Editor zur handschriftlichen Eingabe zweidimensionaler Formeln in den Computer

Dirk Hopf

10. Dezember 1996

Diplomarbeit nach einem Thema von Prof. Dr. Günter Hotz
in der Technischen Fakultät, Fachbereich 14 – Informatik,
der Universität des Saarlandes, Saarbrücken

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, 10. Dezember 1996

Inhaltsverzeichnis

Einleitung

Computer werden für immer vielfältigere Aufgaben genutzt und können durch Algebrasysteme sogar symbolisch rechnen. Diese Rechnungen werden in der Sprache der Mathematik ausgedrückt, wobei die traditionelle Schreibweise die Zweidimensionalität des Papiers ausnutzt. Heutige Computeralgebrasysteme (CAS) verlangen jedoch die Eingabe wie früher die Fernschreiber in linearer Form. Dagegen benutzen sie schon längere Zeit die zweidimensionale Schreibweise, um die Lösung auf dem Bildschirm auszugeben. Dieser Gegensatz führt doch unweigerlich zu der Frage, warum nicht auch die Eingabe zweidimensional erfolgt. Der Fortschritt in der Entwicklung der Computer führte durch die immer größer werdende Integrationsdichte zu immer kleineren Computern, so daß tragbare leistungsfähige Computer möglich wurden. Für dieses neue Produkt wurde auch gleichzeitig der „Stift“ als neues Eingabemedium erprobt. Mittels Stift erfolgt die Eingabe direkt auf die Oberfläche des Displays, was dem zweidimensionalen Schreiben auf Papier entspricht. Die Eingabe der Schrift ist also nicht mehr das Problem, jedoch die schlechten Erkennungsraten der zur Zeit vorhandenen Schrifterkennungssoftware (siehe [Fischer]) zeigen, daß die Erkennung der Zeichen sich als keine leichte Aufgabe für den Computer ergibt. Sind die handschriftlichen Zeichen erkannt, stellt sich bei dieser neuen Eingabeform immer noch die Frage, was die zweidimensional angeordneten Einzelzeichen bedeuten. Hier setzt das OFR[1]-Projekt an.

0.1 Das OFR-Projekt

Es gibt bereits sehr gute Computeralgebrasysteme (CAS), wie z.B. Maple, Reduce, Mathematica usw., mit denen kann man Gleichungen lösen, Formeln vereinfachen, Funktionen integrieren, differenzieren, berechnen, zeichnen und viele Dinge mehr. Aber allen gleich ist die linearisierte Eingabe der Formeln mit Hilfe der Tastatur. Obwohl mathematische Formeln traditionell aus zweidimensionalen Gebilden bestehen, wie z.B. Bruch, Potenz, Wurzel, Integral, Summe, Produkt, Matrix, Limes usw. erfolgt die Eingabe trotzdem linear.

[1] Optische Formel Erkennung bzw. Optical Formula Recognition

Beispiel:

$$\int_0^{e^x} \frac{t}{\sqrt{t^2+\log t}}dt \quad \stackrel{\text{Maple}}{\Longrightarrow} \quad \texttt{int(t/(sqrt(t\^{}2+log(t))),t=0..exp(x))}$$

Aufgabe des OFR-Projektes ist deshalb die Entwicklung einer natürlichen benutzerfreundlichen Oberfläche in der handschriftlich Zeichen eingegeben werden können (Abbildung 0.1). Nach der Eingabe werden diese handschrift-

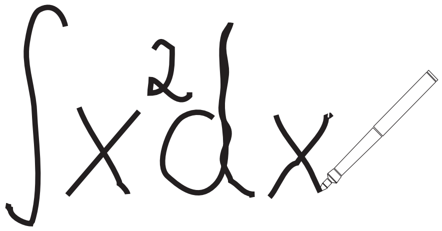


Abbildung 0.1: Handschriftliche Eingabe der Formel mit dem Stift auf den Bildschirm, wobei das Ergebnis direkt auf dem Display sichtbar ist.

lichen Zeichen zuerst als Einzelzeichen erkannt und durch die entsprechenden Zeichen des Computers dargestellt. Sind alle Zeichen einer Formel eingegeben und erkannt, muß anhand der Lage, Größe und Bedeutung der Zeichen die Bedeutung der Formel berechnet und in formatierter zweidimensionaler Schreibweise ausgegeben werden (Abbildung 0.2). Die erkannte Formel kann

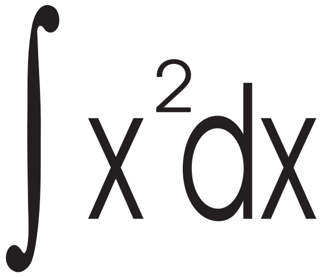


Abbildung 0.2: Erkennen der geschriebenen Zeichen als Formel, d.h. Erzeugung der linearen Formel intern, z.B. in Maple-Format `int(x^2,x)`.

weiter verändert oder zur Berechnung der Lösung einem angeschlossenen Algebrasystem übergeben werden. Die Übergabe der Formel und die Entgegennahme der Lösung erfolgt unsichtbar für den Benutzer in linearer Form. Die lineare Lösung wird für den Benutzer in die entsprechende zweidimensionale Schreibweise umgewandelt und auf dem Bildschirm angezeigt (Abbildung

$$\frac{1}{3}\,x^{3}$$

Abbildung 0.3: Weitergabe der Linearisierung an ein CAS und Erzeugung des Resultats in gewohnter zweidimensionaler Schreibweise.

0.3). Die entstandene Formel kann entweder durch Hinzufügen und Löschen von Zeichen oder durch Berechnungsvorschriften für das CAS verändert werden. Integriert in das Projekt, beschreibt die vorliegende Diplomarbeit den Entwurf und die Implementierung einer Oberfläche zur Eingabe und Verarbeitung der handschriftlichen Zeichen zu einer Formel.

0.1.1 Entstehungsgeschichte des Projekts

Das OFR-Projekt ist aus dem Projekt „Ein Arbeitsplatz zum rechnerunterstützten handschriftlichen Rechnen mit mathematischen Formeln“ von R. Marzinkewitsch (siehe [Marzin]) entstanden. Anfangs bestand das System aus einem Digitalisiertablett angeschlossen an einen PC, der wiederum über die serielle Schnittstelle mit einer Workstation von Sun verbunden war. Das Digitalisiertablett wurde zur handschriftlichen Eingabe der Formeln benutzt und bestand aus einer Scannerfläche, auf die mit einem Stift geschrieben werden konnte. Der Nachteil des Tabletts war, daß der Benutzer seine Eingabe nicht an der gleichen Stelle sah, an der er sie gemacht hatte, sondern nur auf dem Bildschirm des Computers. Dieser Nachteil wurde durch das vom Projekt entwickelte spezielle Digitalisiertablett, eine Kombination aus einem einfachen Digitalisiertablett und einem LCD-Display, beseitigt. Später wurden das spezielle Digitalisiertablett und der PC durch einen PC mit LCD-Display und integriertem Stift ersetzt (siehe [Quapp] Seite 8). Die Verbindung zur Workstation blieb weiter bestehen. Durch die stetige Entwicklung der Computer konnte das System ab 1994 allein auf einem PC mit integriertem Stift, Pentop genannt, betrieben werden.

0.1.2 Aufbau des Pentop-Systems

Der Pentop ist mit einem LCD-Display ausgestattet, auf das mit einen Stift in der Größe eines Kugelschreibers geschrieben werden kann. Der Stift ersetzt eine Maus, wobei er dieselben Funktionen zur Verfügung stellt.

Stift über bzw. auf dem Display bewegen	$\leftrightarrow$	Mausbewegungen
Stiftspitze auf Display drücken	$\leftrightarrow$	linker Mausknopf
zusätzliches Drücken des seitlichen Knopfes	$\leftrightarrow$	rechter Mausknopf

Durch einen schnelleren Prozessor konnte auf die früher benutzte serielle Verbindung zu einer Workstation verzichtet werden, so daß das System allein auf dem PC lauffähig ist. Die Programmteile, die unter UNIX auf der Workstation liefen, wurden komplett auf den PC unter Windows portiert. Zusätzlich wurde das C++GUI Toolkit wxWindows[2] benutzt, um die einheitliche Implementierung der gesamten graphischen Benutzerschnittstelle, für PC's unter Windows und Linux sowie für Workstations unter Unix mit X-Windows Oberfläche (Motif oder XView), in C++ zu ermöglichen. Die Module sind vollständig in C++ übertragen worden. Dabei wurden die verschiedenen Teile als Klassen implementiert (Abbildung 0.4).

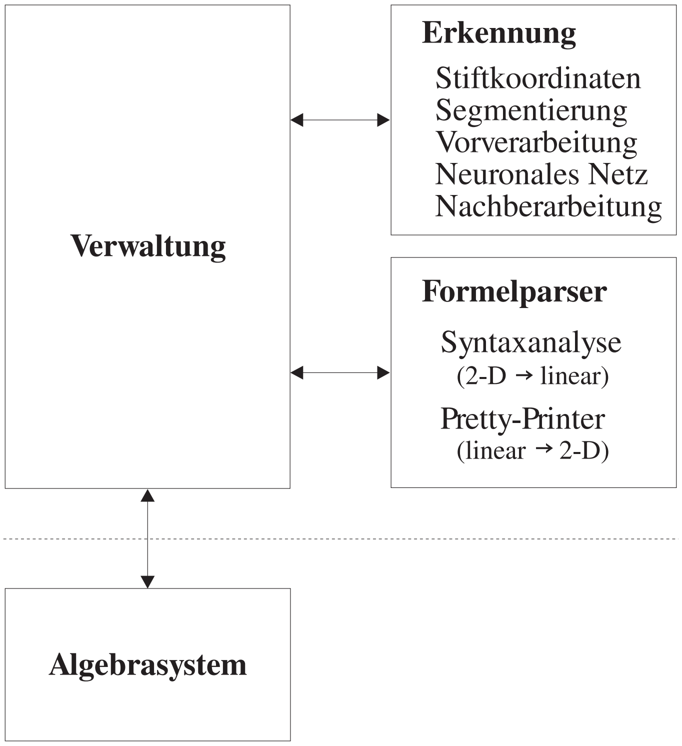


Abbildung 0.4: Struktur des OFR-Systems

Die *Verwaltung*, die *Erkennung*, der *Formelparser* und die Schnittstelle zum Algebrasystem wurden alle durch jeweils eine Klasse implementiert.

[2]wxWindows wurde von Julian Smart, Artificial Intelligence Applications Institute, Universität Edinburgh entwickelt. Die uns vorliegende Version darf zu nichtgewerblichen Zwecken benutzt werden.

Die *Verwaltung* besteht aus einer Klasse, die die Benutzereingaben entgegennimmt und an das jeweilige Modul weitergibt bzw. das jeweilige Kommando ausführt. Sie kann mehrere Seiten mit jeweils mehreren Formeln verwalten. Die Formeln können wiederum in verschiedenen Formaten (Klassen) vorliegen. Die Verwaltung verarbeitet die verschiedenen Formelklassen, indem die Klassen von einer virtuellen Basisklasse *FORMEL* abgeleitet werden. *FORMEL* definiert die zur Verwaltung benötigten Schnittstellen über rein virtuelle Funktionen. Die davon abgeleitete Klasse muß alle geforderten Funktionen bereitstellen, diese werden über die dynamische Bindung zur Laufzeit ausgewählt und aufgerufen.

Die *Erkennung* besteht aus den Modulen Stiftkoordinaten, Segmentierung, Vorverarbeitung, Neuronales Netz und Nachbearbeitung. Man gibt ihr alle Koordinaten beim Schreiben und bekommt die erkannten Zeichen zurück (näheres in Kapitel 2, Seite 45ff.).

Der *Formelparser* ist in der jeweiligen Formelklasse enthalten, so daß damit auch mehrere Formelparser in das System integriert werden können. In der Klasse sind die für die Syntaxanalyse, Linearisierung, zweidimensionale Bildschirmdarstellung, Speicher-, Ladeoperationen usw. benötigten Datenstrukturen und Funktionen enthalten. Im Projekt integrierte Formelparser sind der „Parser für unscharf zweidimensional attributierte Grammatiken“ von F. Weigel und der in Kapitel 1 vorgestellte Formeleditor.

Die Schnittstelle zwischen CAS und Verwaltung wurde portabel gehalten, da sie nur die technischen Voraussetzungen für die Verbindung beschreibt. Die syntaktische Anpassung der Notationen verschiedener Algebrasysteme würde ein zusätzliches Parsen einer Zwischensprache in die Sprache des CAS notwendig machen. Da der Formelparser die Zeichen der Formel sowieso schon parsen muß, ist hier die Umformung in einem Schritt möglich. Auch muß keine für alle Algebrasysteme allgemeingültige Zwischensprache entwickelt werden, die möglicherweise bestimmte Funktionalitäten zukünftiger CAS nicht bereitstellen kann. Die Schnittstelle sorgt also nur für den Transport der Formel, die als reiner ASCII-Text vorliegt.

0.2 Die Hardware

Als Hardware steht dem Projekt ein tragbarer Notebook-Computer der Marke Compaq Concerto zur Verfügung, der einen Intel 486 mit 33 MHz, 8 MB Speicher und ein monochromes LCD-Display mit Stift besitzt. Weiterhin ein Notebook der Marke NEC VERSA EC 50 Pen mit Intel 486DX2-50, 12 MB Speicher und Farb-LCD-Anzeige mit Stift. Zur Zeit läuft das System auf beiden Pentops nur unter MS-Windows, da nur für dieses Betriebssystem Stifttreiber vorhanden sind.

Als Workstation dient eine SPARCstation SLC mit 12 MB Hauptspeicher. Sie wird vorwiegend zum Trainieren der Neuronalen Netze benutzt, da diese Lernphase, je nach Größe des Netzes, mehrere Stunden dauert.

0.3 Die Aufgabenstellung der Diplomarbeit

Wie sieht ein „Editor zur handschriftlichen Eingabe zweidimensionaler Formeln in den Computer“ aus?

Der Editor integriert sich als Formelparser (Abbildung 0.4, Seite 4) in das vorhandene Gesamtsystem. Da die Schnittstelle zwischen *Verwaltung* und *Formelparser* festgelegt ist (siehe Abschnitt 0.1.2), muß der Editor folgende Funktionalität erbringen:

Eingabe Eine handschriftlich eingegebene zweidimensionale Formel ist eine Menge von Zeichen eines Alphabets (siehe Anhang A), die durch ihre Größe und Lage zueinander die Bedeutung der Formel ergeben.

Ausgabe Es gibt zwei Ausgaben. Eine für den Benutzer sichtbare, zweidimensionale Darstellung auf dem Bildschirm und eine unsichtbare für das angeschlossene Computeralgebrasystem in linearer Notation.

Hauptaufgaben eines Editors

- Eingabe einer Formel oder Bearbeitung einer bestehenden Formel
- Parsen der Eingabe zu einer internen Darstellung
- Transformation in eine lineare Formel gleicher Bedeutung
- Weitergabe der linearen Formel an ein CAS (Verwaltung)
- Entgegennehmen der linearen Lösung (Verwaltung)
- Transformation in interne Darstellung
- Darstellen der Lösung als zweidimensionale Formel

Wie kann die Eingabe geparst werden?

Es gibt verschiedene Ansätze beim Parsen der Zeichen zu einer Formel.

Offlineparser Beim Offlineparsen wird zuerst die gesamte Formel eingegeben, in dieser Eingabephase können die handschriftlichen Zeichen frei auf dem Bildschirm plaziert werden. In der zweiten Phase, der Syntaxanalyse, versucht der Parser, die einzelnen Zeichen anhand bestimmter Regeln einer Grammatik zu einer syntaktisch korrekten Formel zusammenzubauen. Dieser Ansatz eines Parsers für zweidimensionale Eingaben wurde bereits von F. Weigel implementiert. Sein Konzept beruht auf dem Parsen unscharf zweidimensional attributierter Grammatiken. Die verwendeten kontextfreien Regeln enthalten die Geometrieinformation nur in Form von Attributen. Der Parser erzeugt gleichzeitig mehrere Syntaxbäume bottom up, wobei die Knoten der Bäume eine Güte besitzen. Abhangig von der Güte wird der darunterliegende Teilbaum weiter benutzt oder verworfen. Entsteht ein Syntaxbaum, dessen Blätter genau den Zeichen der Formel entsprechen, entspricht dieser einer möglichen Lösung mit der Wahrscheinlichkeit der Güte seiner Wurzel. Jeder Knoten trägt seine Geometrieinformation (Attribute) und die Güte seines Unterbaumes. Die Attribute bzw. die Güte eines Knotens berechnen sich mit Hilfe von Updatefunktionen u_i bzw. Bewertungsfunktionen b_i aus den Attributen und der Güte seiner Kinder. Die Funktionen hängen von der Regel ab, mit der der Knoten erzeugt wurde, besitzen aber alle die folgenden Wertebereiche:

$$\begin{aligned} u_i &: \mathbb{R}^{n_i} \to \mathbb{R} \\ b_i &: \mathbb{R}^{m_i} \to [0,1] \end{aligned}$$

Den Anfang dieser rekursiven Berechnungen stellen die Blätter des Syntaxbaumes, d.h. die Zeichen der Formel, dar. Hier entsprechen die Attribute der Größe und Plazierung der geschriebenen Zeichen. Die Güte kann wahlweise durch eine Konstante oder durch die Güte der Erkennung festgelegt werden. Schwachpunkte dieses Ansatzes bzw. der Implementierung sind die durch die Ungenauigkeit der Eingabe im Zusammenspiel mit der unscharfen Attributierung entstehenden Mehrdeutigkeiten und das Fehlen einer Hilfe bei der Nichterkennung bestimmter Teilausdrücke.

Onlineparser Beim Onlineparsen wird ein Zeichen direkt nach der Eingabe durch die Regeln einer Grammatik in die bestehende, möglicherweise noch unvollständige Formel eingefügt. Die zweite Phase, zu dem Zeitpunkt zu dem die Linearisierung der eingegebenen Formel benötigt wird, besteht nur noch in der Überprüfung der syntaktischen Korrektheit der Formel. Durch das direkte Einfügen kann sich der Parser bei einer entstandenen Mehrdeutigkeit für eine beliebige Möglichkeit entscheiden. Diese Entscheidung wird durch die graphische Ausgabe und Formatierung der Zeichen dem Benutzer sichtbar gemacht und kann wenn nötig sofort korrigiert werden. Dieser Ansatz entspricht in vielen Teilen dem des Offlineparsers und kann durch

Modifikation des Parsers und eine Erweiterung der Grammatik erreicht werden. Die Einschränkung der freien Plazierung der Zeichen führt zu einem syntaxgesteuerten Editor, wie er im folgenden beschrieben wird.

Editor Das Grundkonzept ist die Darstellung der vorhandenen Daten und eines Cursors als Stelle zum Einfügen neuer Daten.

Beispiel:

$$a + b$$

Jedem Cursor entspricht ein eindeutiges Feld, in dem sich der Stift befinden muß, damit dieser Cursor sichtbar wird. Die Positionierung des Cursors erfolgt durch den Stift und hängt syntaxgesteuert von der vorhandenen Eingabe ab. Vor und hinter jedem Zeichen kann ein Cursor erscheinen. Zusätzliche Eingabefelder stellt die Syntaxsteuerung erst nach Eingabe bestimmter Schlüsselsymbole bzw. Schlüsselwörter bereit. Ein Beispiel hierfür ist die Möglichkeit der Angabe der Grenzen eines Integrals. Diese können erst nach dem Schlüsselsymbol, in diesem Fall dem Integralzeichen, eingegeben werden. Ohne diese Steuerung könnte der Benutzer nur schwer das richtige Eingabefeld herausfinden. Mit Hilfe des Cursors und den Eingaben baut der Editor eine Struktur auf, die der Formel entspricht und welche die Grundlinie und Größe aller eingegebenen Zeichen festlegt. Diese Festlegung wird sofort nach dem Einfügen eines neuen Zeichens auf dem Bildschirm angezeigt. Zusätzlich zur Syntaxsteuerung läßt der Editor nur eine zusammenhängende Formel entstehen, so daß alle Eingaben eine eindeutige Bedeutung haben und anhand dieser eindeutig im Kontext plaziert werden können.

Warum ein Editor? Bei jeder Arbeit am Computer sollte der Computer dem Benutzer so früh wie möglich mitteilen, ob die bisherige Eingabe, korrekt und eindeutig ist. Diese Rückkopplung zwischen Computer und Benutzer entspricht beim Offlineparsen eher der bei der Erkennung von eingescannten Vorlagen als der von handschriftlich eingegebenen Vorlagen, in unserem Fall Formeln, da eine Meldung erst nach Eingabe der vollständigen Formel erfolgt. Durch die ungenaue zweidimensionale Schreibweise können Eingaben entstehen, die nicht korrekt oder nicht eindeutig sind. Alle möglichen, korrekten Lösungen aus mehrdeutigen Eingaben können angezeigt und dadurch vom Benutzer die richtige ausgewählt werden. Dagegen kann aus syntaktisch falschen Eingaben, die nur durch ungenaue Eingabe entstanden sind, keine mögliche Lösung erzeugt werden. Die betroffenen Zeichen müssen gelöscht und neu geschrieben werden.

Die direkte Rückkopplung beim Onlineparsen entsteht durch die direkte Bearbeitung der Eingabe, die sofortige Auflösung bei entstehenden Problemen und die Visualisierung der Bedeutung der Eingabe. Die oben angesprochene Mehrdeutigkeit kann dadurch aufgelöst werden, daß sich der Parser für die wahrscheinlichste Möglichkeit entscheidet und das Ergebnis sichtbar macht. Durch die direkte Rückkopplung ist also höchstens die letzte Eingabe falsch verstanden worden und kann durch Überschreiben revidiert werden. Da der Offlineparser im Onlineparser enthalten ist, könnte ein möglicher Entwurf auf dem bisher implementierten Parser für unscharf zweidimensional attributierte Grammatiken aufsetzen, wenn dieser dokumentiert worden wäre.

Durch den Cursor wird die direkte Rückkopplung erweitert, da er eine Vorausschau aller möglichen Eingabefelder sowie deren Grundlinie und Schriftgröße darstellt. Damit die Information, die der Cursor darstellt, genutzt werden kann, muß jedoch die Menge aller möglichen Stellen reduziert werden. Die Reduktion erfolgt durch die Einschränkung der Eingabereihenfolge mit Hilfe einer Syntaxsteuerung, die festlegt, welcher Cursor sichtbar wird. Beim Editor können durch den Cursor auch Zeichen in den vorhandenen Kontext eingefügt werden, wobei sich alle nachfolgenden Zeichen automatisch nach hinten verschieben.

0.4 Aufbau der Arbeit

In Kapitel 1 wird zuerst das Grundkonzept des Cursors festgelegt. Der darauffolgende Abschnitt befaßt sich zuerst mit der Eingabe von Blockschrift und danach mit den Besonderheiten, die bei der Eingabe von Formeln auftreten können. Die auftretenden Besonderheiten werden dazu benutzt, das Grundkonzept des Cursors zu erweitern. Im Abschnitt „Transformation“ geht es darum, die Unterschiede zwischen der linearen und der traditionellen zweidimensionalen Schreibweise vorzustellen. Ein Parser stellt meistens seine Eingabe als Syntaxbaum dar; da der Syntaxbaum aber beim Onlineparsen ständigen Veränderungen ausgesetzt ist, wird im Abschnitt „Datenstruktur“ ein Strukturbaum beschrieben, der während der Eingabephase nur schwach variiert, d.h. keine großen bereits vorhandenen Teilbäume umstrukturieren muß. Die Vereinfachung gegenüber dem Syntaxbaum macht jedoch eine Nachbearbeitung nötig. Im nächsten Abschnitt wird die Implementierung des Formeleditors durch die Implementierung der unterschiedlichen Knoten des Strukturbaums als Klassen beschrieben. Die geforderte Funktionalität der Schnittstelle zum bestehenden OFR-System wird durch die Klasse *Formel*, die der Wurzel des Strukturbaums entspricht, gebildet. In der Implementierung wird auch die Erweiterung der vorhandenen Funktionen bzw. Strukturen beschrieben, um weitere Schreibweisen, Funktionen und Strukturen darstellen zu können. Im letzten Abschnitt werden Ausblicke über weitere Einsatzgebiete des Editors bzw. des Cursorkonsepts gegeben.

Das bisher verwendete Alphabet enthält nur einen Bruchteil der in der Mathematik verwendeten Zeichen, so daß in Kapitel 2 die Erweiterung des Alphabets und die damit verbundenen Probleme der Erkennung beschrieben werden. Zuerst wird eine Beschreibung der Module des bestehenden Systems geliefert, woraus ersichtlich wird, daß lediglich die Segmentierung von den Problemen der Erweiterung betroffen ist. Danach wird die Datenstruktur „Erkennungsgraph“ vorgestellt, in der alle Informationen enthalten sind, mit denen der Parser entscheiden kann, welche der möglichen Lösungsvorschläge die wahrscheinlich richtige Lösung ist. Da die Erkennung keine notwendigen Informationen über den Kontext der Eingabe hat, kann sie nur mögliche Lösungsvorschläge machen. Die Vorschläge können dafür aber das gesamte Wissen der Erkennung, d.h. die Besonderheiten des benutzen Alphabets und vor allem die individuellen Schriftproben des Benutzer, berücksichtigen. Dadurch, daß der Graph alle Informationen an den Parser weitergibt, benötigt der Parser kein Wissen über die individuelle Schrift des Benutzers, sondern muß nur das Wissen zum Zusammenbau der Zeichen zur Verfügung stellen.

Danksagungen

Ich bedanke mich herzlich bei Herrn Prof. Dr. Günter Hotz für die interessante Themenstellung und bei Herrn Dr. Joachim Hartmann für seine interessante Vorlesung „Neuronale Netze“, die mich zu diesem Projekt führte. Frank Weigel und Thomas Chadzelek gilt mein Dank für die ausgezeichnete Zusammenarbeit am OFR-Projekt.

Ihre freundliche Unterstützung, ihre Geduld und ihre wertvollen Ratschläge trugen wesentlich zum Gelingen dieser Arbeit bei. Allen Kommilitonen und Mitarbeitern an diesem Lehrstuhl danke ich für das hervorragende Arbeitsklima.

Insbesondere aber danke ich meinen Eltern, die mir ein Studium überhaupt erst ermöglichten; ihnen und meiner ganzen Familie widme ich dieses Werk.

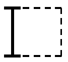
Kapitel 1

Formeleditor

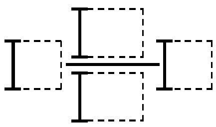
1.1 Grundsätzliche Vorgehensweise

Beim Bewegen des Stiftes über dem Display erscheint an bestimmten Stellen, abhängig von der bestehenden Eingabe und der Lage des Stiftes, ein Cursorsymbol. Nur an diesen Cursorstellen kann ein Zeichen eingefügt werden. Bei einer leeren Formel gibt es nur eine einzige Cursorposition. Abhängig von den neu eingefügten Eingaben entstehen weitere Stellen, an denen der Cursor erscheinen kann.

Beispiel:



leere Formel



Bruch

Das linke Bild zeigt die einzige Cursorstelle, einschließlich Eingabebereich, einer leeren Formel. Auf dem Bildschirm sind die gestrichelten Bereiche nicht sichtbar, auch der Cursor am linken Rand des Bereiches erscheint erst dann, wenn sich der Eingabestift in dem gestrichelten Bereich befindet. Wird z.B. ein Bruchstrich eingegeben, entstehen drei neue Cursorpositionen (rechtes Beispielbild), je eine für die Eingabe von Zähler, Nenner und weiterer Zeichen hinter dem Bruch.

1.2 Cursorkonzept

Durch die ungenaue zweidimensionale Anordnung bzw. die ungenauen Größenverhältnise bei der uneingeschränkten handschriftlichen Eingabe der

Zeichen einer Formel können beim Offlineparser Mehrdeutigkeiten entstehen. Im Gegensatz zum Offlineparser können beim Editor keine Mehrdeutigkeiten entstehen, da hier die handschriftliche Eingabe durch den Cursor eingeschränkt wird, jedes Zeichen sofort nach der Eingabe nach genau einer Bedeutung in den Kontext eingefügt wird und dem Benutzer diese Bedeutung durch die traditionelle Schreibweise der Formel direkt anzeigt wird. Diese direkte Rückkopplung zum Benutzer wird durch den Cursor zusätzlich erweitert, indem er eine Vorausschau aller möglichen Eingabebereiche darstellt. Sind diese Cursorstellen und die damit verbundenen Bereiche eindeutig, so ist auch deren Bedeutung eindeutig.

Bei der Vielfalt der verschiedenen Schreibweisen von Formeln wäre ohne eine Einschränkung der Schreibreihenfolge eine Vorausschau der Bedeutung durch den Cursor nicht möglich, da alle möglichen Eingabefelder sich überlappen würden und so eine eindeutige Zuordnung zu einem Cursor unmöglich machen. Durch die Festlegung einer Schreibreihenfolge kann die Anzahl der Cursorstellen so reduziert werden, daß fast alle Eingabefelder eindeutig werden. Diese Einschränkungen des Benutzers sollten jedoch so gering und intuitiv einfach wie möglich sein.

Um mögliche Einschränkungen besser beschreiben zu können, werden zuerst nachfolgende Begriffe erläutert, die eine Unterscheidung der Eingaben erlauben. Einerseits gibt es Eingaben, die bereits vorhandene Zeichen erweitern und mit diesen auf einer gemeinsamen Grundlinie stehen, im weiteren als *lineare Eingaben* bezeichnet, und andererseits jene, die einem Parameter einer Funktion entsprechen und nicht auf einer gemeinsamen Grundlinie mit dem Funktionsnamen bzw. -symbol stehen, im weiteren als *zweidimensionale Parameter* bezeichnet. Lineare Eingaben, die Parameter einer Funktion darstellen, werden noch zusätzlich als *lineare Parameter* bezeichnet. Mit diesen Bezeichnungen können die eingegebenen Funktionen in *linear eingegebene Funktionen*, das sind Funktionen ohne zweidimensionale Parameter, und in *zweidimensional eingegebene Funktionen*, das sind Funktionen mit mindestens einem zweidimensionalen Parameter, unterteilt werden. Eine weitere Unterteilung spaltet die Eingaben in *Schlüsselsymbole*, das sind Funktionssymbole wie $\int$, $\sum$, $\prod$ usw., in *Schlüsselwörter*, das sind Funktionsnamen wie lim, sin, cos usw., und in *einfache Zeichen* auf (siehe Abbildung 1.1).

Da der Zusammenhang zwischen den zweidimensionalen Parametern und der restlichen Eingabe erst durch das zugehörende Schlüsselsymbol bzw. Schlüsselwort entsteht, ist es eine geringe und intuitiv einfache Einschränkung, daß die Schlüssel vor den zweidimensionalen Parametern eingegeben werden müssen. Ohne diese Forderung könnte der Editor auch nicht entscheiden, wie er die zweidimensionalen Parameter einer noch nicht spezifizierten Funktion innerhalb des Kontextes behandeln bzw. plazieren sollte. Durch obige Einschränkung und die daraus resultierenden Eindeutigkeit des Cursors

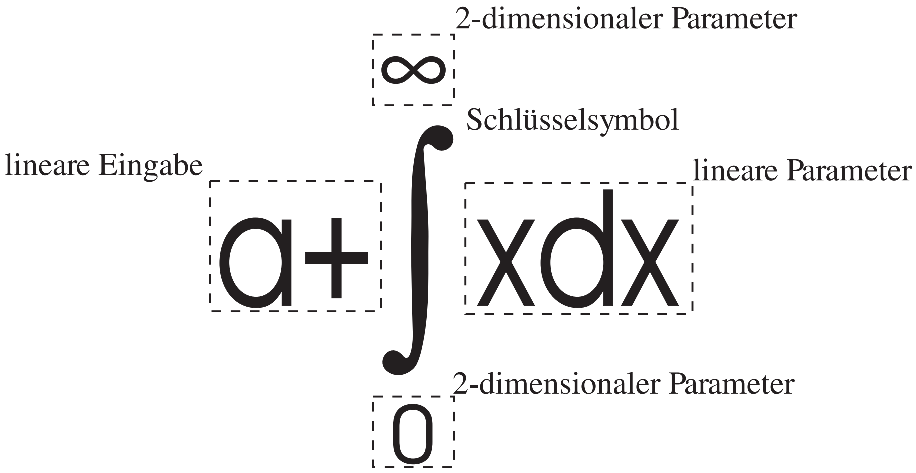


Abbildung 1.1: Bezeichnungen der verschiedenen Eingaben

verbessert sich auch die Erkennung von bestimmten Buchstaben (näheres in Abschnitt 1.3.1, Seite 15), da durch den Cursor auch die Grundlinie und Normhöhe der dort möglichen Eingaben angezeigt wird.

Nur wenn beim Aufsetzen des Stiftes der Cursor vorhanden ist, kann das geschriebene Zeichen in die Formel eingefügt werden, da nur dann dem Editor eine eindeutige Einfügestelle in die Formel vorliegt.

1.3 Eingabe von Blockschrift

Wie bei einem einfachen Texteditor gibt es auch hier zwei Modi zum Eingeben von Zeichen. Standardmodus ist das Einfügen eines Zeichens, hier wird das neue Zeichen an der Cursorstelle eingefügt und die Zeichen dahinter verschoben, auch wenn es im Moment des Schreibens wie ein Überschreiben aussehen kann. Zweiter Modus ist das Überschreiben von Zeichen, hier ersetzt das neue Zeichen das Zeichen hinter dem Cursor. Zwischen den beiden Modi kann mittels Aufruf des entsprechenden Menüeintrags `einfügen/überschreiben` oder durch speziellen Tastendruck umgeschaltet werden.

1.3.1 Lineare Eingabe

Zu jedem möglichen Cursor gibt es einen Bereich, in dem der Stiftpointer liegen muß, damit der Cursor sichtbar wird. Bei der linearen Eingabe einfacher Zeichen wie Buchstaben, Ziffern und die mathematischen Symbole $+ \; - \; * \; /$ steht das neueingegebene Zeichen mit bereits eingegebenen Zeichen auf einer Grundlinie, wobei folgende Fälle auftreten können:

- Leere Formel

- Vor einem bereits eingegebenem Zeichen

  $a + b$ — einfügen $a + b$ — überschreiben

- Zwischen zwei Zeichen

  $a + b$ — einfügen $a + b$ — überschreiben

- Hinter dem letzten Zeichen der Formel oder eines zweidimensionalen Funktionsparameters

  $a + b$ — einfügen $a + b$ — überschreiben

Liegt der Aufsetzpunkt eines Zeichens in dem gestrichelten Bereich, wird das Zeichen an der entsprechenden Cursorstelle eingefügt bzw. das Zeichen hinter der Stelle durch das neue überschrieben.

Das untere Ende des Cursors entspricht einer gedachten Grundlinie und die Größe entspricht der Schriftgröße eines Großbuchstabens. Das eingegebene Zeichen sollte in Bezug auf Lage und Größe mit dem Cursor übereinstimmen (siehe Abbildung 1.2), da zum Beispiel das 'P' und das 'p' nur anhand der

Abbildung 1.2: Lage und Grösse der Zeichen

Lage zur gedachten Grundlinie erkannt werden kann. Das 'O' und das 'o' kann nur anhand der Größenverhaltnisse zum Cursor erkannt werden.

Die obige Festlegung des Bereichs in Bezug auf den Cursor beruht auf der Tatsache, daß der erste Aufsetzpunkt beim Schreiben eines Zeichens auf der vorgegebenen Grundlinie für fast alle Zeichen des verwendeten Alphabets (siehe Anhang A, Seite 58f.) in diesem Bereich liegt. Ausnahmen sind β, μ und ρ. Dadurch ist beim Schreiben fast aller Zeichen des Alphabets der gleiche Cursor im Moment des Aufsetzens sichtbar.

1.3.2 Eingabe zweidimensionaler Parameter

Zusätzlich zu den Eingabefeldern bei der linearen Eingabe werden durch das Cursorkonzept aus Abschnitt 1.2 und die damit verbundene Einschränkung der Schreibreihenfolge folgende weitere Eingabefelder benötigt:

- Funktionen ohne Schlüsselsymbol und mit mindestens einem zweidimensionalem Parameter, wie

  $b^{\square}$ Potenz $\qquad$ $b_{\square}$ Index

- Funktionen mit einem Schlüsselsymbol und mindestens einem zweidimensionalem Parameter, wie

  $\int_{\square}^{\square} \square$ Integral, Summe, Produkt $\qquad$ $\sqrt{\square}$ Wurzel $\qquad$ $\frac{\square}{\square}$ Bruch

- Funktionen mit zwei mehrdeutigen Schlüsselsymbolen, wie der Binomialkoeffizient im Gegensatz zu der Klammerung mit runden Klammern

  $\binom{\square}{\square}$ Binomialkoeffizient $\qquad$ $(\square)$ Klammern

- zweidimensional eingegebene Funktionen mit einem Schlüsselwort, wie

  $\lim_{\square} \square$ Limes

- zweidimensional eingegebene Datenstrukturen variabler Größe mit zwei eindeutigen Schlüsselsymbolen, wie

  $\begin{bmatrix} \square & \cdots & \square \\ \vdots & \ddots & \vdots \\ \square & \cdots & \square \end{bmatrix}$ Matrix

- in obigen Schreibweisen enthaltene Funktionen, wie Exponentialfunktion e^x in Potenz, Ableitung der Form $\frac{\partial f}{\partial x}$ in einem Bruch und $\sin x$ bei den linearen Eingaben

Die obigen Strukturen sind nicht alle möglichen, sondern nur ein häufig benutzter Teil, der bei der Implementierung des Editors berücksichtigt worden ist. Der Editor kann um weitere Strukturen erweitert werden. Wie eine Erweiterung funktioniert, wird im Abschnitt 1.6.19 erklärt.

Die Eingabefelder der zweidimensionalen Parameter, die durch ihre Lage (z.B. Integralgrenzen) oder durch die Größe des Schlüsselsymbols (z.B. Nenner und Zähler) von den anderen abgesetzt sind, entsprechen denen bei der linearen Eingabe.

An bestimmten Strukturen ist ersichtlich, daß nicht für alle Bereiche die Disjunktheit gefordert werden kann. So überschneiden sich bei Basis und Exponent einer Potenz die Bereiche teilweise. Für die überlappten Bereiche ist der Cursor nicht eindeutig, so daß zusätzliche Parameter benötigt werden, die Stelle eindeutig zu machen.

Potenz Da sich beim Exponent gegenüber der Basis die Grundlinie und Normhöhe deutlich ändern, können sie zur Auswahl der richtigen Stelle benutzt werden. Für die Potenz wird auch die Berechnung des Cursors so erweitert, daß der sichtbare Cursor sich beim Übergang vom eindeutigen in den mehrdeutigen Bereich nicht ändert. Durch diese Erweiterung bleibt der Cursor für fast alle Zeichen des verwendeten Alphabets konstant, jedoch kann es vorkommen, daß der Cursor des Exponenten beim Aufsetzen sichtbar war, obwohl ein Zeichen in den darunterliegenden Bereich eingegeben wurde. Das Zeichen kann aber anhand seiner Grundlinie und Normhöhe eindeutig dem unteren Bereich zugeordnet werden.

Binomialkoeffizient Die Eingabe von Binomialkoeffizienten ist erschwert, da sie weder ein eindeutiges Schlüsselsymbol noch ein eindeutiges Eingabefeld besitzen. Durch die Forderung, daß die öffnende und schließende Klammer als Schlüsselsymbole vor der weiteren Eingabe vorhanden sein müssen, vergrößert sich die Wahrscheinlichkeit, sich eher für den Binomialkoeffizienten als für den geklammerten Ausdruck zu entscheiden, da bei der Eingabe eines geklammerten Ausdrucks die Schreibreihenfolge fast immer erst öffnende Klammer, dann Ausdruck und erst zum Schluß schließende Klammer lautet. Die Vergrößerung der Wahrscheinlichkeit kann dadurch erreicht werden, daß die Funktion, die alle möglichen Stellen berechnet, diese Wahrscheinlichkeit in die Berechnung der Güte einer Stelle einbringt. An der Stelle mit der größten Güte wird das neue Zeichen eingefügt. Da nur durch die Stiftposition keine eindeutige Entscheidung getroffen werden kann, gibt es auch nach Eingabe der Klammern keinen sichtbaren Cursor für die Bereiche des Binomialkoeffizienten. Durch die Reduzierung der Größe der einzugebenden Zeichen, d.h. einen deutlichen Unterschied zur Größe der Zeichen bei der linearen Eingabe und eine weitere Einschränkung der Eingabereihenfolge, indem bei Eingabe von $\binom{n}{k}$ das n vor dem k eingegeben werden muß, kann die Eingabe fast immer als Binomialkoeffizient erkannt werden. Die Ausnahmen entstehen, wenn das erste obere Zeichen entweder ein 'y' oder ein 'p' mit einer großen Unterlänge ist, die schreiberabhängig vom handschriftlich eingegebenen 'Y' bzw. 'P' nur schwer zu unterscheiden sind. Die Großbuchstaben könnten auch der Anfang eines geklammerten Ausdrucks sein, da ihre Grundlinie und Normhöhe denen der linearen Eingabe sehr nahe kommen kann.

Matrix Um Matrizen variabler Größe eingeben zu können, müssen eindeutige Eingabebereiche für alle bereits vorhandenen sowie für alle zukünftigen Elemente der Matrix zur Verfügung stehen. Die Eindeutigkeit der Eingabebereiche bereits vorhandener Matrixelemente kann durch die Formatierung in eindeutige Spalten und Zeilen sichergestellt werden. Eine Möglichkeit der Erweiterung ist eine zusätzliche leere Spalte am linken Rand und eine leere Zeile am unteren Rand, die zur Vergrößerung der Matrix zur Verfügung stehen. Wird in ein Element dieser Spalte (Zeile) ein Zeichen eingefügt, erweitert sich die Matrix um eine Spalte (Zeile), wobei die zusätzliche leere Spalte (Zeile) wieder angehängt wird. Analog werden beim Löschen von Zeichen zwei leere Spalten am rechten Rand (Zeilen am unteren Rand) durch eine ersetzt. Bei dieser Implementierung kann eine Matrix von oben nach unten und von links nach rechts aufgebaut werden. Erste mögliche Einfügestelle ist das Element (1,1). Nach der Eingabe des Elements (1,1) sind die drei weiteren Elemente (1,2), (2,1) und (2,2) vorhanden. Analog zur obigen Methode kann eine zusätzliche Erweiterung nach links und oben aussehen. Die äußeren Klammern passen sich automatisch an die Größe der Zeilen und Spalten an.

1.4 Transformation

Der Editor muß bei der Linearisierung eine syntaktisch korrekte, zweidimensional eingegebene Formel in eine lineare gleicher Bedeutung transformieren. Die Umkehrung, der sogenannte *Pretty-Printer*, erzeugt für eine lineare Lösung vom CAS eine entsprechende Formel mit zweidimensionalen Parametern.

In der linearen Darstellung wird die eindeutige Bedeutung durch Klammern, die Rangfolge der Operatoren und ihre Assoziativität realisiert (Tabelle 1.1). Klammern und lineare Parameterlisten machen einen Ausdruck sehr lang und unübersichtlich. In der zweidimensionalen Schreibweise werden die meisten Klammern nicht benötigt.

Stufe[a]	Operator	Funktion
6	$(\ )\ ,\ f(x)$	Klammerung, Funktion
6	$\int\ \ d$	Integral
6	$\frac{a}{b}$	Bruch
5	a^b	Potenz
4R	$\prod$	Produkt
3L	$*$	Multiplikation
2R	$\frac{\partial}{\partial x} f(x)$	Ableitung
2R	$\sum\ ,\ \lim$	Summation, Limes
1L	$+,-$	Addition und Subtraktion

Tabelle 1.1: RANGFOLGE DER OPERATOREN UND IHRE ASSOZIATIVITÄT. DIESE RANGFOLGE IST IN DIE RANGFOLGE DES ALGEBRASYSTEMS EINGEBETTET.

[a] R–rechts-assoziativ, L–links-assoziativ

Beispiel:

$$(a + b\hat{\ }(c + d))/(e + f) \equiv \frac{a + b^{c+d}}{e + f}$$

Bei der Transformation von der linearen in die zweidimensionale Schreibweise können die Klammern für Bruch und Potenz entfallen.

Bei bestimmten Transformationen ins Zweidimensionale müssen manchmal Klammern ergänzt bzw. durch andere ersetzt werden.

Beispiel:

$$\begin{aligned} sum(a_i + b_i\, ,\, i = 1..n) &\equiv \sum_{i=1}^{n}(a_i + b_i) \\ \text{im Gegensatz zu} & \\ sum(a_i * b_i\, ,\, i = 1..n) &\equiv \sum_{i=1}^{n} a_i * b_i \end{aligned}$$

Bei der Linearisierung müssen die an verschiedenen Stellen plazierten Funktionsparameter berechnet und für das CAS zu einem Funktionsaufruf zusammengefaßt werden.

1.5 Datenstruktur

Mit Hilfe des Cursors können die im Abschnitt 1.3.2 dargestellten Strukturen eingegeben werden. In die Strukturen können wieder Strukturen eingefügt werden. Werden die Teile einer Struktur in einer Vater-Kinder-Beziehung abgelegt, entsteht durch die rekursive Eingabe ein Baum.

Ein Parser stellt normalerweise seine Eingabe als Syntaxbaum dar. Anhand des Syntaxbaumes kann die Korrektheit einer Eingabe überprüft und gegebenenfalls die gewünschte Ausgabe erzeugt werden. Der Syntaxbaum ist je nach Struktur der Grammatik beim Onlineparsen starken Veränderungen ausgesetzt. Vor allem bei der Eingabe von Klammern und Schlüsselsymbolen in die vorhandene Formel müssen große Teilbäume verändert werden. Außerdem stellt sich die Frage, wie ein Syntaxbaum für unvollständige Ausdrücke aufgebaut ist und sich durch weitere Eingaben verändert.

1.5.1 Strukturbaum

Der Strukturbaum ist ein spezieller Syntaxbaum, dessen Struktur sich beim inkrementellen Parsen nur geringfügig ändert, d.h. es müssen keine bereits vorhandenen Teilbaüme ihre Struktur verändern. Beim Hinzufügen eines Schlüsselsymbols wird nur der entsprechende Unterbaum (siehe Anhang B) zu einem neuen Kind eines bereits vorhanden Knotens, beim Einfügen eines einfachen Zeichens wird nur das neue Blatt angehängt. Beim Strukturbaum werden alle Zeichen in einem Knoten zusammenfaßt, die eine gemeinsame Grundlinie besitzen und nicht durch ein Schlüsselsymbol vom Kontext abgetrennt sind.

Beispiel:
Ein Beispiel eines möglichen Syntax- und Strukturbaums für $\sum_{i=1}^{n} i + n$ ist in Abbildung 1.3 zu sehen. Das in der Abbildung benutzte 'E' steht für einen Ausdruck bzw. Teilausdruck, die Reihenfolge der Kinder von $\sum$ ist: untere Grenze, obere Grenze und beim Syntaxbaum noch der zu summierende Ausdruck.

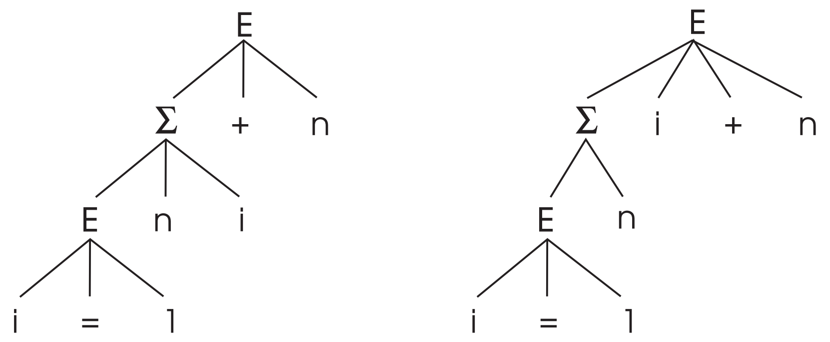


Abbildung 1.3: Möglicher Syntaxbaum (links) und Strukturbaum (rechts)

Außer der Reihenfolge der Zeichen und Strukturen wird keine weitere syntaktische Struktur gespeichert. Als Beispiel sind die Grenzen der Summe durch ihre Lage in jeweils einem eigenen Unterbaum gespeichert, die linearen Parameter wie zu summierender Ausdruck, stehen auf der selben Ebene wie das '$\sum$' und die davor und dahinterliegenden Zeichen. Der Ausdruck unter einer Wurzel kann mit weiteren Zeichen auf einer Grundlinie stehen, wird aber durch die Wurzel vom Kontext abgetrennt und daher in einem Knoten zusammengefaßt.

Der Cursor bestimmt, an welcher Stelle des vorhandenen Strukturbaumes die neue Struktur eingehängt wird. Das Umherwandern mit dem Stift auf dem Bildschirm entspricht dem Umherwandern in dem entsprechenden Baum. Das Löschen eines Zeichens ist dem Löschen bestimmter Knoten gleichzusetzen. Den Blättern des Baumes entsprechen die sichtbaren Zeichen, den inneren Knoten entsprechen Funktionen, die zweidimensionale Parameter besitzen.

Da beim Strukturbaum für die linearen Eingaben keine weitere syntaktische Struktur als die Reihenfolge berechnet wird, muß vor der Linearisierung die Syntax des entsprechenden Strukturbaumes überprüft werden, da nicht jedem Strukturbaum eine korrekte Formel entspricht. Auch die linearen Parameter, die mit anderen Zeichen auf einer Grundlinie stehen und in einem Knoten des Baumes zusammengefaßt sind, müssen berechnet und ihren Funktionen zugeordnet werden. Bei einer nicht korrekten Formel wird die Fehlerstelle durch ein Rechteck markiert und das Fehlersymptom beschrieben.

Beispiele:

1)	$\int_0^{\boxed{}} x dx$	Fehlersymptom:	Obergrenze fehlt
2)	$\boxed{(}a + b * (c + d)$	Fehlersymptom:	öffnende Klammer hat keine passende schließende Klammer

Da die Syntax der Formel auch durch das CAS überprüft wird, kann sich die Syntaxüberprüfung des Editors auf die Berechnung der Funktionsparameter und die durch die Linearisierung der Formel verschwindenden Fehler beschränken. Eine unkorrekte Klammerung kann verschwindende Fehler produzieren, da durch die Linearisierung eine korrekte Formel entstehen kann.

Beispiel:
Da ein Bruch $\frac{a}{b}$ zu $(a)/(b)$ linearisiert wird, verschwindet die unkorrekte Klammerung der folgenden Formel bei der Linearisierung und eine falsche Linearisierung wird erzeugt.

$$\frac{(a+b}{c)+d} \quad \overset{\text{Maple}}{\Longrightarrow} \quad ((a+b)/(c)+d)$$

Da alle Zeichen, die auf einer Grundlinie stehen und nicht durch ein Schlüsselsymbol abgetrennt sind, einem linear geschriebenem Ausdruck entsprechen, kann ihre Syntax lokal mit einem einfachen linearen Parser mit entsprechender Grammatik überprüft werden, wobei durch die enthaltene Rangfolge von Operatoren und ihrer Assoziativität (Tabelle 1.1, Seite 19) auch die Berechnung und Zuordnung der Funktionsparameter möglich ist. Der einfache Parser kann entweder mit Hilfe der Parser-Generatoren „Lex“ und „Yacc“ sowie einer entsprechenden Grammatik automatisch erzeugt werden oder wegen der einfachen Aufgabe von Hand als Kellerautomat implementiert werden.

Fazit Durch das Cursorkonzept und die Darstellung der Daten in einem Strukturbaum konnte das zweidimensionale Parsen auf die syntaxgesteuerte, kontrollierte Eingabe zweidimensional plazierter Zeichen mit anschliessendem linearen Parsen der linearen Eingaben des Strukturbaums reduziert werden.

1.6 Implementierung

Durch die besondere Kapselung der sichtbaren Strukturen im Strukturbaum kann jeder Knoten als eine für sich und seine Kinder verantwortliche Klasse implementiert werden. Dabei sind alle sichtbaren Zeichen in der Klasse *Zeichen*[1] zusammengefaßt. Die grundlegenden Funktionen und Variablen zum Aufbau und zur Durchmusterung der Knoten des Baumes sind in der Klasse *Element* deklariert. Da jede Klasse einem Knoten entspricht, sind alle Klassen von der Basisklasse *Element* abgeleitet (siehe Abbildung 1.4).

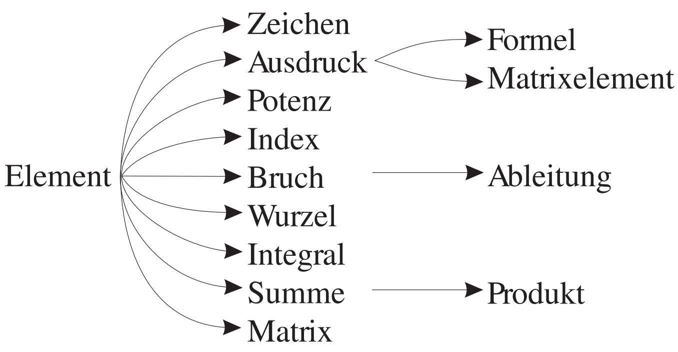


Abbildung 1.4: Vererbungshierachie der Klassen

[1]Die Namen der Klassen entsprechen den Namen in der Implementierung.

1.6.1 Element

Die Datenstruktur der Klasse *Element* stellt unter anderem das umfassende Rechteck eines jeden Knotens zur Verfügung. Dieses Rechteck beschreibt den Nutzungsraum der jeweiligen Klasse und entspricht bei der Klasse *Zeichen* der Größe und Lage des sichtbaren Zeichens auf dem Bildschirm in der Einheit Pixel.

In einer Klasse des Editors können mehrere Kinder (Elemente) des Strukturbaumes zusammengefaßt werden. Diese muß dafür sorgen, daß alle ihre Kinder über disjunkte Flächen verfügen. Ausnahmen bilden hier die Klassen *Potenz* und *Binomialkoeffizient*. Jede Klasse bestimmt selbst darüber, wo eine Einfügestelle in ihrem Teilbaum ist. Die Position des Stiftes bzw. der Bereich, in dem der Stift liegen muß, darf nur im eigenen Rechteck liegen. In

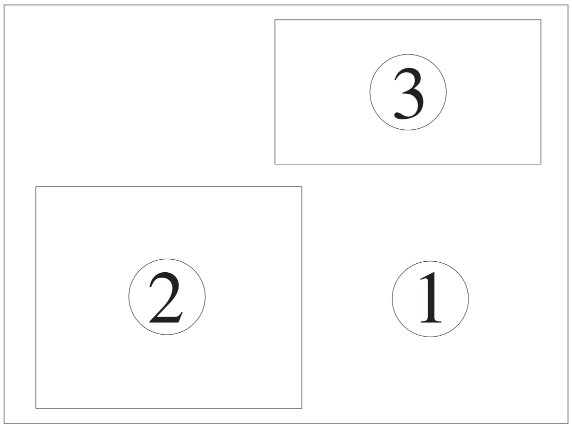


Abbildung 1.5: Modell der disjunkten umfassenden Rechtecke

Abbildung 1.5 sieht man drei Rechtecke, wobei jedem Rechteck eine Instanz einer Klasse entspricht. Da die Rechtecke 2 & 3 in 1 liegen, sind sie die Kinder von 1.

Die Knoten des Strukturbaumes sind nicht nur von der Wurzel zu den Blättern verbunden, sondern durch die `Vater`[2]-Pointer von *Element* auch in entgegengesetzter Richtung. Dagegen stellt jede Klasse die Verbindungsstruktur zu den Kindern individuell zur Verfügung, da die Anzahl der Kinder von Klasse zu Klasse unterschiedlich ist.

C++ unterstützt neben dem Klassenkonzept auch virtuelle Funktionen, mit ihrer Hilfe können abgeleitete Klassen spezialisierte Versionen einer Basisklassen-Funktion bereitstellen, ohne daß beim Aufruf die abgeleitete

[2]Alle Wörter in Schreibmaschinenschrift kommen als Variablen- oder Funktionsnamen im Programmtext vor.

Klasse explizit angegeben werden muß. Dieses Konzept der virtuellen Funktionen kann in der Implementierung des Editors dazu benutzt werden, daß eine Klasse bei einem Funktionsaufruf der Kinder ihres Strukturknotens, die verschiedenen Klassen angehören können, nicht auf den jeweiligen Typ beim Aufruf einer Funktion achten muß. Die virtuellen Funktionen müssen schon in der Basisklasse deklariert werden. Zusätzlich zur Deklaration kann auch eine allgemeingültige Funktion programmiert werden, die später durch die spezialisierten Funktionen überdeckt wird.

Diese Funktionalität wird bei den Funktionen zum Einfügen, Löschen, Formatieren und Anzeigen von Knoten benutzt. Durch die Deklarierung der Funktionen auf der untersten Ebene entsteht auch ein einheitliches Erscheinungsbild und eine eindeutige Schnittstelle zur Kommunikation der Funktionen untereinander. Vor allem in der Klasse *Ausdruck* erleichtert die dynamische Bindung der virtuellen Funktionen die Implementierung, da hier fast alle anderen Klassen als Kinder vorkommen können.

Layout

Das sichtbare zweidimensionale Layout der Zeichen wird durch die Funktion `format` erzeugt. Die Funktion arbeitet auf zwei Arten. Bei der Wurzel angefangen, wird die gesamte Formel neu gesetzt, indem jeder Knoten `format` für seine Kinder aufruft und die formatierten Kinder nach seiner jeweiligen Bedeutung formatiert. Beim Funktionsaufruf der Kinder wird deren Größe (Normhöhe) und Lage (x-Koordinate des linken Randes und Grundlinie) festgelegt. Die Funktion der Kinder benutzt die Parameter, um sich und gegebenenfalls ihre eigenen Kinder zu formatieren, d.h. das umfassende Rechteck festzulegen. Die Rekursion endet bei den Blättern, die den sichtbaren Zeichen entsprechen. Nach Beendigung der Funktion ist der gesamte Strukturbaum neu gesetzt. Wird nur ein Zeichen in einen bereits formatierten Strukturbaum eingefügt, werden abhängig von der Struktur nur bestimmte Teilbäume neu formatiert. Hierbei beginnt der Algorithmus beim Vater v der Wurzel w des neu eingefügten Teilbaums und arbeitet sich von w über die weiteren von w abhängigen Kinder von v über den Vater von v bis zur Wurzel des Strukturbaums. Bei dieser Methode werden alle Zeichen, die in der sichtbaren Darstellung vor der Eingabestelle sind, nicht verändert, alle Zeichen die bei Eingabe bestimmter Schlüsselsymbole (z.B. Bruch) mit diesen Schlüsseln eine neue Bedeutung erhalten (z.B. Zähler und Nenner), neu formatiert (Größe und Lage) und alle in der sichtbaren Darstellung dahinterliegenden Zeichen nur verschoben. Die Darstellung der Formel mit Hilfe der Funktion `print` vereinfacht sich daher zur Darstellung eines jeden Blattes an der durch das *umfassende Rechteck* festgelegten Stelle und in dessen Größe.

Cursor

Eine weitere wichtige virtuelle Funktion namens `Cursor_setzen` sorgt für die Plazierung des Cursors und das Auffinden der damit verbundene Stelle im Baum. Als Parameter bekommt die Funktion die Koordinaten des Stifts auf dem Display. Als erstes wird die Funktion der Wurzelklasse aufgerufen, diese überprüft, ob die Koordinaten in ihrem *umfassenden Rechteck* liegen. Ist dies der Fall, untersucht sie alle ihre Kinder, in dem sie die Funktion `Cursor_setzen` der Kinder mit den gleichen Parametern aufruft. Jede Klasse entscheidet individuell anhand dieser Funktion, ob der Cursor in ihrem Bereich liegt und wenn ja dort gesetzt werden kann. Die Funktion terminiert bei einem Blatt, wenn die Cursorstelle gefunden werden konnte, bei einem inneren Knoten, wenn die Stiftkoordinaten in keinem der umfassenden Rechtecke seiner Kinder liegen.

Dem *Cursor* auf dem Bildschirm entspricht ein `Element`-Pointer auf ein Element e, ein `Vater`-Pointer auf dessen Vater und eine Variable zur Angabe der Richtung, d.h. ob der Cursor vor oder hinter dem Element e steht (siehe

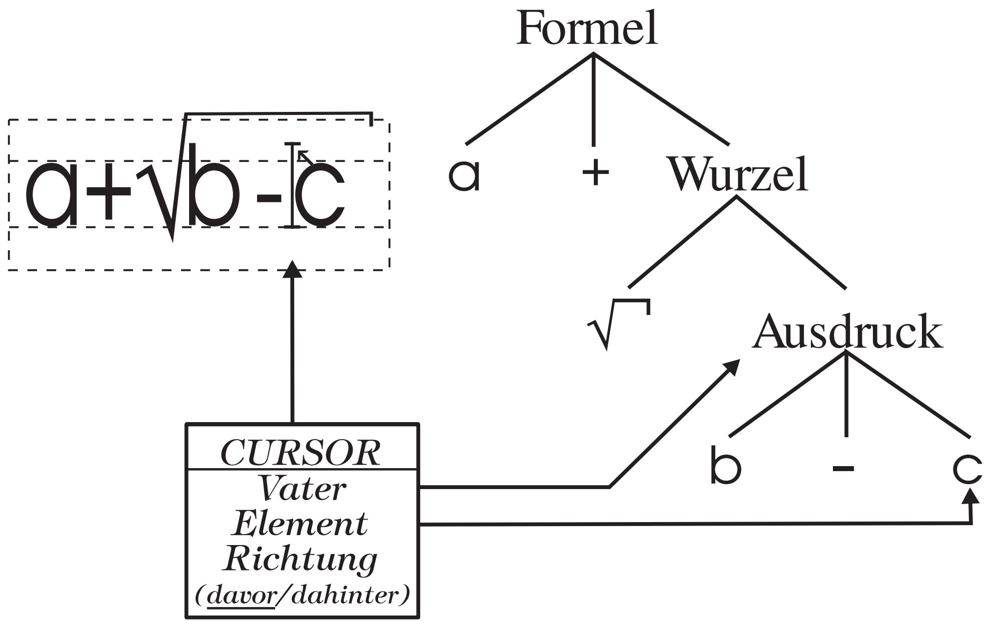


Abbildung 1.6: Zusammenhang zwischen dem Cursor auf dem Bildschirm und im Baum

Abbildung 1.6). Abhängig von der Formel kann der `Element`-Pointer unspezifiziert sein. Dagegen darf der `Vater`-Pointer nicht unspezifiziert sein, da die dort angegebene Klasse für das Einfügen eines neuen Zeichens verantwortlich ist. Vorwiegend ist eine Instanz der Klasse *Ausdruck* für das Einfügen neuer Zeichen verantwortlich, da die meisten Eingaben lineare Eingaben sind, die in dieser Klasse zusammengefaßt sind. Jeder mögliche zweidimensionale Parameter einer Funktion entspricht jeweils einer Instanz *Ausdruck*. Die zwei weiteren Angaben des Cursors geben die genaue Einfügestelle an. Ist der Ausdruck dagegen leer, so zeigt der `Vater`-Pointer auf die Instanz des Ausdrucks, der `Element`-Pointer auf ein neutrales Element und die Richtung ist unspezifiziert. Weiter Klassen für das Einfügen sind *Potenz* und *Index*, da sie keine Schlüsselsymbole besitzen (näheres im Abschnitt *Potenz*).

Während der Bewegungen des Stiftes (über dem Display) wird die Funktion `Cursor_setzen` immer mit den neuesten Koordinaten aufgerufen, so daß der Cursor dem Stift folgt. Beim Schreiben der Zeichen verändert der Cursor seine Lage nicht. Kommt das erkannte Zeichen von der Erkennung, kann es anhand des Cursors in den Strukturbaum eingehängt werden.

Linearisierte Formel

Die Funktion `ausgabe` der Wurzelklasse berechnet die linearisierte Darstellung der gesamten Formel durch geeignetes Aneinanderhängen der Linearisierungen seiner Kinder. Die Berechnung setzt sich rekursiv bis zu den Blättern fort.

1.6.2 Zeichen

Eine Instanz der Klasse *Zeichen* entspricht genau einem Zeichen aus dem verwendeten Alphabet (siehe Anhang A), indem dort ein eindeutiger Zeichencode gespeichert wird. Der Zeichencode ist durch das Alphabet eindeutig festgelegt. In dieser Klasse enden die meisten rekursiven Funktionen wie `print`, `Cursor_setzen`, `format`, `ausgabe` ...

Durch die Definition der virtuellen Basisklassen-Funktion `print` sorgt *Zeichen* für die Darstellung der Zeichen auf dem Bildschirm. Beim Aufruf von `print` wird das Zeichen anhand der durch das umfassende Rechteck festgelegten Größe und Lage (in Pixeln) auf den Bildschirm ausgegeben.

Die Funktion `Cursor_setzen` sorgt für die Berechnung der Einfügestellen vor, zwischen und hinter Zeichen sowie der Einfügestellen für noch nicht vorhandene Funktionen ohne Schlüsselsymbol, wie bei Potenz und Index. Die jeweilige Funktion `examine_Potenz` oder `examine_Index` untersucht die Bedingungen, ob eine Einfügung möglich ist. Die Funktionen und jeweiligen Darstellungen auf dem Bildschirm werden in ihrem jeweiligen Abschnitt noch näher erklärt.

Die Formatierung eines Zeichens durch `format` besteht nur in der Übernahme der Parameter `x-Koordinate`, `Grundlinie` und `Normhöhe` sowie der Rückgabe der `Breite` und `Höhe` (Normhöhe + Unterlänge) des Zeichens.

Da ein angeschlossenes Algebrasystem nur ASCII-Zeichen versteht, müssen bei der Linearisierung nicht darin enthaltene Zeichen speziell kodiert werden. Das verwendete CAS MapleV benutzt z.B. zur Kodierung der griechischen Buchstaben deren Namen. Beispiele hierfür sind α als *alpha*, β als *beta* und γ als *gamma*. Die griechischen Großbuchstaben werden wie die Kleinbuchstaben kodiert, wobei der Anfangsbuchstabe groß geschrieben wird. Beispiele hierfür sind Γ als *Gamma*, Δ als *Delta* und Θ als *Theta*. Die Kodierung ist von dem angeschlossenen CAS abhängig (weitere Hinweise dazu in Abschnitt 1.6.16).

1.6.3 Ausdruck

Die Klasse *Ausdruck* stellt alle Hilfsmittel zur Erzeugung und Veränderung linearer Zeichenketten zur Verfügung.

Ein *Ausdruck* ist eine Kette von Einzelzeichen und Strukturen, wobei die Strukturen ein- oder zweidimensional sein können. Im weiteren werden Einzelzeichen und Teilausdrücke unter dem Begriff *Elemente* zusammengefaßt. Alle *Elemente* hängen in horizontaler Richtung aneinander und werden in vertikaler Richtung an der Grundlinie ausgerichtet (siehe Abbildung 1.7).

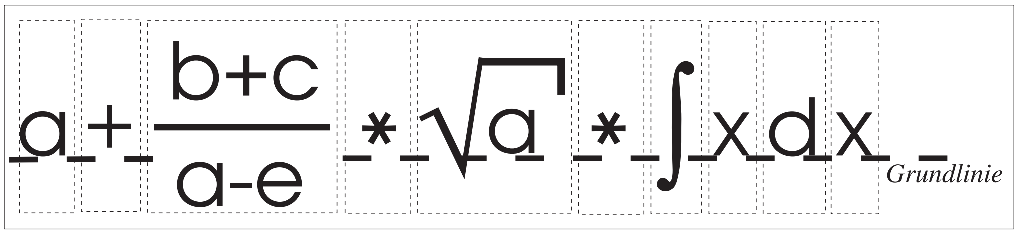


Abbildung 1.7: Aussehen eines Ausdrucks

Da ein „Ausdrucksknoten“ im Strukturbaum eine unbekannte aber kleine Anzahl Kinder hat, wurde statt einer Baumstruktur eine schon implementierte doppeltverkettete Liste aus LEDA zur Speicherung der Kinder benutzt. Durch die Kapselung der Klasse ist die konkrete Art der Implementierung allen anderen Klassen und deren Funktionen verborgen, so daß trotz der Entartung zu einem Graphen weiter von einem Baum gesprochen werden kann (siehe Abbildung 1.8).

Beim Einfügen überprüft *Ausdruck* das Zeichen, ob es ein Schlüsselsymbol, das letzte fehlende Zeichen eines Schlüsselwortes oder ein einfaches Zeichen ist. Liegt ein Schlüssel vor, wird eine Instanz der zugehörigen Klasse erzeugt, die nach ihrer Konstruktionsvorschrift auch ihre Kinder erzeugt. Liegt ein *einfaches Zeichen* vor, wird eine Instanz der Klasse *Zeichen* mit dem jeweiligen Zeichencode erzeugt. Das neue erzeugte Zeichen bzw. die neu erzeugte

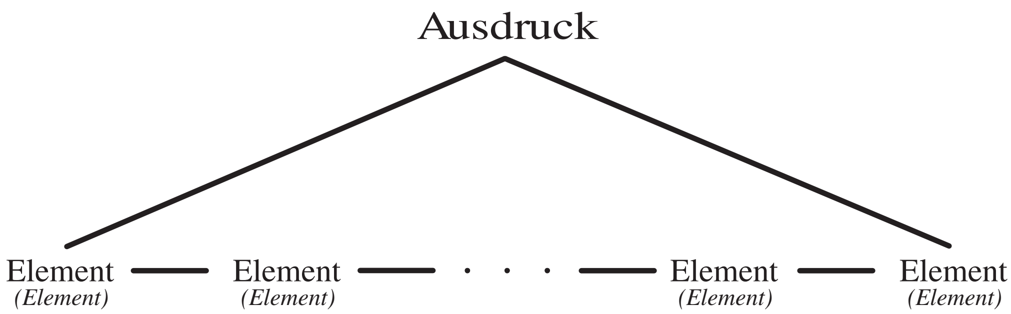


Abbildung 1.8: SIMULATION DES AUSDRUCKSBAUMS DURCH EINE VERKETTUNG ZWISCHEN ERSTEM UND LETZTEM KIND

Struktur wird an der durch den Cursor spezifizierten Stelle in den vorhandenen Baum eingehängt.

Nach dem Einhängen in den formatierten Baum muß der betroffene Teilbaum neu formatiert werden, indem dem neuen Knoten von *Ausdruck* die `Grundlinie`, die `Normhöhe` und die individuelle `x-Koordinate` übergeben wird. Anhand dieser Vorgaben berechnet die virtuelle Funktion `format` die Lage und Größe des umfassenden Rechtecks und somit auch das Aussehen des Knotens auf dem Display. Ausnahme ist hier der leere Ausdruck (leere Formel oder zweidimensionaler Parameter), er stellt als minimales Eingabefeld ein Eingabefeld der Größe eines Zeichens in der für den Ausdruck festgelegten Schriftgröße bereit.

Danach ist die Darstellung der Zeichen wieder gemäß der durch den Editor verstandenen Bedeutung, so daß `print` in einer einfachen Schleife nur den virtuellen `print`-Aufruf aller Knoten des Strukturbaumes auslösen muß. Da ein `print` nicht immer alle Zeichen neu zeichnen muß, besteht durch das Setzen des Parameters `reprinter` die Möglichkeit, daß `print` nur die veränderten (verschobenen oder verkleinerten) Zeichen neu zeichnet. Bei der Veränderung eines Zeichens oder eines ganzen Teilbaumes merkt sich der entsprechende Knoten in der Variable `reprint`, ob der unter ihm liegende Teilbaum ein verändertes Zeichen besitzt oder nicht. Beim Zeichnen der veränderten Teile wird die Variable wieder zurückgesetzt.

Die schwache Varianz des Strukturbaums gegenüber dem Syntaxbaum ist während der „langen" Eingabephase von Vorteil, macht aber vor der Linearisierung eine Berechnung und syntaktische Überprüfung der linearen Parameter notwendig. Der Berechnung liegt die Rangfolge der Funktionsoperatoren und ihre Assoziativität (Tabelle 1.1, Seite 19) zu Grunde.

Beispiel:

$$\int \sum_{i=0}^{k} x_i + y \, dx \quad \stackrel{\text{Maple}}{\Longrightarrow} \quad int(\, sum(\, x[i],\, i = 0..k)\, + y,\, x)$$

Die Implementierung der Berechnung und Überprüfung kann wegen der linearen Darstellung im Strukturbaum mittels der Parser-Generatoren „Lex“ und „Yacc“ sowie einer entsprechenden Grammatik automatisch erzeugt werden oder wie im vorliegenden Fall wegen der einfachen Aufgabe von Hand als Kellerautomat implementiert werden. Der Kellerautomat liest alle Elemente der Klasse *Ausdruck* der Reihe nach. Findet er Schlüsselsymbole, -wörter und öffnende Klammern, legt er sie auf den Keller. Beim Auftreten des ersten Zeichens hinter einer vollständigen Funktion, bestimmt durch Tabelle 1.1, oder beim Auftreten einer schließenden Klammer wird der oberste Eintrag des Kellers überprüft und bei Korrektheit vom Keller entfernt. Ist das Kellerelement eine Funktion, werden für den Zeitraum, in der das Element als oberstes liegt, alle gelesenen Zeichen als Parameter der Funktion angesehen und dem entsprechenden Knoten des Strukturbaumes mitgeteilt. Nach dem erfolgreichen Parsen aller Vorkommen von *Ausdruck* kennt jede Funktion ihre Parameter, so daß durch den Aufruf von `ausgabe` die Linearisierung berechnet werden kann.

Innerhalb dieser Klasse lassen sich Teilformeln kopieren, ausschneiden und an jeder Stelle auch wieder einfügen. Die Teilformeln entsprechen einem Teil der Elemente der Klasse, wobei mit einem Element auch dessen gesamter Unterbaum enthalten ist. Durch die Forderung, daß nur ganze Unterbäume kopiert bzw. ausgeschnitten werden können, entstehen beim Einfügen in eine Formel keine Mehrdeutigkeiten, da mit den zweidimensionalen Parametern auch immer das entsprechende Schlüsselsymbol eingefügt wird. Die Einschränkung ist nicht schwerwiegend, da es nicht sinnvoll ist, nichtzusammenhängende Teile, wie z.B. Zähler und Nenner eines Bruches ohne den Bruchstrich, zu kopieren.

1.6.4 Formel

Die Klasse *Formel* stellt den Wurzelknoten dar und entspricht in der Funktionalität weitestgehend der Klasse *Ausdruck*. In der Implementierung wird *Formel* von *Ausdruck* abgeleitet, womit sie deren Funktionalität erbt.

Die Funktion `Einfügen` überprüft die Grundlinie und Normhöhe des eingegebenen Zeichens mit der gesetzten Cursorstelle. Da diese Stelle nicht eindeutig sein muß, muß anhand des Zeichens eine gegebenenfalls andere Cursorstelle berechnet werden (siehe Abschnitt 1.3.2, Seite 16ff.). Auch kann an dieser Stelle entschieden werden, ob ein bestimmtes Zeichen als Klein- oder als Großbuchstabe eingegeben wurde. Weiterhin kann aus mehreren Zeichen, die von der Erkennung als verschiedene Möglichkeiten berechnet wurden, mit Hilfe der Erkennungsgüte, Lage und Größe zum Kontext ein Zeichen ausgewählt werden. Ein Beispiel hierfür ist die Entscheidung zwischen einem Minus und einem Bruchstrich. Ein Strich, der länger als 2/3 der Höhe h des Cursors ist, was in etwa der Breite eines 'x' in der Schriftgröße h entspricht, ist ein Bruchstrich und kein Minus. An dieser Stelle der Implementierung besteht auch die Möglichkeit, mehrere unvollständige Zeichen zu einem vollständigen Zeichen zusammenzufügen, da die Entscheidung über die Kombination vom Kontext abhängt und nicht von der Erkennung entschieden werden konnte (siehe Kapitel Erkennung Abschnitt 2.3, Seite 51).

Zusätzlich zur Funktionalität von *Ausdruck* bildet *Formel* auch die Schnittstelle zum OFR-System im allgemeinen und zur Verwaltung im speziellen (siehe Abschnitt 0.1.2, Seite 5). Durch die zusätzliche Ableitung der Klasse *Formel* von *FORMEL*, der rein virtuellen Klasse zur Definition der Schnittstelle zwischen der Verwaltung und den Formeln, ist das Format der Kommunikation definiert, so daß nur noch die dort geforderte Funktionalität zur Verfügung gestellt werden muß. Da die Schnittstellenspezifikation durch die Einbindung des Editors in das bisherige OFR-System feststand, konnte bei der Entwicklung des Editors darauf geachtet werden, daß keine zusätzliche Umwandlung der Formate zwischen beiden Seiten nötig wurde. Über die Schnittstelle bekommt der Editor die Zeichen von der Erkennung und gibt nach dem Einfügen des Zeichens die Größe der entstandenen Formel der Verwaltung bekannt. Anhand der Größe aller Formeln einer Seite weiß die *Verwaltung*, welchen Formeln sie den Auftrag geben muß, sich auf dem Bildschirm zu zeichnen. Die meisten der geforderten Funktionen finden bereits in *Ausdruck* ihre Implementierung und müssen daher nur mit den entsprechenden Funktionen verbunden werden.

1.6.5 Potenz

Rechts über den Zeichen 'a'–'z', 'A'–'Z', 'α'–'ω', 'Γ'–'Ω', '0'–'9' und ')' besteht die Möglichkeit, einen Exponenten hinzuzufügen. Ein sichtbares Merkmal ist der verkleinerte Cursor, wie in Abbildung 1.9 zu sehen. Die Menge der zulässigen Zeichen, die unter dem Exponent stehen dürfen, wird von der Funktion `examine_Potenz` bestimmt. Wird rechts über einem zulässigen Zeichen ein neues Zeichen eingegeben – hierbei muß die Grundlinie und Normhöhe des neuen Zeichens der durch den Cursor vorgegebenen Grundlinie und Normhöhe entsprechen (siehe Abschnitt 1.3.1, Seite 15) – wird eine neue Instanz der Klasse *Potenz* in den Strukturbaum eingefügt und das neue Zeichen zum ersten Zeichen des Exponenten gemacht. Der Exponent ist eine Instanz der Klasse *Ausdruck*. Dieser Exponent kann wie in der Klasse *Ausdruck* beschrieben erweitert und verändert werden. Neben dem Exponenten merkt sich *Potenz* nur das Zeichen unter dem Exponenten, nicht die Basis. Dies entspricht dem Konzept des Strukturbaumes. Die Basis der Potenz ist von den Zeichen davor und dahinter abhängig und kann unter dem Einfügen

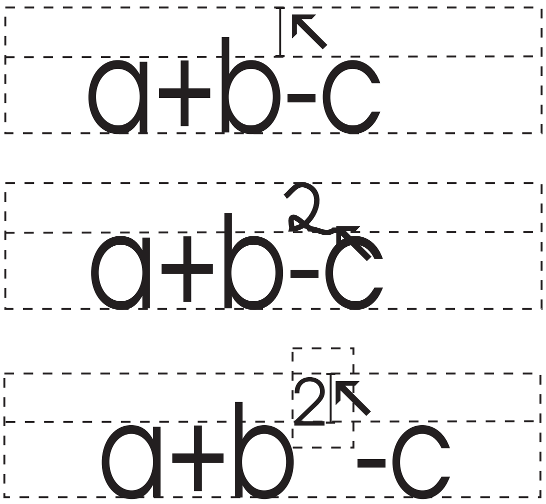


Abbildung 1.9: Einfügen eines Exponenten

weiterer Zeichen stark variieren. Nur das Zeichen direkt unter dem Exponent ist mit ihm eindeutig verbunden.

Wenn noch kein Exponent vorhanden ist, sorgt die Funktion `Cursor_setzen` der Klasse *Zeichen* für das Anzeigen des Cursors über dem Zeichen, nachdem sie von der Funktion `examine_Potenz` die Genehmigung erhalten hat. Der `Element`-Pointer des *Cursors* zeigt dabei auf das Zeichen unter dem Cursor, der `Vater`-Pointer auf eine noch nicht verwendete Instanz *Potenz* und die `Richtung` ist ohne Bedeutung. Wenn ein Zeichen als Exponent eingefügt wird, wird der Knoten, auf den der `Element`-Pointer zeigt, durch den Knoten, auf den der `Vater`-Pointer zeigt, im Strukturbaum ersetzt. Der alte Knoten, der dem Zeichen unter dem Cursor entspricht, wird wie der Knoten für den Exponent an den Knoten *Potenz* angehängt. Die Funktion `Cursor_setzen` der Klasse *Potenz* sorgt erst nach dem Einfügen in den Strukturbaum für das Setzen des Cursors.

Die Funktion `format` hat nur die Aufgabe, abhängig von der Größe des Zeichens unter dem Exponenten die Schriftgröße des Exponenten festzulegen. Die Schriftgröße des Exponenten bestimmt dessen Gesamtgröße und dadurch die Größe des umfassenden Rechtecks der Potenz. Nachdem alle Teile der *Potenz* formatiert sind, setzt sich die Formatierung an dessen Vaterknoten fort.

Beim Löschen des gesamten Exponenten wird das entsprechende Vorkommen der Klasse *Potenz* gelöscht und das Zeichen unter dem Exponenten an der gleichen Stelle wie die *Potenz* in den übergeordneten Knoten eingefügt. Das Zeichen unter dem Exponenten kann nur zusammen mit dem Exponenten gelöscht werden, da es dem Schlüsselsymbol entspricht und der Exponent zu einem zweidimensionalen Parameter ohne Schlüssel würde. Alle Einfüge- und Löschoperationen ziehen eine Formatierung des übergeordneten Knotens nach sich.

1.6.6 Index

An die Zeichen 'a'–'z', 'A'–'Z', 'α'–'ω' und 'Γ'–'Ω' lassen sich Indizes anhängen. Ein sichtbares Merkmal ist der verkleinerte Cursor rechts unterhalb des Zeichens (siehe Abbildung 1.10). Die Entscheidung, ob das Zei-

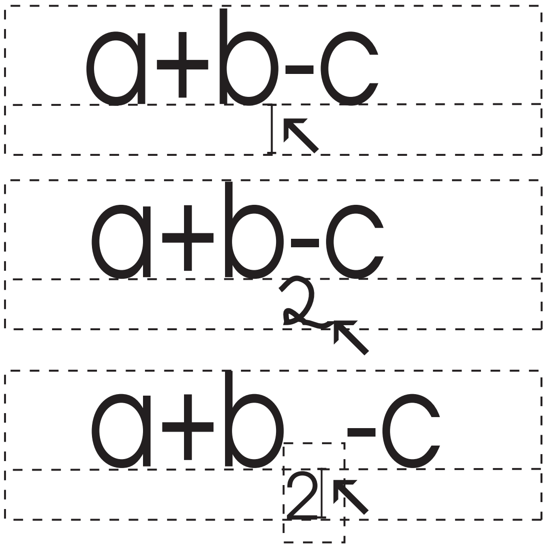


Abbildung 1.10: Einfügen eines Index

chen über dem Cursor ein zulässiges Zeichen ist, wird in der Funktion `examine_Index` getroffen. Alle weiteren Funktionen sind analog zur Klasse *Potenz* und wurden leicht modifiziert übernommen.

1.6.7 Bruch

Eine weitere Struktur ist die Klasse *Bruch*. Durch Zeichnen eines Bruchstriches wird ein Knoten *Bruch* in den vorhandenen Strukturbaum eingehängt. Dieser ist für die Teile Zähler, Bruchstrich und Nenner verantwortlich. Hier gibt es drei Möglichkeiten der Eingabe:

- Einfügen eines leeren Bruches (Abbildung 1.12), indem der Bruchstrich in der Mitte der Höhe des Cursors gezogen wird. Die vorhandenen Zeichen werden nach hinten geschoben. Die Länge des gezeichneten Bruchstriches ist hier ohne Bedeutung, da die Formatierung die Länge des Bruches an die Größe von Zähler und Nenner anpaßt. Zähler und Nenner entsprechen jeweils einer Instanz der Klasse *Ausdruck*, so daß ihre Größe einem leeren Ausdruck entspricht.

- Einfügen eines Bruches mit Nenner (Abbildung 1.13), indem ein Bruchstrich im oberen Teil des Cursors beginnt. Die nachfolgenden Zeichen werden, abhängig von der Länge des Striches, vom ersten Element hinter dem Cursor bis zum letzten noch vollständig unter dem Bruchstrich stehenden Element, zum Nenner.
- Einfügen eines Bruches mit Zähler (Abbildung 1.14), indem ein Bruchstrich gezogen wird, der im unteren Bereich des Cursors beginnt. Abhängig von der Länge des Striches werden analog zum Einfügen eines Bruches mit Nenner auch hier die nachfolgende Zeichen, abhängig von der Länge des Striches, zum Zähler.

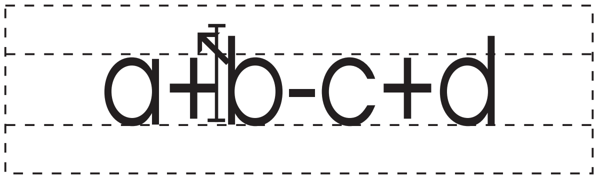


Abbildung 1.11: Ausdruck vor Einfügen eines Bruchstriches

Die Funktion `Cursor_setzen` gibt die Verantwortung, einen Cursor zu setzen an die Funktion der Kinder, des Zählers und Nenners vom Typ *Ausdruck*, weiter.

Die Funktion `format` sorgt durch ihre Formatierung für einen eindeutigen Bruch, indem sie die Länge des Bruchstriches nach Veränderungen an Zähler und Nenner deren Länge anpaßt.

Die Funktion `ausgabe` kann nur eine Linearisierung liefern, wenn der Zähler und der Nenner eine Ausgabe liefern, d.h. unter anderem nicht leer und korrekt geklammert sind. Im korrekten Fall entsteht:[3]

'(' + `ausgabe`(Zähler) + ')/(' + `ausgabe`(Nenner) + ')'

1.6.8 Ableitung

Ist das jeweils erste Zeichen des Zählers und Nenners eines Bruches ein Partialsymbol, so wird daraus eine Instanz der Klasse *Ableitung*. Wird eines dieser Partialsymbole gelöscht, wird *Ableitung* wieder zu *Bruch*.

Die meisten Funktionen der Klasse *Ableitung* können von *Bruch* übernommen werden. Die Funktion `ausgabe` muß an den besonderen Aufbau der Parameter bei Ableitungen angepaßt werden.

[3] + hier im Sinne von Zeichenkettenkonkatenation (Comskee)

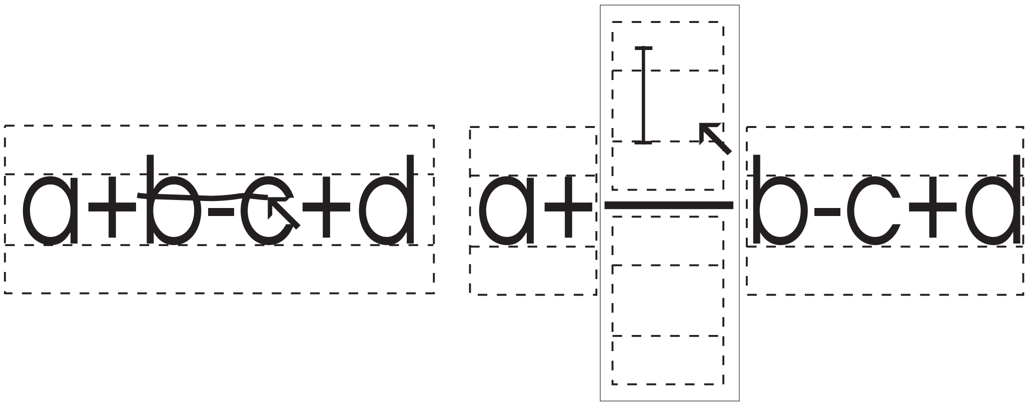


Abbildung 1.12: Einfügen eines leeren Bruches

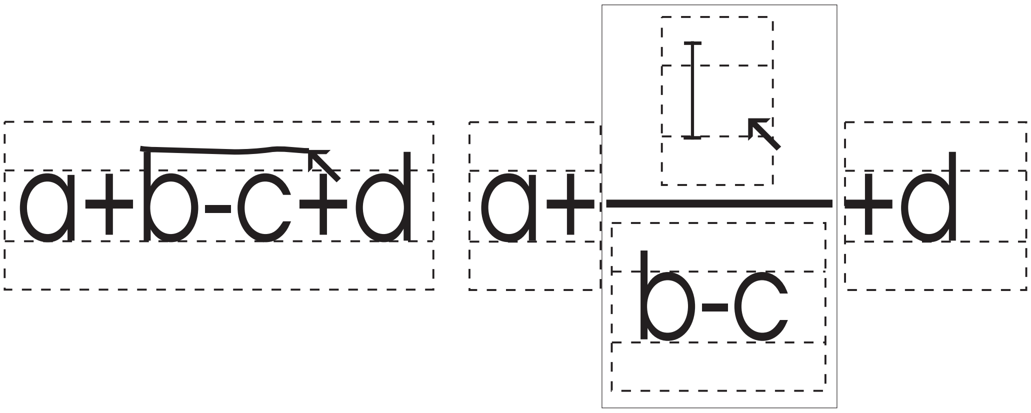


Abbildung 1.13: Einfügen eines Bruches mit Nenner

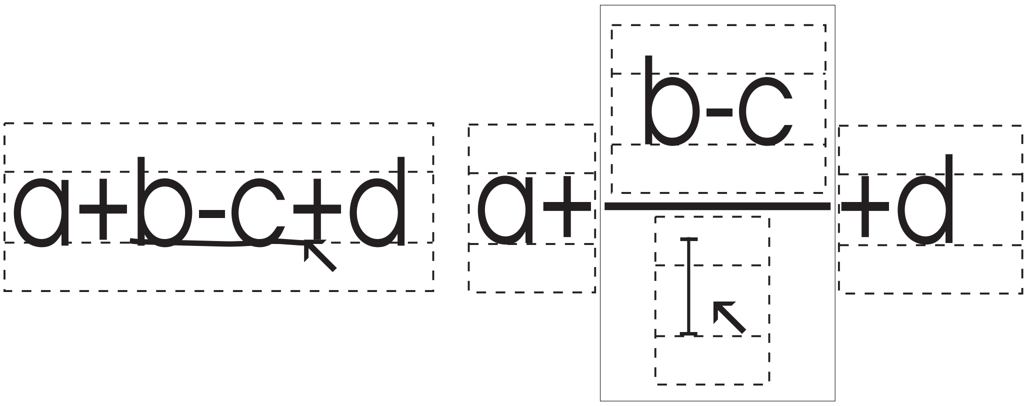


Abbildung 1.14: Einfügen eines Bruches mit Zähler

Beispiel[4]:

$$\frac{\partial^2}{\partial y \partial x} x^5 * y^2 \quad \stackrel{\text{Maple}}{\Longrightarrow} \quad \mathit{diff}(x\hat{\ }5 * y\hat{\ }2, x, y)$$

Steht die abzuleitende Funktion hinter dem Bruchstrich, ist aus der Rangfolge der Operatoren und ihrer Assoziativität (Tabelle 1.1, Seite 19) zu erkennen, daß bei fehlender Klammerung die Funktion vor der ersten Addition endet.

1.6.9 Wurzel

Mit Hilfe des Wurzelzeichens kann die Klasse *Wurzel* erzeugt werden, wobei die Länge des Zeichens den Ausdruck unter der Wurzel bestimmt. Beim Einfügen einer Wurzel in einen bestehenden Ausdruck beginnt der Ausdruck unter der Wurzel mit dem Zeichen hinter dem Cursor und endet mit dem letzten Zeichen, das noch vollständig unter dem Wurzelzeichen steht (siehe Abbildung 1.15).

Die Teilbäume des Strukturbaums, die den Zeichen unter der Wurzel entsprechen, werden zu den Kindern des Wurzelausdrucks, der vom Typ *Ausdruck* ist.

Durch die Formatierung paßt sich das Wurzelzeichen den Veränderungen des Ausdrucks unter der Wurzel an. Es vergrößert sich beim Einfügen und verkleinert sich beim Löschen von Zeichen automatisch. In Abbildung 1.15c-e sieht man die möglichen Einfügestellen am Ende der Wurzel, eine noch innerhalb (Abbildung 1.15c), die nächste bereits außerhalb der Wurzel (Abbildung 1.15d) und die dritte entspricht einer möglichen Stelle zum Erzeugen eines Exponenten noch unter der Wurzel (Abbildung 1.15e). An der ersten Stelle kann der Ausdruck unter der Wurzel erweitert werden, an der zweiten Stelle der übergeordnete Ausdruck, der damit schon dem Vater der Wurzel gehört. Dieser Vater ist für das Einfügen an dieser Stelle verantwortlich.

Das Resultat von `ausgabe` ergibt sich z.B. für MapleV folgendermaßen:

'sqrt(' + `ausgabe`(Ausdruck unter der Wurzel) + ')'

Die Erzeugung einer Linearisierung der Wurzel ist deshalb nur bei einer erfolgreichen Linearisierung des Ausdrucks unter der Wurzel möglich.

1.6.10 Summe

Durch Eingabe des Zeichens $\sum$ kann eine *Summe* erzeugt werden. Ober- und unterhalb des Summenzeichens können die jeweiligen Grenzen des Audrucks angegeben werden (siehe Abbildung 1.16). Wie bei der traditionellen

[4] Maple-spezifische Interpretation der Parameter

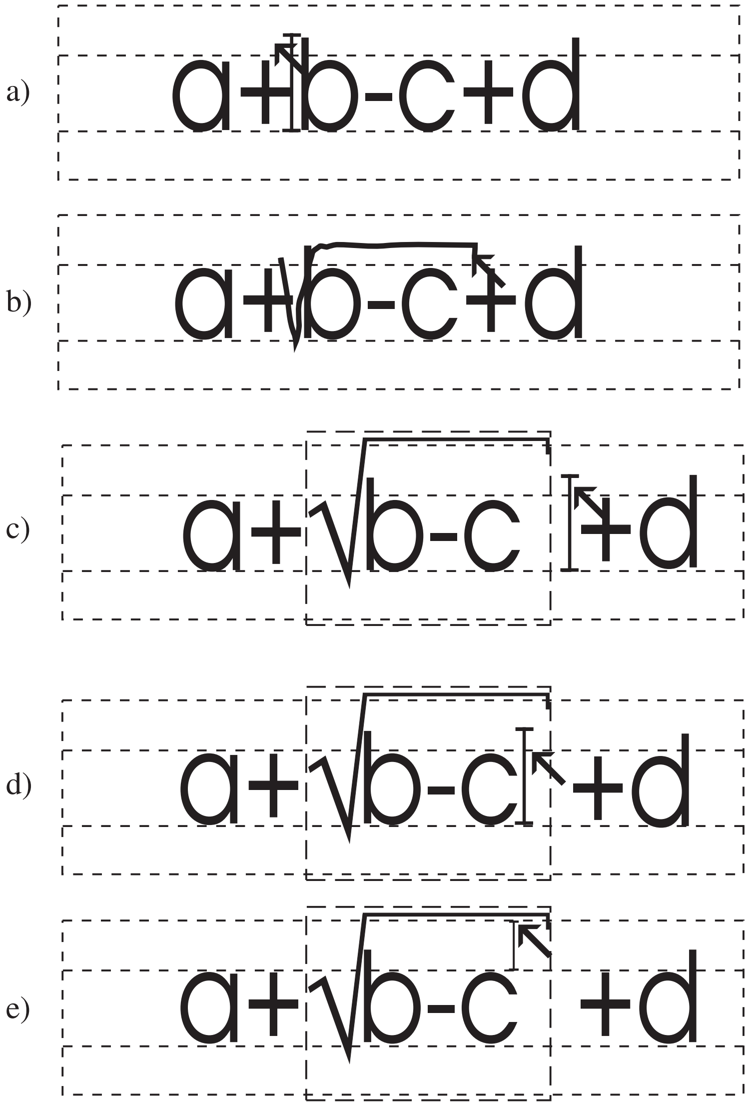


Abbildung 1.15: Einfügen einer Wurzel

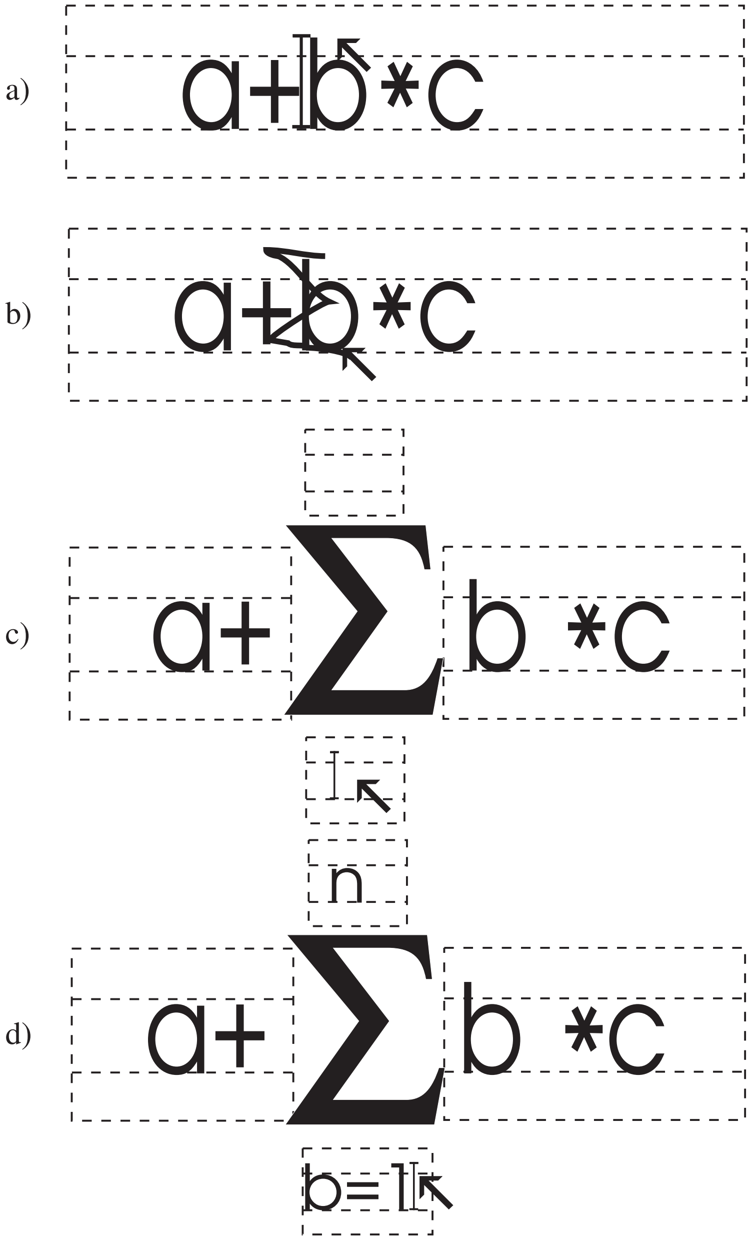


Abbildung 1.16: Einfügen einer Summe

zweidimensionalen Schreibweise der Grenzen, die in der Implementierung Instanzen von *Ausdruck* sind, ist die Größe der Schrift und damit auch der Cursor gegenüber der Grundgröße verkleinert. Der Ausdruck, über den summiert wird, ist von der Rangfolge der Operatoren und ihrer Assoziativität (Tabelle 1.1, Seite 19) abhängig und wird durch den übergeordneten Ausdruck (siehe Abschnitt 1.6.3, Seite 29) erst für die Linearisierung berechnet. Eine Linearisierung der Summe ist nur möglich, wenn

- unter dem Summenzeichen eine Zuweisung steht, durch die die Summationsvariable und deren Startwert festgelegt wird,
- über dem Summenzeichen das Ende der Summation festgelegt wird,
- der Ausdruck hinter dem Summenzeichen linearisierbar und nicht leer ist.

1.6.11 Produkt

Analog zur *Summe* kann mit $\prod$ ein *Produkt* erzeugt werden, dessen Grenzen wieder ober- und unterhalb des Produktzeichens angeben werden können.

Die meisten Funktionen können von der Klasse *Summe* unverändert, bzw. leicht modifiziert übernommen werden.

1.6.12 Integral

Durch das Integralzeichen wird die Struktur *Integral* erzeugt. Über und unter dem Integralzeichen erscheint jeweils ein Cursor, bei dem die Grenzen angeben werden können. Analog zu *Summe* und *Produkt* erscheint auch hier der Cursor kleiner (siehe Abbildung 1.17).

Die Arbeitsweise der meisten Funktionen dieser Klasse entspricht der bereits beschriebener Klassen, so daß hier nur die erweiterte Linearisierungsfunktion `ausgabe` bzw. deren Bedingungen beschrieben werden. Die Bedingungen zur Linearisierung des Integrals sind:

- Ober- und Untergrenze sind angegeben und linearisierbar (bestimmtes Integral)
  oder beide fehlen (unbestimmtes Integral)
- Integrationsausdruck und Integrationsvariable sind vorhanden und linearisierbar, wobei nur ein 'd' dazwischen steht.
- das 'd' steht auf derselben Ebene wie das Integralzeichen

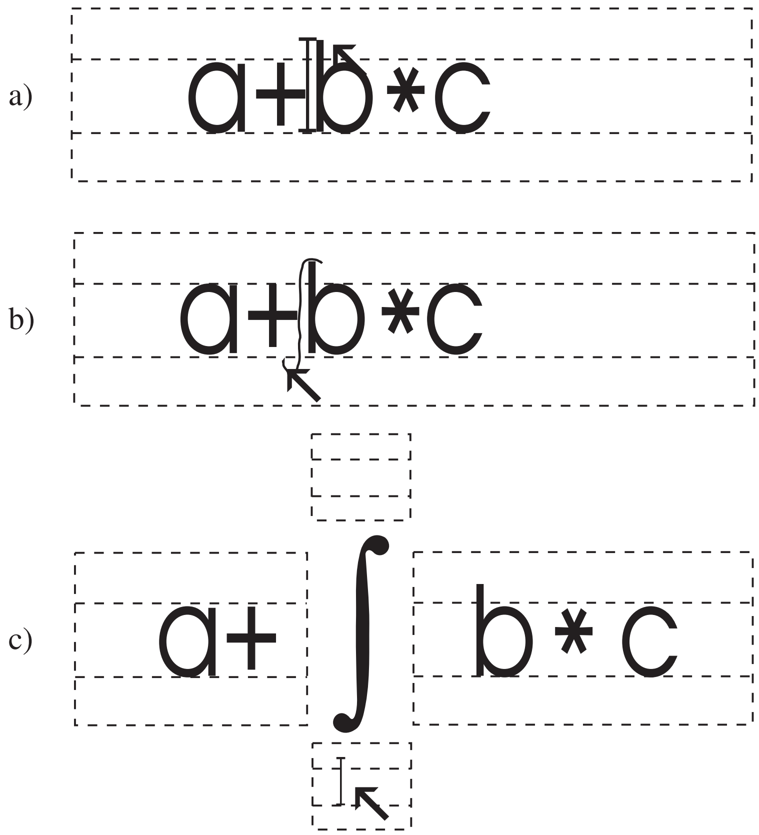


Abbildung 1.17: Einfügen eines Integrals

Die Berechnung des Integrationsausdrucks und der Integrationsvariable findet wie für alle linearen Parameter in der Funktion `parsen` der Klasse *Ausdruck* statt.

1.6.13 Matrix

Zwischen zwei korrespondierenden eckigen Klammern, zwischen denen noch keine weiteren Zeichen stehen, kann eine *Matrix* erzeugt werden. Es wurden speziell die eckigen Klammern als Schlüsselsymbole gewählt, da durch den Binomialkoeffizienten die runden Klammern mit einer bestimmten Bedeutung belegt sind. Ohne diese Einschränkung könnte der Editor eine 2×1 Matrix (Vektor) von einem Binomialkoeffizienten nicht unterscheiden, da eine Unterscheidung nur mit einem Typsystem (z.B. aus [Weber]) möglich ist.

Wie Matrizen verschiedener Größe eingegeben werden können, wird an dieser Stelle nicht weiter erläutert, da der implementierte Ansatz bereits in Abschnitt 1.3.2, Seite 18, beschrieben wurde.

Da jedes Element einer Matrix ein mathematischer Ausdruck ist, entspricht jedem Element zur Laufzeit eine Instanz der Klasse *Matrixelement*, die eine geringfügig modifizierte Variante der Klasse *Ausdruck* darstellt. Die Modifizierung besteht nur darin, alle Elemente einer Matrix zusammenzufassen, so daß die Funktionen der Klasse *Matrix* auf ein bzw. mehrere bestimmte Elemente einfach zugreifen können.

Die Funktion `format` formatiert alle Elemente einer Spalte der Matrix zentriert zu ihrer Spalte und alle einer Zeile richtet sie an einer gemeinsamen Grundlinie aus. Bei einer Veränderung, durch Hinzufügen oder Löschen, wird zuerst das jeweilige Element formatiert, mit der daraus resultierenden neuen Größe verschiebt die Matrix alle davon abhängigen Elemente. Die äußeren Klammern werden an die jeweilige Größe der Zeilen und Spalten angepaßt.

Eine sichere Methode, daß bei der Eingabe kein Element vergessen wird, ist bei der Linearisierung der Matrix darauf zu achten, daß alle Elemente angegeben sind. Fehlt ein Element, wird diese Stelle markiert und die Linearisierung abgebrochen. Viele Matrizen sind jedoch nur spärlich mit verschiedenen Werten besetzt, so daß durch die sichere Methode sehr viele gleiche Werte eingegeben werden müssen. Eine einfachere Methode wäre, da oft der Wert 0 vorkommt, alle Elemente mit diesem Wert vorzubelegen und durch die Eingabe eines Wertes den vorbelegten Wert zu überschreiben.

Beispiel: Die Einheitsmatrix, eine äußerst spärlich besetzte Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Bei der zweiten Methode müssen nur die Einsen auf der Hauptdiagonale eingegeben werden.

Da die Bereiche mit gleichen Werten meistens einer Linie und einem Dreieck, manchmal auch einem Viereck entsprechen, werden in der Mathematik nur die Eckpunkte der Bereiche angegeben und zusätzlich durch Punkte verbunden. Bei Diagonalmatrizen wird sogar bei den dreieckigen Bereichen oft nur der Wert in der rechtwinkligen Ecke angegeben.

Beispiel: symmetrische Tridiagonalmatrix

$$\begin{bmatrix} s & 1 & & & & 0 \\ 1 & s & & & & \\ & & s & \ddots & & \\ & & \ddots & s & & \\ & & & & s & 1 \\ 0 & & & & 1 & s \end{bmatrix}$$

Da aber kein Typsystem benutzt wird und das CAS die genaue Größe der Matrix benötigt, muß durch mindestens eine vollständig ausgefüllte Spalte und Zeile die Größe festgelegt werden.

Implementiert ist die sichere Methode der Überprüfung; durch die eigene Kapselung der Klasse *Matrix* kann auch eine der anderen Methoden leicht implementiert werden.

1.6.14 Binomialkoeffizient

Die Voraussetzung für die Eingabe eines Binomialkoeffizienten ist ein leeres rundes Klammerpaar. Bei einer Eingabe zwischen die beiden Klammern, deren Normhöhe kleiner als die Hälfte der Normhöhe des Cursors ist und deren Grundlinie oberhalb der Mitte der Höhe des Cursors liegt, wird der gesamte Ausdruck als Binomialkoeffizient angesehen.

1.6.15 Limes

Die Implementierung der Klasse *Limes* ist ähnlich der der Klassen *Integral*, *Summe* und *Produkt*. Einziger Unterschied ist die Verwendung eines Schlüsselwortes im Gegensatz zu den Schlüsselsymbolen der bisherigen Klassen. Die bisherige lineare Eingabe der Klasse *Ausdruck* muß jetzt noch zusätzlich überprüft werden, ob durch das neu eingefügte Zeichen ein Schlüsselwort entsteht und gegebenenfalls muß die zugehörende Struktur im Strukturbaum erzeugt werden.

1.6.16 Linearisierung

Die Linearisierung und der im nächsten Abschnitt beschriebene Pretty-Printer sind die vom jeweiligen CAS abhängigen Funktionen. Die Linearisierung steht, von den übrigen Klassenfunktionen getrennt, in einer eigenen Datei. Die restlichen Klassenfunktionen sind in jeweils einer Datei unter dem Namen der Klasse zusammengefaßt.

Die Linearisierung einer Formel wird durch die Klassenfunktion `ausgabe(`*lösung, von, bis*`)` berechnet, wobei die Berechnung bei der Klasse *Formel*, der Wurzel des Strukturbaums, beginnt.

Dabei berechnet jede Klasse (Knoten) die lineare Darstellung mit Hilfe der Kinder und hängt sie an die Zeichenkette *lösung* an. Spezialfall ist die Klasse *Ausdruck*, die bei Angabe der Parameter *von* und *bis* nur die lineare Darstellung des Teilausdrucks anhängt, der bei Element *von* beginnt und bei Element *bis* endet. Diese Teilausdrücke entsprechen den linearen Parametern von Funktionen, die damit ausgeschnitten werden und dann im Funktionsaufruf an der richtigen Stelle als Parameter eingefügt werden. Die Ergebnisse bei der Berechnung der linearen Parameter durch die Klassenfunktion `parsen` der Klasse *Ausdruck* werden in spezielle Knoten des erweiterten Strukturbaumes eingetragen (siehe Anhang B Seite 60). Da der erweiterte Strukturbaum im weitesten Sinne dem einfachen Strukturbaum entspricht, werden die Knoten in den einfachen Strukturbaum eingehängt und nur von der Linearisierungsfunktion `ausgabe` benutzt. Die Angaben in diesen Knoten sind nur so lange korrekt, wie kein Zeichen eingegeben oder gelöscht wird und werden erst für die Linearisierung wieder berechnet.

1.6.17 Pretty-Printer

Der Pretty-Printer ist die Umkehrung der Linearisierung einer Formel. Die Ergebnisse eines angeschlossenen Algebrasystems kommen als lineare Zeichenfolge. Diese Ergebnisse werden durch den Pretty-Printer in einen Strukturbaum transformiert, dessen innere Knoten aus Instanzen der im Kapitel Formeleditor beschriebenen Klassen und dessen Blätter aus Instanzen der Klasse *Zeichen* bestehen. Mit Hilfe der Parser-Generatoren „Lex" und „Yacc" und Regeln für die lexikalische und syntaktische Analyse kann ein Modul in C erzeugt werden, mit dem lineare mathematische Formeln geparst werden können. Zu jeder Regel der Grammatik können C++-Funktionen angegeben werden, die ausgeführt werden, wenn die Eingabe damit reduziert wird. Mit diesen Funktionen wird der zur Eingabe passende Strukturbaum erzeugt.

1.6.18 Transformation in das Format des Offlineparsers

Die Umwandlung einer Formel vom Format des Editors (*Formel*) in das des Parsers von F. Weigel (*Formula*) ist durch die Klassenfunktion `trafoToFormula` in jeder Klasse des Editors realisiert. Durch die Funktion wird der Formelbaum durchwandert und für jedes Blatt, also jedes sichtbare Zeichen, ein Zeichen vom Format des Parsers erzeugt. Im Format des Parsers liegen die Formeln nur als Kette von Zeichen vor, wobei die Reihenfolge der Zeichen beliebig ist und einem Zeichen ein durch das verwendete Alphabet festgelegter Zeichencode, die Koordinaten der linken oberen Ecke, die Höhe und Breite entsprechen.

Die umgekehrte Transformation, vom Parser zum Editor, läuft über die Linearisierung durch den Parser und das anschliessende Parsen durch den Pretty-Printer des Editors.

1.6.19 Erweiterung

Die zur Verfügung gestellten Funktionen stellen einen in sich abgeschlossenen Bereich der Mathematik dar, der um weitere Funktionen bestimmter Bereiche vergrößert werden kann.

Der Editor kann durch weitere Klassen erweitert werden, wobei folgende Dinge beachtet werden müssen. Die Bedeutung weiterer Einfügestellen muß durch die Lage und Größe des Cursors, durch bestimmte Schlüsselsymbole oder Schlüsselwörter festgelegt sein. Neue Einfügestellen, die nur durch ihre Lage an bestimmten Stellen entstehen, werden wie bei Potenz und Index in der Funktion `Cursor_setzen` der Klasse *Zeichen* festgelegt. Die weitere Implementierung der Funktionen der neuen Klasse kann bei den Klassen *Potenz* und *Index* abgeschaut werden.

Zusätzliche Schlüsselsymbole in der linearen Eingabe, die weitere Eingabebereiche für die zweidimensionalen Parameter der zugehörigen Funktion bereitstellen, werden durch eine zusätzliche Abfrage des entsprechenden Schlüsselsymbols in der Funktion `einfügen` der Klasse *Ausdruck* erreicht. Die Überprüfung ganzer Schlüsselwörter findet in der Funktion `format` der Klasse *Ausdruck* statt.

Eine Erweiterung des Editors wären z.B. zusätzliche Symbole über Buchstaben wie Vektorpfeile oder die von Mathematikern oft benutzte Tilde und das Dach.

Beispiel:

$$\vec{a} \qquad \tilde{a} \qquad \hat{a}$$

Eine Realisierung wäre durch die Kombinierung der beiden Zeichen zu einem Zeichen direkt nach der handschriftlichen Eingabe möglich, so daß der Editor die Zeichen als ein einziges Zeichen behandelt. Da das CAS aber nur ASCII-Zeichen versteht, muß eine eindeutige lineare Kodierung erfolgen wie z.B. $\tilde{a}$ als *a_tilde*. Die Beispielkodierung ist jedoch nur eindeutig, wenn der Unterstrich als Zeichen in Variablen- und Funktionsnamen nicht vorkommt.

1.7 Speicherverbrauch und Zeitkomplexität

Da nur für eingegebene Schlüssel die Unterbäume des Strukturbaumes zur Laufzeit des Editors dynamisch erzeugt werden, wird nur Speicherplatz linear in der Anzahl der Zeichen einer Formel benötigt. Da jedes eingegebene Zeichen direkt in den Strukturbaum eingehängt wird und danach im schlimmsten Fall alle Knoten des Strukturbaumes neu formatiert werden müssen, ist die benötigte Rechenzeit linear in der Größe des Baumes bzw. linear in der Anzahl der Zeichen. Die gleiche Rechenzeit entsteht auch bei der Linearisierung der Formel, da der Baum einmal ganz durchwandert werden muß.

1.8 Ist das Modell auch für andere Bereiche möglich?

Steht ein Computer ohne Handschrifteingabe zur Verfügung, kann die Cursorsteuerung, d.h. die Funktion `Cursor_setzen`, so erweitert werden, daß die Bewegung des Cursors mit den Cursortasten gesteuert werden kann. Die Eingabe der Zeichen erfolgt mit Hilfe der Tastatur, wobei für die wenigen, nicht auf der Tastatur befindlichen Zeichen z.B. die Funktionstasten belegt werden können.

Die im folgenden Beispiel verwendete Schreibweise chemischer Formeln weist viel weniger *sichtbare Strukturen* auf, als sie bei mathematischen Formeln zu finden sind.
Beispiel:

$$\mathrm{Fe}_2^{+2}\mathrm{Cr}_2\mathrm{O}_4$$

Mit den leicht modifizierten Klassen *Ausdruck* für die lineare Eingabe, *Potenz* für das Hochstellen von Zeichen und *Index* für das Tiefstellen können chemische Formeln eingegeben werden.

Kapitel 2

Erkennung

2.1 Allgemein

Dieses Kapitel befaßt sich mit der Erweiterung der bestehenden Online-Erkennung handgeschriebener Zeichen. Zuerst folgt eine Beschreibung der Erkennung des bestehenden Systems.

Die Erkennung besteht aus den folgenden Bereichen:

- Stiftkoordinaten
- Segmentierung der Koordinaten
- Vorverarbeitung
- Neuronales Netz
- Nachbearbeitung

2.2 Beschreibung des bestehenden Systems

2.2.1 Stiftkoordinaten

Die Eingabe, die der Stift liefert, besteht aus einem Strom von Koordinatenpaaren

$$(x, y) \in \begin{cases} \mathbb{N}^2 & \text{Stift schreibt} \\ (-1, -1) & \text{Stift abgehoben} \end{cases}$$

Bei dem Compaq Concerto besteht ein Zeichen aus ca. 20 bis 140 Koordinatenpaaren.

2.2.2 Segmentierung

Die Aufgabe der Segmentierung ist die Aufteilung des Koordinatenstroms in Teile, im weiteren Segmente genannt. Die Segmente entsprechen den Portionen, die das Neuronale Netz erkennen soll.

Die Stiftkoordinaten, die zwischen dem Aufsetzen und Abheben des Stiftes entstehen, werden zusammengefaßt und im weiteren als eine *Linie* bezeichnet. Ein *Linie* bildet die kleinste Einheit der Segmentierung und kann sogar aus nur einem einzigen Stiftkoordinatenpunkt bestehen. Als erster Gedanke für eine Partitionierung der *Linien* könnte versucht werden, alle Linien eines Zeichens in einem Segment zusammenzufassen. Daß bei den mathematischen Formeln dieser Ansatz nicht zu realisieren ist, wird in den folgenden Abschnitten deutlich.

Vorbetrachtung: Schriftarten

Die Schwierigkeit, die *Linien* der Eingabe in einzelne Zeichen zu segmentieren, ist von der Schriftart abhängig. Die Eingabe kann in folgende verschiedene Schriftarten unterteilt werden, deren Schwierigkeitsgrad in Bezug auf die Aufteilung in vollständige Einzelzeichen von oben nach unten zunimmt (aus [TapSueWak]):

- Schreiben in vorgezeichneten Kästchen
- Blockschrift mit diskreten Zwischenräumen
- Blockschrift mit geringem Zwischenraum
- gebundene Schreibschrift

Die Aufteilung reicht von der einfachsten Segmentierung, beim Schreiben in vorgezeichneten Kästchen, bis zur *internen Segmentierung* von gebundener Schreibschrift in Abhängigkeit von bereits Erkanntem, da mehrere Zeichen aus einer Linie bestehen können. Beim Schreiben von mathematischen Formeln wird Blockschrift mit geringem Zwischenraum verwendet, bei der eine interne Segmentierung nötig ist.

Wie sind Blockschriftzeichen aufgebaut?

Nach obiger Vorbetrachtung stellt sich die Frage aus welchen Teilen die Blockschriftzeichen des verwendeten Alphabets (siehe Anhang A) bestehen. Durch die genaue Untersuchung der Zeichen können sie folgendermaßen charakterisiert werden:

1. eine Linie entspricht maximal einem Zeichen
2. ein Zeichen kann aus
   (a) einer Linie
   (b) sich schneidenden[1] Linien
   (c) sich nicht schneidenden Linien und
   (d) aus deren Kombinationen

   bestehen.
3. alle sich schneidenden Linien gehören zu einem Zeichen.

Die erste Forderung ist einfach zu realisieren, da die kleinste Einheit eine Linie ist. Auch die Forderungen 2a, 2b & 3 sind folgendermaßen eindeutig zu realisieren. Nachdem jede Linie mit allen anderen auf Schnitt oder Berührung getestet wurde, werden alle sich schneidenden Linien zu einem Segment zusammengefaßt. Diese Forderung kann für das gegebene Alphabet so erweitert werden, daß nicht jede Linie mit allen anderen getestet werden muß, sondern ein Zeichen aus maximal zwei aufeinanderfolgenden sich nicht schneidenden Linien besteht, die durch mindestens eine darauffolgende Linie miteinander verbunden sind. Der hier beschriebene Test entspricht der derzeitigen Implementierung. Problematischer ist dagegen Forderung 2c, da der Abstand zwischen zwei Linien eines Zeichens nicht kleiner ist als der Abstand zwischen Linien zweier Zeichen. Die Lösung dieses Problems benötigt eine interne Segmentierung und wurde im bestehenden System vom Parser berechnet. Dieser baut mit Hilfe der Syntaxanalyse des Kontextes mehrere unvollständige Zeichen zu einem vollständigen Zeichen zusammen. Auf diese Weise wurden ':', '=', 'i', 'j' erkannt.

2.2.3 Vorverarbeitung

Die Vorverarbeitung hat die Aufgabe, die Information eines Segments in eine für das Neuronale Netz besser geeignete, komprimierte Eingabe zu transformieren. Hierbei wird die in [Quapp] beschriebene Charakterisierung verwendet. Sie extrahiert bestimmte Merkmale aus einem Segment eines Zeichens.

[1] im Sinne von schneiden oder berühren

Merkmale sind z.B. das Verhältnis von Höhe zu Breite, von Breite zu Anzahl horizontaler bzw. Höhe zu Anzahl vertikaler Bewegungen und die Kontur. Details sind der Diplomarbeit [Quapp] zu entnehmen. Die eigentliche Eingabe wird so in einen Vektor $\vec{x} \in [0,1]^n$ mit $n = 42$ transformiert.

2.2.4 Neuronales Netz

Aus dem Resultat der Vorverarbeitung berechnet das Neuronale Netz mit Hilfe von gelernten (vorberechneten) Matrizen eine Ausgabe der Form

$$\vec{y} \in [0,1]^{out} \quad \text{mit } out = \text{Anzahl der Zeichen.}$$

Bei diesem Netz entspricht jedem Ausgangsneuron ein Zeichen.

Allgemein (aus [Brause] und [Quapp])

Das hier benutzte Netz ist ein „Feed-Forward-Layered Net", also ein nicht rückgekoppeltes, vollständig verbundenes Netz in unserem Fall mit drei Schichten (siehe Abbildung 2.1). Jeder Verbindung zwischen zwei Schich-

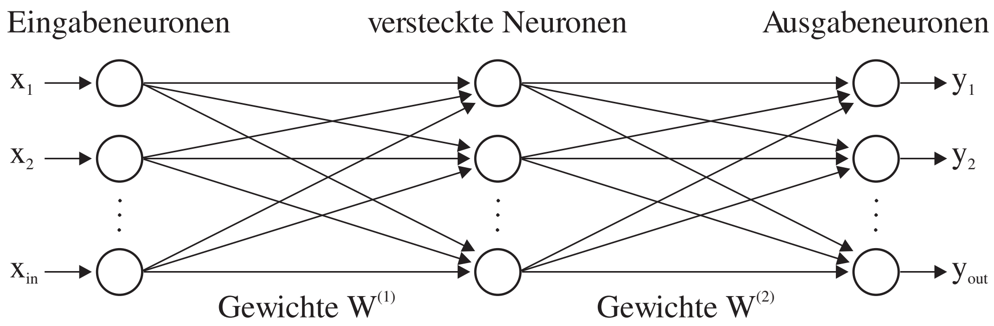


Abbildung 2.1: Feed-Forward-Layered Net mit 3 Schichten

ten entspricht ein Gewicht, welche in einer Matrix zusammengefaßt sind. Die Neuronen (siehe Abbildung 2.2) bilden eine gewichtete Summe der Form

$$y_i(x) = S(z_i(x)) \tag{2.1}$$

$$z_i(x) = \sum_k w_{ik} x_k - s_i \tag{2.2}$$

$$\text{mit} \quad S(z) = \frac{1}{1+e^{-z}} \tag{2.3}$$

wobei y_i die Transferfunktion, z_i die Aktivitätsfunktion abzüglich der Schwelle s_i, w_{ik} das Gewicht des k-ten Eingangs von Neuron i und $S(z)$ die Ausgabefunktion bezeichnet. Die Ausgabefunktion transformiert die Ausgabe der

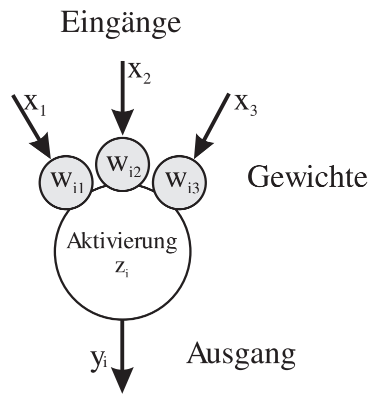


Abbildung 2.2: Modellneuron

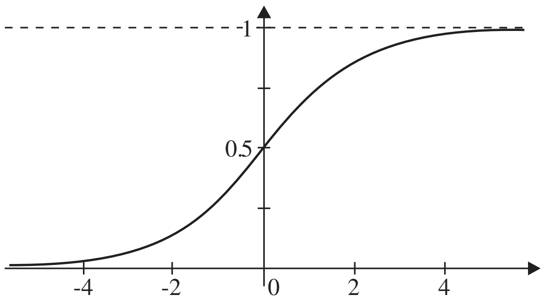


Abbildung 2.3: Fermi-Funktion

Aktivitätsfunktion in den Wertebereich der verwendeten Modellvariante. In unserem Fall bildet die Fermi-Funktion (siehe Abbildung 2.3) in den Wertebereich der reellen Zahlen zwischen 0 und 1 ab.

Durch die folgende Umstellung

$$\begin{array}{rrcll} & x & \mapsto & x' & = & (x_1, \ldots,\ x_n,\ 1)^T \\ \text{und} & w_i & \mapsto & w'_i & = & (w_{i1}, \ldots,\ w_{in},\ -s)^T \end{array}$$

kann Funktion 2.2 auch als

$$\begin{array}{rrcll} & z_i(x') & = & w_i'^T x' & \text{(Skalarprodukt)} \\ \text{bzw.} & z(x') & = & \mathbf{W}x' & \text{(Matrix-Vektor-Multiplikation)} \end{array}$$

geschrieben werden. Hierbei stellt w'_i den Vektor, gebildet aus der i-te Zeile der Matrix $\mathbf{W}$, dar.

Bevor das Neuronale Netz richtig arbeitet, muß es in einer ersten Phase sein Wissen lernen. Während des Lernvorgangs werden Trainingsmuster an den Eingang des Netzes angelegt und durch die aktuellen Gewichtsmatrizen $\mathbf{W}^1(t)$ und $\mathbf{W}^2(t)$ eine Ausgabe berechnet. Danach wird bei dem hier verwendeten Backpropagation-Algorithmus der quadratische Fehler zwischen dieser berechneten Ausgabe und der gewünschten Ausgabe minimiert, indem durch das Gradientenverfahren, das ein lokales Minimum findet, zuerst die Gewichtsmatrix $\mathbf{W}^2(t)$ über die Ausgabeschicht und danach die Gewichtsmatrix $\mathbf{W}^1(t)$ über die versteckte Schicht angepaßt wird. Bei jeder Runde der Lernphase, von $\mathbf{W}(t)$ nach $\mathbf{W}(t+1)$, werden die Änderungen der Gewichtsmatrizen nach vorne propagiert, was dem Verfahren den Namen Backpropagation gab. Vor dem eigentlichen Lernen muß noch die gewünschte Form der Ausgabe festgelegt und die Gewichtsmatrizen mit Zufallszahlen initialisiert werden. In unserem Fall entsprach jedem Ausgabeneuron genau ein Zeichen des verwendeten Alphabets. Die Entsprechung wurde dadurch realisiert, daß die Anzahl der Ausgabeneuronen der Anzahl der Zeichen des Alphabets entsprach und daß das dem Zeichen entsprechende Neuron bei Eingabe des Zeichens als einziges maximal feuert. So wurde bei Eingabe von x_i (i-tes Zeichen des Alphabet) nur das i-te Ausgabeneuron auf 1, alle anderen auf 0 gesetzt. Eine Runde des Lernvorgangs, in der jedes Muster der Probe einmal trainiert wird, reicht zur Approximation der gewünschten Funktion nicht aus, so daß dieser Vorgang mehrmals wiederholt werden muß.

2.2.5 Nachbearbeitung

Durch die Entsprechung eines Neurons mit einem Zeichen besteht die Nachbearbeitung allein im Suchen des maximalen Ausgangsneurons. Die Ausgabe der Erkennung ist eine Liste von Zeichen, wobei die Lage und Größe eines Zeichens durch sein umfassendes Rechteck (Einheit in Pixeln) gegeben ist.

2.3 Erweiterung des Zeichenvorrats

Im bestehenden System wird nur ein kleiner Teil an Zeichen benutzt (siehe Anhang A auf Seite 58), dieser sollte auf einen Zeichenvorrat erweitert werden, der den gesamten Bereich der Mathematik abdeckt.

Durch diese Erweiterung entstehen folgende Probleme:

- Die Segmentierung muß die Linien eines Zeichens zusammenfassen, die sich nicht schneiden, wie i j Θ Ξ ; : ? ! % $= \leq \equiv \geq \subseteq$
- Ähnlichkeiten zwischen Zeichen, die allein durch die Linienzüge nicht zu entscheiden sind, vorwiegend zwischen der Groß- und Kleinschreibung bestimmter Buchstaben.
  c/C, o/O, s/S, u/U, v/V, w/W, x/X, z/Z (Größenunterschied)
  p/P , y/Y (Unterschied der Lage bezüglich einer Grundlinie)
  Die Ähnlichkeit zwischen der Groß- und Kleinschreibung bestimmter Zeichen variiert, da trotz der Vorgabe der Benutzung von Blockschrift die Schriftzeichen verschiedener Schreiber stark variieren.

Diese Probleme betreffen die Segmentierung und das Parsen, das sich an die Erkennung anschließt. Die Erkennung durch das Neuronale Netz wurde in ihrer Funktionsweise nicht verändert. Es wurde nur die Größe des Neuronalen Netzes an das neue Alphabet angepaßt.

2.3.1 Zeichen aus sich nicht schneidenden Linien

Bisherige Lösung

Durch Zeichen, die aus sich nicht schneidenden Linien bestehen, entsteht bei der bisherigen Erkennung aus jedem Segment des handschriftlich eingegeben Einzelzeichens ein unvollständiges Zeichen, auch Teilzeichen genannt. Der Parser baut mit Hilfe des Kontextes und einem festen, von handschriftlichen Proben unabhängigen Schema die Teile zusammen. Bei dem bisherigen (kleinen) Alphabet waren das nur die Zeichen 'i', 'j', ':', '='. Durch die Erweiterung hat sich deren Anzahl vervielfacht.

Forderung an einen neuen Lösungsansatz:

Die Erkennung sollte alle möglichen vollständigen Zeichen, keine Teilzeichen, liefern. Diese Forderung ermöglicht eine strikte Trennung der schreiberabhängigen Erkennung vom schreiberunabhängigen Parsen der erkannten Zeichen zu einer Formel.

Bei der Suche nach allen möglichen Zeichen darf man sich nicht, wie in der lexikalischen Analyse häufig verwendet, auf die längste Kombination beschränken, was folgendes Gegenbeispiel zeigt.

Beispiel: Der lat. Kleinbuchstabe 'i', und der griech. Kleinbuchstabe 'ι'(iota) sind im unteren Teil identisch.

$$\text{bestehende Formel} \quad \overset{Eingabe}{\Longrightarrow} \quad \text{Ergebnis}$$

$$\frac{12}{} \qquad i \qquad \frac{1.2}{\iota}$$

Bei obiger Eingabe würde ohne Kontextinformation die längste Kombination, nämlich der lat. Kleinbuchstabe 'i', erkannt werden. Im Zusammenhang mit bestehendem Kontext können die beiden eingegeben Teile als zwei getrennte Zeichen angesehen werden, nämlich als ein Punkt und der griech. Kleinbuchstabe namens iota.

Neuer Lösungsansatz: der Erkennungsgraph

Eine Möglichkeit ist die Anordnung der von der Erkennung gelieferten Zeichen in einem gerichtetem Graphen, wie in Abbildung 2.4 zu sehen. Jedes Zeichen trägt Angaben über sein umfassendes Rechteck, seine Grundlinie, Normhöhe und Erkennungsgüte mit sich. Dabei entspricht eine mögliche Lösung der Erkennung einem Weg von der Quelle zur Senke. Anhand dieser Angaben und des Kontextes kann der Parser sich für eine Möglichkeit entscheiden.

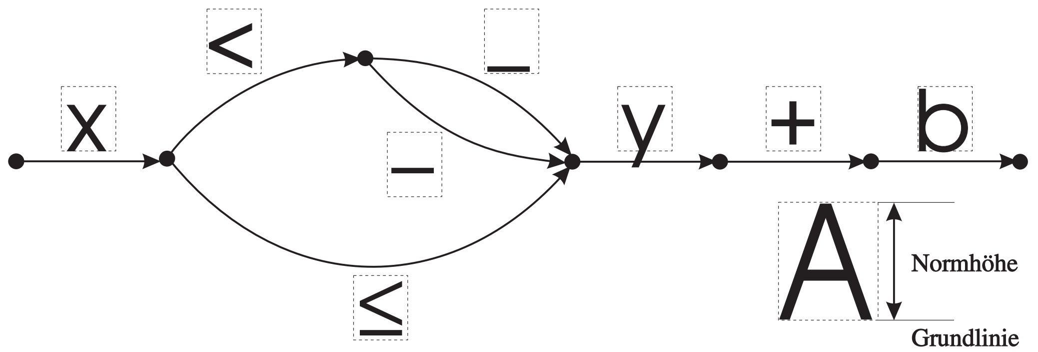


Abbildung 2.4: Ergebnis der Erkennung

Definition 2.1 :
Ein Erkennungsgraph ist ein gerichteter Graph, dessen Kanten mit möglicherweise erkannten Zeichen beschriftet sind. Ein Weg vom Start- zum Zielknoten entspricht einem Ergebnis der Erkennung. Ein Beispiel hierfür ist der Graph in Abbildung 2.4.

Da die Segmentierung durch den geringen Abstand der Zeichen nicht eindeutig entscheiden kann, welche sich nicht schneidenden Linien zu einem Zeichen gehören, sollte das Neuronale Netz nicht nur vollständige Zeichen, sondern auch unvollständige Zeichen, die durch weitere Linien ergänzt werden müssen, erkennen. Durch die Ähnlichkeit der Linien verschiedener Zeichen gibt es auch vollständige Zeichen, die durch weitere Linien zu einem anderen Zeichen ergänzt werden können. Diese vollständigen, erweiterbaren Zeichen und die unvollständigen Zeichen werden im weiteren unter dem Begriff *Teilzeichen* zusammengefaßt.

Das Neuronale Netz berechnet aus einem Liniensegment mehrere (Teil)-Zeichen verschiedener Erkennungsgüte. Diese (Teil)-Zeichen kann man, wie in Abbildung 2.5 zu sehen, in einem Netzgraphen speichern.

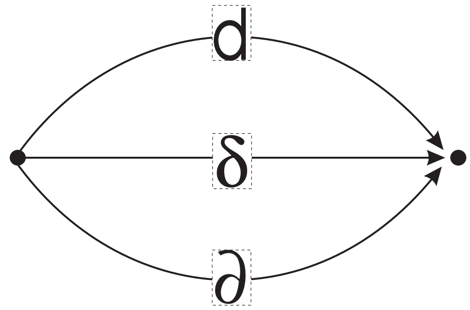


Abbildung 2.5: Ergebnis des Neuronalen Netzes als Graph

Definition 2.2:
Ein Netzgraph ist ein gerichteter Graph, dessen Kanten mit (Teil)-Zeichen beschriftet ist. Das Ergebnis, das das Neuronale Netz aus einem Segment berechnet, entspricht einem Graphen aus einem Startknoten und einem Zielknoten sowie Kanten vom Start- zum Zielknoten, die den erkannten (Teil)-Zeichen entsprechen. Analog berechnet das Netz alle anderen Segmente und hängt die erzeugten Graphen in der Reihe ihres Auftretens aneinander, in dem der Zielknoten des letzten Segments mit dem Startknoten des neuen Segments zu einem Knoten verschmilzt. Der entstandene Graph wird als kombinierter Netzgraph bezeichnet.

Anhand einer Menge von Regeln, wobei eine Regel den Zusammenbau von (Teil)-Zeichen zu einem Zeichen beschreibt, wird der kombinierte Netzgraph in einen Erkennungsgraphen transformiert.

Definition 2.3:
Eine Regel ist folgendermaßen aufgebaut:

$$\textit{Zeichen\_A} \Rightarrow \{\textit{(Teil)-Zeichen\_1}, \ldots, \textit{(Teil)-Zeichen\_n}\}$$

Die Menge der (Teil)-Zeichen, auf der rechten Seite, entsprechen Kanten im Graph, die durch eine Kante mit der Beschriftung Zeichen_A ersetzt werden.

Bei einer Implementierung könnte eine Regel als eine Instanz einer Klasse *Regel* und die Menge aller Regeln als eine Liste (siehe [LEDA]) mit Elementen vom Typ *Regel* programmiert werden. Mit Hilfe einer Funktion 'Regel paßt' und dem Algorithmus[2] auf Seite 55 kann der kombinierte Netzgraph in einen Erkennungsgraphen transformiert werden.

Die im Algorithmus auf Seite 55 verwendeten Klassenfunktionen `paßt` und `trafo` der Klasse *Regel* haben folgendes Verhalten, wobei die Variable R vom Typ *Regel* und k vom Typ *Kante* ist:

$$R.\texttt{paßt}(k) = \begin{cases} \textit{paßt nicht} & \text{wenn} \quad \text{Beschriftung von } k \text{ nicht zu } R \text{ paßt} \\ \textit{paßt} & \text{wenn} \quad \text{Beschriftung von } k \text{ paßt, aber } R \text{ noch} \\ & \qquad\quad \text{nicht vollständig erfüllt} \\ \textit{fertig} & \text{wenn} \quad R \text{ durch Beschriftung von } k \\ & \qquad\quad \text{vollständig erfüllt} \end{cases}$$

$$R.\texttt{trafo}(k) = R'$$

mit R' ist gleich der Regel $R \backslash \{\text{Beschriftung}(k)\}$, wobei zusätzlich gespeichert wurde, daß das Teilzeichen mit der Beschriftung von Kante k bereits erfüllt ist.

Eine Variante der Überprüfung einer Regel ist die Kombination eines regelbasierten Entscheiders mit einem aus Proben trainiertem Neuronalen Netz. Der regelbasierte Entscheider überprüft die Teile einer Regel nur anhand der (Teil)-Zeichen, ohne Lage- und Größeninformation. Hat der Entscheider die vollständige Übereinstimmung der Teile entschieden, überprüft ein Neuronales Netz die Lage- und Größeninformation aller (Teil)-Zeichen einer Regel und berechnet die Güte des aus den Teilen resultierenden Zeichens.

Wenn eine Regel R vollständig erfüllt ist, kann mit $R.\texttt{source}()$ der Startknoten, $R.\texttt{target}()$ der Zielknoten und $R.\texttt{info}()$ die Beschriftung für eine neue Kante im Graphen ermittelt werden.

Arbeitsweise des Algorithmus von Seite 55 Der kombinierte Netzgraph kann in hintereinanderliegende Schichten partitioniert werden. Hierbei entspricht eine Schicht einem einfachen Netzgraphen, also einer Menge von Kanten mit gleichem Start- und Zielknoten, bzw. einem von der Segmentierung geliefertem Segment. Beginnend bei der Quelle des Graphen werden alle

[2] in C++ mit Funktionen aus [LEDA]

Algorithmus zur Transformation eines Netzgraphen in einen Erkennungsgraphen

```
Regel R; Graph G;
passendeRegeln.clear();    // leere Liste
Knoten i_node = G.first_node();    // Startknoten von Graph G
while (G.outdegree(i_node) > 0)
 {
   next_i_node = G.first_adjacent_node(i_node);
   möglicheRegeln = passendeRegeln + alleRegeln;
   passendeRegeln.clear();
   vollständigeRegeln.clear();
   forall_adjacent_edges(i_edge, i_node)
     { // durchlaufe mit i_edge alle Kanten mit Startknoten i_node,
       // aber Achtung es werden Kanten gelöscht
       forall(R, möglicheRegeln)    // durchlaufe mit R alle möglichen Regeln
         {
           switch(R.paßt(i_edge))
             {
               case 'paßt':
                 // R' ist Regel R, bei der i_edge erfüllt
                 R' = R.trafo(i_edge);
                 passendeRegeln.insert(R');
               break;
               case 'fertig':
                 // letzte Forderung von Regel R wurde durch i_edge erfüllt
                 R' = R.trafo(i_edge);
                 vollständigeRegeln.insert(R');
               break;
             }
         }
       if(ZeichenTyp(G.info(i_edge)) == 'Teilzeichen')
                 G.delete_edge(i_edge);
     }
   forall(R, vollständigeRegeln)
     { // Regel R konnte angewendet werden ⇒ erzeuge neue Kante
       G.new_edge(R.source(),R.target(),R.info());
     }
   // alle Kanten ausgehend von i_node transformiert
   i_node = next_i_node;    // weiter mit nächstem Knoten
 }
```

von ihr ausgehenden Kanten einer Schicht anhand der Regeln überprüft und gegebenenfalls transformiert. Entspricht die Kante einem Teilzeichen, wird sie gelöscht. Nach vollständiger Überprüfung einer Schicht sind also keine „Teilzeichen-Kanten“ enthalten, wobei die Liste `passendeRegeln` nun alle Regeln enthält, die für die nächste Schicht noch relevant sind, und die Liste `vollständigeRegeln` alle Regeln, die vollständig erfüllt sind. Für alle Regeln, die vollständig erfüllt sind, wird eine neue Kante in den Graphen eingefügt. Startknoten der neuen Kante ist der Startknoten des ersten erfüllten Teils der Regel, analog ist der Zielknoten der neuen Kante der Zielknoten des letzten erfüllten Teils der Regel. So muß, bis die Senke des Graphen erreicht ist, Schicht für Schicht transformiert werden. Da eine Regel keine mit Teilzeichen beschriftete Kanten produziert, terminiert der Algorithmus nach maximal $O(\#\text{ Kanten} * \#\text{ Regeln})$ Schritten.

2.3.2 Groß-/Kleinschreibung

Liefert die Erkennung zu dem erkannten Zeichen auch dessen Angaben über die Grundlinie und Normhöhe, kann ein nachgeschalteter Parser abhängig vom Kontext die Entscheidung, ob ein eingegebenes Zeichen ein Groß- oder eine Kleinbuchstabe ist, lösen. Der beim Formeleditor vorhandene Cursor erleichtert die Entscheidung, da der Benutzer die gewünschte Grundlinie und Normhöhe der Eingabe sieht. War trotzdem die Entscheidung des Editors falsch, kann durch die direkte Rückkopplung zum Schreiber die Entscheidung revidiert werden.

Um Erkennungsaufgaben, die vom verwendeten Alphabet und von der Schrift des jeweiligen Benutzers abhängen, vollständig in der Erkennung zu kapseln, kann die Entscheidung, welche Zeichen sich ähneln, d.h. wie bei dem Groß-/Kleinschreib-Problem nur durch ihre Grundlinie und Normhöhe sich unterscheiden, auch durch die Erkennung gelöst werden. Treten Zeichen auf, die das Groß-/Kleinschreib-Problem aufweisen, wird die entsprechende Kante im Erkennungsgraphen verdoppelt und eine Kantenmarkierung durch die jeweils andere ersetzt. Durch die zusätzliche Speicherung der jeweiligen Grundlinie und Normhöhe der ähnlichen Zeichen im Erkennungsgraphen, braucht ein der Erkennung nachgeschalteter Parser kein konkretes Wissen über das verwendete Alphabet.

2.3.3 Fazit

Trotz der Vergrößerung des vorhandenen Alphabets und der damit verbundenen zusätzlichen Probleme bei der Erkennung steht durch die Darstellung aller Erkennungsergebnisse im Erkennungsgraphen eine Datenstruktur zur Verfügung, die eine strikte Trennung darstellt zwischen den Aufgaben der Erkennung, d.h. der Berechnung aller Besonderheiten der handschriftlich eingegebenen Zeichen des verwendeten Alphabets und der darin enthaltenen individuellen Schrift des Benutzers, und den Aufgaben des Parsers, d.h. der Kombination der Zeichen zu einer Formel mit anschließender Überprüfung der Syntax.

Anhang A

Alphabete

A.1 Erweitertes Alphabet

Lateinische Kleinbuchstaben

a b c d e f g h i j k l m n o p q r s t u v w x y z

Lateinische Großbuchstaben

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Griechische Kleinbuchstaben

$\alpha\ \beta\ \gamma\ \delta\ \varepsilon\ \zeta\ \eta\ \vartheta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ \pi\ \rho\ \sigma\ \tau\ \upsilon\ \varphi\ \chi\ \psi\ \omega$

Griechische Großbuchstaben

$\Gamma\ \Delta\ \Theta\ \Lambda\ \Xi\ \Pi\ \Sigma\ \Upsilon\ \Phi\ \Psi\ \Omega$

Ziffern

0 1 2 3 4 5 6 7 8 9

Interpunktion & Sonderzeichen

, . ; : ? ! ’ ” ‘ @ # $ % & ˆ ˜ _

Mathematische Symbole

$(\;) \; [\;] \; \{ \; \} \; \partial \; \infty \; \forall \; \exists \; \int \; \sqrt{} \; \neg$

$+ \; - \; * \; / \; \backslash \; \times \; \circ \; \cup \; \cap \; \wedge \; \vee$

$| \; < \; = \; > \; \leq \; \equiv \; \geq \; \in \; \subset \; \subseteq \; \neq \; \notin$

$\rightarrow \; \mapsto \; \Rightarrow \; \Longleftrightarrow$

$\mathbb{N} \; \mathbb{R}$

Bedingung: neue Zeichen müssen ein linear geschriebenes Äquivalent besitzen, wie z.B. griechische Buchstaben $\alpha \leftrightarrow alpha$ ...

A.2 Bisheriges Alphabet

Das bisherige Alphabet bestand aus den lateinischen Kleinbuchstaben, den Ziffern und folgenden Zeichen

$+ \; - \; * \; \int \; \Sigma \; \Pi \; < \; > \; (\;) \; [\;] \; \sqrt{} \; \alpha \; \beta \; \delta \; \mu$

Anhang B

Strukturbäume

Die folgenden Abbildungen entsprechen den Strukturbäumen der implementierten Klassen des Formeleditors (Kapitel 1, Seite 12ff.). Die gestrichelten Kanten in manchen Bäumen gehören nur zu den erweiterten Strukturbäumen und stellen zusätzliche Syntaxinformation zur Linearisierung der *linearen Parameter* der entsprechenden Funktionen dar. Der Name der Wurzel entspricht dem Namen der Klasse und die Namen der Blätter den Variablennamen der Kinder, deren Typ in Klammern darunter steht.

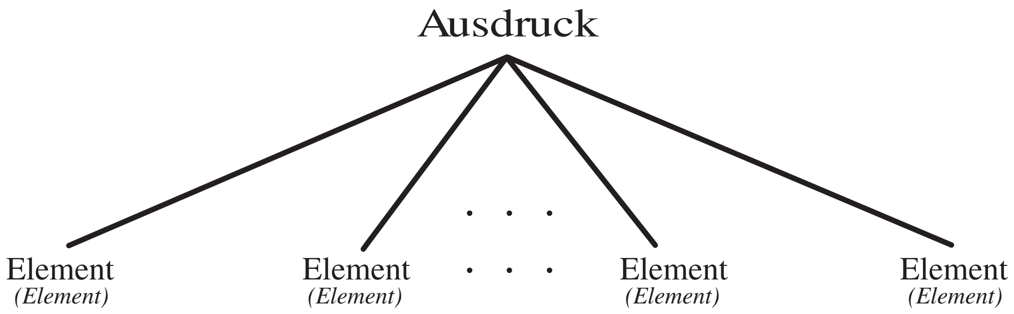


Abbildung B.1: Strukturbaum der Klasse Ausdruck

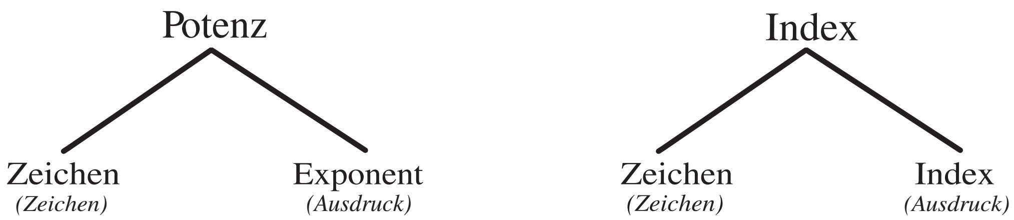


Abbildung B.2: Strukturbäume der Klassen Potenz und Index

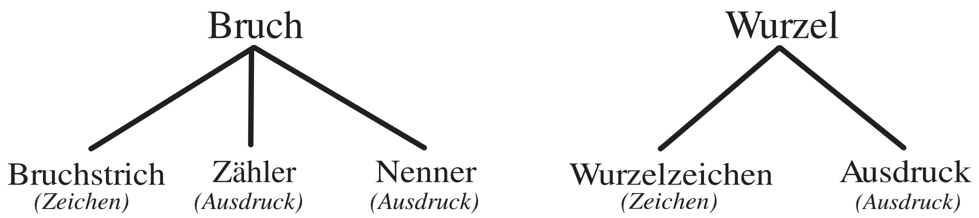


Abbildung B.3: Strukturbäume der Klassen Bruch und Wurzel

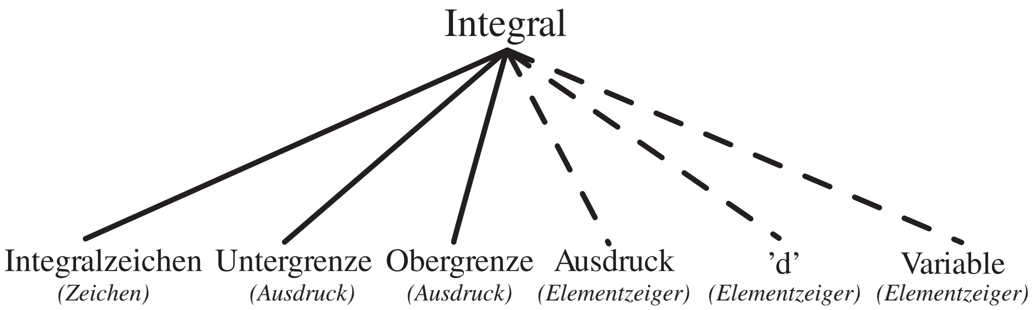


Abbildung B.4: Strukturbaum der Klasse Integral

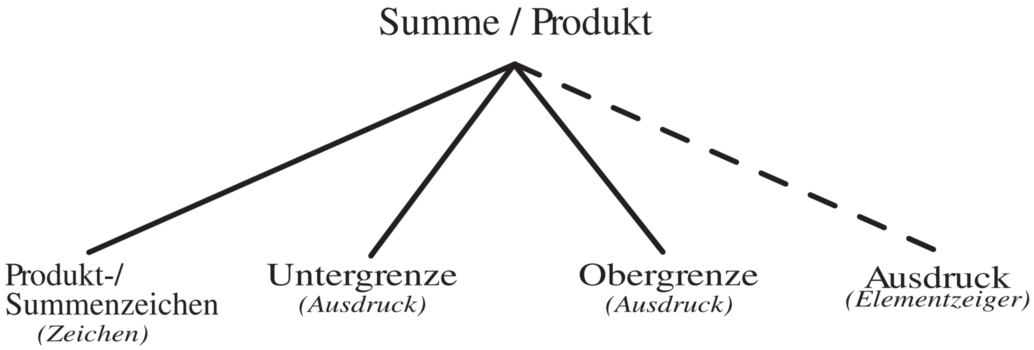


Abbildung B.5: Strukturbaum der Klasse Summe und Produkt

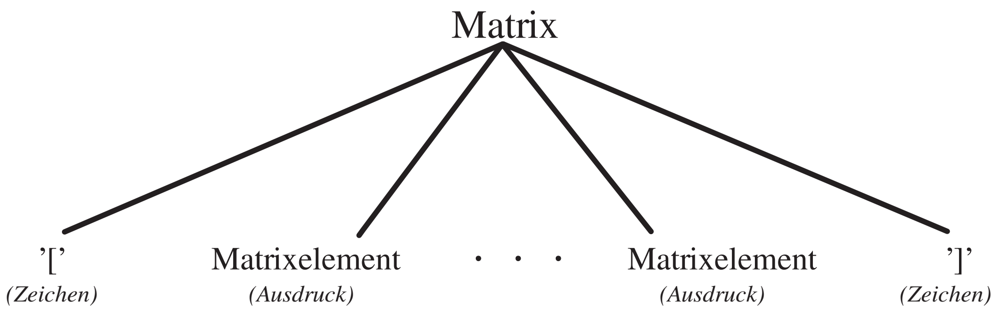


Abbildung B.6: Strukturbaum der Klasse Matrix

Literaturverzeichnis

[Brause] R. Brause,
Neuronale Netze
Teubner Verlag, 1995

[Fischer] C. Fischer,
Pencomputer – Schwachstellen und Verbesserungsansätze
Diplomarbeit, Lehrstuhl Psychologie, Universität Mannheim, 1995

[HeWi] R. Heckmann, R. Wilhelm,
Formula Layout
Technischer Bericht Nr. A 07/95, Universität des Saarlandes, 1995

[Kappes] R. Kappes,
Ein Parser für zweidimensionale mathematische Formeln
Diplomarbeit, Universität des Saarlandes, 1991

[Kölsch] J. Kölsch,
Schreibschrifterkennung durch elastischen Mustervergleich mit wörterbuchbasierter Fehlerkorrektur
Diplomarbeit, Universität des Saarlandes, 1991

[Lipp] Stanley B. Lippman,
C++ Einführung und Leitfaden 2.Auflage
Addison-Wesley, 1991

[LEDA] Stefan Näher,
LEDA, Library of Efficient Data types and Algorithms
Max-Planck-Institut für Informatik, Saarbrücken

[Marzin] R. Marzinkewitsch,
Ein Arbeitsplatz zum rechnerunterstützten handschriftlichen Rechnen mit mathematischen Formeln
Dissertation, Universität des Saarlandes, 1990

[Ritter] S. Ritter,
Handblockschriftenerkennung mit neuronalen Netzwerken
Diplomarbeit, Universität des Saarlandes, 1992

[Quapp] J. Quapp,
Ein- und Mehrschreiber-Systeme zur Handschrifterkennung mit Neuronalen Netzen
Diplomarbeit, Universität des Saarlandes, 1993

[TapSueWak] C. Tappert, C. Y. Suen, T. Wakahara,
The State of the Art in On-Line Handwriting Recognition
IEEE Transactions on Pattern Analysis and Machine Intelligence,
Vol. 12, No. 8, 1990

[Weber] A. Weber,
Type Systems for Computer Algebra
Dissertation, Universität Tübingen, 1993

[Weigel] F. Weigel,
Parsingkonzept für unscharf 2-dimensional attributierte Grammatiken
Diplomarbeit, Universität des Saarlandes, nie fertig gestellt

[WiMa] R. Wilhelm, D. Maurer,
Übersetzerbau
Springer-Verlag, 1992