls nicht genauer beschrieben. Um die Ausmaße des aus einem Zeichen gebildeten Kastens bestimmen zu können, muß es Funktionen *charHeight*, *charDepth* und *charWidth* vom Typ $Char \to Dim$ geben, die Auskunft über die Ausmaße eines Zeichens erteilen. (In TEX sind diese Funktionen durch Nachschlagen in Fonttabellen realisiert.)

Die durch *charWidth* angegebene „natürliche“ Breite eines Zeichens ist die, die beim Zusammensetzen von Wörtern aus Einzelzeichen benutzt wird. Wenn das Zeichen geneigt ist wie z. B. f, ist diese Breite kleiner als die Gesamtbreite des Zeichens. Der Unterschied wird durch die Funktion *charItalic* gegeben, deren Name daher rührt, daß das Anhängen der Extrabreite im Englischen *italic correction* genannt wird. Sie wird sichtbar, wenn ein geneigtes Zeichen mit oberen und unteren Indizes versehen wird; in f_1^1 zum Beispiel entspricht der horizontale Abstand zwischen den beiden Indizes dem *charItalic*-Wert von f.

Beim Formelsatz wird die „italic correction“ manchmal durchgeführt und manchmal nicht. Daher definieren wir zunächst eine Funktion $boxChar : Char \to Box$, die ein Zeichen in einen Kasten mit natürlicher Breite einschließt. Die Benutzer dieser Funktion können dann entscheiden, ob sie die „italic correction“ durchführen wollen, d. h. einen Leerraum von der Größe des Überhangs anhängen.

$$boxChar(ch) \;=\; \mathbf{Box}(h, d, w, \mathbf{Chr}(ch))$$
wobei $h = charHeight(ch),\ d = charDepth(ch),\ w = charWidth(ch).$

Im Falle eines einzelnen Zeichens in einer Formel wird die Korrektur durchgeführt. Wir benutzen dazu eine Funktion $correct : Box \times Dim \to HList$, die definiert ist als

$$correct(b,i) = \begin{cases} [b] & \text{falls } i = 0 \\ [b, hSpace(i)] & \text{sonst} \end{cases}$$

Die Verschiebungen sind 0; wir haben sie gemäß unserer Konvention weggelassen. Damit ergibt sich für *ElemToHList* angewandt auf ein Symbol:

$$ElemToHList(st, cr, \mathbf{Sym}(s)) = correct(boxChar(ch), charItalic(ch))$$
wobei $ch = setSym(st, s)$.

In diesem Falle hat die Gestauchtheit keinen Einfluß auf das Ergebnis.

Überstrichene Teilformeln

Eine überstrichene Formel wie $\overline{x}$ hat zwei sichtbare Bestandteile, nämlich die Teilformel x und den Strich. Dazu kommen noch zwei unsichtbare Bestandteile. Zwischen der Teilformel und dem Strich ist nämlich ein vertikaler Zwischenraum; der Strich sitzt ja nicht direkt auf dem x auf. Über dem Strich ist ein zweiter Leerraum, der erst relevant wird, wenn die ganze Formel unter einem anderen Objekt plaziert wird. Daher besteht die Berechnung von $ElemToHList(st, cr, \mathbf{Over}(ml))$ aus den folgenden Schritten:

1. $b = MListToBox(st, true, ml)$

   Ein Kasten b wird erzeugt, indem die Teilformel ml unter dem Strich in demselben Stil st wie die gesamte Teilformel gesetzt wird. Sie ist allerdings immer gestaucht, egal ob es die ganze Teilformel war oder nicht.

2. $r = rule(th, width(b))$

   Der Strich r wird als ein schwarzer Kasten erzeugt. Seine Dicke th wird durch einen Stilparameter bestimmt:

   $$th = RuleThickness(st).$$

   Seine Länge stimmt mit der Breite von b überein. Die Funktion *width* liefert einfach die *width*-Komponente ihres Argumentkastens zurück:

   $$width(\mathbf{Box}(h,d,w,c)) = w.$$

3. $b' = vBox[vSpace(th), r, vSpace(3 \cdot th), \underline{b}]$

   Es wird also ein Kasten b' aus einer vertikalen Liste zusammengebaut, die von oben nach unten aufgezählt wird. Die horizontale Verschiebung aller vier Bestandteile ist 0. Der Referenzpunkt von b' stimmt mit dem von b überein, was durch die Unterstreichung spezifiziert wird. Der Leerraum über dem Strich ist so dick wie der Strich, und der Abstand vom Strich zu b ist dreimal so breit.

4. Das Ergebnis von *ElemToHList* sollte aber nicht ein Kasten, sondern eine horizontale Liste sein. Daher ist das Ergebnis nicht b' selbst, sondern $[b']$, d. h. $[0 \triangleright b']$.

Unterstrichene Teilformeln

Die unterstrichenen Teilformeln sind genau symmetrisch zu den überstrichenen aufgebaut. Auch sie haben zwei unsichtbare Bestandteile, einen vertikalen Zwischenraum zwischen Teilformel und Strich und einen Leerraum unter dem Strich. Daher besteht die Berechnung von $\mathit{ElemToHList}(st, cr, \mathbf{Under}(ml))$ aus den folgenden Schritten:

1. $b = \mathit{MListToBox}(st, cr, ml)$
   Im Unterschied zu eben ist die Teilformel über dem Strich nur dann gestaucht, wenn die ganze Teilformel gestaucht war.
2. $r = \mathit{rule}(th, \mathit{width}(b))$
   wobei *th* dasselbe wie eben ist.
3. $b' = \mathit{vBox}[\underline{b}, \mathit{vSpace}(3 \cdot th), r, \mathit{vSpace}(th)]$
   Jetzt müssen die Bestandteile der vertikalen Liste natürlich umgekehrt aufgezählt werden. Der Referenzpunkt von b' stimmt wieder mit dem von *b* überein, und die Abstände sind dieselben wie eben.
4. Wieder ist das Ergebnis nicht b' selbst, sondern $[b']$.

Brüche

Wir definieren jetzt die Berechnung von $\mathit{ElemToHList}(st, cr, \mathbf{Frac}(num, den))$ für einen Bruch mit Zähler *num* (englisch *numerator*) und Nenner *den* (englisch *denominator*).

1. $st' = \mathit{fract}(st)$
   Der Stil st' von Zähler und Nenner wird berechnet.
2. $\mathit{numBox} = \mathit{MListToBox}(st', cr, num) \qquad \mathit{denBox} = \mathit{MListToBox}(st', \mathit{true}, den)$
   Zähler und Nenner werden im Stil st' gesetzt. Der Nenner ist immer gestaucht, der Zähler nur, wenn es der ganze Bruch ist.
3. $th = \mathit{RuleThickness}(st)$
   Die Dicke *th* des Bruchstrichs ergibt sich aus demselben Stilparameter wie die Dicke des Strichs bei Über- und Unterstreichungen.
4. $l = \mathit{max}(\mathit{width}(\mathit{numBox}), \mathit{width}(\mathit{denBox}))$
   Die Länge *l* des Bruchstrichs entspricht dem Maximum der Breiten des Zählers und des Nenners.
5. $\mathit{stroke} = \mathit{rule}'(th/2, th/2, l)$
   Der Bruchstrich hat die Dicke *th* und die Länge *l*. Er wird so erzeugt, daß sein Referenzpunkt in der Mitte des linken Randes liegt.
6. Die Abstände *distNum* zwischen Zähler und Bruchstrich sowie *distDen* zwischen Bruchstrich und Nenner werden berechnet (siehe unten).

7. $numSpace = vSpace(distNum)$ $denSpace = vSpace(distDen)$
   Entsprechende Zwischenräume werden erzeugt.

8. $numBox' = rebox(l, numBox)$ $denBox' = rebox(l, denBox)$
   Zähler und Nenner werden in dem zur Verfügung stehenden Raum der Breite l zentriert. Dies geschieht in TEX durch Anhängen von unendlich dehnbarem Kleber auf beiden Seiten des Inhalts von Zähler und Nenner, gefolgt von einem Vorgang, der dem Randausgleich beim Zeilenumbruch entspricht. Dieses komplizierte Vorgehen wurde gewählt, um es geübten TEX-Programmierern zu ermöglichen, durch Einfügen expliziten Klebers in Zähler oder Nenner Brüche mit ungewöhnlichem Aussehen zu konstruieren wie z. B. $\frac{1}{n+1}$. Wir wollen hier auf diese Vorgänge nicht näher eingehen, d. h. keine Definition der Funktion *rebox* angeben.

9. $fracBox = vBox[numBox', numSpace, \underline{stroke}, denSpace, denBox']$
   Nun wird der Bruch von oben nach unten aus dem Zähler $numBox'$, dem Abstand $numSpace$ zum Bruchstrich, dem Bruchstrich *stroke*, dem Abstand $denSpace$ zum Nenner und dem Nenner $denBox'$ vertikal zusammengesetzt. Der Referenzpunkt von *fracBox* ist der des Bruchstrichs.

10. Damit würde *fracBox* so positioniert, daß der Bruchstrich auf der übergeordneten Grundlinie liegen würde. Er sollte jedoch auf der Achse liegen. Daher brauchen wir noch eine Verschiebung nach oben um den Betrag

    $$axh = AxisHeight(st).$$

11. Als optisch kaum wahrnehmbare Subtilität werden zum Schluß auf beiden Seiten Leerräume

    $$delSpace = hSpace(nullDelimiterSpace)$$

    hinzugefügt, wobei *nullDelimiterSpace* eine stilunabhängige Konstante mit einem Wert von üblicherweise 1,2 pt ist.

12. Daher ist das Endergebnis die horizontale Liste

    $$[delSpace, -axh \triangleright fracBox, delSpace].$$

    Die Verschiebung um *axh* ist negativ, da sie nach oben gerichtet sein soll.

Berechnung der Abstände in einem Bruch

Nun müssen wir nur noch erläutern, wie die Abstände *distNum* und *distDen* über und unter dem Bruchstrich berechnet werden. Sie hängen von der Dicke *th* des Bruchstrichs und der Achsenhöhe *axh* ab, die wir oben in Schritt 3 und 10 eingeführt haben. Dazu kommen als weitere Zutaten die Tiefe des Zählers und die Höhe des Nenners sowie einige Stilparameter. Die Berechnung ist in Abbildung 6.5 veranschaulicht.

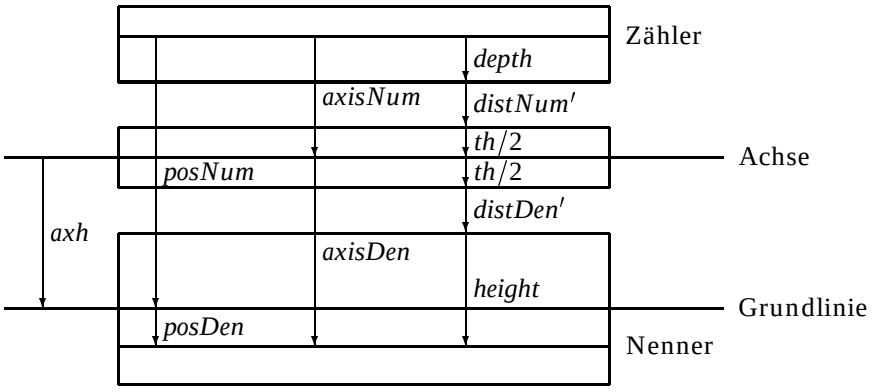

Abbildung 6.5: Abstände in einem Bruch

Zunächst werden aus einigen Stilparametern die gewünschten Abstände *posNum* und *posDen* von der Grundlinie des Zählers zur übergeordneten Grundlinie und von dort zur Grundlinie des Nenners berechnet.

$$posNum = \begin{cases} num1(st) & \text{falls } st = D \\ num2(st) & \text{sonst} \end{cases}$$

$$posDen = \begin{cases} denom1(st) & \text{falls } st = D \\ denom2(st) & \text{sonst} \end{cases}$$

Interessanter ist jedoch der Abstand zur Achse. Unter Benutzung von *axh* werden die Abstände *axisNum* und *axisDen* von der Grundlinie des Zählers zur Achse und von dort zur Grundlinie des Nenners berechnet.

$$axisNum = posNum - axh \qquad axisDen = posDen + axh$$

Daraus lassen sich Rohwerte für den Abstand $distNum'$ zwischen Unterrand des Zählers und Oberrand des Bruchstrichs und für den Abstand $distDen'$ zwischen Unterrand des Bruchstrichs und Oberrand des Nenners bestimmen.

$$\begin{aligned} distNum' &= axisNum - th/2 - depth(numBox) \\ distDen' &= axisDen - th/2 - height(denBox) \end{aligned}$$

Da diese Abstände von *posNum* und *posDen* ausgehend berechnet worden sind, könnte es sein, daß sie zu klein oder sogar negativ sind, was bedeuten würde, daß sich Teile des Bruchs überlagern. Zur Abhilfe gibt es einen stilabhängigen Minimalwert für die Abstände vom Bruchstrich.

$$minDist = \begin{cases} 3th & \text{falls } st = D \\ th & \text{sonst} \end{cases}$$

Wenn die berechneten Abstände zu klein sind, werden sie auf den Minimalwert gesetzt.

$$distNum = max(distNum', minDist) \qquad distDen = max(distDen', minDist)$$

Dies sind die Abstände, die beim Aufbau des Bruchs tatsächlich benutzt werden.

Indizes

Betrachten wir nun die Übersetzung von mathematischen Elementen mit oberen und unteren Indizes.

Die Berechnung von

$ElemToHList(st, cr, \mathbf{Sup}(nuc, sup))$	mit oberem Index *sup*,
$ElemToHList(st, cr, \mathbf{Sub}(nuc, sub))$	mit unterem Index *sub* und
$ElemToHList(st, cr, \mathbf{SupSub}(nuc, sup, sub))$	mit beiden Indizes,

wobei der Kern jedesmal *nuc* ist, wird in den folgenden Schritten durchgeführt:

1a. Falls der Kern ein einzelnes Symbol *s* ist, d. h. $nuc = \mathbf{Sym}(s)$, dann setze

$$nucBox = boxChar(ch) \qquad itCorr = charItalic(ch),$$

wobei $ch = setSym(st, s)$ ist.

Das Symbol *s* wird also durch *setSym* stilabhängig in das Ausgabezeichen *ch* umgewandelt. Das Zeichen wird durch *boxChar* in einen Kasten gesteckt, der *nucBox* genannt wird. Später wird der Überhang *itCorr* des Zeichens *ch* gebraucht.

1b. Falls der Kern kein einzelnes Symbol ist, setze

$$nucBox = ElemToHList(st, cr, nuc) \qquad itCorr = 0.$$

Der Kern wird zum Zeichenkasten *nucBox*. In diesem Fall wird der Überhang *itCorr* immer auf 0 gesetzt.

2. $st' = script(st)$

Der Stil für die Indizes wird bestimmt.

3. Bei **Sup** und **SupSub**: $supBox = MListToBox(st, cr, sup)$
Bei **Sub** und **SupSub**: $subBox = MListToBox(st, true, sub)$

Die Kästen für die Indizes werden berechnet. Beachten Sie, daß untere Indizes immer gestaucht sind, während obere Indizes die Gestauchtheit von ihrem Kontext erben.

4. Zum Zusammenstellen des Endergebnisses werden einige Abstände gebraucht.

Bei **Sup**:	*shiftSup*	(siehe unten)
Bei **Sub**:	*shiftSub*	(hier nicht behandelt)
Bei **SupSub**:	*supDist* und *subDist*	(siehe unten)

5. Nun wird eine horizontale Liste *hl* zusammengestellt.

   Bei **Sup**: $[nucBox,\ hSpace(itCorr),\ -shiftSup \triangleright supBox]$
   Bei **Sub**: $[nucBox,\ shiftSub \triangleright subBox]$
   Bei **SupSub**: $[nucBox,\ vBox(vl)]$
   wobei $vl = [itCorr \triangleright supBox,\ \underline{vSpace'(supDist, subDist)},\ subBox]$

   In allen drei Fällen besteht *hl* im wesentlichen aus einem Kasten für den Kern (*nucBox*), gefolgt von einem Kasten für den Index oder die Indizes. In den beiden Fällen mit einem einzelnen Index ist das einfach der um einen gewissen Betrag nach oben bzw. unten verschobene Indexkasten. Bei **SupSub** dagegen besteht der Kasten für die Indizes aus den beiden Indexkästen übereinandergestellt mit einem geeigneten Leerraum dazwischen.

   Beachten Sie die unterschiedliche Behandlung des Überhangs *itCorr*: Bei **Sub** wird er ignoriert; der Index wird unmittelbar an den Kern gehängt, um zu vermeiden, daß er zu weit wegsteht. Bei **Sup** wird dagegen der Kern um den Überhang erweitert, um zu verhindern, daß der Index sich mit dem Kern überlappt. Konsequenterweise wird bei **SupSub** der obere Index um den Betrag *itCorr* nach rechts geschoben, aber nicht der untere (f_1^1).

6. In allen drei Fällen wird zum Schluß die Liste *hl* am rechten Ende noch um einen Leerraum *hSpace*(*scriptSpace*) erweitert. Dabei ist *scriptSpace* eine stilunabhängige Konstante, üblicherweise 0,5 pt.

Die Positionierung der Indizes

Nun wollen wir für den Fall, daß ein oberer und ein unterer Index zusammen auftreten, die Abstände *supDist* und *subDist* vom Unterrand des oberen Index zur Grundlinie bzw. von der Grundlinie zum Oberrand des unteren Index berechnen. Als Zwischenergebnis fällt dabei auch die bei **Sup** benötigte Distanz *shiftSup* an.

1. Zunächst wird der gewünschte Abstand *supPos* von der Grundlinie des oberen Index zur übergeordneten Grundlinie berechnet. Er ergibt sich als Maximum von drei Beiträgen, die sich aus einigen Stilparametern herleiten.

$$supPos1 = \begin{cases} 0 & \text{falls } isChr(nucBox) \\ height(nucBox) + supDrop(st') & \text{sonst} \end{cases}$$

$$supPos2 = \begin{cases} sup1(st) & \text{falls } st = D \text{ und } cr = false \\ sup2(st) & \text{falls } st \neq D \text{ und } cr = false \\ sup3(st) & \text{falls } cr = true \end{cases}$$

$$supPos3 = depth(supBox) + \tfrac{1}{4}\,xHeight(st)$$

$$supPos = max(supPos1, supPos2, supPos3)$$

Das bei *supPos1* benutzte Prädikat *isChr* ist auf Kästen wie folgt definiert:

$$isChr\,(\mathbf{Box}\,(h,d,w,c)) \;=\; \begin{cases} true & \text{falls } c = \mathbf{Chr}\,(ch) \\ isChr\,(b) & \text{falls } c = \mathbf{HList}\,[0 \triangleright b] \\ false & \text{sonst} \end{cases}$$

Beachten Sie, daß bei *supPos1* der Indexstil st' benutzt wird, während ansonsten immer der Kontextstil *st* vorkommt. In die Berechnung von *supPos2* geht die Gestauchtheit ein. Der Anteil *supPos3* sorgt dafür, daß der Abstand vom Unterrand des oberen Index zur Grundlinie mindestens ein Viertel der *x*-Höhe beträgt.

Der hier berechnete Wert *supPos* ist identisch mit dem Wert *shiftSup*, der beim Setzen von **Sup**-Formeln benötigt wird.

2. Dann wird der gewünschte Abstand *subPos* von der übergeordneten Grundlinie zur Grundlinie des unteren Index berechnet. Das geht so ähnlich wie bei *supPos*, ist aber etwas einfacher.

$$\begin{aligned} subPos1 &= \begin{cases} 0 & \text{falls } isChr\,(nucBox) \\ depth\,(nucBox) + subDrop\,(st') & \text{sonst} \end{cases} \\ subPos2 &= sub2\,(st) \\ subPos &= max\,(subPos1, subPos2) \end{aligned}$$

Der hier berechnete Wert *subPos* ist *nicht* identisch mit *shiftSub*. Die Berechnung von *shiftSub* wird hier nicht geschildert.

3. Nun werden vorläufige Werte $supDist'$ und $subDist'$ für die Abstände vom Unterrand des oberen Index zur Grundlinie bzw. von dort zum Oberrand des unteren Index berechnet. Die Summe $dist'$ dieser beiden Abstände ist also ein Rohwert für die Höhe des leeren Raumes zwischen den beiden Indizes.

$$\begin{aligned} supDist' &= supPos - depth\,(supBox) \\ subDist' &= subPos - height\,(subBox) \\ dist' &= supDist' + subDist' \end{aligned}$$

4. Ähnlich wie bei Brüchen kann es vorkommen, daß diese Abstände zu klein sind. Zur Abhilfe gibt es einen Minimalwert *minDist* für die Größe des Leerraums zwischen den Indizes und einen Minimalwert *minSup* für die Höhe des Unterrands des oberen Index über der Grundlinie. Für diese beiden Minimalwerte gibt es keine eigenen Stilparameter; es werden statt dessen Parameter anderer Art herangezogen und mit Vorfaktoren versehen.

$$\begin{aligned} minSup &= \tfrac{4}{5}\,xHeight\,(st) \\ minDist &= 4 \cdot RuleThickness\,(st) \end{aligned}$$

5. Wenn $supDist'$ und $dist'$ zu klein geraten sind, dann werden sie auf den erforderlichen Minimalwert erhöht.

$$\begin{aligned} supDist &= max(supDist', minSup) \\ dist &= max(dist', minDist) \end{aligned}$$

Der endgültige Wert von *subDist* ergibt sich als Differenz dieser beiden Werte:

$$subDist = dist - supDist$$

Die Korrektur von *dist* erfolgt also effektiv dadurch, daß der untere Index tiefer gesetzt wird, während die Lage *supDist* des oberen Index unverändert bleibt. Die Korrektur von *supDist* erfolgt, indem der obere und der untere Index um den gleichen Betrag höher gesetzt werden. Dadurch ändert sich ihr Abstand *dist* nicht. Die beiden Korrekturen sind also unabhängig voneinander und erzielen das gewünschte Ergebnis.

Damit ist die Definition der Übersetzung *ElemToHList* von mathematischen Elementen in horizontale Listen abgeschlossen.

Klassen

Es fehlt noch die Übersetzung von mathematischen Listen. Zwischen je zwei Elemente einer Liste wird ein Zwischenraum eingefügt, der von der Klasse der beiden Elemente und vom Stil abhängt. Wir wollen jetzt diese Klassen und ihre Berechnung definieren.

Die Menge *Class* der Klassen besteht aus den neun Elementen **Ord**, **Op**, **Bin**, **Rel**, **Open**, **Close**, **Punct**, **Inner** und **None**. Die ersten acht Klassen wurden bereits in Abschnitt 6.3 vorgestellt. Die neunte Klasse **None** wird von uns für mathematische Elemente benutzt, die im TEX-Buch gar keine Klasse haben.

Jedes mathematische Symbol besitzt in TEX eine Klasse. Die Klasse von „+“ z. B. ist **Bin** und die von „=“ ist **Rel**. Wir nehmen an, daß die Klasse eines Symbols durch die Funktion $symClass : Symbol \to Class$ gegeben ist.

Die Klasse eines mathematischen Elements kann mit Hilfe der Funktion $mClass : M \to Class$ berechnet werden. Sie ist wie folgt definiert:

$$\begin{aligned} mClass\,(\mathbf{Sym}\,(s)) &= symClass\,(s) \\ mClass\,(\mathbf{MathSpace}\,(w)) &= \mathbf{None} \\ mClass\,(\mathbf{Over}\,(ml)) &= \mathbf{Ord} \\ mClass\,(\mathbf{Under}\,(ml)) &= \mathbf{Ord} \\ mClass\,(\mathbf{Frac}\,(num, den)) &= \mathbf{Inner} \\ mClass\,(\mathbf{Sup}\,(nuc, sup)) &= nucClass\,(nuc) \\ mClass\,(\mathbf{Sub}\,(nuc, sub)) &= nucClass\,(nuc) \\ mClass\,(\mathbf{SupSub}\,(nuc, sup, sub)) &= nucClass\,(nuc) \end{aligned}$$

Dabei ist $nucClass[\mathbf{Sym}(s)] = symClass(s)$ und $nucClass(ml) = \mathbf{Ord}$ für alle anderen mathematischen Listen *ml*.

Die den einzelnen Fällen zugewiesenen Klassen entsprechen den Berechnungen im TEX-Buch. Dort wird z. B. ein **Over**-Atom während des Formelsatzes in ein **Ord**-Atom umgewandelt, und ein Bruch, der gar kein Atom ist, wird zu einem **Inner**-Atom gemacht.

Als Klasse einer mathematischen Liste nehmen wir die Klasse des ersten Elements, dessen Klasse von **None** verschieden ist, bzw. **None**, wenn es kein solches Element gibt. Für die Funktion $mlClass : MList \to Class$ gilt also:

Die leere Liste hat die Klasse **None**: $mlClass[\,] = \mathbf{None}$

Nichtleere Listen werden so behandelt:

$$mlClass[m_1, \ldots, m_n] = \begin{cases} mClass(m_1) & \text{falls } mClass(m_1) \neq \mathbf{None} \\ mlClass[m_2, \ldots, m_n] & \text{sonst} \end{cases}$$

Der zwischen zwei mathematische Elemente einzufügende Zwischenraum hängt von deren Klassen cl_1 und cl_2 sowie vom Stil *st* ab. Er wird durch die Funktion *space* gegeben. Die Berechnung von $space(st, cl_1, cl_2)$ erfolgt in zwei Schritten:

1. Der Zwischenraum wird in mathematischen Einheiten gemäß Tabelle 6.3 ermittelt. Er ist stilabhängig; die eingeklammerten Zahlen gelten nur für $st = D$ und $st = T$; für *S* und *SS* ist 0 anzunehmen. Wenn eine der beiden Klassen cl_1 und cl_2 **None** ist, ist der Zwischenraum ebenfalls 0.
2. Der Zwischenraum wird von mathematischen Einheiten in absolute umgerechnet. Das erfolgt wie beim Umwandeln von **MathSpace**-Formeln in Leerraumkästen durch Multiplikation mit $Quad(st)/18$, ist also wieder stilabhängig.

Die Übersetzung mathematischer Listen

Nun fehlt nur noch die Definition der Funktion

$$MListToHList : Style \times Bool \times MList \to HList,$$

die mathematische Listen aus *MList* in horizontale Listen verwandelt. Die Grundidee für *MListToHList* ist einfach: wir wandern von links nach rechts durch die mathematische Liste, übersetzen ihre Elemente in horizontale Listen und hängen alle diese Listen zu einer einzigen zusammen. Dieses Vorgehen wird durch zwei Dinge erschwert.

Manche **Bin**-Elemente werden in **Ord**-Elemente verwandelt. Die Umwandlung hängt von den Klassen der beiden Nachbarelemente ab. Bei der Ermittlung dieser Klassen werden **None**-Elemente (horizontale Zwischenräume) übersprungen.

Zwischen je zwei „echte“ Elemente wird ein Zwischenraum eingefügt, der von ihrer Klasse und vom Stil abhängt. Dabei ist ein echtes Element eines, das nicht die Klasse **None** hat. Also werden auch hier **None**-Elemente übersprungen.

Wenn wir beim Links-Rechts-Durchlauf durch die mathematische Liste ein bestimmtes Element bearbeiten, müssen wir also wissen, welche Klasse der nächste linke Nachbar hatte, der kein **None**-Element war. Die Definition von *MListToHList* wird daher auf eine Funktion *doMList* zurückgeführt, die als zusätzliches Argument diese Information hat:

$$doMList : Style \times Bool \times Class \times MList \to HList$$

Wenn wir mit der Bearbeitung einer mathematischen Liste beginnen, gibt es noch keinen linken Nachbarn. Das wird dadurch angezeigt, daß das Zusatzargument von *doMList* **None** ist. Daher definieren wir:

$$MListToHList\,(st, cr, ml) \;=\; doMList\,(st, cr, \mathbf{None}, ml).$$

Die Funktion *doMList* macht aus einer leeren mathematischen Liste eine leere Liste von Kästen:

$$doMList\,(st, cr, leftClass, [\,]) \;=\; [\,]$$

Die Berechnung von $doMList\,(st, cr, leftClass, [m_1, \ldots, m_n])$ mit $n \neq 0$ (nichtleere Liste) erfolgt in mehreren Schritten:

1. $rest \;=\; [m_2, \ldots, m_n]$

   Das ist die Restliste, die noch zu bearbeiten ist, wenn wir mit m_1 fertig sind.

2. $ownClass \;=\; mClass\,(m_1)$ $\qquad\qquad$ $rightClass \;=\; mlClass\,(rest)$

   Die Klasse des Elements m_1, das gerade bearbeitet wird, wird *ownClass* genannt. Die Klasse des linken und rechten „echten“ Nachbarn (beim Überspringen von **None**-Elementen) ist durch das Aufruf-Argument *leftClass* gegeben bzw. wird jetzt berechnet und *rightClass* genannt. Wenn es keinen echten Nachbarn auf einer Seite gibt, ist die betreffende Klasse **None**.

3. $hl \;=\; ElemToHList\,(st, cr, m_1)$

   Das aktuelle Element m_1 wird in die horizontale Liste *hl* übersetzt.

4. Falls $ownClass = \mathbf{Bin}$ und $leftClass \in \{\mathbf{None}, \mathbf{Bin}, \mathbf{Op}, \mathbf{Rel}, \mathbf{Open}, \mathbf{Punct}\}$ oder $rightClass \in \{\mathbf{None}, \mathbf{Rel}, \mathbf{Close}, \mathbf{Punct}\}$, dann setze $ownClass' = \mathbf{Ord}$; sonst sei $ownClass' = ownClass$. Dies entspricht den beiden in Abschnitt 6.3 angegebenen Umwandlungsregeln.

5a. Falls $ownClass' = \mathbf{None}$, entsteht das Endergebnis durch Zusammenhängen von *hl* und $doMList\,(st, cr, leftClass, rest)$.

   Wenn das aktuelle Element ein **None**-Element ist, muß es bei der Nachbarbestimmung übersprungen werden. Daher ist die Klasse des linken Nachbarn des nächsten Elements immer noch *leftClass*.

5b. Falls $ownClass' \neq$ **None**:

(a) $spaceBox = space(st, leftClass, ownClass')$

Das ist der Kasten für den Zwischenraum, der vor dem aktuellen Element einzufügen ist.

(b) $boxList = doMList(st, cr, ownClass', rest)$

Die Restliste wird in eine Kastenliste übersetzt. Die Klasse des linken Nachbarn des nächsten Elements ist die eventuell modifizierte Klasse $ownClass'$ des aktuellen Elements.

(c) Das Endergebnis entsteht durch Zusammenhängen des Kastens *spaceBox* und der horizontalen Listen *hl* und *boxList* zu einer einzigen horizontalen Liste.

Im Unterschied zu dem hier vorgestellten Algorithmus benötigt TEX zwei Durchläufe durch die zu setzende Formel. Im ersten Durchlauf werden die Teilformeln gesetzt, während das Einfügen der impliziten Zwischenräume zwischen benachbarte Elemente erst im zweiten Durchlauf erfolgt. Der Grund dafür, daß wir hier mit einem Durchlauf ausgekommen sind, liegt daran, daß wir viele Arten von Teilformeln und viele zusätzliche Einzelheiten wie Kleber und Strafelemente nicht behandelt haben.

6.5 Funktionale Programmiersprachen

Der hier vorgestellte Algorithmus zum Formelsatz beruht im wesentlichen auf der Definition vieler Funktionen mit Hilfe anderer Funktionen. Diese Funktionen werden oft auf baumartige Strukturen wie die Formelinternform angewandt, wobei zwischen verschiedenen Fällen anhand von *Konstruktoren* wie **Sym** und **Sup** unterschieden wird.

Außerdem werden in den zum Teil sehr langen Berechnungsfolgen Namen für Zwischenergebnisse eingeführt. Diese Namensdefinitionen sind keine Wertzuweisungen im Sinne imperativer Programmiersprachen wie Pascal oder C, da der einmal an einen Namen gebundene Wert sich nachträglich nicht mehr ändert. Das liegt daran, daß wir auch in den Fällen, in denen eine Umdefinition nahegelegen hätte, lieber einen neuen Namen benutzt haben wie z. B. $numBox'$ oder $ownClass'$.

Aus den obengenannten Gründen kann der Algorithmus ziemlich direkt in einer *funktionalen Programmiersprache* implementiert werden. Diese Sprachen erlauben es, Funktionen auf baumartigen Strukturen mit Fallunterscheidung über Konstruktoren elegant zu definieren. Sie gestatten es, Namen an Zwischenergebnisse zu binden, verbieten aber das nachträgliche Umdefinieren dieser Namen durch Wertzuweisungen.

Im folgenden werden wir etwas mehr über funktionale Programmiersprachen sagen; zunächst über Datentypen und Konstruktoren, dann über die Funktionen selbst.

6.5.1 Induktiv definierte Mengen und Datentypen

In Abschnitt 6.4.1 wurde eine alternative Internform für Formeln definiert. Die Art der Definition könnte Ihnen aufgefallen sein. Es handelt sich um eine *induktive Definition* von Mengen. Diese Art der Definition soll, weil sie sehr wichtig ist, etwas näher erläutert werden.

In unserem Beispiel werden zwei Mengen zusammen definiert, die Menge M von mathematischen Elementen und die Menge *MList* von mathematischen Listen, im folgenden einfach Elemente und Listen genannt. Eine Liste ist demnach eine endliche Folge von Elementen, geschrieben als $[m_1, m_2, \ldots, m_n]$. Diese Definition benutzt den noch nicht definierten Begriff „Element“. Die Definition von Elementen zerfällt in zwei Teile. Im ersten Teil wird gesagt, daß für alle s aus der Menge *Symbol* der Ausdruck **Sym**(s) ein Element von M bezeichnet und daß **MathSpace**(n) für alle Zahlen n ein Element von M ist. Im zweiten Teil wird gesagt, wie man aus schon konstruierten Elementen und Listen neue Elemente bauen kann. **Over** und **Under** konstruieren aus einer Liste ein Element, **Frac** und **Sup** aus zwei Listen usw.

Jede induktive Definition zerfällt in diese zwei Teile. Der erste Teil gibt die *Basis* der Menge an. Das ist in unserem Beispiel $\{\textbf{Sym}(s) \mid s \in \mathit{Symbol}\} \cup \{\textbf{MathSpace}(n) \mid n \in \mathbf{R}\}$ für die Menge M. Für diesen Schritt setzt man noch nicht voraus, daß bereits Objekte der Art, die gerade definiert werden soll, existieren.

Der zweite Teil definiert den *Induktionsschritt*. In ihm wird angegeben, wie aus bereits konstruierten Objekten der zu definierenden Mengen neue Objekte konstruiert werden können.

Man kann natürlich leicht solche induktiven Definitionen hinschreiben. Informatiker (und Mathematiker) fragen sich jedoch, was sie bedeuten, d. h. welche Mengen da eigentlich definiert werden. Dazu kann man mehrere Antworten geben. Eine ist die folgende: die definierten Mengen bestehen aus den Elementen, die entweder in den Basismengen sind, oder die durch endlichmalige Anwendung des Induktionsschritts aus Elementen der Basismengen konstruiert werden können.

Ein weiteres Beispiel für induktiv definierte Mengen ist die Menge *Box* der Kastenterme zusammen mit der Menge *Content* der Kasteninhalte. Die Basismenge für *Content* besteht aus den Elementen **White**, **Black** und **Chr**(ch) für Zeichen *ch*. Im Induktionsschritt werden aus bereits konstruierten Kästen mittels **HList** und **VList** neue Kästen gebaut.

Jetzt ist noch zu klären, was induktive Definitionen mit funktionalen Programmiersprachen zu tun haben. Die Antwort ist, daß moderne funktionale Programmiersprachen es den Programmierern erlauben, nach Bedarf induktiv neue Datentypen zu definieren. Wenn Sie also, wie in diesem Kapitel, einen Formelsatzalgorithmus funktional programmieren möchten, so definieren Sie die dazu notwendigen Datentypen für mathematische Elemente, mathematische Listen, Kastenterme und Inhalte mehr oder weniger so, wie Sie es im vorigen Kapitel gesehen haben. Lediglich die Syntax der Programmiersprache muß beachtet werden.

In der funktionalen Programmiersprache Miranda[1] [Tur86] sähen diese Definitionen etwa folgendermaßen aus:

```
mList  ==  [mItem]
```

Durch diese Definition wird der Datentyp `mList` als Liste von Elementen des Datentyps `mItem` definiert, der unserem *M* entspricht. Die eckigen Klammern bedeuten „Liste von“.

```
mItem ::=
   Sym        symbol                |
   MathSpace  num                   |
   Over       mList                 |
   Under      mList                 |
   Frac       mList  mList          |
   Sup        mList  mList          |
   Sub        mList         mList   |
   SupSub     mList  mList  mList
```

Hierdurch wird der Datentyp `mItem` der mathematischen Elemente über verschiedene Fälle definiert. Die einzelnen Fälle werden durch das Zeichen | getrennt. Zu jedem Fall gehört ein sogenannter *Konstruktor* wie z. B. `Sym` und `MathSpace`. Diese Konstruktoren sind Namen, die die Zugehörigkeit zu den einzelnen Fällen angeben. Sie werden in Miranda groß geschrieben im Gegensatz zu den Typ- und Variablennamen, die klein geschrieben werden. In unserer Beschreibung des Formelsatzes hatten wir die Konstruktoren immer fett gesetzt. Beachten Sie auch, daß in Miranda die Argumente der Konstruktoren einfach hinter diese gesetzt werden ohne Klammern oder Kommata, z. B. `Frac num den`. Wir hatten dagegen **Frac**(num, den) geschrieben.

In der funktionalen Sprache Standard ML, kurz SML, [Pau91] sähen die Definitionen ein bißchen anders aus.

```
datatype  mItem  =
   Sym       of  symbol                |
   MathSpace of  num                   |
   Over      of  mList                 |
   Under     of  mList                 |
   Frac      of  mList * mList         |
   Sup       of  mList * mList         |
   Sub       of  mList         * mList |
   SupSub    of  mList * mList * mList
withtype  mList  =  mItem list
```

1 Miranda ist ein eingetragenes Warenzeichen der Research Ltd. in Canterbury, GB.

Die Listenbildung wird hier statt durch die eckigen Klammern durch das nachgestellte Wort `list` angedeutet. Die einzelnen Fälle werden wie in Miranda durch das Zeichen `|` getrennt, werden aber durch `of` und `*` stärker gegliedert. In SML dürfen die Konstruktoren beliebig geschrieben werden, groß oder klein. Bei der Benutzung von Konstruktoren werden die Argumente mit Klammern und Kommata geschrieben, wie wir es auch im Text getan haben, also z. B. `Frac (num, den)`.

Zum Schluß zeigen wir noch die Definition von Kästen, Kasteninhalten, horizontalen und vertikalen Listen, wie sie in Miranda aussehen würde.

```
box      ::=  Box dim dim dim content
content  ::=  White  |  Black  |  Chr char
           |  HList hList  |  VList vList
hList  ==  [shiftedBox]
vList  ==  [shiftedBox]
shiftedBox  ::=  Shift dim box
```

Dabei muß zwischen dem großgeschriebenen Konstruktor `HList` (von uns früher **HList** geschrieben) und dem kleingeschriebenen Typnamen `hList` (früher *HList*) unterschieden werden. Der Konstruktor `Shift` entspricht unserem Zeichen „▷“; also steht `Shift 0 b` für $0 \triangleright b$. Die Konvention, daß eine Verschiebung von 0 nicht extra hingeschrieben werden muß, ist in Miranda (und in SML ebenso) leider nicht anwendbar.

6.5.2 Funktionen

Funktionale Programmiersprachen heißen so, weil ihr Hauptkonzept Funktionen sind, und zwar Funktionen im mathematischen Sinne. Sie liefern, anders als sogenannte Funktionen in imperativen Programmiersprachen wie C, bei wiederholter Anwendung auf die gleichen Argumente und „unter sonstigen gleichen Bedingungen“ immer das gleiche Resultat.

Es gibt eine gewisse Menge eingebauter Funktionen, z. B. die notwendigen arithmetischen Funktionen, und einen Mechanismus, nach Bedarf neue Funktionen zu definieren. Uns interessieren hier Funktionen auf benutzerdefinierten Datentypen, wie wir sie oben kennengelernt haben.

Betrachten wir die Funktionen *ElemToHList*, *MListToHList* und *MListToBox* aus der Beschreibung des Formelsatzes. Sie berechnen den Satz von mathematischen Elementen bzw. mathematischen Listen. Diese Funktionen sind *mutuell rekursiv*, d. h. sie greifen zur Berechnung auf sich selbst bzw. auf sich gegenseitig zurück. Dabei erkennen wir die gleiche Struktur wieder, die wir bei der simultan induktiven Definition der beiden Mengen M und $MList$ bzw. der beiden Datentypen `mItem` und `mList` kennengelernt haben. Wenn *ElemToHList* z. B. eine überstrichene Formel, dargestellt als **Over**(ml), setzen soll, so wendet sie dazu *MListToBox* auf *ml* an. Die Basisfälle können ohne Rekursion erledigt werden.

Betrachten wir jetzt einen Ausschnitt der Definition von `elemToHList` in Miranda. (Wir müssen den Funktionsnamen jetzt klein schreiben, da in Miranda nur Konstruktoren groß geschrieben werden dürfen.)

```
elemToHList  ::  style -> bool -> mItem -> hList
...
elemToHList st cr (MathSpace mw)  =  setSpace st mw
elemToHList st cr (Over  ml)      =  setOver  st cr ml
elemToHList st cr (Under ml)      =  setUnder st cr ml
...
```

Die erste Zeile deklariert den *Typ* der Funktion `elemToHList`: sie hat als erstes Argument einen Stil `style`, als zweites einen Wahrheitswert `bool`, der die Gestauchtheit angibt, und als drittes ein mathematisches Element `mItem`. Das Ergebnis der Funktion ist eine horizontale Liste `hList`. Diese Typdeklaration kann im Programm stehen, muß es aber nicht. Wenn sie fehlt, wird der Typ der Funktion vom Miranda-System aus der Funktionsdefinition erschlossen. Es empfiehlt sich aber, die Deklaration ins Programm aufzunehmen. Einerseits dient sie Lesern des Programms als zusätzliche Dokumentation, und andererseits zwingt sie die Programmierer, sich Gedanken über den Funktionstyp zu machen und ihn formal aufzuschreiben.

Wir haben oben drei Fälle der Funktionsdefinition gezeigt: die Behandlung von Zwischenräumen, von Über- und von Unterstreichungen. Im Gegensatz zu unserer Darstellung im Abschnitt 6.4.3 wird die Erzeugung des Ergebnisses jeweils einer Hilfsfunktion überlassen, um das Mirandaprogramm übersichtlicher zu machen. Alle Fälle der Definition einer Funktion müssen nämlich unmittelbar aufeinander folgen; es ist nicht möglich, zwischendurch Hilfsfunktionen zu definieren, wie wir das in unserer Darstellung des Formelsatzes getan haben.

```
setOver  ::  style -> bool -> mList -> hList
setOver st cr ml  =  [Shift 0 b']
   where b  =  Shift 0 (mListToBox st True ml)
         r  =  rule th (width b)
         th =  ruleThickness st
         b' =  vBox [vSpace th, r, vSpace (3 * th)] b []
```

Dieses Programmstück entspricht bis auf die Art der Darstellung im wesentlichen dem, was wir im Formelsatzabschnitt vorgestellt haben. Statt $0 \triangleright b'$ muß hier `Shift 0 b'` geschrieben werden; die 0-Verschiebung kann nicht weggelassen werden. Man kann aber Funktionen wie `rule` und `vSpace` gleich so schreiben, daß sie statt eines Kastens einen 0-verschobenen Kasten liefern; das haben wir in dem Beispiel oben angenommen.

Die Funktion `vBox` ist etwas komplizierter als ihr informales Gegenstück *vBox*. Statt einer Liste mit einem ausgezeichneten Element erhält sie nämlich drei Argumente: die Liste der Elemente vor dem ausgezeichneten Element, das ausgezeichnete Element selbst und die Liste der Elemente danach. Der Typ von `vBox` ist also

```
vBox  ::  vList -> shiftedBox -> vList -> box.
```

Das `setOver`-Beispiel zeigt ein weiteres Konzept funktionaler Sprachen: *lokale Namen* können als Abkürzungen für Teilausdrücke eingeführt werden. Im Beispiel sind das `b`, `r`, `th` und `b'`. Lokale Namen sind, wie der Name sagt, lokal zu einer Funktionsdefinition und deshalb außerhalb der Definition unbekannt. Sie sollten paarweise verschieden sein, um Verwirrung zu vermeiden. (In Miranda gibt es eine Fehlermeldung, wenn ein lokaler Name zweimal in derselben funktionsdefinierenden Regel benutzt wird.) Ohne die Programmbedeutung zu verändern, können zusätzliche lokale Namen eingeführt werden:

```
setOver st cr ml  =  [Shift 0 b']
   where b  =  Shift 0 (mListToBox st True ml)
         r  =  rule th (width b)
         th =  ruleThickness st
         space1  =  vSpace th
         space2  =  vSpace (3 * th)
         b' =  vBox [space1, r, space2] b []
```

Die Reihenfolge der lokalen Definitionen ist dabei in Miranda beliebig. (In SML hingegen müssen Namen definiert sein, bevor sie zum ersten Mal benutzt werden; das ist in unserem Beispiel nicht der Fall bei `th`.)

Andererseits können bestehende lokale Namen durch den Ausdruck, den sie abkürzen, ersetzt werden, ohne daß sich die Programmbedeutung ändert. Letztlich kann man so alle lokale Namen beseitigen:

```
setOver st cr ml  =
   [Shift 0 (vBox [ vSpace (ruleThickness st),
                    rule (ruleThickness st) (width b),
                    vSpace (3 * ruleThickness st)]
                  (Shift 0 (mListToBox st True ml))  [])]
```

Wozu gibt es dann überhaupt lokale Namen, wenn man auch ohne sie auskommen kann? Ihre Benutzung kann das Programm übersichtlicher und kürzer machen, insbesondere wenn der Ausdruck, der abgekürzt wird, mehrfach vorkommt, wie im Beispiel `ruleThickness st`. In diesem Fall wird auch die Ausführung des Programms im allgemeinen schneller, da der abgekürzte Teilausdruck nur einmal ausgewertet wird.

Die Möglichkeit, in einer funktionalen Sprache lokale Namen einzuführen, sollte nicht mit den Wertzuweisungen imperativer Sprachen wie C verwechselt werden. Wie bereits gesagt, sind Doppeldefinitionen wie `x = 1; x = 2` ein Fehler. Definitionen wie `x = x + 1` sind noch übler; sie werden als rekursiv aufgefaßt (das `x`, das gleich `x + 1` ist) und liefern im günstigsten Fall eine Fehlermeldung; in weniger günstigen Fällen entsteht aber eine nichtterminierende Berechnung.