Proseminar:
Programmiersysteme
WS 2003/2004
Dozent: Prof. Dr. rer. nat. Gert Smolka
Betreuer: MSc Marco Kuhlmann

Proseminararbeit:

„Dokumentensatz: TEX & LATEX "

Patrick Pekczynski
Lilienweg 11
66773 Schwalbach-Elm
E-m@il: the_phantom_dP@web.de

4.Fachsemester Informatik (**Hauptfach**)
3.Fachsemester Informationswissenschaft (**Nebenfach**)

Dieses Dokument wurde mit LATEX2ε (Version: <2001/06/01>)gesetzt.

27. März 2004 12:32

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

> [...] TEX, a new typesetting system intended for the creation of beautiful books — and especially for books that contain a lot of mathematics
>
> — DONALD KNUTH, *The TEXbook* (1984) [1]

Ob in Informatik, Mathematik oder Physik: wenn es um den Satz schöner Dokumente geht, wie *Knuth* es in diesem Zitat formuliert, die viele mathematische Formeln enthalten, so ist das von ihm entwickelte Dokumentensatzsystem TEX die erste Wahl, um Kunst und Technologie (*griech.:* τεχ) im Satz dieser Dokumente zu vereinen. Die Tatsache, dass TEX seit den Anfängen seiner Entwicklung 1978 mit einer **open source**-Lizenz veröffentlich worden ist, stellt einen Hauptgrund dar, warum sich TEX inklusive seiner Erweiterungen, wie z.B. dem Makropaket LATEX, bei Wissenschaftlern und Akademikern im mathematisch-technischen Bereich eine so große Akzeptanz gefunden hat und sich noch immer einer so großen Beliebtheit erfreut, dass es als Standard wissenschaftlicher Publikationen in diesem Bereich gilt. Obwohl TEX und auch LATEX weltweit eine große Nutzerzahl vorweisen können und sich ständig wachsender Beliebteit erfreuen, so wissen längst nicht alle, wie mächtig dieses Tool ist, dass sie in Händen halten und Tag für Tag benutzen. Dem soll in der folgenden Arbeit, zumindest in einer einführenden Form, Abhilfe geschaffen werden.

[1] [Knu84]

2 Historischer Überblick

2.1 TEX - Entwicklung

1977	**Donald Knuth**, Informatikprofessor an der Stanford University, beginnt mit der Entwicklung von TEX, seiner eigenen Dokumentensatzsprache, weil er unzufrieden war mit der Satzqualität seiner Bücher *The Art of Computer Programming*, Band 1-3
1978	Erste brauchbare Ergebnisse mit TEX78 = *alpha version*
1982	Erste stabile Version mit TEX82 = *beta version* Knuth erklärt TEX für öffentliches Eigentum [*public domain*] für nahezu jeden Rechnertyp und jedes Betriebssystem verfügbar
1983	TEX 1.0 erste TEX-Version wie es heute bekannt ist
1985	TEX 2.0 wird veröffentlicht
1990	TEX 3.0 letzte Hauptversion von TEX wird veröffentlicht
dato	Aktuelle TEX-Version = TEX 3.14159

Abbildung 1: TEX - Entwicklungsgeschichte

Da in der Fachliteratur teilweise widersprüchliche Angaben zu den Versionsnummern von TEX referenziert werden, beziehen sich die Daten in Abbildung 1 primär auf die Veröffentlichungen von *Downes* , Mitglied der LATEX3-Projektgruppe, und *Kopka* [2].

2.2 Entwicklungsumgebung

Die erste Version von TEX, TEX78, wurde in **SAIL** geschrieben, der *Stanford Artificial Intelligence Language*, einem **Algol 60**-Derivat, das in Stanford benutzt wurde [3]. Spätere Versionen von TEX wurden in **WEB** [4] geschrieben, einer an TEX und **Pascal** gebundenen Metasprache, die eine Untermenge von Pascal darstellt, erweitert um einen Präprozessor [5]. Mit der Verwendung der WEB-Sprache, die ebenfalls von Donald Knuth entwickelt worden ist, stellte Knuth im Rahmen der TEX-Entwicklung erstmals das Prinzip des **literate programming** vor. Dabei geht es darum, dass die Verwendung der WEB-Sprache zur Programmierung es ermöglicht, Programmcode zusammen mit dessen Dokumentation in einer einzigen Datei zu entwickeln.

[2] vgl. [Dow02], S.1389f, [Kop00], Abs. Vorwort
[3] vgl. Abs.B, S.51 [Gau03]
[4] CWEB für C, noweb für jede andere Programmiersprache
[5] vgl. [Eij92], Abs.33.7, S.228

Durch einen eigens für WEB entwickelten Compiler kann dann aus dem Quellcode mit Programm und Dokumentation wahlweise nur der Programmcode oder nur die Dokumentation extrahiert werden. Bei der Implementierung späterer TEX-Versionen konnte auf diese Weise TEX-Pascal-Code mit TEX-Dokumentation in einer einzigen Datei geschrieben werden, was eine effiziente Wartung des Programmcodes ermöglichte.

2.3 LATEX - Entwicklung

1985	**Leslie Lamport** stellt LATEX 2.09 fertig
1989	**Mittelbach**, **Rowley** und **Schöpf** beginnen nach Begegnung mit Lamport mit der Implementierung und Erweiterung von LATEX ⇒ Beginn des **LATEX3 Projekts**
1994	LATEX2ε offizielle LATEX-Version
1994-1999	Zweimal im Jahr [*01.06. und 01.12.*] erscheint neue LATEX-Version
2001	Seit 01.06. aktuelle LATEX2ε Version

Abbildung 2: LATEX - Entwicklungsgeschichte

Die Daten in Abbildung 2 stützen sich auf die Angaben der LATEX3-Projektgruppe gemäß http://www.latex-project.org/latex2e.html vom 27.01.2003.

3 Dokumentensatzsystem

3.1 Batch-System

Unter dem Gesichtspunkt eines *Dokumentensatzsystems* ist TEX keinesfalls mit anderen Satz- oder Textverarbeitungssystemen vergleichbar. Es handelt sich bei TEX nicht um ein interaktives **WYSIWYG**[6]-System, bei dem während des Satzvorgangs bereits eine Vorschau auf das fertige Dokument möglich ist und auf der Basis eines *Wortprozessors* arbeitet. Vielmehr lässt sich TEX als dreistufiges **Batch-System** bezeichnen, das sich aus den folgenden Komponenten zusammensetzt:

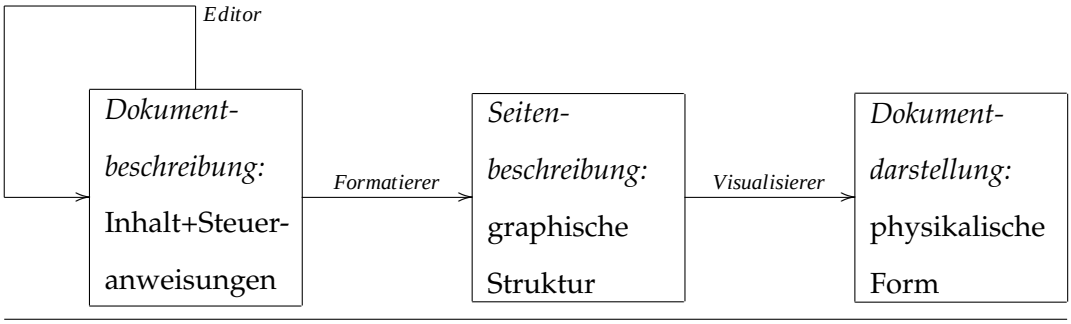


Abbildung 3: Aufbau eines Batch-Systems nach [WH96], Abb.1.2 S.8

Im ersten Teil des Batch-Systems, wird in einem Editor, z.B. *Emacs*, die Dokumentbeschreibung erstellt, d.h. die eigentliche TEX-Datei, die den Text des Dokuementes, sowie TEX-Steuerbefehle in Form von \<*Befehlssequenz*> enthält. In der zweiten Stufe baut der Formatierer bzw. der TEX-Kern aus der im Editor erstellten Quelldatei eine graphische Struktur auf, die sogenannte Seitenbeschreibung. In der dritten und letzten Phase wird anschließend aus den Satzanweisungen der Seitenbeschreibung vom Visualisierer die physikalische Form des fertigen Dokumentes, die Dokumentdarstellung. Die Bezeichnung Batch-System für die in Abbildung 3 dargestellten Phasen gründet auf der Tatsache, dass der Ablauf der einzelnen Schritte sequentiell, d.h. einer nach dem anderen erfolgen muss.

3.2 Phyisches Markup

Abgesehen von der Tatsache, dass es sich bei TEX um ein Batch-System für Dokumentensatz handelt, kann TEX darüber hinaus als **Phyische Markupsprache** bezeichnet werden. Unter **phyischem Markup**, das auch als **visuelles Markup** oder **Präsentationsmarkup** bezeichnet wird, versteht man das Formatieren von Textteilen des zu setzenden Dokumentes mit verschiedenen **low-level**-Formatierbefehlen

[6] What you see is what you get

[7] wie in Abbildung 4 dargestellt.

```
\fbox{\textbf{\Large Bsp - 1}}
\hskip 15pt {\sffamily Bsp - 2}
\newline
\textit{\tiny {Bsp - 3}}
```

Bsp - 1 Bsp -2 *Bsp - 3*

Abbildung 4: Bsp.: low-level-Formatierung

4 Minimalistischer Ansatz

Das **plattformunabhängige** Prinzip von TEX ähnlich einer **Virtuellen Maschine** aus Makrobefehlen, ermöglicht es, dass auf dem TEX-Kern Erweiterungspakete zu TEX laufen, unabhängig davon, welche TEX-Distribution verwendet wird. Diese Erweiterungspakete von TEX werden auch als **Makropakete** bezeichnet. Makropakete umfassen eine bestimmte Menge an Programm- und Befehlsdefinitionen, die ihrerseits wiederum auf die grundlegenden Befehle des TEX-Kerns zurückgreifen, auf dem sie arbeiten. Aus diesem Ansatz heraus sind im Lauf der Zeit neben den zu TEX entwickelten Zusatzprogrammen viele solcher Makropakete enstanden.

4.1 Das LATEX-Paket

Das bekannteste und am weitesten verbreitetste Makropaket, ist das LATEX-Paket. LATEX ist eine Kurzform für **Lamport TEX**, benannt nach seinem Entwickler, Leslie Lamport (vgl. Abschnitt 2.3). Am Beispiel von LATEX wird deutlich, wie die Makroerweiterung von TEX auf der Basis dieses minimalistischen Ansatzes funktioniert. LATEX als komfortables TEX-Zugangspaket legt eine neue **Layer-Struktur** von **high-level**-Befehlen [8] über dem TEX-Kern an und erstellt auf diese Weise eine benutzerfreundliche Oberfläche an Befehlen, deren Fokus sich auf die Dokumentstruktur anstatt des visuellen Markups richtet. So verdeutlicht Abbildung 5, wie in LATEX mit dem `\section{}`-Befehl ein einzelner prägnanter high-level-Befehl zur Abschnittsformatierung gleich mehrere low-level-Befehle zum visuellen Markup des Abschnitts ersetzt.

[7] vgl. [GMS00], Abs.1.3.1, S.7
[8] vgl. [GMS00], Abs.1.1.2, S.3

```
\section{}
```

```
\newcommand\section{
\@startsection
{section}{1}{\z@}
{-3.5ex \@plus -1ex \@minus -.2ex}
{2.3ex \@plus.2ex}
{\normalfont\Large\bfseries}}
```

Abbildung 5: Bsp.: high-level-Formatierung

4.1.1 Logisches Markup

Auf der Grundlage dieser Layer-Schicht von high-level-Formatierungsbefehlen lässt sich LaTeX außerdem als **generische Markupsprache** [9] bezeichnen. Unter **generischem Markup**, das auch als **logisches Markup** oder **deskriptives Markup** bezeichnet wird, versteht man das Ergänzen von Text um Informationen, mit denen das Dokument nach logischen Aspketen strukturiert wird [9]. Dabei geht es ebenso um die *Beschreibung*, welche *inhaltliche Bedeutung* einzelne Textteilen haben sollen. Es geht hierbei weniger um Angaben, in welchem Font, welcher Größe o.ä. ein Textteil gesetzt werden soll, sondern vielmehr um seine Rolle in der logischen Dokumentstruktur. So wird der Aufbau einer logischen Struktur z.B. mit folgenden Befehlen erzielt:

```
\section{}
\subsection{}
\paragraph{}
\subparagraph{}
```

```
\begin{T}...\end{T}
{\em Text}
```

Abbildung 6: Bsp.: logische Befehle - LaTeX

Wie in Abbildung 6 dargestellt, werden logische Strukturen bereits mit einfachen Abschnittsbefehlen oder Hervorhebung im Text erzielt. Ebenso können logische Elemente durch die Gruppierung von Textteilen zu Umgebungen vom Typ T gebildet werden. Das Arbeiten mit solchen *logischen Befehlen* geht hauptsächlich auf die Verfügbarkeit von Prinzipien wie Dokumentklassen und Formatvorlagedateien, den sogenannten **Style-Files**, zurück. So lassen sich im `\documentclass{}`-Befehl andere Dokumentklassen wie z.B. *book*, *report*, *article*, *letter* oder *slides* auswählen, mit denen unterschiedliche logische Strukturen umgesetzt werden. Die Tatsache, dass die tatsächliche Realisierung dieser logischen Strukturen in den einzelnen

[9] vgl. [GMS00], Abs.1.3.1, S.8

Style-Files bzw. Klassendateien unter Verwendung graphischer Formatierungen, d.h. visuellem Markup erfolgt und in dem Dokument selbst lediglich die Befehle zur logischen Gliederung erscheinen, unterstreicht die stärkere Trennung von **Inhalt** in Form von logischen Ebenen und dem **Dokumentlayout**, der graphischen Repräsentation. Auf diese Weise ist ein zentraler Vorteile von LaTeX als logischer Markupsprache, dass Änderungen am Dokument an einer zentralen Stelle erfolgen, z.B. in den Style-Files oder der **Preamble**, dem Dokumentvorspann, und nicht etwa wie in WYSIWYG-Systemen eine inkonsistente Änderung an jeder einzelnen Stelle im gesamten Dokument durchgeführt werden muss. Aufgrund der Ähnlichkeit zu anderen Markupsprachen besteht z.B. die Möglichkeit, eine Konvertierung von LaTeX-Dokumenten zu HTML-Dokumenten durchzuführen.

4.1.2 Möglichkeiten von LaTeX

Neben dem Prinzip des logischen Markup (vgl. Abschnitt 4.1.1) ist LaTeX Grundlage für zahlreiche Ergänzungspakete, die im Zuge der Publikation wissenschaftlicher Dokumente selten verwendet werden. Dennoch heben die Möglicheiten, die diese Erweiterungen zu LaTeX bieten die Mächtigkeit dieses Makropaketes und letztendlich auch die Mächtigkeit von TeX als Basis aller verwendeter Makropakete deutlich hervor. So wird LaTeX beispielsweise zur Dokumentation von Schach (vgl. Abbildung 7), Dame oder Backgammon verwendet. Außerdem bietet LaTeX die Möglichkeit zum Notensatz. So lassen sich Guitar Tabs setzen, mit denen Gitarrengriffe notiert werden, Notation von Schlagzeug und Perkussion, Klaviernoten (vgl. Abbildung 8) bis hin zu kompletten Partituren.

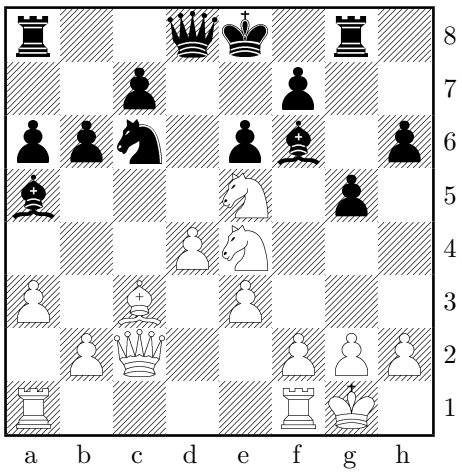


Abbildung 7: Schachdokumentation mit dem *chess*-Paket

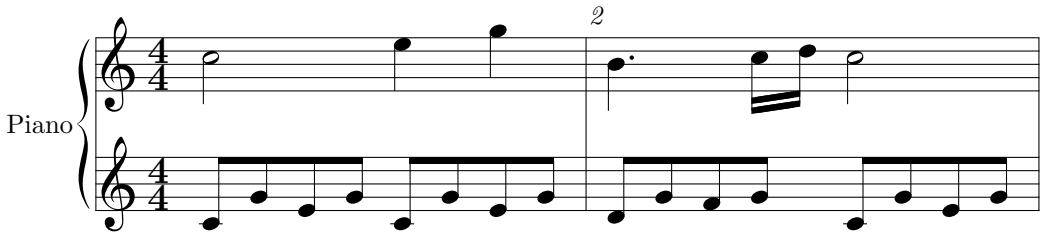


Abbildung 8: Notensatz mit LaTeX

4.2 Das ConTEXt-Paket

Eine vollwertige Alternative zum LATEX-Paket (vgl. Abschnitt 4.1) stellt das von Hans Hagen entwickelte Makropaket ConTEXt [10] dar. Es stellt, ähnlich wie LATEX ein einheitliches Konzept für Makros und Schnittstellen zur Verfügung. Auf diese Weise bietet ConTEXt die Möglichkeit, hochwertige und *interaktive* PDF-Dokumente zu erstellen. Der wichtigste Vorteil von ConTEXt im Vergleich zu LATEX besteht jedoch darin, dass ConTEXt durch einen integrierten XML-Parser dazu in der Lage ist, **XML-Code** zu interpretieren und darzustellen. Die Verarbeitung der Quelldatei (`*.tex`) erfolgt durch TEXEXEC, ein **PERL**-Script zur Erzeugung der fertigen ConTEXt-Dokumente. Bei der Verarbeitung wird zur Erzeugung von Registern, Listen oder Referenzen auf TEXUTIL, ein weiteres PERL-Script, zurückgegriffen.

4.3 Makropakete und Ergänzungsprogramme

Dass sich der minimalistische Ansatz von TEX bewährt hat, spiegelt sich in der großen Zahl an Makropaketen und Eräng zungsprogrammen wieder, die mittlerweile für TEX existieren und von **CTAN**, dem *Comprehensive TEX Archive Network* [11], archiviert und katalogisiert [12] worden sind und zum Download bereitstehen. Abbildung 9 zeigt eine Auswahl von Programme und Makropaketen, die speziell für TEX entwickelt worden sind.

`latex`	Bekanntes Makropaket für TEX (vgl. Abschnitt 4.1)
`context`	Alternative zu LATEX (vgl. Abschnitt 4.2)
`javatex`	Javaimplementierung von TEX
`xmltex`	Parser für XML-Dokumente
`bibtex`	Automatische Bibliographie erzeugen
`makeindex`	Automatischer Dokumentindex
`tex4ht`	(LA)TEX- HTML/XML - Konverter
`xl2latex`	Konvertiert Excel Tabellen zu LATEX-Tabellen
`pdftex`	Erzeugt direkt **PDF** an Stelle von **DVI**
`e-tex`	Unterstützt *rechts-nach-links*-Satz (arabische Sprachen)
`omega`	Mit 16-Bit-Unicode-Zeichen Dokumentensatz in nahezu allen Sprachen (*Arabisch, Chinesich, Sanskrit, Griechisch, ...*)

Abbildung 9: Bsp.: TEX - Erweiterungen

[10] http://www.pramga-ade.com/
[11] http://www.ctan.org/
[12] http://datamining.csiro.au/tex/full.html

5 Der TEX-Prozess

Der Gesamtablauf bei der Verarbeitung einer Quelldatei durch das TEX-System wird in Abbildung 10 skizziert:

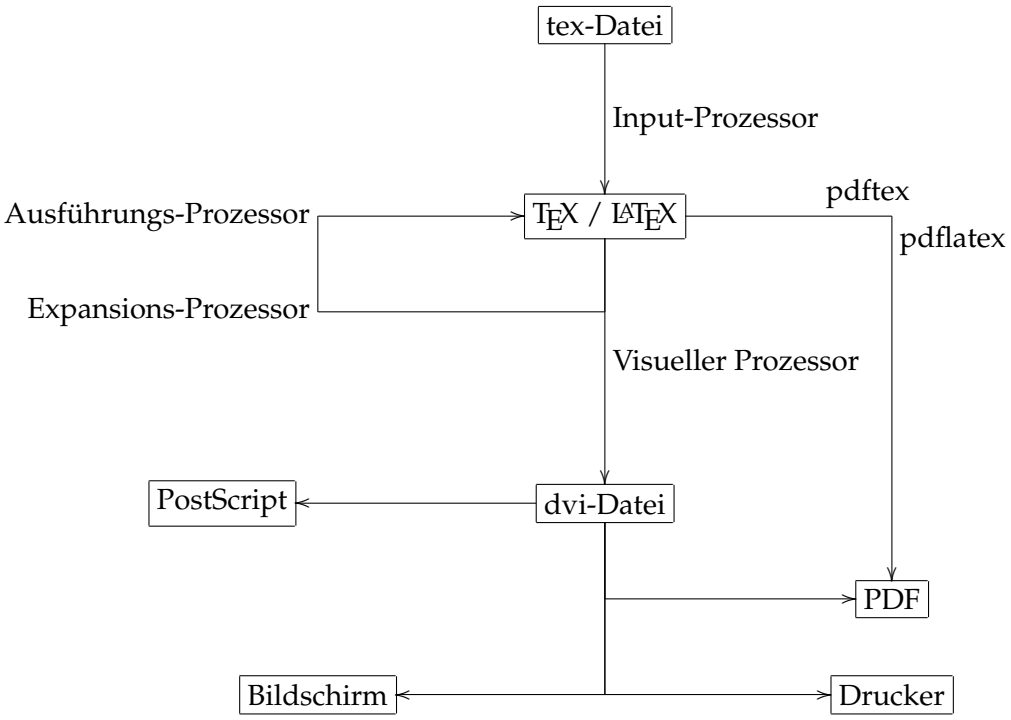


Abbildung 10: Skizze - TEX-Prozess

Die Verarbeitung einer TEX-Datei im Kern des TEX-Systems läuft auf vier verschiedenen Ebenen ab, d.h. der TEX-Prozessor selbst lässt sich in vier grundlegende Bestandteile aufteilen [13]:

1. Eingabe-Prozessor (*input processor*)
2. Expansions-Prozessor (*expansion processor*)
3. Ausführungs-Prozessor (*execution processor*)
4. Visueller Prozessor (*visual processor*)

Abbildung 11: Bestandteile des TEX-Prozessors nach [Eij92]

5.1 Eingabe-Prozessor

Im Eingabe Prozessor wird die Quelldatei, die mit dem TEX-System verarbeitet wird, zeilenweise in **Token** übersetzt, die die internen Objekte von TEX repräsentieren. Diese internen Objekte lassen sich in zwei verschiedene Arten untergliedern. Auf der einen Seite gibt es die **Token für den Satz**, d.h. die Token, aus denen der

[13] vgl. [Eij92], Abs.1.1, S.1f

zu setzende Text des Dokumentes besteht. Auf der anderen Seite werden Token, mit denen Befehlssequenzen beschrieben werden als **control sequence tokens** [14] bezeichnet. Die Ausgabe, die der Eingabe-Prozessor erzeugt besteht aus einer Liste von Token, dem sogenannten **stream of tokens** [14]. Token, die in dieser Ausgabe auftreten können, werden mit sechzehn verschiedenen Kategorien klassifiziert:

0. Escape-zeichen: Beginn eines Steuerbefehls [\]
1. Beginn einer Umgebung: [{]
2. Ende einer Umgebung [}]
3. Math shift: Formelumgebung [$]
4. Zeilen-, spaltentabulator: [&]
5. Zeilenende: [code 13 wie `\endlinechar` Parameter]
6. Parameterzeichen für Makros: [#]
7. Superscript in Mathemodus: [^]
8. Subscript in Mathemodus: [_]
9. Ignorierte Zeichen: werden von der Eingabe entfernt
10. Leerzeichen:
11. Buchstabe:
12. Andere: [Zeichen, die in keiner anderen Kategorie sind, wie z.B. Zahlen oder Satzzeichen]
13. Aktive Zeichen: fungieren als TEX-Befehle ohne ein vorangehendes Escape-zeichen
14. Kommentarzeichen: [%]
15. Ungültiges Zeichen: sollte nicht im Input auftreten

Abbildung 12: Klassifizierung der Token - [Eij92], Abs.2.3, S.8f

5.2 Expansions-Prozessor

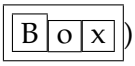
Der wesentliche Vorgang im Expansionsprozessor, der eine zentrale Rolle für TEX spielt, ist der Prozess der **Expansion**. Dabei bedeutet Expansion, dass in diesem Prozessor Tokenfolgen, die als Ausgabe des Input-Prozessors erzeugt worden sind entweder durch andere Tokenfolgen ersetzt werden oder nicht. Expansion ist demnach ein Vorgang bei dem Zeichen durch andere Zeichen ersetzt werden. Token werden in diesem Teil des TEX-Prozessors so weit expandiert, bis sie als Primitive des TEX-Kerns nicht weiter expandiert werden können oder die Expansionsreihen-

[14] [Eij92], Abs.1.1, S.1

folge etwas anderes vorgibt. Spezielle Befehle [15], durch die die Expansionsreihenfolge verändert wird, sind:

- `\expandafter`<*token1*><*token2*> ⇝ <*token1*><*Expansion von token2*>
- `\noexpand`<*token1*> ⇝ *token1* wird nicht expandiert
- `\edef` ⇝ Makrodefinition, bei der der Ersetzungstext in der Definition bereits expandiert wird

5.3 Ausführungs-Prozessor

Der Ausführungs-Prozessor erhält nicht weiter expandierbare Folgen von Token aus dem Expansions-Prozessor als Eingabe. Diese Token stellen Befehlsseqenzen dar, die nicht weiter expandiert werden können, sondern in diesem Teil des Prozesses ausgeführt werden. Darunter fallen Token, die Makrodefinitionen bilden oder eine Zustandsänderung z.B. in den internen Registern von TEX bewirken [16]. Ferner werden auch Sequenzen von Token ausgeführt, die Listen bauen. Dazu gehören Listen aus Zeichen, Abständen oder Boxen [17], den internen Enkapsulierungen von allen setzbaren Zeichen (⇒ 

). Ebenso gibt es für den Ausführungs-Prozessor einen speziellen nicht expandierbaren Befehl um die Ausführung zu verhindern [18]:

- `\relax` ⇝ Ausführung wird unterbrochen

5.4 Visueller Prozessor

Im Visuellen Prozessor werden auf die im Ausführungs-Prozessor erzeugten Listen die wichtigsten TEX-Algorithmen angewendet:

- Automatische Silbentrennung
- Umbruch von Absätzen
- Seiten- und Zeilenumbruch
- Mathematischer Text- und Formelsatz
- Erzeugung der DVI-Ausgabedatei

[15] vgl. [Eij92], Abs.1.3.2, S.3
[16] [Eij92], Abs.1.5, S.5
[17] vgl. [WH96], Abb.9.1, S.233
[18] vgl. [Eij92], Abs.12.5, S.103

5.4.1 DVI-Format

DVI (*kurz.:* ***device independent***), das zusammen mit TEX von Knuth entwickelt wurde, ist die wichtigste Ausgabe, die im TEX-Prozess produziert wird und für die zahlreiche Konvertierungsprogramme wie z.B. `dvi2ps` entwickelt wurden. [19]. Dabei bezeichnet DVI die „Maschinensprache einer abstrakten Maschine “[20] und ist mit ein Hauptgrund für die **Portabilität** von TEX und dessen Ausgabe.

5.4.2 Mathematischer Text- und Formelsatz

Der wichtigste Grund dafür, dass TEX bei Wissenschaftlern und Akademikern eine so weite Verbreitung gefunden hat ist die bemerkenswerte Qualität des mathematischen Text- und Formelsatzes von TEX. Der Algorithmus für das mathematische Formellayout [21] ist von Reinhold Heckmann, einem ehemaligen Mitglied des Lehrstuhls Prof.Dr. Reinhard Wilhelm für Übersetzerbau an der Universität des Saarlandes in **SML** reimplementiert worden [22] [23], um zu unterstreichen, dass man Dokumentensatz auch auf funktionale Basis stellen kann. Ferner bietet sich so die Möglichkeit, ähnliche Systeme auf dieser funktinoalen Basis mit derselben hohen Qualität zu entwickeln, wie TEX sie besitzt.

[19] vgl. [Knu86], Part 31, S.234
[20] [WH96], Abs.9.3, S.227ff
[21] vgl. [WH97], Abs.1, S.1
[22] Paper([WH97]): http://rw4.cs.uni-sb.de/ heckmann/papers/neuform.ps.gz
[23] Source: http://rw4.cs.uni-sb.de/ bdecker/ULI/Dokumente/dokumentenverarbeitung.html

6 Makroprogrammierung

Neben den herausragenden Leistungen von TEX als Dokumentensatzsystem [24] und als Markupsprache [25] handelt es sich bei TEX außerdem um eine Makrosprache bzw. ein Makrosystem, da TEX selbst auf 300 Basisbefehlen und 600 daraus abgeleiteten Makrobefehlen [26] aufbaut. Aus diesem Grund sind zentrale Konzepte von TEX **Makros** und die damit verbundene **Makroexpansion**.

6.1 Makro

Unter einem Makro versteht man einen „Abkürzungsmechanismus für Texte“. [27] Mit einer **Makrodefinition** wird ein Name für einen Ersetzungstext eingeführt, der weitere Makrodefinitionen, Makrobenutzungen, sowie variable Parameter enthalten kann [26]. Auf diese Weise über eine solche Makrodefinition ein **Makroname** mit einem **Ersetzungstext** assoziiert. Für TEX sind Makros definitorische Abkürzungen von Befehlssequenzen [28]. Es handelt sich dabei um parametrisierbare Kontrollsequenzen, in denen Befehle zusammengefasst werden, wie z.B:

```
\def\LpTeX{                                  \LpTeX\ ⇝ (LA)TEX
  \setbox0=\hbox{\scriptsize A}
  \wd0=294912sp
  \setbox1=\hbox{L}
  \wd1=177324sp
  (\box1\raise134500sp\box0)\TeX}
```

Abbildung 13: Beispiel - Makro: (LA)TEX

6.2 Makroexpansion

Vor dem programmiersprachlichen Hintergrund, sind Makros die beste Analogie, die TEX für Prozeduren bieten kann. Ebenso lassen sich Makroaufrufe, bei denen der Prozess der Makroexpansion (vgl. Abschnitt 5.2) eine wichtige Rolle spielt, am ehesten mit prozeduralen Aufrufen vergleichen. Makroaufrufe werden von TEX's

[24] vgl. Abschnitt 3
[25] vgl. Abschnitte 3.2 und 4.1.1
[26] vgl. [Kop00], Vorwort
[27] [WH96], Abs.7.3, S.187f
[28] vgl. [Eij92], Abs.11, S.83

Makrosystem zu einem möglichst späten Zeitpunkt expandiert. So, wie Aktualparameter eines Makroaufrufs daher ohne vorherige Expansion in den Ersetzungstext eingefügt werden, werden Makrodefinitionen ohne weitere Expansion des Ersetzungstextes an den in der Definition festgelegten Makronamen gebunden [29]. Vergleicht man Makros mit Prozeduren kann man in diesem Zusammenhang auch von einer **call-by-name**-Auswertung oder **lazy evaluation** [29] sprechen, d.h. die Parameter werden ohne Auswertung direkt übergeben. Durch Änderung der Expansionsreihenfolge (vgl. Abschnitt 5.2) lässt sich jedoch eine **call-by-value**-Auswertung erreichen.

6.3 Berechenbarkeit

In TEX können grundlegende Rechenoperation auf Registern für ganzzahlige Werte durchgeführt werden. Ebenso gibt es in TEX **if-then-else**-Anweisungen der Form `\if` <*Bedingung*><*then-Zweig*>`\else<` *else-Zweig*. Abgesehen von den grundlegenden Rechenoperationen und if-then-else-Anweisungen, lassen sich mit Hife von Makros in TEX auch Schleifen realisieren [30]:

```
\def\loop#1\repeat{
  \def\body{#1}\iterate}

\def\iterate{
  \let\next\relax
  \body
  \let\next\iterate
  \fi \next }
```

Abbildung 14: Bsp.: Schleife in TEX

Der Makrobefehl für Schleifen kann z.B. mit

```
\loop
  \if <Abbruch-Bedingung> <then-Zweig>
  \else <Anweisung wird iteriert>
\repeat
```

[29] [WH96], Abs.7.3.4, S.194
[30] vgl. [Eij92], Abs.11.8, S.92

aufgerufen werden. Durch die Verfügbarkeit der Grundrechenarten über ganzzahlige Werte, Zuweisungen, if-then-else-Anweisungen und Schleifen wird TEX eine **Turing-vollständige** Sprache.

7 Fallbeispiel: SML

Durch die Mächtigkeit, die TEX's Makrosystem bietet (vgl. Abschnitt 6.3) lassen sich aus anderen Programmiersprachen bekannte TEXniken in TEX als Makro ausdrücken. Am Beispiel einer im Rahmen dieser Arbeit erstellten naiven Listenimplementierung, sowie den auf dieser Implementierung basierenden Funktionen `fold`, `map` und partieller Applikation aus SML lässt sich zeigen, wie Mächtigkeit TEX durch sein Makrosystem ist. Es wird jedoch schnell klar, dass für die Makroimplementierung einer funktionalen Prozedur von wenigen Zeilen mehrere Zeilen Makrocode nötig sind, um dieselbe Funktionalität zu erhalten. Das folgende Fallbeispiel zeigt eine Implementierung von Listenfunktionen, die mit Hilfe der internen **integer**- und **token**-Register von TEX realisiert wurden. Die Listenimplementierung inklusive der in den Beispielen angewendeten, sowie weiteren Funktionen findet sich in Anhang B und C und ist der Arbeit in **list.sty**, sowie **functions.sty** beigefügt.

7.1 Anwendungsbeispiele

7.1.1 Falten, Map und Floatrepresentation

Für die folgenden Beispiele, die auf *fold*, *map* und eine *Floatrepresentation über Listen* zurückgreifen, wird die Beispielliste `\newlist(\examp=4,2,23,8,10,50,N)` verwendet:

- Reversieren mit foldle (über die internen Token-Register von TEX):

  `\rev(\examp)` = [50,10,8,23,2,4]

- Summe aller Elemente:

  `\foldla(\sum//0//\examp)`=97

- Dezimaldarstellung:

  `\foldla(\litonum//0//\examp)`=422381050

- Map modulo 7:

  `\map(\mapmod7//\examp)` = [4,2,2,1,3,1]

- Floatrepresentation über Listen mit integer-Werten

  Primzahlapproximation der Zahl $\pi \approx \frac{833719}{265381}$:

  [3,1,4,1,5,9,2,6,5,3,5,8,1,0,7,7,7,7,1,2,0,4,4,1,9,3,0,6,5,8,1,8,5,7,7,8,1,8,3,0,6,5,1]

7.1.2 Currying

Ebenso wie die im vorigen Abschnitt vorgestellten Funktionen lässt sich in TEX das Prinzip der partiellen Applikation (*Currying*) als Makro implementieren, wie das folgende Beispiel zeigt:

```
\def\mapmul#1#2{
    \count11=#1
    \multiply\count11 by #2
    \def\func{\number\count11} }
\def\foo(#1,#2){#1#2}
\def\mul#1{\foo(\mapmul{12},#1)}
```

Der Funktion `\mapmul` wird als erstes Argument 12 übergeben und die daraus enstandene Funktion an den Makronamen `\mul` gebunden. So ergibt der Aufruf von `\mul4` $\rightsquigarrow$ 48.

8 Ausblick

8.1 Probleme

TEX zeichnet sich in seiner Konzeption durch Qualität, Portierbarkeit, Stabilität und Verfügbarkeit aus. Trotz den bemerkenswerten Algorithmen, der herausragenden Qualität der Dokumente und der in den vorangegangenen Abschnitten vorgestellte Mächtigkeit von TEX auf den Gebieten des Markups und der Makroprogrammierung, zeichnen sich dennoch Probleme ab, die nicht vernachlässigt werden können. Aufgrund des umfangreichen Befehlssatzes von TEX (vgl. Abbildung 6) kann die Mächtigkeit der Sprache lediglich durch eine komplizierte Benutzung zugänglich gemacht werden. Abgesehen davon weist TEX in puncto Robustheit alle Probleme einer ungetypten Sprache auf. So ist beispielsweise ein Makroaufruf mit Parametern möglich, die nicht für die Makrodefinition vorgesehen waren. Die Fehlersuche ist sehr umständlich, da Debugging und Tracing lediglich über Extraschalter wie z.B. `\tracingonline` [31] ermöglicht wird. Ferner ist der Kontrollfluss lediglich

[31] vgl. [Eij92], Abs.32.1.3, S.219

schwer nachvollziehbar. Probleme treten neben dem Coding und Debugging ebenso bei den TEX-Erweiterungen auf. Sobald ausgetauschte Dateien oder verschiedenen Distributionen auf verschiedenen TEX-Erweiterungen wie z.B. verschiedenen Style-Files aufbauen, kommt es nur zu einer eingeschränkten Umsetzung des TEX-Prinzips der Portierbarkeit.

8.2 LATEX3-Projekt

Eine zukunftsträchtige Weiterentwicklung von TEX und vor allem von LATEX als Dokumentensatzsystem ist das LATEX3-Projekt. Dabei handelt es sich um ein Langzeitforschungsprojekt mit dem Ziel, die nächste Version von LATEX zu entwickeln [32]. Bis zur Fertigstellung von LATEX3 beschäftigt sich die Projektgruppe mit der Wartung der aktuellen Version von LATEX2ε.
Die wissenschaftliche Arbeitsweise der Projektmitglieder zeichnet sich durch Vorträge, Tagungen und Anwenderkonferenzen aus, mit dem Ziel, dass LATEX3 einen größeren Umfang an Dokumenten verarbeiten und *Doument Type Definitions* (DTD) für SGML-Dokumente übersetzen soll. Durch die Reimplementierung größerer Programmteile werden mit LATEX3 ferner die Voraussetzungen geschaffen, um Hypertextsysteme zu erstellen, die z.B. auf HMTL und XML zurückgreifen. So wird TEX zur Grundlage einer neuen LATEX-Version, die, ausgezeichnet durch überarbeitete und vereinheitlichte Interfaces, sowie einer erhöhten Robustheit, die **Kunst** und die **Technologie** von TEX auf eine größere Menge standardtisierter Dokumente anwenden, und so TEX's augefeilte Algorithmen und hochwertige Qualität im Dokumentensatz noch populärer machen wird.

[32] vgl. [Kop00], Abs.1.2, S.3 und [GMS00], Abs. Vorwort, S.xi

A Literatur

[Dow02] Michael Downes. *TEX and LATEX2ε*. *Notices of the AMS*, 49(11):1384–1391, 2002. [Download: pdf-file].

[Eij92] Victor Eijkhout. *TEX by Topic, A TEXnician's Reference*. Addison-Wesley, 1992. [Download: pdf-file].

[Gau03] *The (LA)TEX project: A case study of open-source software*, 2003. vorgestellt auf dem TEX Users Group Treffen 2003.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LATEX Companion*. Addison-Wesley, 1994.

[GMS00] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Der LATEX Begleiter*. Addison-Wesley, 2000.

[Knu84] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts;Menlo Park, California;Don Mills, Ontario, 1984.

[Knu86] Donald E. Knuth. *TeX - The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts;Menlo Park, California;Don Mills, Ontario, 1986.

[Kop00] Helmut Kopka. *LATEX - Einführung*, volume 1 of *LATEX*. Addison-Wesley, 2000.

[WH96] Reinhard Wilhelm and Reinhold Heckmann. *Grundlagen der Dokumentenverarbeitung*. Addison-Wesley, 1996.

[WH97] Reinhard Wilhelm and Reinhold Heckmann. *A Functional Description of TEX's Formula Layout*. *Journal of Functional Programming*, 7(5):451–485, 1997. [Download: ps-file].

– Onlinequellen mit Stand vom 27. März 2004 12:32–

B Listen-Implementierung

B.1 Liste

Eine Liste besteht aus kommagetrennten Werten, die mit dem Terminal N begrenzt werden, wobei N für **nil** steht und mit `\nil` verwendet wird.

B.2 Interaktion

```
\def\newlist(#1=#2){\newtoks#1 \noexpand#1={#2}}
\def\printlist#1{\edef\execute{\noexpand\cut\the#1}
  \ifx\the#1\nil [\ ] \else \execute\fi}
\def\cut#1,N{[#1]}
```

B.3 Cons

```
\def\cons(#1::#2){
\edef\execute{\noexpand#2={#1,\the#2}}
\execute}
```

B.4 Append

```
\def\append(#1\@#2){
\edef\execute{\noexpand\filter\the#1\the#2!\noexpand#1}
\ifx\nil\the#1 NIL1
    \ifx\nil\the#2 RELAX \relax
    \else #1=\the#2 \fi
\else \execute \fi
}
\def\filter#1,N#2!#3{#3={#1,#2}}
```

B.5 Head

```
\def\hd(#1(as)#2){
 \edef\execute{\noexpand\head\the#1!\noexpand#2}
 \if\nil\the#1 \def#2{N} \else\execute\fi}
\def\head#1,#2!#3{\def#3{#1}}
```

B.6 Tail

```
\def\tl(#1(as)#2){
 \edef\execute{\noexpand\tail\the#1!\noexpand#2}
 \if\nil\the#1 \toks0={N} \toksdef#2=0 \else\execute\fi}
\def\tail#1,#2!#3{\toks0={#2} \toksdef#3=0}
```

B.7 Foldl

```
\def\foldla(#1//#2//#3)
{
                \hd(#3(as)\x) \tl(#3(as)\xr)
                \ifx\x\nil
                \else
                    #1(\x,#2)
                    \count250=\func \countdef\buffer=250
                    \foldla(#1//\number\buffer//\xr)
                \fi }
\def\foldle(#1//#2//#3)
{               \hd(#3(as)\x) \tl(#3(as)\xr)
                \ifx\x\nil
                \else
                    #1(\x,#2)
                    \toks1=\funclist \toksdef\buffer=1
                    \foldle(#1//\buffer//\xr)
                \fi }
\def\foldl(#1//#2//#3//#4){
  \ifnum#4=0
    \foldla(#1//#2//#3)
  \else
    \ifnum#4=1
       \foldle(#1//#2//#3)
    \else
    \fi
  \fi}
```

B.8 Foldr

```
\def\foldr(#1//#2//#3//#4)
{
  \ifnum#4=0
        \rev(#3)
        \foldl(#1//#2//#3//0)
        \rev(#3)
  \else
    \ifnum#4=1
        \rev(#3)
        \foldl(#1//#2//#3//1)
        \rev(#3)
    \else
    \fi
  \fi}
```

B.9 Reverse

```
%reverse with foldl
\def\rev(#1){
  \hd(#1(as)\check)
  \ifx\check\nil \toks99={N} \toksdef\funclist=99 \relax
  \else \toks100={N} \toksdef\rnull=100 \foldl(\opcons//\rnull//#1//1)
  #1=\funclist \fi}
```

B.10 Length

```
\def\inc(#1,#2){\newcount\func\func=#2\advance\func by 1 }
\def\length(#1){\hd(#1(as)\check)
\ifx\check\nil \def\func{0} \else \foldl(\inc//0//#1//0) \fi}
```

B.11 Map

```
\def\map(#1//#2){
  \toks97={N} \toksdef\modnull=97
  \mapp(#1//#2//\modnull)
  \rev(\modnull)
  #2=\modnull}
\def\mapp(#1//#2//#3){
  \hd(#2(as)\m) \tl(#2(as)\ms)
    \toks110=\ms \toksdef\ms=110
    \ifx\m\nil \relax
    \else
       #1(\m) \cons(\func::#3)
       \mapp(#1//\ms//#3)
    \fi}
```

B.12 Split

```
\def\split(#1,#2,#3){
\newlist(\cutoff=N) \cutoff=#1 \newcount\abort
\if#30 \length(\cutoff) \div(\number\func,2,2)
       \hd(\funclist(as)\x) \abort=\x
\else \abort=#3 \fi
\toks3={N} \toksdef#2=3 \newcount\c \c=0
\hd(\cutoff(as)\x) \tl(\cutoff(as)\xr)  \cons(\x::#2)
\loop
  \ifnum\number\c=\number\abort
  \else
     \hd(\xr(as)\x) \tl(\xr(as)\xr)  \cons(\x::#2)
     \advance\c by 1
\repeat
\cutoff=\xr \rev(#2) #1=\cutoff}
```

B.13 Exists

```
\def\exists(#1,#2,#3){\count11=0 \countdef\func=11 \foldl(#1//0//#2//#3)}
```

B.14 Tabulate

```
\def\tabulate(#1,#2){\toks25={N} \toksdef\tab=25
  \htabulate(#1,0,#2,\tab) \rev(\tab)}
\def\htabulate(#1,#2,#3,#4){
\count30=#2 \countdef\n=30
\ifnum\n=0
\else
\ifnum\n>#3
\else
     #1(#2)
     \cons(\number\func::#4)
     \advance\n by 1
     \htabulate(#1,\number\n,#3,#4)
\fi\fi}
```

B.15 Compare

```
\def\listcomp(#1,#2,#3){
\count100=0 \countdef\l=100 \length(#1) \l=\number\func
\length(#2)
\ifnum\l<\number\func \def\func{0} \else
\ifnum\l>\number\func \def\func{0} \else
  \hd(#1(as)\first) \tl(#1(as)\tfirst)
  \toks50=\tfirst \toksdef\tfirst=50
  \hd(#2(as)\second) \tl(#2(as)\tsecond)
  \toks51=\tsecond \toksdef\tsecond=51
\ifx\first\nil
\else
\ifnum\first=\second \def\func{0} \listcomp(\tfirst,\tsecond,#3)
\else
     \ifnum\first#3\second \def\func{1}
     \else \def\func{0}
     \fi
\fi
\fi\fi\fi}
```

B.16 Isnull

```
\def\isnull(#1){
\hd(#1(as)\h) \tl(#1(as)\hs)
\ifx\h\nil
\else
  \ifx\h=0 \def\func{1} \isnull(\hs) \else \def\func{0} \fi
\fi}
```

B.17 Remnull

Remnull wird bei der Floatrepräsentation benötigt, um führenden Nullen zu entfernen.

```
\def\remnull(#1){
\count51=0 \countdef\removed=51
\hd(#1(as)\xn) \tl(#1(as)\xns)
\ifx\xn\nil
\else
    \ifnum\xn=0
       #1=\xns \advance\removed by 1
    \else \fi
\fi}
```

C Funktionen

C.1 Sum

```
\def\sum(#1,#2){\count11=#1 \countdef\func=11 \advance\func by #2 }
```

C.2 Litonum

litonum beschreibt die Dezimalrepräsentation einer Liste.

```
\def\litonum(#1,#2){\count11=#2 \countdef\func=11
  \multiply\func by 10
  \ifnum#1>10 \multiply\func by 10
  \else \ifnum#1=10 \multiply\func by 10 \else \fi
  \fi
  \advance\func by #1}
```

C.3 Mul

```
\def\mul(#1,#2){\count11=#1 \countdef\func=11 \multiply\func by #2 }
```

C.4 Arithmetische Funktionen für Map

C.4.1 Mapmul

```
\def\mapmul#1(#2){\count11=#2 \multiply\count11  by #1
  \def\func{\number\count11}}
```

C.4.2 Mapdiv

```
\def\mapdiv#1(#2){\count11=#2 \div(#2,#1,1)
  \hd(\divlist(as)\intdiv) \tl(\divlist(as)\divlist)
  \hd(\divlist(as)\intmod)
  \def\func{\intmod,\intdiv}}
```

C.5 Makepair

Umwandlung einer Zahl in ein Tupel aus zwei Zahlen. Benötigt bei der Floatrepräsentation.

```
\def\makepair(#1){\div(#1,10,1)
  \hd(\divlist(as)\first) \tl(\divlist(as)\rest)
  \hd(\rest(as)\snd)
  \def\func{\snd,\first}}
```

C.6 Carry-Funktionen

Funktionen zum Berechnen des Übertrags bei arithmetischen Operationen in der Floatrepräsentation.

C.6.1 Carrycons

```
\def\carrycons(#1,#2,#3){
  \hd(#1(as)\value) \tl(#1(as)\rest) \hd(\rest(as)\carry) \tl(\rest(as)\rest)
  \ifx\value\nil
  \else
    \count32=\value
    \advance\count32 by \number#2
    \cons(\number\count32::#3) #2=\carry
    \carrycons(\rest,#2,#3)
  \fi
  #1=#3
  \rev(#1)}
```

C.6.2 Ccarry

```
\def\ccarry#1{
  \map(\makepair//#1)
  \toks12={N} \toksdef\ccnull=12 \count154=0
  \rev(#1)
  \carrycons(#1,\count154,\ccnull)
  \rev(#1)}
```

C.6.3 Carryadd

```
\def\carryadd#1{
  \exists(\testforcarry,#1,0)
  \ifnum\number\func=0 \else \ccarry#1 \carryadd#1 \fi}
```

C.6.4 Testforcarry

```
\def\testforcarry(#1,#2){\count11=#2 \countdef\func=11
  \count100=#1 \ifnum\number\count100>9 \func=1 \else \relax\fi}
```

C.7 Plusone

```
\def\plusone(#1){\count11=#1 \advance\count11  by 1
  \def\func{\number\count11}}
```

C.8 Null-Funktionen

Führende Nullen hinzufügen.

C.8.1 Addnull

```
\def\addnull(#1){\count11=0 \def\func{\number\count11}}
```

C.8.2 Prefixnull

```
\def\prefixnull(#1,#2){
  \count11=0
  \loop
     \ifnum\number\count11=#2
     \else
         \cons(0::#1) \advance\count11 by 1
  \repeat}
```

C.9 Inc

```
\def\inc(#1,#2){\count11=#2 \countdef\func=11 \advance\func by 1 }
```

C.10 Div-Funktionen

Divisionsfunktionen.

C.10.1 Div

```
\def\div(#1,#2,#3){
  \count230=#1 \countdef\result=230
  \count229=0 \countdef\left=229
  \count228=0 \countdef\right=228
  \count227=#3 \countdef\abort=227
  \count226=0 \countdef\loopup=226
  \toks98={N} \toksdef\divlist=98
  \ifnum\number\result=#2
       \ifnum#2=0 \cons(div by zero::\divlist)
        \else \cons(0,1::\divlist) \fi
  \else
  \ifnum\number\result>#2
        \divleft(\result,#2,\left)
        \cons(\number\left::\divlist)
        \loop
             \ifnum\number\loopup=\number\abort
             \else
                 \divright(\result,#2,\right)
                 \cons(\number\right::\divlist)
                 \right=0
                 \advance\loopup by 1
         \repeat
  \else
        \ifnum\number\result<#2
        \cons(0::\divlist)
        \loop
            \ifnum\number\loopup=\number\abort
```

```
            \else
                  \divright(\result,#2,\right)
                  \cons(\number\right::\divlist)
                  \right=0
                  \advance\loopup by 1
         \repeat
         \else
         \fi
   \fi
\fi
\rev(\divlist)
}
\subsubsection{Divleft}
\begin{verbatim}
\def\divleft(#1,#2,#3){
\ifnum\number#1=#2 \advance#1 by -#2 \advance#3 by 1
\else
   \ifnum\number#1>#2 \advance#1 by -#2 \advance#3 by 1 \divleft(#1,#2,#3)
   \else
   \fi
\fi
}
```

C.10.2 Divright

```
\def\divright(#1,#2,#3){
\ifnum\number#1=0
\else
    \ifnum\number#1=#2 \relax
    \else
         \ifnum\number#1<#2
           \multiply#1 by 10
           \ifnum\number#1<#2 \cons(0::\divlist)
           \divright(#1,#2,#3)
           \else \divleft(#1,#2,#3) \fi
         \else
         \fi
    \fi
\fi}
```

C.11 Mod

```
\def\mod(#1,#2){ \div(#1,#2,1)
\hd(\divlist(as)\d) \tl(\divlist(as)\dtail)
\count37=\d \countdef\modarg=37 \multiply\modarg by -#2
\advance\modarg by #1 \def\func{\number\modarg}}
```

C.12 Mod für Map

```
\def\mapmod#1(#2){ \mod(#2,#1)}
```

C.13 Float

Liste als Float darstellen. Ist noch beschränkt in der Länge.

```
\def\float#1{
 \hd(#1(as)\z) \tl(#1(as)\zr) \split(\zr,\splitlist,0) #1=\zr
 \foldl(\litonum//0//\splitlist//0) \newcount\first \first=\number\func
 \foldl(\litonum//0//#1//0)
 \z.\number\first\number\func}
```

C.14 Float-Multiplikation

C.14.1 Floatmul

```
\def\floatmul#1(#2){ \map(\mapmul#1//#2) \carryadd#2}
```

C.14.2 fmul

```
\def\fmul(#1,#2){
\count123=0 \countdef\prefix=123
\toks122={} \toksdef\mult=122
\toks123={} \toksdef\store=123
\toks124=#2 \toksdef\arg=124 \length(\arg)
\toks127=#1 \toksdef\total=127
\count14=\number\func \countdef\totallength=14
\advance\totallength by -1 \mult=\total
\tabulate(\addnull,\number\totallength)  \append(\mult\@\tab)
\hd(\arg(as)\t) \tl(\arg(as)\tr) \toks126=\tr \toksdef\tr=126
\hd(\tr(as)\check)
\ifx\check\nil \floatmul\t(\mult)\store=\mult
\else
\floatmul\t(\mult)\store=\mult
\loop
   \ifnum\number\totallength=1
       \mult=\total
       \advance\prefix by 1 \advance\totallength by -1
       \tabulate(\addnull,\number\totallength)  \append(\mult\@\tab)
       \prefixnull(\mult,\number\prefix)
       \hd(\tr(as)\t)  \floatmul\t(\mult)
       \floatadd(\store,\mult) \store=\alist
   \else
           {\mult=\total
            \global\advance\prefix by 1
            \global\advance\totallength by -1
            \tabulate(\addnull,\number\totallength)
            \append(\mult\@\tab)  \prefixnull(\mult,\number\prefix)
            \hd(\tr(as)\t)
               \ifx\t\nil
               \else  \tl(\tr(as)\tr)
                     \global\toks126=\tr  \global\toksdef\tr=126 \fi
            \floatmul\t(\mult)
            \floatadd(\store,\mult) \global\store=\alist}
\repeat \fi \toksdef\funclist=123}
```

C.15 Floatadd

```
\def\floatadd(#1,#2){
\toks240={N} \toksdef\alist=240 \count240=0 \countdef\fssum=240
\count239=0 \countdef\acarry=239
\length(#1) \fssum=\number\func \length(#2) \acarry=\number\func
\ifnum\number\fssum>\number\func
      \advance\fssum by -\number\func       \advance\fssum by -1
      \tabulate(\addnull,\number\fssum) \append(#2\@\tab)
\else
      \advance\func by -\number\fssum       \advance\func by -1
      \acarry=\number\func
      \tabulate(\addnull,\number\acarry) \append(#1\@\tab)
\fi
\fssum=0 \acarry=0
\rev(#1) \rev(#2)
\toks239=#1 \toksdef\upper=239 \toks238=#2 \toksdef\lower=238
\loop
  \hd(\upper(as)\first) \tl(\upper(as)\upper)
  \toks239=\upper \toksdef\upper=239
  \hd(\lower(as)\second) \tl(\lower(as)\lower)
  \toks238=\lower \toksdef\lower=238
  \ifx\first\nil
  \else
     \fssum=\first \advance\fssum by \second \advance\fssum by \number\acarry
     { \div(\number\fssum,10,1)
       \hd(\divlist(as)\left) \tl(\divlist(as)\divlist) \hd(\divlist(as)\right)
       \global\acarry=\left \global\fssum=\right
     }
     \cons(\number\fssum::\alist)
\repeat
}
```

C.16 Floatsub

```
\def\floatsub(#1,#2){
\toks240={N} \toksdef\alist=240 \count240=0 \countdef\fssub=240
\count239=0 \countdef\scarry=239
\length(#1) \fssub=\number\func \length(#2) \scarry=\number\func
\ifnum\number\fssub>\number\func
      \advance\fssub by -\number\func       \advance\fssub by -1
      \tabulate(\addnull,\number\fssub) \append(#2\@\tab)
\else
      \advance\func by -\number\fssub       \advance\func by -1
      \scarry=\number\func
      \tabulate(\addnull,\number\scarry) \append(#1\@\tab)
\fi
\fssub=0 \scarry=0
\rev(#1) \rev(#2)
\toks239=#1 \toksdef\upper=239 \toks238=#2 \toksdef\lower=238
\loop
  \hd(\upper(as)\first) \tl(\upper(as)\upper)
  \toks239=\upper \toksdef\upper=239
  \hd(\lower(as)\second) \tl(\lower(as)\lower)
```

```
  \toks238=\lower \toksdef\lower=238
  \ifx\first\nil
  \else
     \fssub=\first
     \advance\fssub by -\number\scarry
     \edef\nfs{\number\fssub}
     \ifnum\nfs<\second
         \advance\fssub by 10
         \advance\fssub by -\second
         \scarry=1
     \else
         \advance\fssub by -\second \scarry=0
     \fi
     \cons(\number\fssub::\alist)
\repeat   \rev(#1)\rev(#2)}
```

C.17 Float-Division

C.17.1 Fdiv

```
\def\fdiv(#1,#2,#3){
\global\toks109=#1 \global\toksdef\resetfst=109
\global\toks108=#2 \global\toksdef\resetsnd=108
\toks110={N} \toksdef\fdivlist=110
\count190=0 \countdef\result=190
\global\def\reset{\toks139=\resetfst \toksdef\upper=139
                  \toks138=\resetsnd \toksdef\lower=138}
\toks140={} \toksdef\fsublist=140
\reset
\count130=0 \countdef\left=130
\count131=0 \countdef\right=131
\count132=#3 \countdef\abort=132
\count133=0 \countdef\loopup=133
\isnull(\lower)
\ifx\func=1 \cons(fdiv by zero::\fdivlist)
\else
       \listcomp(\upper,\lower,=)
       \ifnum\func=1 \cons(1,0::\fdivlist)
       \else
              \listcomp(\upper,\lower,>)
              \ifnum\func=1
                  \fdivleft(\upper,\lower,\left)
                  \cons(\number\left::\fdivlist)
                  \loop
                     \ifnum\number\loopup=\number\abort
                     \else
                        {\fdivright(\fsublist,\lower,\right)}
                        \cons(\number\right::\fdivlist)
                        \right=0  \advance\loopup by 1
                  \repeat
              \else
              \fi\fi\fi   \rev(\fdivlist)}
```

C.17.2 Fdivleft

```
\def\fdivleft(#1,#2,#3){
\hd(#1(as)\checkfst) \hd(#2(as)\checksnd)
\ifx\checkfst\checksnd
\remnull(#1) \remnull(#2) \global\toks109=#1 \global\toks108=#2\else\fi
\listcomp(#1,#2,=)
\ifnum\func=1 \floatsub(#1,#2) \reset \global\advance#3 by 1
       \global\fsublist=\alist
\else
  \listcomp(#1,#2,>)
  \ifnum\func=1
       \floatsub(#1,#2) \reset
       \global\advance#3 by 1
       \global\fsublist=\alist
       \fdivleft(\fsublist,#2,#3)
  \else
  \fi
\fi
}
```

C.17.3 Fdivright

```
\def\fdivright(#1,#2,#3){
\count43=0 \countdef\l=43
\isnull(#1)
\ifnum\func=1
\else
  \listcomp(#1,#2,<)
  \ifnum\func=1
       \rev(#1) \cons(0::#1) \rev(#1)  \remnull(#1)
       \length(#1) \l=\number\func \length(#2)
       \ifnum\number\l>\number\func
           \cons(0::#2) \toks108=#2
       \else
       \fi
       \listcomp(#1,#2,<)
       \ifnum\func=1
           \global\cons(0::\fdivlist) \global\toks110=\fdivlist
           \fdivright(#1,#2,#3)
       \else \fdivleft(#1,#2,#3)
       \fi
  \else \fi\fi}
```

C.18 Opcons

Implemtierung einer *op::*-Funktion für Token-Listen.

```
\def\opcons(#1,#2){
\edef\execute{\noexpand#2={#1,\the#2}} \execute
\toks99=#2 \toksdef\funclist=99}
```