Literate Programming

Johannes Hüsing

Vortrag bei der Essener Linux-User-Gruppe am 19.10.1999

1 Idee des Literate Programming

Den Begriff „Literate Programming“ halte ich für schwer übersetzbar. „Belesenes Programmieren“? Kaum. Nach der ursprünglichen Idee seines Erfinders Donald Knuth [5] sollte diese Art der Programmierung im Wesentlichen einen menschlichen Leser im Sinn haben, der das Programm liest, als einen Computer, der das Programm ausführt.

Einige Zitate mögen das Anliegen des Literate Programming verdeutlichen:

> I wouldn't have much respect for an aeronautical engineer who had never admired a superb airplane, nor for a structural engineer who had never studied a beautiful bridge. Yet I, like most programmers, was in roughly that position with respect to programs. That's tragic, because good writing requires good reading—you can't write a novel if you've never read one. But the fault doesn't rest entirely with us programmers: Most programs are written to be executed, a few are written to be maintained, but almost no programs are written so someone else can read them.
>
> Jon Bentley [1]

> The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reïnforce each other.
>
> Donald E. Knuth [5]

> The idea of a literate program as a text book should be extended even further. I would like to see a literate program as an (in)formal argument of the correctness of the program.
>
> Thus a literate program should be like a textbook on mathematics. A mathematical textbook explains a theory in terms of lemma and theorems. But the proofs are never formal in the sense that they are obtaind by symbol

> manipulation of a proof checker. Rather the proofs are by so called "informal rigour", i.e. by very precise and unambiguous sentences in a natural language.
>
> Eric W. van Ammers [7]

Donald Knuth führte diesen Begriff auch mit einem propagandistischen Hintergedanken ein. In den siebziger Jahren wurde der Feldzug gegen die Verwendung des Sprungs (des `goto`) unter dem Namen „strukturiertes Programmieren" gefochten. Ein Teil des Erfolgs dieser Kampagne war vielleicht auch darauf zurückzuführen, daß ein Begriff besetzt wurde. Die Gegner dieser Bewegung hätten einen schweren Stand mit dem Vorhaben gehabt, „unstrukturiertes Programmieren" zu verfechten. Ähnlich sollten die mutmaßlichen Gegner der Idee des „Literate Programming" in die mißliche Lage getrieben werden, einen „illiterate", also analphabetischen, Programmierstil zu verteidigen.

Weil auch in der englischen Sprache der Begriff „Literate Programming" neu war und mit Leben gefüllt werden mußte, werde ich im Verlauf dieses Textes keine Probleme damit haben, ihn ab und zu wörtlich ins Deutsche zu übersetzen.

2 Aus einer Quelle

Ein grundlegendes Prinzip bei dem Programmieransatz ist, daß man sich die Arbeit nicht doppelt machen soll. Ein Programm soll für den Menschen lesbar sein. Der Computer soll dabei aber nicht vergessen werden — schließlich will man etwas von ihm. Die Aufbereitung für den Rechner soll ohne größeren Aufwand vonstatten gehen.

Der Ansatz Knuths war, aus einem gemeinsamen Quelldokument eine Version zu erzeugen, die für den Menschen lesbar ist, und eine, die der Compiler oder Interpreter verarbeiten kann (siehe Abbildung 1).

Die Sprache, die Knuth erfand, nannte er WEB. Lange vor dem Gespinst aus Hyperlinks, das seit einigen Jahren den Globus umzieht, gebrauchte er die Metapher des Webens für seine Methode. Software ist für ihn nicht monolithisch, sondern aus vielen Gedankengängen zusammengesetzt. Diese Gedankengänge setzen sich für den Menschen auf andere Weise zusammen als für den Computer. Ein wesentliches Element ist daher die *Reihenfolge* der Gedankengänge. Ein Compiler wird vielleicht darauf bestehen wollen, daß zu Beginn alle im Programm verwendeten Bibliotheken genannt werden. Ein Mensch hat bis zum ersten Aufruf von Funktionen aus einer Bibliothek vielleicht schon vergessen, daß sie am Anfang definiert wurde.

Daher ermutigt WEB dazu, das Programm in handlich portionierte Splitter ("chunks") zu zerschlagen, die in die für den Menschen lesbare Reihenfolge gebracht werden. Sie müssen natürlich mit den Informationen versehen werden, wie die für den Computer zu bewältigende Reihenfolge ist. Donald Knuth nannte die Aufbereitung für den Menschen „weben" ("weave") und die für den Rechner „verstricken" ("tangle"). Die Programmiererin stellt also das Wollknäuel zur Verfügung. Der Computer übernimmt die Rollen der Oma und der Katze.

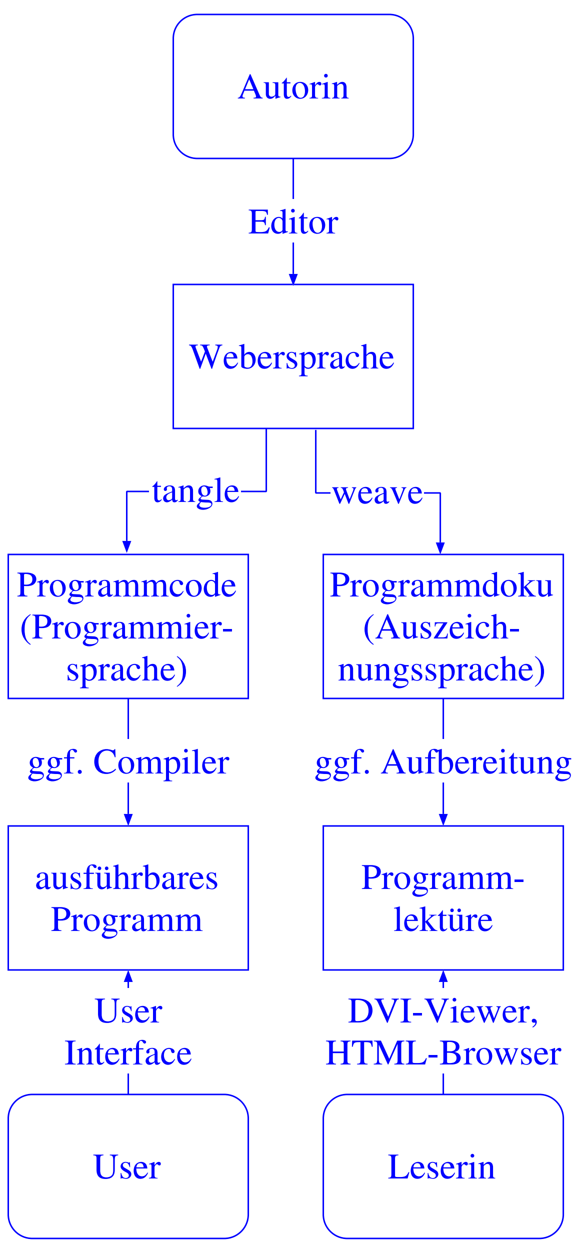

Abbildung 1: Gemeinsames Quelldokument für Computer- und Menschversion

3 Programmdokumentation: Vergleich und Beispiele

3.1 Literate Programming und Dokumentation

Literate Programming ist nicht Programmdokumentation im Sinne eines Handbuchs! Die Beschreibung, was die einzelnen Befehle eines Programms machen, ist der Auflistung der möglichen Interaktionen des Users mit der Maschine und deren Folgen manchmal unähnlich. Da ein Programm aber in seiner Funktionsweise seiner Komponenten der Leserin nähergebracht werden soll, ist der Schritt zur richtigen Bedienung nur kurz. Wer ein Fahrrad unter fachlicher Leitung auseinandergenommen und wieder zusammengesetzt hat, wird unmittelbar einsehen, wie Bremse und Gangschaltung zu bedienen sind. Allerdings ist nicht für alle, die sich einmal auf's Rad setzen wollen, erforderlich, schon einmal einen Freilauf von innen gesehen zu haben. Für ein Anwenderhandbuch eines Programms ist Literate Programming also weder notwendig noch unbedingt hinreichend, obwohl sich aus der Erklärung des Codes eine nurnutzerorientierte Dokumentation leichter herausfiltern läßt.

Wenn auch eine Dokumentation noch keine literarische Programmierung ist und mit einem literarischen Programm sich die Dokumentation nicht erschöpft hat, so ist man ja schon zufrieden, wenn ein Programm auf den Anwender hin dokumentiert ist. Daher soll ein kleiner Exkurs auf die verschiedenen Ansätze der Dokumentation im Umfeld von Linux nicht ausbleiben. An den Beispielen in den Abschnitten 3.3, 3.4 und 3.5 soll auch gezeigt werden, welche Ideen des Literate Programming in welchem Maße in verschiedenen Ansätzen verwirklicht sind.

3.2 Formatvielfalt der Dokumentation

Auch Dokumentationen haben verschiedene Erscheinungsformen. In der Linux-Welt ist das `nroff`-Format aus den Manpages bekannt; daneben existiert `gnu info`, das aktive Querverweise erlaubt. Einige wollten die Dokumentation auch schwarz auf weiß nach Hause tragen, daher war eine LaTeX-Version auch begehrt. Wollte man ein Dokument mit aktiven Querverweisen auch mit so neumodischen Gerätschaften wie Web-(nicht WEB!)-Browsern ansehen, brauchte man die Dokumentation im HTML-Format. Noch etwas später vereinheitlichte Adobe die Ansätze „schwarz auf weiß“ und „dynamische Querverweise“ mit dem Portable Document Format. Es lebe die Vielfalt.

3.3 Inhaltliches Markup mit Linuxdoc und DocBook

Was tut ein Schreiberling, der alle zufriedenstellen will, sowohl die tty-Konsole-Traditionalisten als auch die Link-Surfer sowie die Typographie-Puristen? Er wähle eine Sprache, die sich all die nötigen Informationen erhält, und Formatierungswerkzeuge, die die Quelle auf die je nach Zweck gewünschten Formate herunterbrechen.

Es hat sich durchgesetzt, von der inhaltlichen Auszeichnung eines Dokuments als der höchsten Form der Informationsrepräsentation zu denken und die physische Auszeichnung geringer zu schätzen[1]. Textteile, Bilder, Formeln erscheinen hierbei zusammen mit

[1] Die physische Auszeichnung, die die Position, Größe und Gestaltung der Textteile und Bilder auf dem

der Information, welche Rolle sie im Text spielen. Ein Beispiel kann als ein Beispiel markiert werden, oder auch als eingerückter Textblock im Kasten. Der Vorteil an der Markierung als Beispiel ist, daß in der Ausgabe der Dokumentation als rein technische Spezifikation alle Beispiele leicht erkannt und weggelassen werden können, während sie in einer Dokumentation für den Anfänger vielleicht eine prominente Stellung im Text und ein eigenes Verzeichnis bekommen.

Die Linux-Welt hat sich im Zusammenhang mit dem Linux Documentation Project [4] eine Sprache gewählt, die die inhaltliche Auszeichnung im weitem Maße unterstützt. Diese Sprache, die Standardized General Markup Language (SGML), wurde vor 13 Jahren internationaler Standard [2]. Dieser Standard legt aber nur die grundsätzliche Syntax fest, die Befehle (entities) und ihre Beziehung zueinander werden je nach Zweck des Dokuments in neuen Klassen namens Document Type Definitions (DTD) abgelegt. In Abbildung 2 sieht man einen Beispieltext. Für den HTML-Erfahrenen ist zu erkennen, dass die inhaltliche Auszeichnung nicht konsequent ausgeführt wurde. So ist der Dateiname `sgml-tools-1.0.x.tar.gz` nicht als Dateiname gekennzeichnet, sondern seine Repräsentation in Schreibmaschinenschrift (`tt`) vorgegeben.

Seit einiger Zeit nennt sich das Projekt nicht mehr Linuxdoc, weil nichts Linux-Spezifisches an ihr ist, sondern SGML-Tools. Aus Kompatibilitätsgründen heißt die DTD weiter Linuxdoc. Die Stärke des Projekts ist, dass (selbstredend frei verfügbare) Skripte existieren, die die Aufgabe der Formatierung für verschiedene Zwecke übernehmen. Folgende Formate werden von SGML-Tools unterstützt:

- bloßer Text,
- LaTeX, somit auch DVI, Postscript und mit dem PDFTeX-Paket auch PDF,
- HTML,
- GNU Info,
- LyX zur Weiterverarbeitung mit (K)LyX,
- RTF zur Weiterverarbeitung mit büroüblichen Textverarbeitungssystemen.

Leider ist das SGMLtools-Projekt verwaist. Die Leitfigur des Projekts hat ihren Posten als Koordinator aus beruflichen Gründen zur Verfügung gestellt. Obendrein scheint die Unterstützerfront zu bröckeln. Da die DTD nicht mehr Linux-spezifisch ist, sehen einige keinen Grund mehr, nicht zu einer weitergehenden DTD namens DocBook zu wechseln. Dieser Befehlssatz aus (nach meiner und `grep | wc`s Schätzung) 1910 entities leistet all das, was SGMLtools macht, und noch viel mehr. Seit Version 2.0 ist SGMLtools eine Teilmenge dieses Standards, kann damit aber das alte Linuxdoc-Format nicht mehr unterstützen.

Bildschirm oder der Druckseite bestimmt, ist eigentlich sehr hoch zu schätzen. Sie hat ihren festen Platz in der Ansteuerung der grafischen Benutzeroberfläche oder des Druckers. Sie hat im Gefolge des What-you-see-is-what-you-get-Unwesens eine Stellung beansprucht, die ihr nicht zustand, und darunter hat ihr Ruf gelitten.

```
<!-- This is the SGML-Tools User's Guide, originally by Matt Welsh.  Updated by
     many people.  See guide.txt or guide.ps for formatted output.
-->

<!doctype linuxdoc system>

<article>

<title>SGML-Tools User's Guide
<author>by Matt Welsh.  Updated by Greg Hankins, rewritten by Eric S. Raymond.
<date>1.0 ($Revision: 1.14 $), 10 November 1997
<abstract>
This document is a user's guide to the SGML-Tools formatting system,
a SGML-based system which allows you to produce a variety of
output formats.  You can create plain text output (ASCII and ISO-8859-1),
DVI, PostScript, HTML, GNU info, LyX, and RTF output from a single document
source file.  This guide describes SGML-Tools version 1.0.
</abstract>

<toc>

<sect>Introduction
<p>
SGML-Tools is a suite of programs to help you write source documents
that can be rendered as plain text, hypertext, or high-quality typeset
markup suitable for printing books.
<p>
This document is the user's guide to the SGML-Tools document processing system.
It contains more or less everything you need to know to set up
SGML-Tools and write documents using it.  See <tt/example.sgml/ for
an example of an SGML document that you can use as a model for your
own documents.

...

<sect1>What SGML-Tools Needs
<p>
The file <tt/sgml-tools-1.0.x.tar.gz/ contains everything that you
need to write SGML documents and convert them to groff, LaTeX{},
PostScript, HTML, GNU info, LyX, and RTF.  In addition to this
package, you will need some additional tools for generating formatted
output.
```

Abbildung 2: Ein Beispieltext in SGML-Tools: Ein Ausschnitt aus der Bedienungsanleitung

DocBook ist ohne Frage mächtig und breit unterstützt. Den Werkzeugen, die dieses Format weiterverarbeiten, merkt man aber an, daß mit zunehmender Komplexität die Geschmeidigkeit leidet. Mit Parsern und dokumentenerzeugenden Programmen zu industrieüblichen Preisen geht es sicher leichter von der Hand, hier habe ich allerdings keine Erfahrungen und werde sie auch bis zum nächsten Lottogewinn nicht haben.

Zudem ist der Markt der Auszeichnungssprachen in Bewegung. Der Siegeszug der Extensible Markup Language (XML) hält an, und DocBook wird diesem Format angeglichen. Wenn Literate Programming Zukunft haben soll, ist es auf die Anbindung an diese Entwicklungen angewiesen. Mit Entsetzen mußte ich allerdings feststellen, daß die Mailingliste zu diesem Thema (`http://www.egroups.com/group/xml-litprog-l/`) schon seit Monaten eingeschlafen ist.

3.4 Perldoc und das .pod-Format

Vielerorts, etwa bei manchen Programmiersprachen, wartet man nicht auf die „große Lösung“, sondern verwirklicht eine Teillösung. So zum Beispiel bei Perl.

Perl ist das Schweizer Taschenmesser unter den Skriptsprachen. Die mächtige Verarbeitung regulärer Ausdrücke und die Ersparnis so umständlicher Dinge wie das Deklarieren von Variablen macht diese Sprache vielseitig und zugleich bequem wie einen alten Turnschuh. Vor lauter Begeisterung hackt man los, sieht schnell Ergebnisse, ohne sich lang mit Vereinbarungen aufzuhalten – und sieht sich am Ende vor einem Berg in ASCII gegossener schlauer Einfälle.

Da die Dokumentation immer vergessen wird, so dachte man sich bei Perl, kommt sie mit in den Quellcode. So hat man sie immer bei sich. Die Sprache, die diese Dokumentation ermöglicht, heißt Plain Old Document (Pod). Sie ermöglicht physische Auszeichnung durch Überschriften in zwei Hierarchien, Listenbildung mit freier Wahl der Einrückung, Tabellensatz, Querverweise und Formatangaben. Dies leistet sie mit einem schmalen Befehlssatz von zehn Befehlen und 13 Formatanweisungen.

Das Prinzip ist ein ferner Abglanz der Idee von WEB: Der Quelltext läuft so, wie er ist, als Programm, die Verstrickung in ein Programm ist nicht nötig. Das Weben übernimmt ein Programm namens perldoc. Es schneidet alle Teile, die als Dokumentation gekennzeichnet sind, heraus und setzt sie in das nroff-, LaTeX-, HTML-Format oder blanken Text um. Auch Framemaker, eine langjährige Marktmacht in der Dokumentenverwaltung, wird als Ausgabeformat unterstützt. Der Code hat in der Dokumentation keinen Platz, das Programm wird dokumentiert, aber nicht erzählt. Auch die Reihenfolge ist nicht frei wählbar.

3.5 LaTeX: Das docstrip-Werkzeug

Auf etwas mehr Nähe zum Literate-Programming-Ansatz sollte man eigentlich bei den LaTeXern hoffen. In der Tat ist hier die Idee, die Dokumentation mit dem Programmtext in eine Datei zu schreiben, ebenfalls verwirklicht worden, und zwar mit dem LaTeX-Programm docstrip.

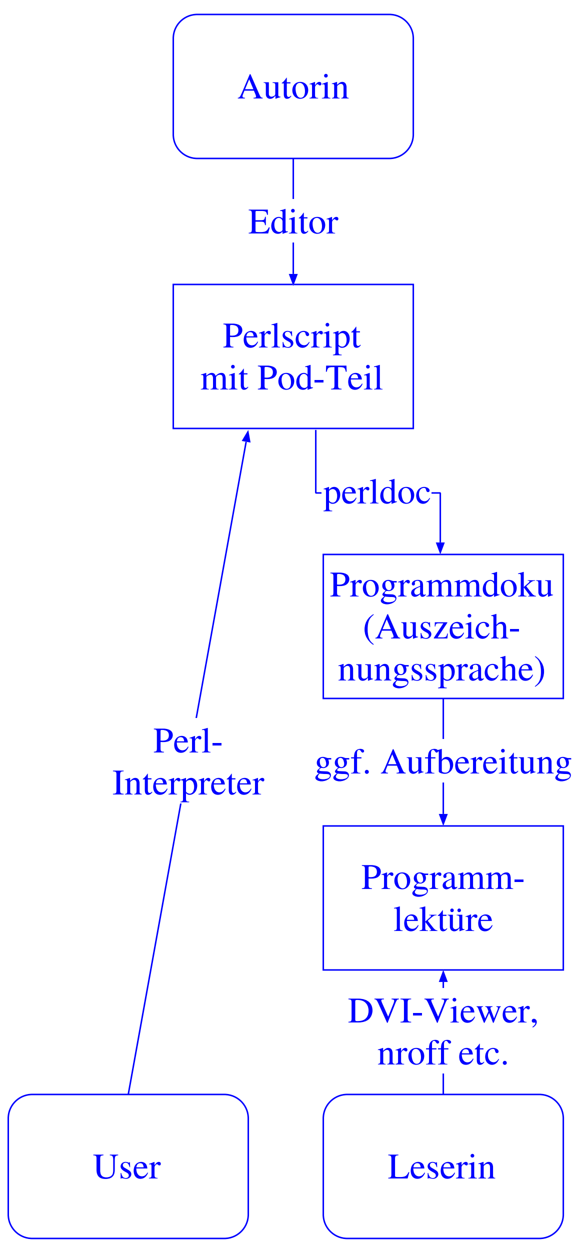

Abbildung 3: Ein Perlscript mit enthaltener Dokumentation ist so ausführbar, wie es ist

Die Syntax für Quelldokumente, die mit docstrip behandelt werden, ist denkbar einfach. Die Dokumentation wird mit einem einfachen „%“, dem Kommentarzeichen in TEX, auskommentiert. Die Befehle, die dem Computer mitteilen, was zu tun ist, werden nicht auskommentiert.

Das Werkzeug docstrip tilgt beim Anlegen des Packets alle Zeilen, die mit *einem* Prozentzeichen beginnen, und legt so den Stil an. Zeilen, die mit *zwei* Prozentzeichen beginnen, werden gelassen. So können normale Kommentare auch für das Packet mitgeführt werden.

4 Vom Meister selbst: WEB

Die Sprache WEB unterstützt als Auszeichnungssprache TEXund als Programmiersprache Pascal. Über sie wurde schon einiges gesagt. Sie unterstützt das Prinzip des Webens und Verstrickens. Ein verwobenes Programmdokument (ein Ausschnitt daraus ist in Abbildung 4 zu sehen) ist in kleine, numerierte Abschnitte aufgeteilt, die den Programmsplittern entsprechen, die im Quelldokument definiert wurden. Der Sinnzusammenhang wird durch eine einfache Syntax von Querverweisen hergestellt.


LITERATE PROGRAMMING

2. This program has no input, because we want to keep it rather simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the *output* file.

Since there is no input, we declare the value $m = 1000$ as a compile-time constant. The program itself is capable of generating the first m prime numbers for any positive m, as long as the computer's finite limitations are not exceeded.

⟦The program text below specifies the “expanded meaning” of ‘⟨ Program to print … numbers 2 ⟩’; notice that it involves the top-level descriptions of three other sections. When those top-level descriptions are replaced by their expanded meanings, a syntactically correct program will be obtained.⟧

⟨ Program to print the first thousand prime numbers 2 ⟩ ≡
program *print_primes*(*output*);
  const $m = 1000$;
    ⟨ Other constants of the program 5 ⟩
  var ⟨ Variables of the program 4 ⟩
    begin ⟨ Print the first m prime numbers 3 ⟩;
    end.

This code is used in section section 1.

3. Plan of the program. We shall proceed to fill out the rest of the program by making whatever decisions seem easiest at each step; the idea will be to strive for simplicity first and efficiency later, in order to see where this leads us. The final program may not be optimum, but we want it to be reliable, well motivated, and reasonably fast.

Let us decide at this point to maintain a table that includes all of the prime numbers that will be generated, and to separate the generation problem from the printing problem.

⟦The WEB description you are reading once again follows a pattern that will soon be familiar: A typical section begins with comments and ends with program text. The comments motivate and explain noteworthy features of the program text.⟧

⟨ Print the first m prime numbers 3 ⟩ ≡
  ⟨ Fill table p with the first m prime numbers 11 ⟩;
  ⟨ Print table p 8 ⟩

This code is used in section section 2.

4. How should table p be represented? Two possibilities suggest themselves: We could construct a sufficiently large array of boolean values in which the kth entry is *true* if and only if the number k is prime; or we could build an array of integers in which the kth entry is the kth prime number. Let us choose the latter alternative, by introducing an integer array called $p[1 \mathinner{..} m]$.

In the documentation below, the notation ‘$p[k]$’ will refer to the kth element of array p, while ‘p_k’ will refer to the kth prime number. If the program is correct, $p[k]$ will either be equal to p_k or it will not yet have been assigned any value.

⟦Incidentally, our program will eventually make use of several more variables as we refine the data structures. All of the sections where variables are declared will be called ‘⟨ Variables of the program 4 ⟩’; the number ‘4’ in this name refers to the present section, which is the first section to specify the expanded meaning of ‘⟨ Variables of the program ⟩’. The note ‘See also …’ refers to all of the other sections that have the same top-level description. The expanded meaning of ‘⟨ Variables of the program 4 ⟩’ consists of all the program texts for this name, not just the text found in §4.⟧

⟨ Variables of the program 4 ⟩ ≡
p: **array** $[1 \mathinner{..} m]$ **of** *integer*; { the first m prime numbers, in increasing order }

See also section sections 7, 12, 15, 17, 23, and 24.

This code is used in section section 2.

5. The output phase. Let's work on the second part of the program first. It's not as interesting as the problem of computing prime numbers; but the job of printing must be done sooner or later, and we might as well do it sooner, since it will be good to have it done. ⟦And it is easier to learn WEB when reading a program that has comparatively few distracting complications.⟧

Since p is simply an array of integers, there is little difficulty in printing the output, except that we need to decide upon a suitable output format. Let us print the table on separate pages, with rr rows and cc columns per page, where every column is ww character positions wide. In this case we shall choose $rr = 50$, $cc = 4$, and $ww = 10$, so that the first 1000 primes will appear on five pages. The program will not assume that m is an exact multiple of $rr \cdot cc$.

⟨ Other constants of the program 5 ⟩ ≡
  $rr = 50$; { this many rows will be on each page in the output }
  $cc = 4$; { this many columns will be on each page in the output }
  $ww = 10$; { this many character positions will be used in each column }

See also section section 19.

This code is used in section section 2.

6. In order to keep this program reasonably free of notations that are uniquely esque, ⟦and in order to illustrate more of the facilities of WEB,⟧ a few macro definitions for low-level output instructions are introduced here. All of the output-oriented commands in the remainder of the program will be stated in terms of five simple primitives called *print_string*, *print_integer*, *print_entry*, *new_line*, and *new_page*.

⟦Sections of a WEB program are allowed to contain *macro definitions* between the opening comments and the closing program text. The general format for each section is actually tripartite: commentary, then definitions, then program. Any of the three parts may be absent; for example, the present section contains no program text.⟧

⟦Simple macros simply substitute a bit of code for an identifier. Parametric macros are similar, but they also substitute an argument wherever ‘#’ occurs in the macro definition. The first three macro definitions here are parametric; the other two are simple.⟧

define *print_string*(#) ≡ *write*(#)
    { put a given string into the *output* file }
define *print_integer*(#) ≡ *write*(# : 1)
    { put a given integer into the *output* file, in decimal notation, using only as many digit positions as necessary }

Abbildung 4: Ein verwobenes WEB-Dokument (Ausschnitt)

Der im Lesedokument abgedruckte Pascal-Code ist ebenfalls mit Hilfe der in LATEX üblichen Computer-Modern-Schriften gesetzt worden. Speziell sind folgende Symbole eingeführt worden: `:=` wird $\rightarrow$, `<=` $\leq$, `>=` $\geq$, `<>` $\neq$, `AND` $\wedge$, `OR` $\vee$, `NOT` $\neg$, `IN` $\in$ und `5E20` wird $5 \cdot 10^{20}$.

Während das typographische Dokument auf Lesbarkeit optimiert ist, hat der Teil, der den Computercode generiert, nicht umsonst den Namen tangle erhalten. Der Code soll nämlich nicht nur für den Computer bearbeitbar, sondern auch für den Menschen unleserlich sein, damit der Mensch nicht auf die Idee kommt, am Computercode weiterzuprogrammieren und die eigentliche Quelle zu vernachlässigen. Die im Open-Source-Bereich verpönte obfuscation, die absichtliche Vernebelung der offengelegten Quellen, ist in diesem speziellen Fall erwünscht.

Eine Veränderung der Computerwelt wurde bei der Entwicklung von WEB nicht vorhergesagt. Sie spielt aber der Entwicklungsmöglichkeit des Literate Programming vor die Füße. Die Rede ist von dem Siegeszug von Hypertext im Zusammenhang mit dem World Wide Web. Nun ist es möglich, ein Programm mit einer Hypertextsprache zu kommentieren. Die Leserin kann in einem elektronischen Dokument direkt in die Gedankensplitter springen, die mit dem aktuellen Teil des Programms in Sinnzusammenhang stehen.

5 Verallgemeinerung: Noweb

Die Webersprache, aus der das Dokument erzeugt wird, muß ein Grundverständnis sowohl der Programmiersprache als auch der Auszeichnungssprache haben, um beides setzen zu können. Bei der Vielfalt der Programmiersprachen, zu der sich auch mittlerweile eine respektable Sammlung an Auszeichnungssprachen (oder auch nur DTDs) gesellt haben, müßte es eine Unmenge an Webersprachen geben. Donald Knuth brachte der Welt noch CWEB, das das sehr populäre C mit TEXverbindet (bzw. sie voneinander trennt). Aber wenn ich ein Python-Programm in HTML kommentieren will, benötige ich eine neue Sprache, und wenn ich ein Shellskript mit DocBook kommentieren will, wieder eine neue und so weiter.

Im Prinzip aber müßte eine solche Präprozessor- oder Webersprache nicht viel können. Sie müßte, im Falle von LATEX zum Beispiel, lesbaren LATEX-Code erzeugen. Wenn der Autor diesen Code schon als lesbaren LATEX-Code schreibt, müßte der weave-Teil des Programms nur den Code-Teil erkennen und als Code formatieren, während es den Rest einfach so lassen kann. Der tangle-Teil hat es noch leichter, indem er nur den Code behält. Er wäre ein umgekehrtes Perldoc, das ja selbst den Code verwirft und nur die Dokumentation formatiert.

Diesen hohen Abstraktionsgrad erfüllt das Programm noweb von Norman Ramsey. Es erkennt

- die Trennmarken zwischen Kommentaren und Code und
- die Bezeichnungen der Codesplitter und ihre Querverweise.

So kann der codeerzeugende Teil des Pakets, notangle, völlig unabhängig von der Programmiersprache den computerlesbaren Code erzeugen.

Auf seiten von noweave ist die Sprachunabhängigkeit der Auszeichnungssprache nicht gegeben. Das Dokument wird in der Reihenfolge des Quelldokuments gesetzt, die Verweise im Quelltext müssen aber in Querverweise verwandelt werden. Mindestens hierzu muß noweave den Wiedergabe der Querverweise in der Auszeichnungssprache kennen. Dieser Teil der Sprache ist leider nicht zuende entwickelt. Außerdem leistet noweave durch seine mangelnde Spracherkennung keinen speziellen Satz der Sprache. Manche halten dies allerdings für einen Vorteil.

Wie sieht die Syntax von Noweb aus? Im Quelltext beginnt man mit einleitenden Kommentaren in der Auszeichnungssprache. Ein Stück Code wird auf folgende Weise verpackt:

```
<<Begrüßung>>=
print "Hello, World";
@
```

Er beginnt mit beschreibenden Stichworten in doppelten spitzen Klammern und endet mit dem Klammeraffen, gefolgt von einem Leerschlag. Beide Markierungen müssen am Anfang einer Zeile stehen.

Die beschreibenden Stichworte zählen gleichzeitig als Sprungmarke. Ein Stück Programmcode kann so mit einem Platzhalter ausgedrückt werden und erst an anderer Stelle ausführlich erscheinen. Ein Ausdruck wie

```
<<Elefant in den Kühlschrank>>=
<<Kühlschranktür auf>>
<<Elefant rein>>
<<Kühlschranktür zu>>
@
```

ist daher nicht ungewöhnlich, wenn man dem Leser an dieser Stelle ersparen will, die genaue Prozedur des Öffnens der Kühlschranktür auseinanderzusetzen. Dat andere Loch, dat krieje mer spätor.

6 Praktische Umsetzung

6.1 Simultane Quellformatierung unter Emacs

Wer literarisch programmiert, muß die Programmier-, die Auszeichnungs- und die Webersprache beherrschen. Daher ist es eine gute Idee, daß Noweb schlank daher kommt, so daß man nicht viel zu lernen hat. Mittlerweile verlangt man allerdings auch vom Editor einige Eigenschaften wie die automatische Einrückung von Text. Auch der Editor muß alle drei Sprachen beherrschen und obendrein erkennen, in welchem Splitter welche Sprache gesprochen wird.

Der Noweb-Modus [6] von Emacs leistet dies. Ein im Noweb-Modus befindliches Dokument hat in der ersten Zeile einen Eintrag, der beschreibt, um welche Programmier- und welche Auszeichnungssprache es sich handelt. Anhand der Markierungen „`<<*>>`“ und „`@ `“ baut sich der Modus eine Liste aus Begrenzungen auf und vergleicht die Position des Points (das ist Emacssprech für Cursor) mit dieser Liste, um die aktuellen Sprachen zu ermitteln. Hier schaltet Emacs dann in den Untermodus um, der zur Sprache des aktuellen Splitters paßt.

Ein kleines Problem liegt in der optischen Hervorhebung des Textes im Editor. In meiner Implementation von Emacs kommen sowohl der font-lock-mode als auch hilit mit den wechselnden Formatierungen durcheinander. Eine Abhilfe schafft eine Funktion, die immer nur den aktuellen Splitter formatiert.

```
;; im noweb-Modus: Markiere aktuellen Splitter

(defun noweb-hilit-chunk () (interactive)
  "Im noweb-mode: Nutzt hilit, um aktuellen Splitter zu markieren."
  (let ((chunk-region (noweb-chunk-region)))
    (hilit-highlight-region (car chunk-region) (cdr chunk-region))))
```

Besonders bequem ist es, wenn diese Markierung immer automatisch vorgenommen wird, wenn ein neuer Splitter betreten wird. Dazu muß man den Funktionsaufruf im `noweb-select-mode-hook` unterbringen, der immer dann abgefragt wird, wenn der Point in einen neuen Splitter gerät.

```
;; im noweb-Modus: Markiere aktuellen Splitter immer,
;; wenn du ihn betrittst.

(add-hook 'noweb-select-mode-hook 'noweb-hilit-chunk)
```

Und ab damit in die `.emacs`-Datei. Die Splitter bilden einen natürlichen und intuitiven Bereich zur lokalen Markierung. Wenn sie dem Geist des Literate Programming zufolge kurz sind, kostet die Markierung auch kaum Zeit.

Diese Vereinfachung funktioniert bei mir nicht, wenn man `hilit-highlight-region` durch `font-lock-fontify-region` ersetzt. In die Feinheiten der beiden Markierungsmethoden unter Emacs habe ich mich zu wenig hineinbegeben, um sagen zu können, warum.

6.2 Die Verwendung von make

Um aus dem Mutterdokument sowohl lesbare Kommentare als auch einen ausführbaren Code zu erzeugen, müssen mehrere Schritte ausgeführt werden. Der noweave- und notangle-Schritt läuft ab, erst dann schreitet der Compiler zu Werke. Wurde die Kommentierung in LaTeX gemacht, muß LaTeX auch ein paar Male durchlaufen.

Wer programmiert, automatisiert solche Schritte und konzentriert sich dann auf das Wesentliche. Zur Abkürzung dieser Prozedur existiert das GNU-make-Werkzeug. Makefiles sind ein unverzichtbarer Bestandteil komplizierterer Softwareprojekte. Aber auch bei einfachen Programmen, wenn sie erst auf diese Weise präprozessiert werden müssen, ist ein solches Vorgehen gerechtfertigt.

In Abbildung 5 ist ein noch einfaches Beispiel aufgeführt. Einfach deshalb, weil es sich um ein Perlskript handelt, das nicht kompiliert und gelinkt werden muß. Die grundlegende Syntax eines Makefiles ist einfach:

```
dafür: nötig
<TAB>Arbeitsschritte, um aus "nötig" "dafür" zu machen
```

Diese Vorschriften können mehrmals in einem Makefile vorkommen. Ungewöhnlich ist, daß make auf dem Tabulator zur Einrückung der Arbeitsschritte besteht. Was tut make, wenn es die Makefiles ausliest? Zunächst können make Argumente übergeben werden, die alle Erwähnungen von „dafür“ enthalten können. Wenn make ohne Argumente aufgerufen wird, werden alle vor den Doppelpunkten stehenden Teile erzeugt.

Vor jedem Arbeitsschritt wird nachgesehen, ob die Datei „dafür“ nicht doch neuer ist als alle „nötig“-Dateien. Dann scheint es nicht „nötig“ zu sein, „dafür“ etwas zu tun, da man offenbar die aktuelle Version von „dafür“ vor sich hat.

Wenn man das Makefile in Abbildung 5 durchgeht, so sieht man, daß am Anfang einige Definitionen stehen. Zunächst wird der Programmname definiert. Der Quelltext bekommt das Anhängsel `.nw`. Darüber hinaus soll eine feste Datei namens `hugodoc.tex`

```
CODENAME = hello
SRC = $(CODENAME).nw
DOCNAME = $(CODENAME)doc

all: docs code

docs: docs-ps

code: $(SRC)
	if test -e $(CODENAME).pl; then chmod u+w $(CODENAME).pl; fi
	notangle $(SRC) > $(CODENAME).pl
	# I would like rather to see you edit the .nw source than this...
	chmod u-w $(CODENAME).pl

docs-ps: $(DOCNAME).tex $(CODENAME).tex
	latex $(DOCNAME)
	latex $(DOCNAME)
	latex $(DOCNAME)
	dvips -D600 $(DOCNAME)

$(CODENAME).tex: $(SRC)
	if test -e $(CODENAME).tex; then chmod u+w $(CODENAME).tex; fi
	noweave -x -latex -delay $(SRC) > $(CODENAME).tex
	# I would like rather to see you edit the .nw source than this...
	chmod u-w $(CODENAME).tex
```

Abbildung 5: Ein Makefile zu einem einfachen in noweb geschriebenen Programm

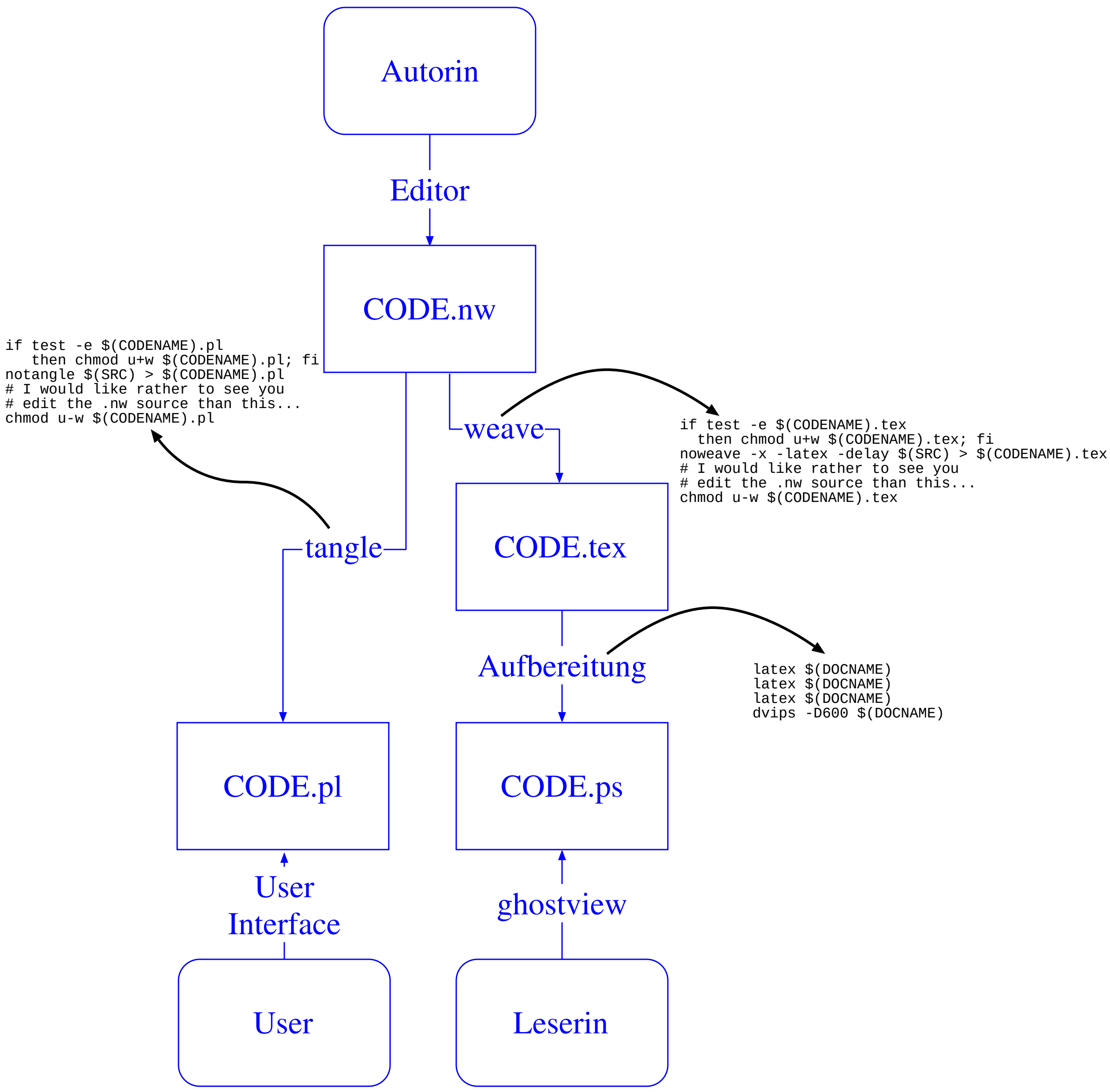

Abbildung 6: Die Elemente des Makefiles im Ablauf des Webens und Verstrickens

existieren, in der die Dokumentation steht. Sie enthält das LaTeX-Skelett mit Paketaufrufen, die im Quelldokument nur unübersichtlich wären. Die meisten entwickeln ihr Programm eben doch im Editor und nicht mit ständigem Blick auf die DVI-Darstellung.

Wenn man „alles“ (`make` oder `make all`) generieren will, so teilt sich „alles“ in die Postscript- und Code-Ausgabe. Hier sind keine Arbeitsschritte angegeben; die Arbeitsschritte zur Generierung der Postscript- oder Codeausgabe sind dann unter `docs:` respektive `docs-ps:` und `code` vermerkt.

Für den Code ist nur die Mutterdatei nötig. Aus dieser macht notangle eine für den Compiler geordnete Datei. Dieser Code ist immer noch gut lesbar – man läßt die Katze nicht lang genug mit dem Wollknäuel spielen. Daher gibt man derjenigen, die sich am Code vergreifen will, über die eingeschränkte Schreibberechtigung qua `chmod` etwas auf die Finger.

In einer Compilersprache müßte jetzt noch kompiliert werden. Dies kann man sich in diesem Beispiel sparen. Stattdessen wird die Postscript-Datei erzeugt, zunächst mit LaTeX („Du mußt es dreimal sagen!“ [3]) und dann mit dvips.

Die LaTeX-Datei selbst wird erzeugt durch noweave. Die Optionen der Reihe nach:

`-x` sorgt dafür, daß die Splitter mit Seitenzahlen versehen werden, und erleichtert die Querverweise.

`-delay` Im Normalfall wird das gewobene Dokument mit dem LaTeX-Skelett versehen. Wir lassen das hier mit der Option `-delay` weg, um mehr Gestaltungsfreiheit zu haben (zum Beispiel um die KOMA-Klassen wählen zu können).

`-latex` Sorgt für den Output in LaTeX. Andere Möglichkeiten hier sind `-tex` oder `-html`.

Wenn make aufgerufen wird, sucht es im aktuellen Verzeichnis nach `Makefile` und arbeitet diese Datei ab.

6.3 Probleme in der Konstellation

Der Noweb-Modus in Emacs kann zwar reibungslos innerhalb eines Splitters in der aktuellen Sprache formatieren. Die Auszeichnung innerhalb des Editors über den `font-lock-mode` funktioniert aber nur mangelhaft. Dieser Modus schien nicht darauf vorbereitet zu sein, daß innerhalb eines Dokuments nach verschiedenen Syntaxen formatiert werden muß. Wie in 6.1 erwähnt, läßt sich `hilit` befriedigend für diese Aufgabe einspannen.

Außerdem sollte man eines lassen: Innerhalb einer Struktur (einer Schleife oder einer Unterroutine) einen Kommentarsplitter einzufügen. Der Modus geht nämlich davon aus, daß er zu Beginn eines neuen Codesplitters mit der Einrücktiefe Null beginnen kann. Wenn man eine Unterroutine in mehrere Splitter zerlegt und einzeln kommentieren will, muß man sie also zu Beginn über die Verweise darauf verwalten.

Die Fehlersuche mit handelsüblichen Debuggern macht Übersichtsprobleme. Ein Compiler oder Interpreter bezieht sich, wenn er sagt: „Scalar found where operator expected at line 67“, auf die Zeile 67 im Programmcode und nicht im Originaldokument. In diesem muß man dann nach der originalen Zeile suchen. Diese Suche wird vereinfacht, indem man notangle mit der Option `-L` aufruft. Darauf wird jeder Codesplitter in der Programmdatei mit einem Vermerk eingeleitet, wo in der Originaldatei die Zeile steht. Wenn die Ausgabe des Debuggers lesbar ist, kann man sich ein relativ kurzes Emacs-Makro vostellen, das sich gleich zur richtigen Stelle im Originaldokument durchhangelt. Im Falle von Perl reagiert sogar der Interpreter auf diese Kommentare! und die Nachricht lautet: "Scalar found where operator expected at sleuth.nw line 220". Nichtsdestoweniger ist man durch die Mehrzahl an Aufrufen, die make verursacht, beim Debuggen etwas weiter vom Code weg als ohne diesen Zwischenschritt.

7 Open Source und Literate Programming

Eigentlich sollten die beiden Ansätze Open Source und Literate Programming zu unzertrennlichen Schwestern geworden sein. Wenn man die Quellen seines Tuns offenlegt, offenbart man einen hochgeschätzten Wesenszug. Noch netter allerdings wäre es, wenn man seinen Lesern den Spaß am Lesen erhielte, indem man seine Software mit einem essayistischen Ansatz schreibt.

Auf diesem Wege geschriebener Code ist leicht wartbar, für die Autorin selbst, aber auch und vor allem für andere. Gedankengänge sind leichter nachzuvollziehen und somit auch durch andere leichter erweiterbar.

Ein wichtiger Punkt kommt noch hinzu. In der Welt der Open Source hört man immer den Ruf: „Ich würde gerne selbst zu diesem großartigen Projekt beitragen. Aber ich gehöre zum gemeinen Fußvolk der Nicht-Hacker.“ Ein Ruf, der ebenso regelmäßtig beantwortet wird mit: „Dann schreibe Handbücher! Dazu haben viele Hacker keine Lust, und einen zu großen Abstand zum Fußvolk.“ Diese Antwort verhallt jedoch häufig ungehört. Das kann gut daran liegen, dass man, bevor man ein Handbuch schreiben kann, den Quellcode des Programms in groben Zügen nachvollziehen muss. Einer Dokumentation, die auf reiner Anwendungserfahrung beruht, würde ich jedenfalls nicht trauen. Wäre der Code hingegen literarisch programmiert, könnte man mit einem auch begrenzten Verständnis für die Details die für den User wesentlichen Teile aus dem dokumentierten Teil filtern. Dies ist eine redaktionelle Tätigkeit, die der Hacker vielleicht immer noch gerne links liegenläßt. Ein engagierter User mit solidem Halbwissen kann aber genau dies leisten und so aus einem literarisch abgefaßten Code ein wertvolles Handbuch schreiben (siehe auch die Erläuterungen in Abschnitt 3.1).

Literatur

[1] Jon Bentley and Donald Ervin Knuth. Programming pearls: A literate program. *Communications of the Association for Computing Machinery*, 29:364–369, May 1986. Nachdruck in D. E. Knuth. Literate Programming (1992).

[2] ISO-8879: Information processing – text and office systems – standard generalized markup language (SGML). Technical report, International Organization for Standardization, 1986.

[3] Johann Wolfgang Goethe. *Faust. Der Tragödie erster Teil.* Aufbau-Verlag Berlin und Weimar, 1965. Vers 1531.

[4] Greg Hankins. The Linux Documentation Project. `http://tech.appl-opt.physik.uni-essen.de/linux/LDP`.

[5] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984. Nachdruck in D. E. Knuth. Literate Programming (1992).

[6] Thorsten Ohl. noweb-mode. `http://heplix.ikp.physik.tu-darmstadt.de/~ohl/`.

[7] Eric W. van Ammers. Second observation on Literate Programming. Persönliche Kommunikation, zitiert in den LPFAQ (`http://shelob.ce.ttu.edu/daves/lpfaq/`).