8 Texteditoren

In diesem Kapitel betrachten wir interaktive Editoren. Das sind Programme, mit denen man Texte, allgemeiner gesagt Dokumente, erstellen und verändern kann. Die Texte können Programme sein, aber auch Manuskripte von Artikeln oder Büchern und auch Seiten im World Wide Web.

Der Benutzer eines solchen Editors gibt einen Text ein, der im Rechner in einer Datei abgespeichert wird. Diese Datei kann wieder geladen werden, um weiter bearbeitet zu werden. Der Editor bietet zumindest die folgende Funktionalität an:

- Der Benutzer kann einen ihn interessierenden Teil des Textes auswählen, z. B. dadurch, daß er zeilen-, bildschirm-, wort- oder abschnittsweise, durch Angabe einer Zeilennummer oder einer symbolischen Marke, durch Durchlaufen einer hierarchischen Struktur oder durch eine Beschreibung des Teils (assoziativ) dorthin „wandert".
- Auf diesem Teil kann er dann Edieroperationen ausführen. Dazu gehören das Einfügen von Text, das Löschen, das Kopieren und das Ersetzen. Allerdings bieten Editoren häufig auch „globale" Edieroperationen an, mit denen der Benutzer mehrere oder alle Vorkommen eines Textmusters an verschiedenen Stellen im Text gleichzeitig bearbeiten kann.
- Ein Teil des interessierenden Textstücks, eventuell auch dieses Stück zusammen mit seinem Kontext, werden auf dem Darstellungsmedium, heute immer einem Bildschirm, dargestellt.

Dieses Kapitel hat als Ziel, die Funktionalität, den Aufbau und die Implementierung moderner Texteditoren vorzustellen und die historische Entwicklung der Texteditoren zu beschreiben. An der historischen Entwicklung interessiert vor allem, auf welchen Wegen (und Umwegen) sich das heutige konzeptuelle Editormodell entwickelt hat. Die ersten Generationen von Texteditoren gehorchten nämlich anderen Modellen, oft mangels der Einsicht der Entwickler in die Möglichkeiten der rechnergestützten Textverarbeitung. Da wurden eher Schreibmaschinen, Fernschreiber und Lochkartenstanzer nachgebildet. Natürlich schränkten auch die verfügbare Gerätetechnik und die verglichen zu heute schwache Rechnerleistung die Möglichkeiten der Entwickler ein. Erst mit der Erfindung der Maus und anderer Zeigeinstrumente waren die Voraussetzungen für moderne Editoren mit graphischer Oberfläche gegeben.

Die historische Entwicklung des Editormodells ist exemplarisch für viele neue Entwicklungen in der Informatik. Ein neues Medium, ein neues Konzept kommt auf. Seine vollen Möglichkeiten werden erst verkannt. Technische Unzulänglichkeiten

verhindern eine Zeit lang die volle Entfaltung. Durch die gesammelte Phantasie vieler Forscher und Entwickler entfalten sich nach und nach alle in dem Medium steckenden Möglichkeiten, bis alle Welt sich fragt, warum man nicht von Anfang an das volle Potential erkannt hat.

Der Aufbau dieses Kapitels ist wie folgt: In den Abschnitten 8.1 und 8.2 wird der Editor aus der Sicht des Benutzers bzw. des Entwicklers beschrieben. Darauf folgt ein Gang durch die historische Entwicklung von Texteditoren, in dem vor allen Dingen Wert auf die Identifikation von entscheidenden Änderungen des Editormodells gelegt wird. Abschnitt 8.4 beschreibt die Funktionalität von Editoren etwas ausführlicher. Danach werden einige Editoren beschrieben, die die historische Entwicklung dokumentieren oder die heute noch von Bedeutung sind. Es sind dies der *EDT*, der *TECO*, der *vi*, der *emacs* und der *FrameMaker*. Es schließt sich ein Abschnitt über die Implementierung bildschirmorientierter Editoren an. Die Darstellung folgt in Teilen dem immer noch lesenswerten Übersichtsartikel [MvD82] von N. Meyrowitz und A. van Dam.

8.1 Benutzersicht

8.1.1 Editormodelle

Der Benutzer eines Texteditors wird mit einem bestimmten ***konzeptuellen Modell*** des Editors konfrontiert, also einer abstrakten Sicht, was ein Text ist, welches seine Bestandteile sind, mit welchen Edieroperationen man Texte bearbeiten kann und was deren Effekte sind. Dieses konzeptuelle Modell war die Grundlage, die der Entwickler bei der Realisierung des Editors hatte. Der Benutzer erwirbt durch Unterricht, Lesen von Handbüchern und Benutzung des Editors sein individuelles ***Benutzermodell***. Entspricht dieses dem konzeptuellen Modell und ist jenes kohärent, so sollte es für den Benutzer recht einfach sein, sein Benutzermodell zu erweitern und z. B. den Effekt von ihm unbekannten Operationen vorherzusagen. Ist das konzeptuelle Modell schon nicht kohärent, d. h. werden z. B. für sehr ähnliche Operationen sehr verschiedene Bezeichnungen oder Notationen verwendet, oder sind sie über verschiedene Menüs verteilt, dann bleibt dem Benutzer nicht viel mehr übrig als Kommandos und ihre Bedeutung bzw. den Aufbau der Menüs auswendigzulernen.

Das Benutzermodell ist meist ein Teilmodell des konzeptuellen Modells. Wenn das System sehr mächtig ist, so hat jeder Benutzer sein privates Modell des Systems, welches die von ihm benötigten Operationen enthält und mit dem er sich behelfen gelernt hat. Der eine Benutzer, der nie einen Serienbrief verschickt, wird die Serienbrief-Funktion seines PC-Textsystems ignorieren, der andere, der nie eine mathematische Formel schreibt, die Möglichkeiten, Formeln einzugeben.

8.1.2 Bedienoberfläche

Der Benutzer kommuniziert mit dem Editor über dessen *Bedienoberfläche* (user interface). Zur Oberfläche gehören die Eingabegeräte Tastatur, Maus oder andere Zeigeinstrumente und als Ausgabegerät der Bildschirm. Das Zeigeinstrument bewegt ein Zeigesymbol auf dem Bildschirm. Die Veränderung seiner x- und y-Koordinaten wird ständig abgetastet (meist von einem Fenstersystem) und in eine entsprechende Veränderung der Position des Zeigesymbols übersetzt. Das Drücken der Mausknöpfe löst Unterbrechungen (Interrupts) im System und damit Aktionen aus wie das Setzen der Schreibmarke, die Aktivierung von Textstücken oder anderen Objekten oder das Erscheinen von Menüs.

Mit Hilfe der Eingabegeräte kommuniziert der Benutzer mit dem Editor in einer *Sprache*. Dieser Begriff ist hier recht weit gefaßt und überdeckt auch nichtlinguistische Elemente, da er auch die Benutzung von Funktionstasten, das Bewegen der Maus und das Klicken eines Mausknopfes umfaßt. Die *Syntax* der Eingabesprache legt fest, welche Formen von Kommandos erlaubt sind. Das können im Falle textueller Kommandos Zeichenfolgen sein, die mit irgendwelchen Kontrollzeichen oder dem Escape-Symbol beginnen, um nicht mit Eingabezeichen verwechselt zu werden, oder „normale“ Eingabezeichen, die allerdings in einem eigenen Kommandomodus eine spezielle Interpretation haben. Andererseits können es auch Mausklicks auf Textstellen oder Schaltflächen, Menüstellen und Bildrolleisten (Scrollbars) sein.

Die abstrakte Sicht von Texten und die Menge der Editoroperationen bilden die Basis für die *Semantik* dieser Eingabesprache. Die Semantik legt die Bedeutung von Kommandowörtern bzw. angeklickten Flächen fest, also die Zuordnung von Kommandos zu Editoroperationen, hier auch *semantische Routinen* genannt. Oft gibt es mehrere äquivalente Kommandos für eine Editoroperation. Dies wird dazu benutzt, Anfängern und erfahrenen Benutzern verschiedene Oberflächen anzubieten. Der Anfänger bekommt eine narrensichere Oberfläche angeboten, in der er sich allerdings für manche Operationen durch mehrere Menüs quälen muß. Der erfahrene Benutzer lernt sogenannte Hotkeys auswendig, das sind ein oder zwei Zeichen lange Escape- oder Kontrollsequenzen.

Textorientierte Kommandosprachen sollten einen konsequenten syntaktischen Aufbau haben, also etwa nur eine *Präfixschreibweise*. Ein Zeichen für die Operation wird dabei gefolgt von einem Zeichen für den Operanden oder seinen Typ. Die Eingabe `dw` steht also für *delete word*, `db` für *delete backward*, `cw` für *change word* und `cb` für *change backward*. Die Präfixschreibweise wird unhandlich, wenn Kommandos mehrere Operanden haben. Für Kommandos mit zwei Operanden bietet sich *Infixschreibweise* an; die beiden Operanden stehen vor bzw. nach dem Operator.

Leider sind natürliche Sprachen nicht unter dem Aspekt entworfen worden, natürliche Bezeichnungen für die Operationen eines Editors (oder eines beliebigen anderen Systems) mit jeweils verschiedenen Zeichen unseres Alphabets beginnen zu lassen. Deshalb werden die übriggebliebenen Zeichen mehr oder weniger systematisch den übriggebliebenen Operationen zugeordnet. Dann steht z. B. `h` für ein

Zeichen rückwärts und `l` für ein Zeichen vorwärts, `j` für eine Zeile tiefer und `k` für eine Zeile hoch. Natürlich steckt auch dahinter eine Systematik. Diese vier Zeichen liegen nämlich in der zweituntersten Tastaturreihe direkt nebeneinander. Solch eine Benennung von zusammengehörenden Operationen gemäß ihrer Positionen auf der Tastatur nennt man *topologische Benennung*.

Eine nur tastaturorientierte Eingabe hat ihre Grenzen bei Editoren mit großer Funktionalität. Dann müssen nämlich die Tasten mehrfach durch Kombination mit den Shift-, den Kontroll- und eventuell noch einer Alt-Taste überladen werden. Nur noch Profis können die Unmenge der sinnvollen Kombinationen behalten. Der Rest bekommt mit der Lieferung der Software einen Merkstreifen, der seine Tastatur ziert. Darauf sind kurz die Bedeutungen der vielen belegten Kombinationen beschrieben.

Menütechniken bieten die Möglichkeit, mit wenigen Konzepten eine systematische Bedienoberfläche zu realisieren. Aus einer in der Fensterleiste dargestellten Menge von sogenannten *Pulldown-Menüs* oder aus *Popup-Menüs*, die auf Mausklicks erscheinen, kann man baumartig in weitere Menüs verzweigen, bis das gesuchte Kommando erreicht ist. Die Zuordnung der Kommandos zu den verschiedenen Zweigen des Baumes muß wieder sehr systematisch sein, weil sonst der Suchaufwand für den Benutzer groß wird. Werden die Menübäume tief, so wird die Benutzung mühselig.

8.2 Systemsicht

In diesem Abschnitt wird eine mögliche Architektur für einen Editor vorgestellt. Eine solche Architektur identifiziert die Komponenten des Editors, ihre Aufgaben und die Art, wie sie zusammenarbeiten. Die Architektur realer Editoren ist so ähnlich wie die hier vorgestellte. Abbildung 8.1 zeigt die Komponenten und den Kontroll- und Datenfluß zwischen ihnen.

Der *Kommandoprozessor* analysiert die Benutzereingabe auf legale Kommandos hin und aktiviert die von ihnen bezeichneten Operationen. Bei Editoren mit tastaturorientierten Oberflächen besteht die Benutzereingabe aus einem Strom von Eingabezeichen. Moderne Editoren mit Graphikoberfläche laufen im allgemeinen unter einem Fenstersystem (X-Windows, MS Windows). Mausklicks auf Schaltflächen werden von diesem Fenstersystem abgefangen und als Ereignis an den Kommandoprozessor des Editors weitergemeldet. Dieser kann die verschiedenen Ereignisse interpretieren und die entsprechenden Operationen ausführen.

Die Operationen des Editors bauen sich aus primitiven Operationen verschiedener Art auf. Die verfügbaren Arten sind das Wandern, das Edieren, das Zeigen und das Darstellen. Edieroperationen werden immer vom Benutzer ausgelöst, und das Darstellen geschieht automatisch als Folge der anderen Operationen.

Die *Edierkomponente* des Editors führt Edieroperationen wie Einfügen, Löschen und Ersetzen aus. Sie besitzt einen *Edierpunkt*. Dieser zeigt, wenn er existiert, auf irgend-

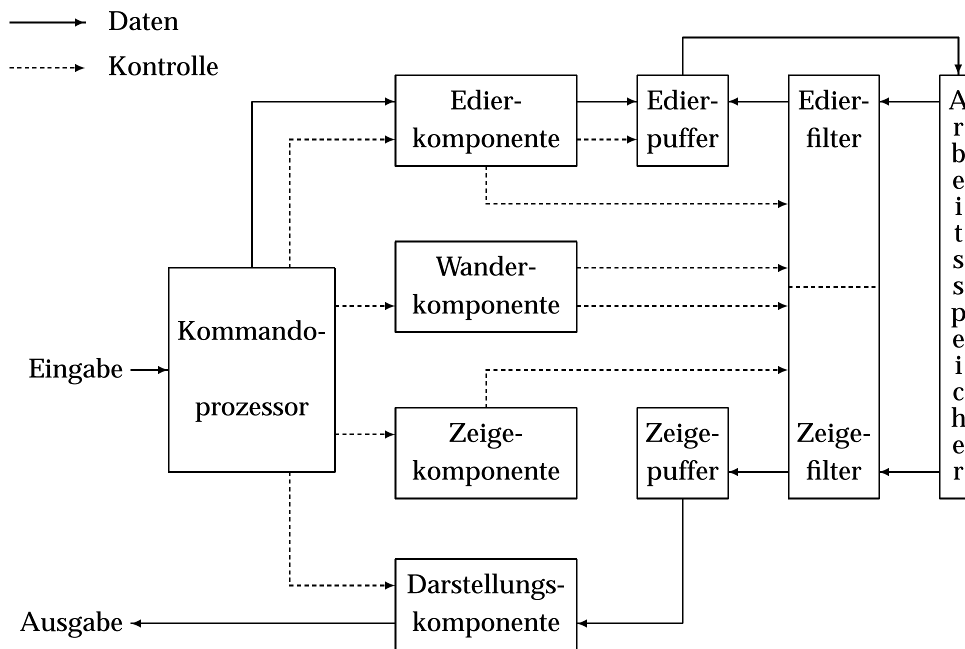

Abbildung 8.1: Editorarchitektur gemäß [MvD82]

eine Stelle im Text, und zwar die Stelle, an der eine zu diesem Zeitpunkt ausgeführte Edieroperation Effekt hat. Er ist explizit veränderbar über Wanderoperationen und implizit durch Seiteneffekte von Edieroperationen; wenn z. B. eine Zeile, in der sich der Edierpunkt befindet, gelöscht wird, sitzt er anschließend auf der darauffolgenden Zeile.

Der *Edierfilter* bestimmt aus dem Edierpunkt und einigen Editorparametern den Inhalt des *Edierpuffers*, das ist der Teil des Textes, der von einer Edieroperation betroffen wäre. Das kann z. B. das aktuelle Wort sein, die aktuelle Zeile oder der aktuelle Paragraph. Dabei ist *aktuell* immer dadurch definiert, daß dieses Objekt den Edierpunkt enthält. Er sitzt, selbst wenn man ihn nicht sieht, entweder am Anfang oder am Ende des Edierpuffers.

Zur Übertragung von Textstücken zwischen aufeinanderfolgenden Edieroperationen verfügen Editoren über weitere Puffer, sogenannte *Zwischenablagen* (clip boards). Gelöschte Textstücke werden da hineingespeichert (cut), um eventuell an anderer Stelle wieder eingefügt zu werden (paste). Für diesen Zweck können Texte auch hineinkopiert werden (copy). Damit kann man das cut-copy-paste-Editorparadigma realisieren.

Die *Zeigekomponente* wählt den Teil des Textes aus, der dem Benutzer gezeigt werden soll. Sie füllt mit Hilfe des Zeigefilters eine rechteckige Fläche mit Text. Dabei faltet sie eventuell zu lange Zeilen. Sie besitzt in Analogie zur Edierkomponente

einen *Zeigepunkt*. Er sitzt am Anfang des zu zeigenden Teils des Textes. Dieser Teil wird an die Darstellungskomponente übergeben, um auf einem Ausgabemedium für den Benutzer lesbar dargestellt zu werden. Der Zeigepunkt ist ebenfalls explizit durch Wanderoperationen und implizit durch Seiteneffekte von Edieroperationen veränderbar. Er ist auch unabhängig vom Edierpunkt verschiebbar, bei bildschirmorientierten Editoren z. B. durch Rolleisten. Der Zeigepunkt und einige Editorparameter bestimmen den Inhalt des *Zeigepuffers*, das ist dieses erwähnte Stück Text, welches an die Darstellungskomponente übergeben wird. Wird ein Stück Text gelöscht, welches Teil des Inhalts des Zeigepuffers ist, so sollte es aus der externen Darstellung verschwinden. Dazu stellt die Darstellungskomponente fest, daß sie die externe Darstellung auffrischen sollte. Sie ruft den *Zeigefilter* auf, welcher den neuen Inhalt des Zeigepuffers berechnet und diesen wieder an die Darstellungskomponente übergibt.

Es klingt so, als ob die Inhalte von Edier- und Zeigepuffer immer gekoppelt wären. Das muß aber nicht der Fall sein. Man kann z. B. mit assoziativen Änderungskommandos auch außerhalb des Zeigepuffers edieren. Die assoziative Suche nach einem Vorkommen eines gegebenen Textmusters bestimmt einen neuen Edierpunkt, nämlich den Anfang des gefundenen Vorkommens (wenn es eines gibt). Dort wird dann unsichtbar die gewünschte Edieroperation ausgeführt. Der Zeigepunkt wird (z. B. im *vi*) erst nach Ausführen der Edieroperation in die Umgebung des Edierpunktes gesetzt. In einem bildschirmorientierten Editor kann man ein Textstück aktivieren, dann mit der Rolleiste dieses Stück aus dem Fenster, also auch aus dem Zeigepuffer bewegen und dann das aktivierte Stück löschen. Anschließend aber wird der Zeigepunkt dem Edierpunkt wieder nachgeführt.

Es bleibt noch übrig, die Aufgabe der *Darstellungskomponente* zu beschreiben. Sie erhält den Inhalt des Zeigepuffers, der noch unabhängig vom Ausgabegerät ist, und stellt ihn auf dem Ausgabemedium dar. Dabei benutzt sie die Möglichkeiten des speziellen Geräts. Da der Neuaufbau eines Bildschirms eventuell lange dauert, besonders wenn er über eine langsame Leitung mit dem Rechner verbunden ist, vergleicht die Darstellungskomponente in diesem Fall den alten mit dem neuen Bildschirminhalt und vermeidet die Neuausgabe von unveränderten Teilen des alten Inhalts.

8.3 Die historische Entwicklung von Editoren

Interaktive Editoren kamen zusammen mit Betriebssystemen auf, die interaktives Arbeiten im Timesharing-Betrieb erlaubten. Bis dahin hatten Benutzer ihre Programme als Lochkartenstapel und ihre Daten auf Magnetbändern an der Tür ihres Rechenzentrums abgegeben. Zur Erinnerung für jüngere Generationen von Informatikern: Eine Lochkarte bestand aus 80 Spalten. In jeder der Spalten konnte durch eine entsprechende Kombination von Löchern ein Zeichen dargestellt werden. Auf eine Karte paßte also eine maximal 80 Zeichen lange Anweisung. Meist gingen die hinteren 8 Spalten noch für eine Kartennummer ab.

Der einzige mit dem Rechner zur Interaktion befähigte Mensch war der Operateur an seiner sogenannten Operateurskonsole. Dort konnte er die Prozesse im Rechner verfolgen und helfend eingreifen. Der Benutzer bekam seinen Kartenstapel nach einiger Zeit zusammen mit einem Protokoll zurück. Auf dem Protokoll standen im günstigen Fall die Ergebnisse seines Laufs, im ungünstigen Fall eine Liste von Fehlermeldungen. War sein Programm fehlerhaft, so setzte er sich zur Korrektur an einen Lochkartenstanzer, legte seinen Kartenstapel oder auch nur die Menge der zu korrigierenden Karten in einen der Eingabeschächte und stanzte statt der fehlerhaften Karten neue. Dabei konnte er eventuell eine fehlerhafte Karte benutzen, um die korrigierte Karte zu erstellen. Seine Möglichkeiten waren die folgenden:

- Er konnte Spalten, also Darstellungen eines Zeichens, von der fehlerhaften Karte auf die neue kopieren.
- Er konnte Spalten auf der fehlerhaften Karte überspringen, ohne die neue Karte voran zu bewegen.
- Er konnte Spalten auf der neuen Karte stanzen, ohne die fehlerhafte Karte voran zu bewegen.

Damit hatte er einen sehr eingeschränkten Lochkarteneditor zur Verfügung, der natürlich elektromechanisch arbeitete.

8.3.1 Zeileneditoren

Dieses Editormodell stand Pate für einige der ersten *Zeileneditoren*. Sie erhielten ihren Namen davon, daß sie Texte abstrakt als Folge von (anfänglich gleichlangen) Zeilen ansahen und dem Benutzer entsprechend jeweils eine von ihm ausgewählte Zeile zur Ansicht und zur Korrektur anboten. Ein anderes Vorbild für Zeileneditoren war die elektrische Schreibmaschine mit einem Magnetspeicher. Hier konnte eine einmal geschriebene Zeile wieder aus dem Speicher geladen und mit einer sehr beschränkten Menge von Edieroperationen verändert werden. Beiden Vorbildern war gemeinsam, daß sie nur ein sequentielles Durchgehen des Textes erlaubten. Deswegen überrascht es nicht, daß frühe interaktive Editoren die Beschränkung auf sequentielle Vorwärtsverarbeitung von diesen Vorbildern übernahmen. Teilweise war hierfür aber auch die Tatsache verantwortlich, daß die zugrundeliegenden Dateisysteme nur sequentielle Dateien anboten.

Meist waren die Zeilen in einem Text durchnumeriert. Wandern, Löschen und Kopieren konnte man nur durch Angabe dieser Zeilennummern. Zeilen einfügen konnte man durch Angabe einer Nummer, die zwischen zwei existierenden Zeilennummern lag. Die Korrektur einer vorhandenen Zeile geschah meist dadurch, daß der Benutzer eine Korrekturzeile unter die zu ändernde Zeile schrieb. Ein `x` unter einem Zeichen etwa hieß „Zeichen löschen", ein `i` vor einer Zeichenkette und unter einem Zeichen besagte, daß die Zeichenkette vor oder hinter diesem Zeichen eingesetzt werden sollte.

Ältere Benutzer von Siemens-Rechenanlagen können sich sicher an den *EDT* erinnern, einen Editor dieses Typs. Er ist in Abschnitt 8.5.1 beschrieben.

Eine ebenfalls von den Vorbildern übernommene (und oft auch durch Eigenschaften des Dateisystems bedingte) Einschränkung war die feste Zeilenlänge. Alle Zeilen waren gleich lang oder hatten zumindest die gleiche Maximallänge. Das Herstellen von überlangen Zeilen führte meist zum Verlust des Zeilenendes und nicht etwa zum Anlegen einer neuen Zeile. Hier führte das treue Beharren auf dem konzeptuellen Modell der Vorbilder zu einem für ernsthafte Arbeit unakzeptablen Systemverhalten.

Ein großer Fortschritt war die Einführung von *mustergesteuerter Suche*, heute meist *assoziative Suche* genannt. Eine Anwendungsstelle einer Edieroperation konnte mittels eines angegebenen Textmusters statt über eine Zeilennummer oder langwieriges Blättern ausgewählt werden. Allerdings sah das Editormodell den Text immer noch als eine Folge von Zeilen. Vorkommen von Mustern wurden nur innerhalb von Zeilen gesucht. Ein Vorkommen, welches sich über ein Zeilenende erstreckte, konnte deshalb nicht mit Hilfe eines Musters gefunden werden.

Der nächste Fortschritt bestand in der Aufgabe der Vorstellung von einer Zeile als dem Äquivalent einer Lochkarte. Zeilen konnten jetzt variabel und fast beliebig lang sein. Eine Konsequenz dieser Änderung des Modells war die Betrachtung der Zeile als edierbares Objekt unabhängig von der externen Darstellung des Textes. Das führte zu größerer Effektivität der assoziativen Suche. Allerdings funktionierte sie immer noch nicht über Zeilengrenzen hinweg. Lange Zeilen konnten jetzt nicht mehr ganz in einer Bildschirmzeile dargestellt werden. Sie wurden in mehrere Bildschirmzeilen gefaltet.

Zwei weitere verbliebene Probleme waren, daß Zeilen, die das gegebene Längenlimit übertrafen, immer noch hinten gekürzt wurden, und daß Edieroperationen ähnlich wie die Suche nicht über Zeilenenden hinweg möglich waren.

8.3.2 Stromeditoren

Die obigen Beschränkungen zeilenorientierter Editoren wurden mit der Betrachtung des ganzen Textes als eines *Stroms* von Zeichen hinfällig. Edier- und Suchoperationen waren jetzt auf dem ganzen Text möglich. Meist wurden die Zeilenendezeichen im Text gelassen. Sie waren aber edierbar wie andere Zeichen und trennten für den Editor den Text nicht mehr in eine Folge von Zeilen. Der Text wurde nur zur Darstellung auf dem Bildschirm umbrochen, wobei die Zeilenendezeichen beachtet wurden. Der einflußreichste Stromeditor war *TECO* [BBN73] (Abschnitt 8.5.2).

8.3.3 Bildschirmorientierte Editoren

Das Edieren wurde erheblich komfortabler, seit Bildschirme mit adressierbaren *Bildschirmmarken* (Cursor) verfügbar wurden. Die Stelle, an der die Bildschirmmarke steht, ist jetzt impliziter Parameter der Edieroperationen. Der Operand muß

nicht mehr umständlich textuell beschrieben werden. Der Benutzer kann die Bildschirmmarke mit Kommandos an die ihn interessierende Stelle im Text bewegen. Dann kann er an dieser Stelle Zeichen vorwärts wie rückwärts löschen, vorhandene Zeichen überschreiben und neue einfügen.

Dieses Editormodell wurde sowohl mit der zeilenorientierten Sicht als auch mit der stromorientierten Sicht des Textes kombiniert. E.T. Irons [ID72] schlug ein Modell vor, welches den meisten modernen Editoren zugrundeliegt. Der Text wird jetzt als der Inhalt des rechten unteren Quadranten einer Textebene aufgefaßt. Das erste Zeichen des Textes befindet sich also im Ursprung dieser Ebene. Zeilen können beliebig lang sein, und der Text kann beliebig viele Zeilen enthalten. (Tatsächlich gibt es sowohl für die Zeilenlänge als auch für die Textlänge Begrenzungen, die allerdings selten von Benutzern erreicht werden.) Der gesamte Bildschirm oder ein Fenster auf einem graphischen Bildschirm dienen als Betrachtungsausschnitt auf den Text, unter dem der Benutzer jeweils die aktuelle Version des Textteils sieht. Die Bildschirmmarke, und damit gekoppelt der Ausschnitt, können mit Kommandos über den Text bewegt werden.

Die beiden populärsten Editoren in der UNIX-Gemeinde, der *vi* und der *emacs*, gehören zu dieser Klasse von Editoren. Sie werden in den Abschnitten 8.5.3 und 8.5.4 beschrieben.

8.3.4 Editoren mit graphischer Oberfläche

Ein weiterer entscheidender Fortschritt für den Komfort beim Textedieren bestand in der Erfindung der Maus und ähnlicher Zeigeinstrumente. D. Engelbart, der Erfinder der Maus, konnte mit ihr eine komfortable und über viele Rechneranwendungen uniforme Bedienoberfläche realisieren. Man kann Dateien durch Anklicken mit der Maus aktivieren und öffnen, Kommandos durch Anklicken einer entsprechenden Schaltfläche (Icons) ausführen, Menüs auf- und zumachen, Punkte darin auswählen und beliebige zusammenhängende, auch zeilenübergreifende Stücke eines Texts durch Überstreichen mit der Maus als Operanden für Edieroperationen auswählen. Diese Stücke können ausgeschnitten, kopiert und an anderer Stelle eingesetzt werden (cut, copy, paste). Das erste kommerzielle System dieser Art war der Xerox STAR [Xer82].

8.3.5 Struktureditoren

Manche Textsorten, z. B. Programme, haben eine durch ein System von Regeln, eine Grammatik, festgelegte Struktur. Nur Zeichenketten, die mittels dieser Grammatik ableitbar sind, werden als korrekt akzeptiert. Beispiele hierfür finden sich in Abschnitt 3.7.

Eine while-Schleife in Pascal etwa ist durch die folgende Produktion beschrieben:

while-Anweisung → **while** *Bedingung* **do** *Anweisung*

while und **do** sind dabei *Schlüsselwörter*. Sie dürfen nur in bestimmten Kontexten verwendet werden, also z. B. um Schleifen zu definieren. *Bedingung* und *Anweisung* sind sogenannte Nichtterminale. Für sie gibt es wieder Produktionen, die Zeichenreihen geeigneter Struktur ableiten.

Die Idee der *Struktureditoren*, auch *syntaxorientierte Editoren* genannt, ist es, das Wissen über den Aufbau der Programme in den Editor zu stecken. Wenn der Benutzer also `while` tippt oder mit der Maus die while-Schaltfläche klickt, so wird an der aktuellen Einfügungsstelle eine while-Schablone erzeugt, die aussieht wie die rechte Seite der Produktion:

while *Bedingung* **do** *Anweisung*

while und **do** sind dabei neue Textbestandteile, und *Bedingung* und *Anweisung* sind Platzhalter. Sie können durch Schablonen für Bedingungen und Anweisungen ersetzt werden. Wenn ein Benutzer sein ganzes Programm mit Hilfe von solchen Schablonen erstellt, so kann er kaum noch syntaktische Fehler begehen. Nur bei der Eingabe terminaler Bestandteile, also von Konstanten, Bezeichnern und Operatoren, kann er Fehler machen, die aber leicht lokal korrigiert werden können.

Die der Programmiersprache zugrundeliegende Grammatik liefert die Abstraktionen, mit denen der Editor arbeitet. Der Benutzer kann als Operanden von Edieroperationen syntaktische Konstrukte der Sprache, also Anweisungen, Ausdrücke, Deklarationen usw. angeben. Es macht keinen Sinn, daß sich Benutzer und Editor über Zeilen oder Worte unterhalten.

Die Schachtelung syntaktischer Konstrukte definiert auf natürliche Weise eine hierarchische Struktur von Programmen, den (konkreten oder abstrakten) Syntaxbaum. Ihm ähnelt auch die Datenstruktur für die Darstellung von Programmen im Speicher. Der Editor hat in dieser Datenstruktur einen etwas anders gearteten Edierpunkt, nämlich einen Zeiger auf die Wurzel des Baums für das aktuelle Konstrukt. Dieser Edierpunkt ist mit der externen Darstellung gekoppelt, meist durch eine hervorgehobene Darstellung des aktuellen Konstrukts. Die Zeigekomponente ist dafür verantwortlich, daß die interne baumartige Datenstruktur in eine gut lesbare externe Darstellung mit geeigneter Einrückung umgebrochen wird. Diese Aufgabe heißt im Englischen *Pretty Printing*.

Struktureditoren sind für Anfänger gut geeignet. Diese verbringen erfahrungsgemäß einen großen Teil ihrer Programmierübungen damit, Syntaxfehler aus ihren Programmen zu entfernen. Erfahrene Programmierer beherrschen die Syntax und sind eher genervt durch diese aufwendige Methode, Programme zu schreiben. Ein System zum teilweise automatisierten Erstellen strukturorientierter Editoren ist der Cornell Program Synthesizer Generator [RT89].

8.3.6 Hypertexteditoren

Hypertexte, also etwa Texte des SGML-Dokumenttyps HTML (siehe Abschnitt 3.5), enthalten als neue Elemente *Verweise* (links). Diese sind in der HTML-Quelle textuell

dargestellt. Ein Verweis auf ein Dokument in der Datei `document`, der mit dem Namen `Doku` dem Leser gezeigt werden soll, sieht wie folgt aus:

```
<A HREF="document.html">Doku</A>.
```

Solche Quelltexte sind genau wie normaler Text edierbar.

Die meisten HTML-Dokumente werden durch sogenannte Filter aus anderen Dokumenten, etwa LaTeX-, *FrameMaker*- oder *Word*-Dokumenten, mehr oder weniger automatisch erzeugt. Für den Rest gibt es sogenannte HTML-Editoren, die den Benutzer bei der Erstellung von HTML-Dokumenten auf verschiedene Weise unterstützen. Einige HTML-Editoren, wie z. B. der im Netscape Navigator Gold eingebaute, arbeiten wie graphische Editoren. Der Benutzer findet auf einer Fensterleiste die wesentlichen Bestandteile von HTML-Doumenten vor. Instanzen davon lassen sich an mit der Maus gewählten Dokumentstellen einfügen. Der Editor fragt nach notwendigen Zusatzinformationen und erzeugt dann die dazu korrespondierende HTML-Spezifikation. Außerdem bietet er dem ungeübten Benutzer Beispiele für alle komplexen Objektarten an, die er kopieren und dann abändern kann.

8.4 Funktionalität von Editoren

Auf der Basis einer gewissen Kenntnis der Architektur von Editoren kommen wir zurück zur Benutzersicht, und zwar zu einer etwas detaillierteren Darstellung der Funktionalität von Editoren. Die Funktionalität besteht in der Menge der Funktionen, die der Editor dem Benutzer anbietet. Dem Benutzer ist es wichtig, *was* diese Funktionen tun. Der Implementierer hat sie zu realisieren und zwar effizient. Wir bezeichnen jetzt häufig etwas als *intern*, wenn es ein Teil der Implementierung ist, und als *extern*, wenn es ein Teil der Benutzersicht des Editors ist. Wir beschränken uns dabei auf die Funktionalität heute noch gebräuchlicher Editoren.

8.4.1 Wandern

Wandern verändert den Edierpunkt und oft gekoppelt damit den Zeigepunkt. Der *vi* z. B. behält den Edierpunkt immer im Zeigepuffer. Würde er herauswandern, wird der Zeigepunkt nachgezogen.

Nach der Bewegung des Edierpunkts wird automatisch ein neuer Inhalt des Edierpuffers berechnet. Der Edierpunkt ist eine Stelle in der internen Datenstruktur des Editors. Mit ihm ist die Schreibmarke gekoppelt , die an der entsprechenden Stelle in der externen Darstellung des Textes steht.

Wanderoperationen erstrecken sich je nach den vom Editor angebotenen Abstraktionen vom einfachen zeichen-, wort-, zeilen- oder abschnittsweisen Durchlaufen des Textes, über das bildschirmweise Blättern, die Bewegung zu symbolischen Marken, die assoziative Suche, das Durchlaufen hierarchischer Strukturen bis zum Verfolgen von Hyperlinks zu anderen Dokumenten.

Eine gewisse Grundausstattung an Wanderfunktionen besitzt jeder Editor. Dazu gehören „nächstes Zeichen“, „vorangehendes Zeichen“ „nächste Zeile“ und „vorangehende Zeile“. Diese kann man z. B. mit den Cursortasten auslösen. Beim Erreichen des Zeilenendes bzw. -anfangs schaltet der Editor zur nächsten bzw. vorangehenden Zeile fort oder verharrt auf dem erreichten Zeichen. Beim Anstoßen an den oberen bzw. unteren Fensterrand wird meist das Fenster um eine Zeile herauf- bzw. heruntergerollt.

Weiß der Editor, was ein Wort ist, z. B. eine nichtleere und nicht nur aus Leerzeichen bestehende Zeichenkette, die von Leerzeichen oder Zeilenanfängen bzw. -enden begrenzt ist, so erlaubt er auch die Wanderung vorwärts oder rückwärts über eine gewisse Anzahl von Worten hinweg. Will sich der Benutzer den Text konsekutiv ansehen, so wird er dazu das fensterweise „Blättern“ verwenden.

Editoren mit graphischer Oberfläche unterstützen das Wandern mit Hilfe des Zeigeinstruments durch Hinzeigen und Klicken in den sichtbaren Textbereich. Dabei werden Edier- und Zeigepunkt kombiniert bewegt. Der Zeigepunkt kann aber auch getrennt vom Edierpunkt bewegt werden, und zwar durch das Rollen des Fensters.

Zusätzlich bieten Editoren assoziative Suche an. Dies reicht vom Suchen nach Textstücken bis zum Suchen nach Vorkommen von recht komplexen Textmustern. Im Zeileneditor *ex*, den man aus dem Editor *vi* ansprechen kann, kann man reguläre Ausdrücke zur Spezifikation von Mustern angeben. Einige der Möglichkeiten sind die folgenden:

- Jedes „gewöhnliche“ Zeichen steht für sich selbst.
- Ein Punkt steht für jedes Zeichen außer dem Zeilenendezeichen.
- Sei w eine Folge von Zeichen. Dann steht $[w]$ für jedes beliebige Zeichen aus w.
- r^* steht für beliebige Folgen von Worten, für die r steht.

Hier ist ein Beispiel. Das Muster `a[bc]*.` steht für Worte, die mit `a` anfangen, sich mit beliebig langen Folgen bestehend aus `b`s und `c`s fortsetzen und in einem beliebigen Zeichen, welches nicht das Zeilenendezeichen ist, enden.

8.4.2 Zeigen und Darstellen

Hier geht es um die Darstellung des aktuellen Bildes eines Textteils. Die beiden Funktionen werden in den folgenden Schritten ausgeführt:

1. Bestimmung des Inhalts des Zeigepuffers.
2. Formatierung, also Zeilenumbruch, Pretty Printing usw.
3. Abbildung auf das physikalische Ausgabegerät.

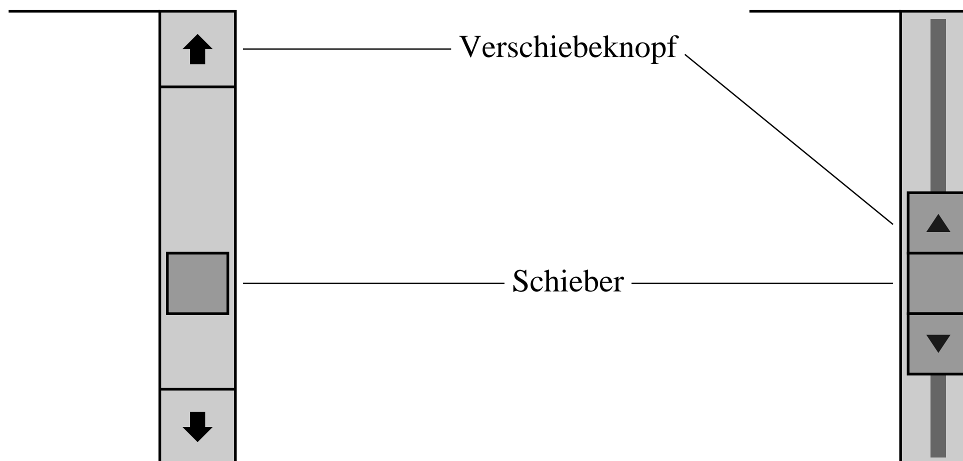

Abbildung 8.2: Vertikale Bildrolleisten in MS Word (links) und Sun Textedit (rechts)

Der Inhalt des Zeigepuffers wird durch den aktuellen Zeigepunkt und einige Editorparameter bestimmt. Letztere kann meist der Benutzer setzen. Einige Editoren bieten z. B. an, Bemerkungen in den Text zu schreiben. Diese Bemerkungen möchten und dürfen manche Benutzer sehen, manche aber auch nicht. Deshalb kann man einstellen, ob die Zeigekomponente diese Bemerkungen mit zur Darstellung auswählt oder nicht.

Editoren mit graphischer Oberfläche erlauben ein Verschieben des Zeigepunktes unabhängig von der Bewegung des Edierpunktes durch Rollen des Fensters. Bildrolleisten, siehe Abbildung 8.2, erlauben verschiedenartiges Rollen des Fensters. Der Benutzer kann mit der Maus den *Schieber* der Rolleiste anklicken und ziehen. Damit gekoppelt bewegt sich dann das Fenster. Außerdem kann er die *Verschiebeknöpfe* anklicken und festhalten. Damit rollt das Fenster zeilenweise über den Text. Zum dritten kann er die Bildrolleiste oberhalb bzw. unterhalb des Schiebers anklicken. Dann rollt das Fenster jeweils um eine Fenstergröße nach oben bzw. unten. Im X-Windows-Fenstersystem (mit twm) gibt es nur den Schieber, keine Verschiebeknöpfe. Mit der mittleren Maustaste kann man den Schieber ziehen oder durch einen Klick auf eine bestimmte Stelle springen lassen. Ein Druck auf die linke Maustaste bewegt den Schieber ein kleines Stück nach oben und damit den Text im Fenster nach unten. Dabei ist es gleichgültig, ob der Klick über, unter oder im Innern des Schiebers erfolgt. Die rechte Maustaste bewirkt eine entsprechende Verschiebung nach unten.

Für die Formatierung gibt es eine große Breite an Möglichkeiten. Diese beginnt mit dem Falten von Zeilen, die länger als die Fensterbreite sind. Dabei beginnt die Darstellung jeder Textzeile am Anfang einer Fensterzeile. Hier kann man mitunter einstellen, ob hinter jedem beliebigen Zeichen oder nur hinter Wortgrenzen gefaltet wird. Wenn, wie z. B. im *vi*, das Falten optisch nicht erkennbar ist, kann man eine

explizite Darstellung der Zeilenendezeichen einschalten.

Im nächsten Schritt kann man die Beziehung zwischen internen Textzeilen und Zeilen der externen Darstellung aufgeben, indem ein Zeilenumbruch über Paragraphen gemacht wird. Einstellbar sind dann die verschiedenen Arten der Zeilenausrichtung, die wir in Abschnitt 4.3 kennengelernt haben: linksbündig, rechtsbündig oder mit Randausgleich. Programme werden im allgemeinen durch Einrückungen lesbar gemacht. Jede weitere Schachtelung von Anweisungen, Blöcken und Deklarationen wird durch eine weitere Einrückung dargestellt.

Formatierung ist ein weites Feld, welches in diesem Buch in den Abschnitten 4.3 und 4.4 gesondert behandelt wird. Die Abbildung auf ein physikalisches Ausgabegerät wird in Kapitel 9 besprochen.

8.4.3 Edieren

Jede Edieroperation braucht eine Spezifikation ihres Anwendungsbereichs. Für manche Operationen ist er implizit gegeben; für manche muß der Benutzer ihn angeben. Der Anwendungsbereich kann die Einheiten der

- internen Darstellung des Dokuments, also Zeichen und Zeilen im Zeileneditor und Teilbäume in einem Struktureditor, oder
- der Abstraktion des Benutzers, also Worte, Abschnitte, Listen, Überschriften, Anweisungen und Deklarationen, benutzen.

Einige Techniken zur Spezifikation des Anwendungsbereichs wurden schon beschrieben. Man kann z. B. den Bereich über reguläre Ausdrücke spezifizieren. In Editoren mit einer textuellen Kommandosprache kann man gewisse physikalische bzw. logische Einheiten als Operanden in Kommandos angeben, z. B. Zeilen oder Worte.

In Editoren mit graphischer Oberfläche kann man den Bereich durch Überstreichen mit der Maus bei gedrückter Maustaste spezifizieren, und auch durch Hinzeigen, Klicken und darauffolgendes *Ausdehnen* (adapt). Das Ausdehnen kann verschiedene Formen haben. Es setzt eine Schachtelung von logischen Einheiten voraus, z. B.

- Zeichen, Worte, Zeilen, ganzer Text oder
- Zeichen, Worte, Sätze, Paragraphen oder
- Ausdruck, elementare Anweisung, Anweisungsliste, Block, strukturierte Anweisung.

Jeder Ausdehnungsschritt führt von der aktuell ausgewählten Einheit zur nächsten umfassenden. Er kann durch einen Mausklick auf die aktuelle Einheit oder auf einen Menüpunkt „Ausdehnen“ ausgelöst werden.

Oft werden Stufen einer Hierarchie auch ansteigenden Zahlen schnell aufeinanderfolgender Mausklicks zugeordnet. Ein Mausklick setzt den aktuellen Zeiger neu, und zwar zwischen zwei Zeichen. Zwei Mausklicks wählen das umfassende Wort aus, drei die umfassende Zeile und vier den ganzen Text.

Eine weitere, nicht unwichtige Funktion ist das Erzeugen von Text. Manche Editoren sind ständig in einem *Einfügemodus*. Wann immer der Benutzer etwas tippt, wird es an der Stelle der Schreibmarke eingefügt. Andere kennen daneben auch einen *Überschreibemodus*. In diesem überschreibt der Editor Zeichen für Zeichen den Text hinter der Schreibmarke. Da man selten einen Text durch einen genau gleich langen ersetzen wird, ist es sinnvoll, das Überschreiben auf den Anwendungsbereich zu beschränken.

Gelöscht wird immer der aktuelle Anwendungsbereich. Löschen ist eine oft mit Ängsten behaftete Aktion, weil der Benutzer annimmt, was (eventuell unabsichtlich) gelöscht wurde, sei ein für alle mal weg. Deshalb ist es wichtig, diese Aktion „angstfrei" zu gestalten. Dazu gehören das Absichern gegen unabsichtliches Löschen durch die Nachfrage nach einer Bestätigung, die Möglichkeit, die Löschaktion rückgängig machen zu können (undo), und die Zwischenspeicherung des gelöschten Textstücks in einem Löschpuffer.

Das Ändern des Anwendungsbereichs kann man sich theoretisch wie die Kombination von Löschen mit anschließendem Einsetzen vorstellen. Insofern treffen die oben genannten Punkte bezüglich der Absicherung gegen unabsichtliche Fehler auch darauf zu. In der Praxis bieten Editoren aber eigenständige Änderungsoperationen an. Dazu gehören das Überschreiben von Text, die Substitution eines Textes (des aktuellen Anwendungsbereichs) durch einen anderen Text und das assoziative Ersetzen eines Vorkommens eines Suchmusters durch das korrespondierende Vorkommen eines Ersetzungsmusters. Letztere Operation kann man auch global, also auf alle Vorkommen eines Suchmusters im Text oder auf alle Vorkommen in einem angegebenen Bereich anwenden. Als letztes bleibt die Zeichenvertauschung, die recht nützlich ist, wenn ein Benutzer dazu neigt, Verdreher zu produzieren.

Es wurde schon oben beschrieben, mit welchen Mitteln man Textstücke kopieren und verschieben kann. Meist werden dazu Ausschneiden, Kopieren und Einsetzen (cut, copy, paste) benutzt. Als Hilfsmittel dienen dabei Zwischenablagen. Diese können implizit sein. Dann landet jeder ausgeschnittene oder kopierte Text in einer von eventuell mehreren Zwischenablagen, die als Stapel oder als Schlange organisiert sind. Oder es kann zusätzlich wie beim *vi* benannte Zwischenablagen geben. Dann kann der Benutzer sowohl beim Abspeichern wie beim Entnehmen den Namen einer Zwischenablage angeben.

8.5 Einige Fallbeispiele

In diesem Abschnitt betrachten wir kurz einige Editoren aus den Klassen, die im historischen Überblick beschrieben wurden. Die Editoren *EDT* und *TECO* sind dabei von eher historischem Interesse.

8.5.1 *EDT*

Der *EDT* ist ein Editor des BS2000-Betriebssystems von Siemens [Sie80]. Er gehört zur Klasse der Zeileneditoren. Er nutzt nicht die ganze Fläche des Bildschirms, sondern macht Ausgaben und erwartet Eingaben immer nur am unteren Bildschirmrand. Der bereits vorhandene Bildschirminhalt verschiebt sich dabei nach oben.

Der *EDT* verwaltet eine Folge von mit Zeilennummern versehenen Zeilen. Am Beginn des Edierprozesses und nach einem Kommando `@renumber` entsprechen diese Zeilennummern der Folge der natürlichen Zahlen 1, 2, 3, ... Im Verlaufe des Edierprozesses können beliebige aufsteigende Folgen von Dezimalbrüchen als Zeilennummern entstehen, z. B. 1, 2, 2.2, 2.3, 2.756, 5, 7.3, ... Der Benutzer „befindet sich“ zu jedem Zeitpunkt in einer dieser Zeilen (aktuelle Zeile). Eingegebener gewöhnlicher Text wird zum Inhalt dieser Zeile, während Kommandos, die immer mit `@` beginnen, andere Effekte haben.

Von Natur aus, d. h. ohne spezielle Kommandos gegeben zu haben, hat der *EDT*-Benutzer keinen Blick auf den Editorinhalt, die Art seiner Numerierung oder die aktuelle Zeile. Erst spezielle Zeigekommandos bewirken die Ausgabe einer Reihe numerierter Zeilen. Der Benutzer kann die ausgegebenen Zeilen lesen, aber sonst nichts weiter direkt damit anfangen. Das Ausgeben von Zeilen hat auch keine Auswirkung darauf, welche Zeile die aktuelle Zeile ist.

Neben diesen Sichtkommandos gibt es Kommandos zum Löschen, Einfügen und Verändern von Zeilen. Löschkommandos bewirken das Verschwinden der angesprochenen Zeilen, ohne daß sich die Numerierung der übrigen Zeilen ändert. Beim Einfügen einer numerierten Zeile besteht die Gefahr, eine bereits existierende Zeile mit derselben Nummer zu überschreiben. Daraus ergibt sich andererseits eine sehr grobe Methode zum Ändern von Text.

Feinere Veränderungen können durch ein Kommando ausgelöst werden, das ein Stück Text durch ein anderes ersetzt. Um z. B. einen typischen Rechtschreibfehler zu korrigieren, muß der Benutzer sinngemäß eingeben

```
@in Zeile 17, ändere 'Dokumnet' in 'Dokument'.
```

(Dafür gibt es eine spezielle Syntax, mit der wir den Leser aber nicht belasten wollen.) Das System sucht dann das erste Vorkommen der Zeichenkette „Dokumnet“ in der betreffenden Zeile und ersetzt es durch „Dokument“. Diese Zeichenketten müssen nicht unbedingt ganze Wörter sein; ein Befehl mit `'ne' in 'en'` würde dasselbe tun, vorausgesetzt die Zeichenkette „ne“ kommt nicht irgendwo vor „Dokumnet“ in der betreffenden Zeile vor. Bei dieser verkürzten Form des Befehls würde z. B. aus „In einem Dokumnet“ der noch schlimmere Text „In eienm Dokumnet“.

Änderungsbefehle können auch auf Zeilenbereiche oder auf den gesamten Text angewandt werden. Es gibt dabei auch eine Variante, die nicht nur das erste Vorkommen, sondern alle Vorkommen der angegebenen Zeichenkette ersetzt.

Im *EDT* sind noch viele andere Befehle verfügbar, die hier nicht besprochen werden. Dazu gehört insbesondere ein mächtiges (aber schwer zu benutzendes) Makrosystem, das es geschickten Benutzern erlaubt, sich selbst neue Kommandos aus den bestehenden zu definieren.

Mit Hilfe dieser Makros wurde eine *EDT*-Erweiterung programmiert, die bequemeres Ändern von Zeilen ermöglicht. Die zu ändernde Zeile wird auf dem Bildschirm dargestellt, und der Benutzer kann in einer Bildschirmzeile direkt darunter Veränderungen spezifizieren. Gewöhnliche Buchstaben ersetzen die darüber stehenden und die Sonderzeichen `/` und `^` dienen zum Löschen und Einfügen. Im folgenden Beispiel ist also die obere Zeile die Ausgabe des *EDT*, die untere die Eingabe des Benutzers.

```
In ainem Dokumnet gibtt es es Texte und Bider.
   e          en      /    ///          ^l
```

Diese Art, Änderungen zu spezifizieren, erinnert noch sehr stark an den im Abschnitt 8.3 geschilderten Lochkarteneditor.

8.5.2 *TECO*

Der Editor *TECO* (Text Editing and Corrector) [BBN73] der Firma *Digital Equipment Corporation* ist ein prominentes Mitglied der Familie der Stromeditoren. Er erlaubt das sequentielle Bearbeiten von „Seiten" einer Datei. Eine Seite entspricht dabei etwa der Größe des Editorpuffers, d. h. des dem Editor zur Verfügung stehenden Arbeitsbereichs im Hauptspeicher. Durch Einfügen und Löschen kann eine Seite in gewissen Grenzen wachsen oder schrumpfen, ohne daß auf den Hintergrundspeicher geschrieben werden muß.

Die Seiten werden nacheinander von einer *Eingabedatei* eingelesen, bearbeitet und in eine *Ausgabedatei* ausgegeben. Der Befehl `P` bewirkt das Wechseln zur nächsten Seite. Zurückblättern auf frühere Seiten ist nicht möglich. Um eine Korrektur auf einer früheren Seite durchführen zu können, muß *TECO* neu gestartet werden. Zum Abschluß eines *TECO*-Laufs muß der ganze Rest der Eingabedatei auf die Ausgabedatei übertragen werden.

Jede Seite besteht aus Zeilen, die aber nicht explizit numeriert sind. Zum aktuellen Zustand gehört außer dem Inhalt der aktuellen Seite eine aktuelle Position, die in einer bestimmten Zeile zwischen zwei Zeichen liegt. Diese Zeile heißt die aktuelle Zeile. Ähnlich wie beim *EDT* sind allerdings weder die aktuelle Seite noch die aktuelle Position direkt sichtbar. Der Benutzer muß Informationen über den Seiteninhalt explizit anfordern:

`T` gibt den Text von der aktuellen Position bis zum Ende der aktuellen Zeile aus.

`0T` gibt die ganze aktuelle Zeile aus.

n`T` gibt *n* Zeilen aus.

`HT` gibt die ganze aktuelle Seite aus.

Die aktuelle Position kann zeichenweise, zeilenweise oder relativ zur ganzen Seite verändert werden. Der Befehl `nC` rückt die aktuelle Position um *n* Zeichen weiter. Die Zahl *n* kann dabei negativ sein. Eine fehlende Zahl wird als 1 angenommen. Der Befehl `nL` rückt die aktuelle Position an den Anfang der *n*-ten Zeile hinter der aktuellen Zeile. Wieder kann *n* negativ sein, und fehlendes *n* wird als 1 betrachtet. Der Befehl `L` rückt also die aktuelle Position an den Anfang der nächsten Zeile, `-1L` an den Anfang der vorhergehenden und `0L` an den Anfang der aktuellen. Der Befehl `J` bewegt an den Anfang der Seite und `ZJ` ans Ende.

Weiterhin gibt es auf ähnliche Art aufgebaute Befehle zum Löschen einiger Zeichen, einiger Zeilen oder der ganzen aktuellen Seite. Durch den Befehl `I`*text*`$` wird der angegebene Text an der aktuellen Position eingefügt.

Schließlich gibt es noch einen Befehl, der nach einer vom Benutzer angegebenen Zeichenkette sucht und die aktuelle Position ans Ende des ersten Vorkommens dieser Zeichenkette setzt. Dieser Befehl hat zwei Varianten: eine lokale, die nur auf der aktuellen Seite sucht, sowie eine globale, die solange weiterblättert, bis die Zeichenkette gefunden wird oder das Ende der Eingabedatei erreicht ist. Es gibt auch Befehle, die eine Zeichenkette suchen und durch eine andere ersetzen.

Der Benutzer kann sich selbst neue *TECO*-Befehle in einer kleinen Programmiersprache definieren. Diese Sprache verfügt über Marken und Sprünge, bedingte Befehle und Schleifen fest vorgegebener Länge. Zahlen können in „Registern“ abgespeichert und manipuliert werden. Auch Textstücke können in Registern gespeichert werden und von dort entweder in den Haupttext eingefügt oder als Kommandofolge ausgeführt werden.

8.5.3 *vi*

Der *vi* [JH80b] ist ein bildschirmorientierter Editor: Während der Arbeit mit dem *vi* wird immer eine gewisse Anzahl von Zeilen am Bildschirm dargestellt. Darin befindet sich irgendwo die Bildschirmmarke (Cursor), die die aktuelle Position bzw. die Lage des Edierpunkts anzeigt. Der Edierpunkt liegt immer auf einem Zeichen des Dokuments, das kein Zeilenendezeichen ist.

Die Bildschirmansicht entspricht im allgemeinen nicht exakt einem Abbild einer Folge von Zeilen aus dem Dokument. Zeilen, die länger sind, als der Bildschirm breit ist, werden nämlich „gefaltet“, das heißt, sie werden durch mehrere Bildschirmzeilen dargestellt. Optisch gibt es leider keinen Unterschied zwischen der Sicht auf zwei kurze Zeilen und der Sicht auf eine lange Zeile, die zwei Bildschirmzeilen einnimmt. Der Unterschied wird am Verhalten beim Bewegen, Löschen und Einfügen im Bereich dieser zwei Bildschirmzeilen erkennbar.

Der *vi* enthält Rudimente eines früheren nicht-bildschirmorientierten Editors, des *ex* [JH80a]. Die Funktionalität des bildschirmorientierten Anteils des *vi* ist genügend groß, so daß sich alle Operationen direkt oder mit einzelnen *ex*-Befehlen

durchführen lassen. Es ist nicht nötig, auf Dauer in den *ex*-Modus zu gehen und dort mehrere Befehle hintereinander abzusetzen, obwohl das möglich wäre.

Der *vi* hat beim bildschirmorientierten Arbeiten mehrere Modi: im Kommandomodus werden alle Tastendrücke als Kommandos interpretiert, während im Einfügemodus eingegebene Zeichen in den Text eingefügt werden. Ein Vorteil des *vi*-Prinzips ist, daß die meisten Kommandos nur einen einfachen Tastendruck erfordern. Die Existenz verschiedener Modi hat aber auch Nachteile. Da nicht angezeigt wird, in welchem Modus man sich gerade befindet, muß der Benutzer sich auf seine Erinnerung verlassen. Wenn er den aktuellen Modus vergessen hat, gibt es keine Möglichkeit zu entscheiden, ob z. B. ein Druck auf die Taste `x` den Buchstaben `x` in den Text einfügt oder das Zeichen unter dem Bildschirmzeiger löscht, ohne es auszuprobieren (oder ein ähnliches Experiment mit einer anderen Taste anzustellen).

Es gibt Befehle, die den Edierpunkt im Text verschieben oder die Sicht auf den Text verändern, d. h. den auf dem Bildschirm sichtbaren Textausschnitt anders wählen. Da der Edierpunkt durch die Bildschirmmarke dargestellt wird, gilt die Einschränkung, daß er immer im sichtbaren Teil des Textes liegen muß. Daher rufen sich positionsverändernde und sichtverändernde Operationen manchmal implizit gegenseitig auf. Ein Befehl, der den Edierpunkt verschiebt, läßt die Sicht unverändert, wenn seine neue Position im sichtbaren Bereich liegt; andernfalls wird die Sicht so weit verschoben, daß die neue Position auf jeden Fall sichtbar ist. Ein sichtverändernder Befehl läßt den Edierpunkt relativ zum Text unverändert, wenn die Sichtveränderung so klein ist, daß er sichtbar bleibt. Andernfalls wird der Edierpunkt als Seiteneffekt geeignet verschoben.

Der *vi* kennt nur Zeichen und Zeilen, aber keine Seiten. Er kennt weiterhin Wörter, Sätze, Absätze und Klammerpaare, die er nach festen Regeln erkennt, die vom Benutzer nicht verändert werden können. Für all diese Elemente gibt es Positionierungsbefehle. Es ist ferner möglich, nach einzelnen Zeichen innerhalb der aktuellen Zeile sowie nach Worten aus einer vom Benutzer gegebenen regulären Sprache im ganzen Text zu suchen. Der Benutzer kann auch die aktuelle Position (unsichtbar) markieren und später wieder anspringen.

Der Operator `d` gefolgt von einem beliebigen Positionierungsbefehl löscht das Textstück von der aktuellen Position bis zum Zielpunkt des Positionierungsbefehls. Für spezielle, häufig vorkommende Löschsituationen gibt es Abkürzungen. Der gelöschte Text ist nicht einfach weg, sondern befindet sich in einer Zwischenablage, von wo aus er an einer anderen Stelle wieder eingefügt werden kann. Der Operator `y` arbeitet ähnlich, kopiert aber das Textstück in die Zwischenablage, anstatt es zu löschen.

Der Befehl `.` wiederholt das letzte textverändernde Kommando an einer neuen aktuellen Position, soweit möglich. Der Befehl `u` macht den Effekt des letzten textverändernden Kommandos rückgängig. Eine Zahl vor einem Kommando gibt an, wie oft es wiederholt werden soll.

Auf den Befehl `:` hin öffnet sich eine Kommandozeile am Unterrand des Bild-

schirms. Dort können dann globale Ersetzungsbefehle und Dateizugriffe in der Sprache des Editors *ex* eingegeben werden. Nach jedem Befehl kehrt das System in den bildschirmorientierten Kommandomodus zurück, so daß auch der Effekt dieser Befehle sofort sichtbar wird.

Der *vi* kann mit vielen verschiedenen Bildschirmen effizient betrieben werden. Dazu müssen diese Bildschirme geeignet beschrieben werden. Die Beschreibungen werden in einer Art kleiner Datenbank abgelegt. Wenn der *vi* an einem Bildschirm bestimmter Art aufgerufen wird, dann schaut er in der Datenbank nach, welche Befehle er diesem speziellen Bildschirm geben muß, damit ein Zeichen verschwindet, der Text eins hoch oder eins runter rutscht usw.

8.5.4 *emacs*

Der *emacs* [Sta81, CR91] ist ein bildschirmorientierter Editor, der noch etwas moderner ist als der *vi*. Im Gegensatz zum *vi* gibt es nur einen Grundmodus. Oberflächlich betrachtet werden darin gewöhnliche Zeichen direkt in den Text eingefügt, während Kontrollzeichen Kommandos sind. In Wahrheit ist aber auch ein gewöhnliches Zeichen als Kommando anzusehen, dessen Effekt es ist, gerade dieses Zeichen in den Text einzufügen. Das macht einen Unterscheid, weil es dem *emacs*-Benutzer möglich ist, die Zuordnung von Tasten zu Kommandos beliebig zu verändern. Der Benutzer könnte also die Taste `x` statt an das Kommando „`x` einfügen“ an die Operationen „`u` einfügen“ oder „ein Zeichen löschen“ binden (was beides nicht besonders sinnvoll wäre).

Weiterhin zeichnet sich *emacs* durch eine ungewöhnlich mächtige Sprache zur Makrodefinition aus. Bei dieser Sprache handelt es sich nämlich um einen Lisp-Dialekt. Daher können im Prinzip alle berechenbaren Funktionen in *emacs* programmiert werden. Es gibt z. B. in *emacs* programmierte Spiele, die die Bildschirmorientiertheit direkt ausnutzen.

Mit dem *emacs* werden fertige Makropakete mitgeliefert, die den *emacs* an Anwendungen anpassen. Für jede bedeutendere Programmiersprache gibt es ein solches Paket, das die *emacs*-eigenen Begriffe von Wort, Satz oder Abschnitt der lexikalischen und syntaktischen Struktur dieser Sprache anpaßt und spezielle Kommandos zur Verfügung stellt, die die wichtigsten Sprachkonstrukte erzeugen und als Ganzheit bearbeiten.

Eine Weiterentwicklung von *emacs* erlaubt es zudem, mit der Maus im Text zu positionieren oder Textstücke als Argumente von Kommandos zu selektieren.

8.5.5 Edieren mit *FrameMaker*

Als interaktives Dokumentsystem muß *FrameMaker* die Funktionalität von Batch-Formatiersystemen mit der von Editoren vereinigen. *FrameMaker* bietet die folgenden grundlegenden Editorfunktionen:

Mit der Maus kann ein Einfügepunkt definiert werden. Wenn danach Text eingegeben wird, wird er an dieser Stelle eingesetzt. Mit der Taste „Backspace" kann Text links von der Einfügestelle gelöscht werden – Zeichen für Zeichen. Dies bezieht sich auf Text, der gerade eingetippt wurde wie auch auf solchen, der schon vorher da war.

Mit der Maus lassen sich auch in verschiedener Weise Dokumentteile selektieren, die dann als Argumente für darauffolgende Kommandos dienen können. Selektierte Dokumentteile werden weiß auf schwarz statt schwarz auf weiß dargestellt.

Mit der rechten Maustaste läßt sich (fast) jederzeit ein Menü aufrufen, das die folgenden Funktionen bietet:

- Undo (oder bei Tasteneingabe `!eu`, wobei in *FrameMaker*-Menüs das Ausrufezeichen für die Escape-Taste steht): Die letzte Veränderung wird rückgängig gemacht. In der deutschen Version heißt es übrigens „Rückgängig" statt „Undo" mit Tastaturkode `!br` statt `!eu`.
- Cut (`!ex`): Ein vorher selektierter Dokumentteil verschwindet aus dem sichtbaren Dokument und liegt dann in einer nicht sichtbaren Zwischenablage.
- Copy (`!ec`): Ähnlich, aber der Teil wird in die Zwischenablage kopiert, verschwindet also nicht aus dem Dokument.
- Paste (`!ep`): Der Inhalt der Zwischenablage wird an der Einfügestelle ins Dokument eingesetzt. Die Zwischenablage wird dadurch nicht gelöscht.
- Clear (`!eb`): Der selektierte Dokumentteil verschwindet, ohne daß die Zwischenablage verändert wird. Der betroffene Dokumentteil ist also außer durch ein „Undo" nicht mehr restaurierbar.

Das Kopieren von Dokumentteilen geht also folgendermaßen:

1. Selektiere den zu kopierenden Teil mit der Maus.
2. Rufe das Ediermenü auf und selektiere darin „Copy"
   (alternativ: tippe `!ec`).
3. Erzeuge einen Einfügepunkt mit der Maus.
4. Rufe das Ediermenü auf und selektiere darin „Paste"
   (alternativ: tippe `!ep`).

Es gibt auch eine sogenannte „Quick-Copy"-Funktion, die die Zwischenablage nicht benutzt. Voraussetzung dafür ist, daß der zu kopierende Text und die Zielstelle gleichzeitig sichtbar sind. Der Ablauf ist folgendermaßen:

1. Erzeuge einen Einfügepunkt mit der Maus.
2. Drücke die Meta-Taste und selektiere gleichzeitig den zu kopierenden Teil mit der Maus.

Die *FrameMaker*-Operationen respektieren die logische Struktur. Wenn z. B. eine Fußnotenreferenz[1] selektiert wird, dann erscheint nicht nur diese Zahl weiß auf schwarz, sondern auch der Text der Fußnote. Durch einen Cut verschwinden sowohl die Zahl als auch der Fußnotentext, die anderen Fußnoten werden umnumeriert und die Seiteneinteilung kann sich ändern. Nach einem Paste erscheint am Einfügepunkt eine Zahl, die Fußnote selbst erscheint am unteren Rand der Seite, die Fußnoten werden wieder neu durchnumeriert und die Seiteneinteilung kann sich ändern.

8.6 Implementierung von bildschirmorientierten Editoren

Im folgenden werden wir etwas genauer auf die Implementierung von bildschirmorientierten Editoren wie *vi* oder *emacs* eingehen.

8.6.1 Puffer und Sicht

Der bearbeitete Text (oder eine Kopie davon) liegt in einem Hauptspeicherbereich, genannt der *Puffer*. Er ist in Zeilen strukturiert, die aus Einzelzeichen bestehen. In diesem Text ist eine Position ausgezeichnet, die *aktuelle Position*. Beim *vi* liegt diese auf einem Zeichen, beim *emacs* konzeptuell zwischen zwei Zeichen.

Der Benutzer hat immer eine gewisse *Sicht* auf den Text. Ein bestimmter Teil des Textes um die aktuelle Position herum ist *sichtbar*, indem er auf dem Bildschirm dargestellt wird. Die aktuelle Position wird durch die Position des Bildschirmzeigers (Cursors) angedeutet.

Die Sicht entspricht im allgemeinen nicht exakt einem Abbild einer Folge von Zeilen aus dem Puffer. Im *vi* gibt es folgende Unterschiede zwischen der Sicht und einem Stück des Pufferinhalts:

1. Zeilen, die länger sind, als der Bildschirm breit ist, werden „gefaltet“, das heißt, sie werden durch mehrere Bildschirmzeilen dargestellt.
2. Ein Tabulatorzeichen im Puffer wird auf dem Bildschirm durch eine bis acht Leerstellen dargestellt.
3. Ein Kontrollzeichen im Text wird in der Bildschirmsicht durch zwei Zeichen dargestellt, z. B. „Cntl-A“ durch `^A`.

Beim Arbeiten mit dem Editor werden alle Operationen zunächst im Puffer durchgeführt. Die Sicht wird gemäß den oben genannten Regeln dann entsprechend verändert. Dabei ist darauf zu achten, daß Veränderungen am Bildschirm relativ zeitaufwendig sind. Insbesondere sollte der komplette Neuaufbau des Bildschirms möglichst vermieden werden.

1 Eine Zahl oben an einem Wort.

8.6.2 Organisation des Puffers

Für den Puffer muß eine Datenstruktur gefunden werden, die es erlaubt, die wichtigsten Operationen effizient durchzuführen. Wegen der eben erwähnten Langsamkeit von Bildschirmveränderungen dürfen Operationen, die die Sicht stark verändern, auch intern auf dem Puffer etwas länger dauern, ohne daß diese zusätzliche Verzögerung unangenehm auffällt.

Eine naheliegende Datenstruktur ist eine doppelt verkettete Liste von Zeilen. Jede Zeile besteht also aus einem Verweis auf die vorhergehende und die nachfolgende Zeile sowie einem Inhalt, der als Feld von Zeichen abgespeichert ist. Es ist günstig, auch noch die Zeilenlänge dazuzunehmen.

```
type  line  =
   record
      nextline:       ^line;
      previousline:   ^line;
      length:         integer;
      content:        string
   end
```

Die aktuelle Position ist durch ein Paar `(^line, integer)` aus einem Zeiger auf die aktuelle Zeile und dem Index des aktuellen Zeichens in dieser Zeile gegeben. Dazu kommen noch je ein Zeiger auf die erste und die letzte Zeile des Puffers sowie Informationen zur Verwaltung der Sicht.

Mit dieser Datenstruktur lassen sich leicht Zeilen einfügen und löschen. Auch Befehle wie „eine Zeile nach oben“ und „eine Zeile nach unten“ lassen sich leicht realisieren. Problematischer sind schon Befehle wie „10 000 Zeilen nach unten“ oder „gehe nach Zeile 10 000“, da dazu von der aktuellen bzw. der ersten Zeile aus 10 000 mal den `nextline`-Verweisen gefolgt werden muß. Dieser Aufwand ist im allgemeinen tolerierbar, da ein solcher Befehl nahezu sicher einen kompletten Bildschirmneuaufbau impliziert.

Betrachten wir nun Operationen innerhalb einer Zeile. Bewegungen nach links oder rechts sowie Ersetzungen sind unproblematisch. Beim Einfügen und Löschen von Zeichen dagegen muß der gesamte Rest der Zeile hinter der Veränderungsstelle verschoben werden. Bei „gewöhnlichen“ Texten, wo die Zeilen etwa so lang sind wie der Bildschirm breit, ist das noch kein Problem. Wenn der Implementierer allerdings auch das effiziente Bearbeiten extrem langer Zeilen ermöglichen will, muß er weitere Datenstrukturen einführen, etwa durch eine Zerlegung von Zeilen in eine Liste unabhängig manipulierbarer Teilzeilen.

8.6.3 Inkrementelle Sichtaufbereitung

Ein vollständiger Neuaufbau des Bildschirms ist zeitaufwendig und wirkt auch störend auf den Betrachter. Nach Edieroperationen, die die Sicht nur geringfügig

verändern, sollten also Bildschirmoperationen ausgelöst werden, die den Bildschirminhalt nur lokal verändern.

Moderne Bildschirme bieten Operationen zum Verschieben des Bildschirmzeigers, zum Löschen von Zeilenenden, zum Einfügen und Löschen von Zeichen, zur Ausgabe von Text ab einer bestimmten Position, zum Hinauf- oder Hinabschieben des Bildschirminhalts usw. an. Informationen darüber, wie diese Operationen ausgelöst werden und wie lange sie brauchen, sind in einer Bildschirmbeschreibung enthalten, die der Editor lesen kann. Die Aufgabe des Editors ist es, gewünschte Veränderungen des Bildschirms durch eine möglichst effiziente Folge von Bildschirmoperationen zu erzielen. Dabei muß der Implementierer darauf achten, daß die Zeit zum Finden einer guten Operationsfolge und ihrer Ausführung nicht die Zeit zur Ausführung einer schlechten Folge übersteigen sollte.

Eine relativ einfache Möglichkeit zur inkrementellen Sichtaufbereitung ist die folgende: der Editor bestimmt den genauen Effekt jeder möglichen Operation auf die Sicht und löst eine in seinem Programm dafür vorgesehene Folge von Bildschirmoperationen aus. Nehmen wir z. B. an, der Benutzer gibt den Befehl, die aktuelle Position um eine Zeile nach oben zu schieben. Sei die aktuelle Position in Zeile i an der Stelle j. Im einfachsten Fall besteht die entsprechende Sichtveränderung daraus, den Cursor eins nach oben zu versetzen, und der Editor kann die entsprechende Bildschirmoperation aufrufen. Von dieser Grundregel gibt es die folgenden Ausnahmen (von denen manche auch zusammen eintreten können):

- Wenn die Zeile i die erste des Textes ist, dann bleibt der Cursor, wo er ist, und der Bildschirm muß piepen oder auf andere Weise seinen Unmut äußern.
- Wenn die Position j in der Zeile $i-1$ infolge der Konventionen des Editors unerreichbar ist (weil sie z. B. beim *vi* jenseits des Zeilenendes liegt oder auf die nichtletzte Stelle eines tabulatorerzeugten Leerraums fällt), dann wird der Cursor nicht nur nach oben, sondern auch horizontal auf eine erreichbare Position verschoben.
- Wenn die Zeile i oder $i-1$ so lang ist, daß sie mehrere Bildschirmzeilen ausfüllt, dann kann es erforderlich sein, den Cursor auf dem Bildschirm um mehr als eine Zeile nach oben zu schieben.
- Wenn die Zeile i die oberste Zeile auf dem Bildschirm ist, dann ist Zeile $i-1$ gar nicht sichtbar. Der Editor muß dann den ganzen Bildschirminhalt nach unten schieben (wobei die unterste Bildschirmzeile aus der Sicht verschwindet) und die Zeile $i-1$ am Oberrand des Bildschirms ausgeben. Alternativ könnte er in einem solchen Fall den ganzen Bildschirm neu aufbauen. Für jeden Bildschirm gibt es eine gewisse Obergrenze für das Ausmaß einer Verschiebung des Inhalts nach unten (scrolling backwards), ab der der komplette Neuaufbau schneller ist. Ähnliches gilt bei Verschiebung nach oben (scrolling) mit einer eventuell anderen Obergrenze.

Das oben angedeutete Verfahren ist bei Vorliegen gewisser besonderer Situationen nicht optimal. Wenn z. B. in dem Fall, daß die Zeile i die oberste Bildschirmzeile ist,

alle auf dem Bildschirm sichtbaren Zeilen den gleichen Inhalt haben, dann ist das Verschieben des Bildschirminhalts nach unten unnötig. Wenn sogar noch die Zeile $i-1$ den gleichen Inhalt hat, dann bräuchte überhaupt nichts getan zu werden (was aber den Benutzer verwirren könnte).

Diese besonderen Situationen können ausgenutzt werden, wenn ein anderes Verfahren benutzt wird. Der Editor verfügt dabei über ein rechteckiges Feld der Größe des Bildschirms. Darin speichert er ab, wie (seiner Meinung nach) gerade der Bildschirm aussieht. Jede Edieroperation wird zunächst auf dem Puffer durchgeführt. Dann wird daraus eine neue Sicht berechnet und mit der in dem Feld verglichen, um eine möglichst gute Folge von Bildschirmoperationen zu finden, die die alte Sicht in die neue abbilden. Dieses Verfahren ist aufwendiger als das erstgenannte, liefert aber dafür in manchen Situationen bessere Ergebnisse.

Allgemein ist zu sagen, daß der Editor-Implementierer dafür sorgen muß, daß die Veränderungen im Puffer von den Veränderungen der Sicht korrekt nachvollzogen werden, so daß die von den Regeln des Editors geforderte Beziehung Puffer – Sicht immer erhalten bleibt. Mitunter wird diese Beziehung durch die Auswirkungen eines Implementierungsfehlers oder durch äußere Einflüsse gestört, wie z. B. durch eine auf dem Bildschirm auftauchende Botschaft des Systemverwalters. Danach ist sinnvolles Arbeiten in der Regel nicht mehr möglich. Die Editoren bieten für diesen Fall ein Kommando an, das den Bildschirm löscht und dann die Sicht vom Puffer aus vollständig rekonstruiert.