9 Seitenbeschreibungssprachen

Eine druckbare Darstellung ist (meist) das Ergebnis eines Formatierprozesses. Sprachen zur Beschreibung von druckbaren Darstellungen können nach verschiedenen Kriterien klassifiziert werden:

- ihrem Berechnungsmodell,
- ihrem Grad an Universalität,
- der Kompaktheit ihrer Darstellungen,
- ihrem Grad an Maschinen- und Geräteunabhängigkeit,
- ihrem Sprachenparadigma: deskriptiv oder prozedural.

In diesem Kapitel betrachten wir zunächst *Bitmatrizen*, dann in Abschnitt 9.2 *prozedurale* und *deskriptive* Sprachen im allgemeinen. Dort wird auch eine überblicksartige Beschreibung der beiden prozeduralen Sprachen *dvi* und *PostScript* gegeben. In den Abschnitten 9.3 und 9.5 werden sie dann detailliert besprochen. Dazwischen behandeln wir die Übersetzung von TEX's Kastentermen nach *dvi*. Am Schluß stellen wir in Abschnitt 9.6 *abstrakte Maschinen* als ein Hilfsmittel zum Erreichen von Geräteunabhängigkeit vor.

9.1 Bitmatrizen

Ein *Rasterausgabegerät* definiert auf seinem Ausgabemedium (Papier oder Bildschirm) eine große Zahl von Bildpunkten, die in Zeilen und Spalten angeordnet sind. Das Gerät ist in der Lage, die Bildpunkte einzeln zu behandeln. Im einfachsten Falle werden gewisse Bildpunkte schwarz gefärbt, die anderen bleiben weiß. Kompliziertere Geräte können Bildpunkte mit einem Grauton oder einer Farbe versehen.

Eine mögliche Seitenbeschreibung für ein solches Gerät ist eine Matrix, deren Einträge den einzelnen Bildpunkten entsprechen. Die Einträge sind im einfachen Schwarz-Weiß-Fall Bits, d. h. entweder „ja“ oder „nein“, die angeben, ob der betreffende Punkt zu schwärzen ist. Für die komplizierten Geräte wären die Einträge Zahlen, die einen Grauwert oder Codes für Farben darstellen. Wir nennen auch diese Matrizen in Erweiterung des Begriffes Bitmatrizen.

Bitmatrizen sind

- universell, denn alles, was (auf einem Rasterausgabegerät) druckbar ist, läßt sich mit ihnen darstellen;
- voluminös (mehrere MByte pro Seite bei hoher Auflösung); das ist nicht nur ein Platz-, sondern auch ein Zeitproblem, da die Daten ja vom Rechner zum Ausgabegerät transportiert werden müssen;
- geräte- bzw. auflösungsabhängig; eine Konversion in eine Bitmatrix für ein Gerät anderer Auflösung bzw. anderer Graustufen- oder Farbdarstellung ist im allgemeinen schwierig.

Das Arbeiten mit Bitmatrizen stellt große Anforderungen an die Kommunikation zwischen Rechner und Drucker und/oder den Speicherausbau des Druckers. Entweder schickt der Rechner eine vollständig aufbereitete Seite an den Drucker, der sie zwischenspeichert und dann den Druckvorgang beginnt, oder der Rechner schickt die Bitmatrix während des Druckvorgangs Stück für Stück an den Drucker. Zumindest Laserdrucker müssen dabei immer mit „Stoff“ versorgt sein, da sie ihre Drucktrommel nicht anhalten können. Also sind Rechner wie Drucker während des Druckvorgangs sehr mit der Übertragung der Bitmatrizen beschäftigt. Außerdem würde dieser Ansatz die Erstellung der Bitmatrix vom Formatierungsrechner verlangen und diesen dadurch zusätzlich belasten.

9.2 Prozedurale und deskriptive Sprachen

Die im vorangehenden Abschnitt eingeführten Bitmatrizen kann man sich als extrem einfach aufgebaute Programme vorstellen. In diesen Programmen gibt es im wesentlichen nur zwei Arten von Befehlen. Die eine Art wird durch ein 1-Bit bezeichnet und bewirkt, daß der Drucker einen Bildpunkt schwarz färbt und dann zum nächsten Bildpunkt vorrückt. Die andere, durch ein 0-Bit bezeichnete Art bewirkt das Vorrücken zum nächsten Bildpunkt ohne eine Schwarzfärbung.

Prozedurale Sprachen (Seitenbeschreibungssprachen)

Um die im vorigen Abschnitt erwähnten Nachteile von Bitmatrizen zu vermeiden, sind Drucker „intelligenter“ gemacht worden, so daß sie mehr Kommandos verstehen können als die beiden eben beschriebenen. Man hat ihnen dazu einen eigenen Prozessor und eigenen Speicher gegeben. Auf diesem eingebauten Rechner ist eine *abstrakte Maschine* realisiert. Eine abstrakte Maschine ähnelt einem käuflich erwerblichen *realen Rechner*. Sie hat also Speicher, Register und einen Prozessor mit einer Maschinensprache. Nur ist diese Maschine im allgemeinen einer speziellen Anwendung gewidmet, also als Allzweckprozessor schlecht zu gebrauchen, und nicht in Hardware, sondern in Software realisiert. In Abschnitt 9.6 werden abstrakte Maschinen eingehender besprochen.

In unserem Fall bietet die abstrakte Maschine eine zum Drucken von Dokumenten geeignete Menge von Funktionen an, wie z. B. „drucke ein a in Font *f*“ oder „ziehe eine Linie von Position (x_1, y_1) nach Position (x_2, y_2)“. Die Ausgabe des Textverarbeitungsprogramms ist dann ein Programm, welches solche Funktionen aufruft. Wir nennen ein solches Programm *prozedural*, da es nicht bloß beschreibt, wie eine Seite aufgebaut ist, sondern auch, wie sich dieser Seitenaufbau durch einzelne Druckaktionen erzeugen läßt.

Meist ist ein solches Programm, dessen Ausführung ein bestimmtes Druckbild produziert, erheblich kleiner als eine entsprechende Bitmatrix. Dadurch wird die Kommunikation zwischen Rechner und Drucker stark reduziert. Die Universalität von Bitmatrizen kann oft simuliert werden, aber auf Kosten der Kompaktheit.

Das Erzeugen einer prozeduralen Seitenbeschreibung beansprucht das Dokumentverarbeitungssystem weniger als das Erzeugen einer Bitmatrix, da Operationen zum Drucken von Buchstaben und Linien viel besser dem in Dokumentsystemen normalerweise verwendeten Abstraktionsniveau entsprechen als das Drucken einzelner Bildpunkte.

Die Gesamtheit der Programme, die von einem bestimmten Druckertyp (genauer: von der in diesen Typ eingebauten abstrakten Maschine) verstanden werden, bildet eine Programmiersprache. Glücklicherweise hat nicht jeder Druckertyp seine eigene Sprache, sondern viele verschiedene Druckertypen verstehen dieselbe Sprache, so daß Textverarbeitungsprogramme weitgehend unabhängig vom verwendeten Drucker arbeiten können.

Beispiele für Sprachen zur prozeduralen Seitenbeschreibung sind *dvi* [Knu86a, Part 31], die Ausgabesprache von TEX, sowie *PostScript* [Ado90] von Adobe.

Deskriptive Sprachen (Seitenaufbausprachen)

In einer deskriptiven Sprache wird beschrieben, wie eine Seite aus Basiselementen aufgebaut ist. Basiselemente sind Buchstaben und Linien verschiedener Art, die horizontal und vertikal zu größeren Strukturen kombiniert werden. Eine Seitenbeschreibung hat also eine Baumstruktur. Mit den Knoten des Baums sind Darstellungsattribute verbunden, z. B. die Ausmaße der beschriebenen Strukturen, interne Koordinatensysteme, eventuell auch Angaben über ihre Durchsichtigkeit, die bei sich überlagernden Teilstrukturen zum Tragen kommen.

Diese Art der Beschreibung ist im allgemeinen nicht universell. Der Seiteninhalt ist auf die vorgegebenen Basiselemente beschränkt. Wählt man einzelne Bildpunkte als Basiselemente, erreicht man Universalität auf Kosten riesiger Darstellungen.

Der Druckprozeß wird durch einen Durchlauf durch die hierarchische Struktur gesteuert, der in einer bestimmten Standardreihenfolge erfolgt. Dabei sammelt der Prozeß Darstellungs- und Plazierungsinformation, um die Inhaltsteile an den gewünschten Stellen und in der gewünschten Darstellung zu drucken. Von der

standardmäßigen Durchlaufreihenfolge kann abgewichen werden, wenn dies von den Darstellungsattributen verlangt wird. Dies hat Konsequenzen, wenn sich undurchsichtige Bereiche überlappen.

Die in Abschnitt 6.4.3 vorgestellte Internform von TEX aus ineinandergeschachtelten Kästen kann als eine solche deskriptive Seitenbeschreibung aufgefaßt werden. Sie wird allerdings nicht direkt ausgegeben, sondern noch vom TEX-System selbst in die prozedurale Sprache *dvi* übersetzt (siehe Abschnitt 9.4). Die Layoutbeschreibung von ODA [App90] ist ein anderes Beispiel einer deskriptiven Seitenbeschreibung.

dvi und *PostScript* – ein Überblick

In diesem Buch werden wir die prozeduralen Sprachen *dvi* [Knu86a, Part 31] und *PostScript* [Ado90] genauer betrachten. Die Sprache *dvi* enthält nur relativ einfache Befehle zum Ausgeben von Zeichen oder Linien sowie zum Ansprechen von Speicherplätzen. Die Sprache *PostScript* erlaubt auch komplexe graphische Operationen und besitzt höhere Kontrollkonstrukte, wie bedingte Anweisungen, Schleifen und Prozeduren, sowie Datenstrukturen wie Felder. Obwohl es daher wahrscheinlich für einen Druckerhersteller viel einfacher ist, *dvi* zu implementieren als *PostScript*, ist heutzutage *PostScript* die Seitenbeschreibungssprache, die von den meisten Druckertypen verstanden wird.

Die beiden Sprachen haben ein gemeinsames Grundprinzip, das hier vorgestellt wird, bevor die beiden Sprachen weiter unten eingehend besprochen werden.

Wie bereits erwähnt, bestehen beide Sprachen aus Befehlen an eine abstrakte Maschine, die von dem Prozessor im Drucker realisiert werden muß. In beiden Fällen besitzt diese abstrakte Maschine unter anderem einen rechteckigen *Seitenraum*, der später auf die vom Drucker erzeugten Seiten abgebildet wird. Positionen in diesem Seitenraum können durch Koordinaten (Zahlenpaare) in einem druckerunabhängigen Koordinatensystem bezeichnet werden. Die abstrakte Maschine verwaltet eine *aktuelle Position* im Seitenraum, auf die sich die meisten Befehle explizit oder implizit beziehen.

Der Hauptteil eines *dvi*- bzw. *PostScript*-Programms besteht aus Befehlsfolgen, die die einzelnen Seiten eines Dokuments erzeugen. Vor der Ausführung einer solchen Befehlsfolge ist der Seitenraum leer. Die Befehle in der Folge legen dann Text oder graphische Objekte in den Seitenraum. Die meisten Befehle tun dies automatisch so, daß der „Anfang“ des Objekts an der aktuellen Position liegt und daß sich die aktuelle Position während des Hineinlegens zum „Ende“ des Objekts verschiebt. Außerdem gibt es Befehle, die die aktuelle Position explizit verändern.

Die in den Seitenraum gelegten Objekte werden noch nicht wirklich ausgedruckt, sondern zunächst nur im Speicher des druckerinternen Prozessors gesammelt und können zumindest in *PostScript* auch wieder gelöscht werden. Erst wenn die geplante Seite vollständig im Seitenraum dargestellt ist, folgt der eigentliche Druckbefehl, der bewirkt, daß der Inhalt des Seitenraums auf ein Blatt Papier gedruckt

wird. Der Druckbefehl löscht während seiner Ausführung den Seitenraum, so daß er danach wieder leer ist.

Der Druckbefehl wird normalerweise den Inhalt des Seitenraums nicht in der Reihenfolge drucken, in der er hineingefüllt worden ist. Da viele Drucker ein Blatt nur in einer Richtung durchziehen ohne anzuhalten oder umzukehren, muß das zu druckende Material vor dem eigentlichen Druckvorgang gemäß der vertikalen Lage im Seitenraum sortiert werden. Dieses Sortieren kann schon während der Füllung des Seitenraums geschehen oder erst durch den Druckbefehl veranlaßt werden.

Da die meisten Drucker letztendlich doch nichts anderes als Rasterausgabegeräte sind, muß zu irgendeinem Zeitpunkt auch die Übersetzung von Buchstaben und Graphikobjekten in eine Pixeldarstellung erfolgen, die die gewünschte Form möglichst gut widergibt. Diese Übersetzung kann wiederum schon bei der Füllung des Seitenraums erfolgen oder erst durch den Druckbefehl ausgelöst werden. Dies hängt auch davon ab, wie die Druckerhersteller die Verwaltung des Seitenraums implementiert haben, ob als Bitmatrix oder als Liste der positionierten graphischen Objekte. Die Implementierung ist an sich beliebig, solange nur die von der abstrakten *dvi*- bzw. *PostScript*-Beschreibung geforderte Funktionalität gewährleistet ist.

9.3 Die Sprache *dvi*

Die Sprache *dvi* (device independent) ist die Ausgabesprache von TEX. Sie ist in Part 31 von „TEX: the Program“ [Knu86a] ausführlich dokumentiert. *dvi* ist die Maschinensprache einer abstrakten Maschine, nennen wir sie die *dvi-Maschine*. *dvi*-Programme können von implementierten *dvi*-Maschinen interpretiert werden, die ihre Ausgabe auf Drucker oder Bildschirm schicken. Es gibt auch Übersetzungsprogramme, die *dvi*-Programme nach *PostScript* übersetzen, z. B. *dvips* [Rok95].

Der Seitenraum der *dvi*-Maschine ist mit einem Koordinatensystem versehen, dessen Ursprung in der linken oberen Ecke liegt. Horizontale Koordinaten nehmen demnach von links nach rechts zu und vertikale von oben nach unten. Während also die Lage des Ursprungs und die Orientierung der Achsen von vornherein festgelegt sind, kann der verwendete Maßstab am Anfang des *dvi*-Programms frei gewählt werden. Er gilt dann allerdings unveränderbar für das gesamte Programm.

Die *dvi*-Maschine hat sieben Register (Speicherplätze). Die Register *h* und *v* enthalten die horizontalen und vertikalen Koordinaten der sogenannten *aktuellen Position*. Die Register *w* und *x* enthalten Zahlenwerte zum Modifizieren von *h* und die Register *y* und *z* solche zum Modifizieren von *v*. Das siebte Register *f* enthält eine Zahl, die den aktuellen Font beschreibt. Wenn ein Objekt in den Seitenraum gelegt wird, so erscheint es immer an der gerade gültigen aktuellen Position (h, v). Wenn es sich um einen Text handelt, wird er in dem aktuellen Font *f* ausgegeben.

Die *dvi*-Maschine hat außerdem einen Keller, dessen Einträge jeweils aus sechs Zahlen bestehen. Die Inhalte der sechs Register *h*, *v*, *w*, *x*, *y* und *z* können durch

spezielle Befehle in diesem Keller abgespeichert und wieder daraus hervorgeholt werden. Das Register *f* kann nicht gekellert werden.

Eine *dvi*-Datei enthält ein von TEX oder LATEX erzeugtes *dvi*-Programm, das eine Folge von Befehlen darstellt. Jeder Befehl besteht aus einem Operator, der als ein Byte kodiert wird, und aus einer Folge von Operandenbytes, deren Struktur und Länge vom Operator abhängt. Im folgenden werden die möglichen *dvi*-Befehle aufgelistet. Dabei wird zunächst der Operator in menschenlesbarer Form zusammen mit seinen Operanden angegeben. Die Länge der Operanden in Bytes wird durch eine kleine hochgestellte Zahl bezeichnet; c^4 bedeutet also, daß der Operand *c* heißt und 4 Byte lang ist. Hinter dieser abstrakten Beschreibung des Operators mit seinen Operanden folgt eine Zahl in Klammern. Diese ist der in der *dvi*-Datei zur Kodierung des Operators dienende *Operatorcode*, d. h. das Byte (in Dezimaldarstellung), das den Operator bezeichnet. Am Ende wird die Funktion des Operators erklärt.

set_char_0 (0) bis **set_char_127** (127)
: Der Befehl **set_char_***i* mit dem Operatorcode *i* (für $0 \leq i \leq 127$) druckt das Zeichen mit der Nummer *i* aus dem durch *f* beschriebenen Font so, daß sein Referenzpunkt an der aktuellen Position, d. h. an der Stelle mit den Koordinaten (h,v) liegt. Dabei erhöht sich *h* um die Breite des Zeichens (die auch negativ sein kann). Beachten Sie, daß diese Befehle keine Operanden haben; die Nummer des zu druckenden Zeichens ist in den Operator selbst hineinkodiert. Alle in einem gewöhnlichen Text vorkommenden Zeichen können auf diese Weise gedruckt werden.

set1 c^1 (128)
: Wie eben, aber die Nummer des zu druckenden Zeichens wird als ein Operandenbyte mitgegeben. Damit werden Zeichen mit Nummern aus dem Bereich von 128 bis 255 gedruckt; das sind gewisse Graphik- und Sonderzeichen.

set2 c^2 (129)
: Wie eben, aber die Nummer des zu druckenden Zeichens ist 2 Byte lang, d. h. aus dem Bereich von 256 bis 65535. Chinesische und japanische Schriftzeichen werden mit solchen Nummern dargestellt.

set3 c^3 (130) und **set4** c^4 (131)
: Dasselbe mit Zeichennummern, die 3 bzw. 4 Byte lang sind. Dafür gibt es noch keine Anwendungen.

set_rule a^4 b^4 (132)
: Druckt einen schwarzen Kasten mit Höhe *a* und Breite *b* (je 4 Byte) so, daß die linke untere Ecke an der Position (h,v) liegt. Dabei erhöht sich *h* um die Breite *b*. Dieser Befehl wird auch zur Erzeugung waagerechter und senkrechter Linien benutzt. Diese ergeben sich als Spezialfälle des schwarzen Kastens, wenn die Höhe *a* bzw. Breite *b* sehr klein sind.

put1 c^1 (133) bis **put_rule** a^4 b^4 (137)
: Diese Befehle arbeiten wie die entsprechenden *set*-Befehle, aber ohne das Register *h* zu verändern.

right1 b^1 (143) bis **right4** b^4 (146)
Der Operand *b* wird als Zahl mit Vorzeichen interpretiert. Beim Befehl **right1** b^1 stellt *b* also eine Zahl aus dem Bereich von -128 bis 127 dar. Die vier Befehle erhöhen *h* um *b*, was einer horizontalen Bewegung entspricht, die bei $b > 0$ nach rechts führt und bei $b < 0$ nach links.

w0 (147)
Erhöht *h* um den Inhalt von *w*.

w1 b^1 (148) bis **w4** b^4 (151)
Setzt *w* auf *b* und erhöht *h* um *b*.

x0 (152) bis **x4** b^4 (156)
Dasselbe mit Register *x* statt *w*.

down1 a^1 (157) bis **down4** a^4 (160)
Erhöht *v* um *a*.

y0 (161) bis **z4** a^4 (170)
Vertikale Analoga zu **w0** bis **x4** b^4, die sich auf *v* und *y* bzw. *z* beziehen.

fnt_num_0 (171) bis **fnt_num_63** (234)
Diese Befehle haben keine Operanden. Register *f*, d. h. der aktuelle Font, wird auf die im Operator kodierte Zahl aus dem Bereich von 0 bis 63 gesetzt.

fnt1 k^1 (235) bis **fnt4** k^4 (238)
Register *f* wird auf *k* gesetzt.

push (141)
Die Werte von *h*, *v*, *w*, *x*, *y* und *z* (nicht *f*!) werden oben auf den Keller geschoben, bleiben dabei aber erhalten.

pop (142)
Die aktuellen Werte dieser sechs Register werden durch die oben auf dem Keller liegenden ersetzt. Dabei wird der Keller kleiner.

nop (138)
Tut nichts.

xxx1 k^1 x^k (239) bis **xxx4** k^4 x^k (242)
Der erste Operand *k* (1 bis 4 Byte lang) gibt an, wieviele weitere Operandenbytes es gibt. Vom *dvi*-Standpunkt aus ist dieses Kommando undefiniert. Der lange Operand *x* kann aber z. B. *PostScript*-Kommandos enthalten, die beim Übersetzen von *dvi* nach *PostScript* textuell in die Resultatdatei übernommen werden.

Die bis jetzt aufgezählten Befehle werden in dem Teil der *dvi*-Datei benutzt, der die einzelnen Seiten der zu erzeugenden Ausgabe beschreibt. Daneben gibt es aber noch einige weitere Befehle, die vor oder nach der eigentlichen Seitenbeschreibung benutzt werden. Eine *dvi*-Datei besitzt nämlich die folgende übergeordnete Struktur: Sie beginnt mit einem **pre**-Befehl (Präambel). Dann folgen die einzelnen Seiten, eingeleitet durch **bop** (begin of page) und beendet durch **eop** (end of page). Am Ende kommt die Postambel, die mit **post** beginnt und mit **post_post** endet.

Die Präambel besteht aus einem einzigen Kommando, das folgendes Format hat (die hochgestellten Zahlen bezeichnen wieder die Länge der Operanden in Bytes):

pre i^1 ***num***^{4} ***den***^{4} ***mag***^{4} k^1 x^k (247)

Der Operand *i* gibt die Nummer der verwendeten *dvi*-Version an, um Probleme mit nicht zueinander passenden Versionen in *dvi*-Erzeuger und *dvi*-Konsument zu vermeiden. Die nächsten drei Operanden bezeichnen die Einheit des Koordinatensystems; diese ist

$$\frac{mag \cdot num}{1000 \cdot den} \cdot 10^{-7}\,\mathrm{m}.$$

Der *k*-Operand gibt die Länge des *x*-Operanden an. Dieser ist ein Kommentar, wird also nicht interpretiert.

Der Seitenanfangsbefehl hat das Format

bop $c^{4\,10}$ p^4 (139).

Er hat zunächst 10 Operanden, die jeweils 4 Byte umfassen. Diese können die Seitennummer und andere Zusatzinformationen beinhalten. Diese Informationen werden nicht zur Ausgabe gebraucht, sondern für besondere Dienstleistungen wie für das Auffinden gewisser Seiten zum selektiven Drucken oder Navigieren bei der Bildschirmanzeige. Beachten Sie, daß die erste Seite nicht unbedingt die Nummer 1 hat. Der letzte Operand *p* zeigt auf den Anfang der *vorhergehenden* Seite, d. h. er gibt die Nummer des Bytes an, das dem **bop** der vorhergehenden Seite entspricht, gezählt vom Anfang der *dvi*-Datei. Die erste Seite hat $p = -1$. Bei der Ausführung des **bop**-Befehls werden alle Register auf 0 gesetzt und der Keller wird geleert.

Der operandenlose Befehl **eop** (140) zeigt das Ende der Seite an. Er bewirkt, daß das im Seitenraum gesammelte Material wirklich ausgedruckt wird. Dabei wird der Seitenraum gelöscht.

Die Postambel besteht aus einem **post**-Befehl, den Definitionen der im Dokument verwendeten Fonts und einem **post_post**-Befehl. Der Befehl **post**, der nach der letzten Seite kommt, hat die folgende Struktur:

post p^4 ***num***^{4} ***den***^{4} ***mag***^{4} l^4 u^4 s^2 t^2 (248)

Der Operand *p* zeigt auf den Anfang der letzten Seite. Die Operanden *num*, *den* und *mag* haben dieselbe Bedeutung wie in der Präambel und sollten mit denen dort übereinstimmen. Die letzten 4 Operanden geben eine kleine Statistik zur *dvi*-Datei an, die von den meisten *dvi*-Interpretern ignoriert wird. Dabei bezeichnen *l* und *u* das Maximum der vertikalen bzw. horizontalen Seitenausdehnungen, *s* die größte Kellertiefe und *t* die Anzahl der Seiten.

Der Befehl **post_post** hat das Format

post_post q^4 i^1 x^{4-7} (249).

Der Operand *q* zeigt auf den **post**-Befehl am Anfang der Postambel. Der Operand *i* gibt die *dvi*-Versionsnummer an wie in der Präambel. Ganz am Schluß der *dvi*-Datei folgen 4 bis 7 Wiederholungen des Bytes 223, die keine Bedeutung tragen, sondern nur als Endemarkierung dienen. Die schwankende Anzahl wird so gewählt, daß die Länge der gesamten *dvi*-Datei in Bytes ein Vielfaches von 4 ist.

Die beschriebene Grobstruktur der *dvi*-Datei soll es ermöglichen, die Datei sowohl von vorne nach hinten als auch von hinten nach vorne zu interpretieren. Bei letzterem wird zuerst das **post_post** vom Dateiende her gesucht. Dieses gibt an, wo sich das **post** befindet. Von dort aus kann einerseits die Fontinformation in der Postambel gelesen werden, andererseits können die einzelnen Seiten unter Zuhilfenahme der Rückwärtsverweise in umgekehrter Reihenfolge betrachtet werden.

Es bleiben noch die Fontdefinitionen zu beschreiben. Jeder Font, der in der Datei benutzt wird, muß zweimal definiert werden: einmal irgendwo vor seiner ersten Benutzung und einmal in der Postambel. Diese Redundanz dient dazu, die beiden Leserichtungen zu unterstützen. Eine Fontdefinition hat die folgende Struktur:

fnt_def_*i* $k^i\ c^4\ s^4\ d^4\ a^1\ l^1\ n^{a+l}$ (243–246)

Dabei kann *i* Werte von 1 bis 4 annehmen. Es gibt die Länge des Operanden *k* an, der die Nummer bezeichnet, unter der der neu definierte Font benutzt werden kann. Die Operanden *s* und *d* geben Skalierungsinformation; jede Größeninformation aus dem Font wird mit dem Faktor $\frac{mag \cdot s}{1000 \cdot d}$ multipliziert. Der Operand *n* ist der Name einer Datei, die den Font enthält. Die Operanden *a* und *l* geben zusammen die Länge dieses Dateinamens an; *a* die Länge des Namens des Dateiverzeichnisses und *l* die Länge des eigentlichen Dateinamens. Um nicht vorgesehene Mißgeschicke zu vermeiden, wird aus der Fontdatei eine Kontrollzahl berechnet, die auf Übereinstimmung mit dem Operanden *c* getestet wird. Damit kann festgestellt werden, ob die Fontdatei, die TEX beim Berechnen der Wortgrößen benutzt hat, auch tatsächlich mit der beim Ausdruck verwendeten übereinstimmt.

Schlußbemerkungen zu *dvi*

Die mangelnde Ausdruckskraft von *dvi* wird erst durch den Vergleich mit *PostScript* richtig sichtbar. Trotzdem wollen wir hier einige Hinweise geben.

Es gibt in *dvi* nahezu keine Arithmetik. Die einzig mögliche arithmetische Operation ist Addition: der Befehl **right** *b* entspricht $h := h + b$, und **w0** entspricht $h := h + w$. In *dvi* gibt es keine Variablen und keine höheren Kontrollstrukturen wie bedingte Anweisungen oder Schleifen und keine Prozeduren. *dvi* ist also keine universelle Programmiersprache. D. E. Knuth, der *dvi* zusammen mit TEX entwickelt hat, ging davon aus, daß alle notwendige Intelligenz in dem *dvi* erzeugenden Programm steckt. Dort werden alle zur Formatierung nötigen Berechnungen durchgeführt und Kontrollflußentscheidungen getroffen. Die *dvi*-Datei ist dann weniger als Programm anzusehen, sondern eher eine Art Datenformat zum Übermitteln von Informationen vom Dokumentsystem zum Drucker.

Außer den oben beschriebenen Mängeln gibt es noch weitere Defizite, die von dem Dokumentsystem nur schwer ausgeglichen werden können. Auffällig ist, daß die einzigen in *dvi* beschreibbaren graphischen Objekte schwarze Rechtecke sind, deren Kanten parallel zu den Rändern der Seite verlaufen. Diese Rechtecke können zu waagerechten oder senkrechten Linien oder einem schwarzen Punkt entarten. Natürlich ist es möglich, beliebige Graphikobjekte wie z. B. Kreise aus solchen Punkten aufzubauen, aber das ist dann noch aufwendiger als eine Bitmatrix zu benutzen, da jeder Punkt nicht durch ein Bit, sondern durch 9 Bytes beschrieben wird.

Um wenigstens gewisse Graphikobjekte einigermaßen elegant zur Verfügung zu stellen, wird der folgende Trick benutzt: es gibt nicht nur Fonts mit gewöhnlichen Zeichen wie Buchstaben und mathematischen Symbolen, sondern auch solche mit ausgewählten Graphikobjekten. Dazu gehören z. B. kurze Linienstücke und Pfeilspitzen in verschiedenen Richtungen, aus denen sich schräge Linien und Pfeile mit gewissen vorgegebenen Orientierungen zusammensetzen lassen. In LaTeX werden Befehle angeboten, die es erlauben, die auf diese Weise erzeugbaren Objekte zu spezifizieren.

9.4 Die Übersetzung von Kastentermen nach *dvi*

In diesem Abschnitt werden wir zeigen, wie die Kastenterme der Internform von TeX (siehe Abschnitt 6.4.3) in eine vereinfachte Version der Seitenbeschreibungssprache *dvi* (siehe Abschnitt 9.3) übersetzt werden können.

Die Zielsprache

Wir nehmen dabei an, daß der Drucker zwei Register *h* und *v* besitzt, die zusammen die aktuelle Position auf der Seite beschreiben. Eine Erhöhung des *h*-Registers entspricht einer Bewegung nach rechts und die Erhöhung des *v*-Registers einer Bewegung nach unten. Der Drucker soll die folgenden vier Instruktionen verstehen:

- **Right** (x) erhöht das *h*-Register um *x*. Dies entspricht einer Bewegung um *x* nach rechts. Natürlich kann *x* auch negativ sein, so daß die effektive Bewegung auch nach links führen kann. (Die entsprechenden *dvi*-Befehle heißen **right1** bis **right4**, je nach der Länge des Operanden *x* in Bytes.)
- **Down** (y) erhöht das *v*-Register um *y*. Dies entspricht einer Bewegung um *y* nach unten. So wie eben kann *y* auch negativ sein. Die effektive Bewegung führt dann nach oben. (Die entsprechenden *dvi*-Befehle heißen **down1** bis **down4**, je nach der Länge des Operanden *y* in Bytes.)
- **Set** (ch) druckt das Zeichen *ch* an der aktuellen Position und verschiebt diese dabei um die Breite des Zeichens nach rechts. (Die entsprechenden *dvi*-Befehle heißen **set_char_0** bis **set_char_127** sowie **set1** bis **set4**, je nach dem Format des Operanden.)

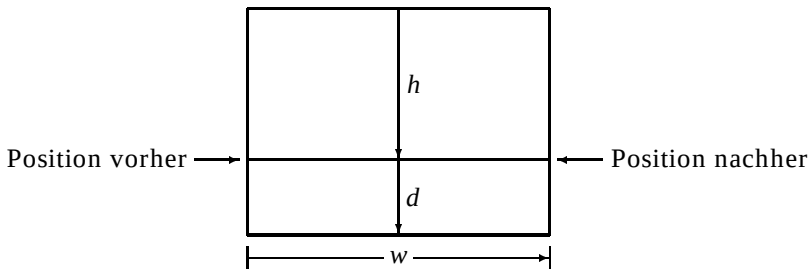

Abbildung 9.1: Invariante für einen Kasten mit Höhe h, Tiefe d und Breite w

- **SetRule** (x,y) druckt ein schwarzes Rechteck der Höhe x und Breite y mit seitenrandparallelen Kanten, dessen linke untere Ecke an der aktuellen Position liegt. Während des Druckens wird die aktuelle Position zur rechten unteren Ecke verschoben. (Der entsprechende *dvi*-Befehl heißt **set_rule**.)

Die Quellsprache

Die Quellsprache für unsere Übersetzung besteht aus den Kastentermen der TEX-Interndarstellung, wie sie in Abschnitt 6.4.3 beschrieben wurde. Wir werden die Beschreibung hier nicht noch einmal wiederholen, sondern nur kurz die Grundzüge in Erinnerung rufen.

Jeder Kasten b wird als $b = \mathbf{Box}(h,d,w,c)$ mit Höhe h, Tiefe d, Breite w und Inhalt c beschrieben. Jeder Kasten hat einen Referenzpunkt, der auf dem linken Rand liegt. Die Höhe gibt den Abstand vom Oberrand zum Referenzpunkt und die Tiefe den Abstand vom Referenzpunkt zum Unterrand an.

Bezüglich des Inhalts kann man verschiedene Arten von Kästen unterscheiden, die wir im weiteren Verlauf jeweils kurz beschreiben werden, bevor wir die Übersetzungsfunktion für sie definieren.

Die Invariante der Übersetzung

Unser Ziel ist es, eine Funktion *code* zu definieren, die Kästen in Folgen von Instruktionen übersetzt. Für die Übersetzung eines Kastens b nehmen wir als Invariante an, daß vor der Ausführung von $code(b)$ die aktuelle Position der Referenzpunkt von b ist, während nach der Ausführung die aktuelle Position am rechten Rand des Kastens in derselben Höhe wie der Referenzpunkt liegen soll (siehe Abbildung 9.1). Einerseits können wir also bei der Definition von $code(b)$ annehmen, daß zu Anfang die aktuelle Position und der Referenzpunkt von b übereinstimmen, und andererseits müssen wir sicherstellen, daß nach Ausführen von $code(b)$ die aktuelle Position sich in der geforderten Endlage befindet. Alle expliziten und impliziten Bewegungen, die von $code(b)$ generiert werden, müssen sich also zu einer horizontalen Bewegung um die Breite w von b aufaddieren.

Nun wollen wir die Funktion *code* für die einzelnen Arten von Kästen definieren.

Zwischenräume

Zwischenräume $\mathbf{Box}(h, d, w, \mathbf{White})$ haben keinen sichtbaren Inhalt, sondern dienen nur als Abstandshalter zwischen anderen Kästen. Daher braucht nichts gedruckt zu werden, sondern es muß nur die aktuelle Position gemäß der Invariante verschoben werden. Also besteht der Code für einen Zwischenraum einfach aus einer Verschiebung um w nach rechts.

$$code(\mathbf{Box}(h, d, w, \mathbf{White})) \;=\; \mathbf{Right}(w)$$

Angesichts dieser Definition mag man sich fragen, was vertikale Zwischenräume, die ja eine Breite von $w = 0$ haben, überhaupt bewirken. Wir werden später bei der Behandlung vertikaler Kombinationen von Kästen sehen, daß zwischen die Übersetzung von zwei Teilkästen Bewegungsinstruktionen eingefügt werden, um von einem Teilkasten zum nächsten zu gelangen. Aufgrund dieser Zwischenbewegungen haben vertikale Zwischenräume dann doch eine Wirkung.

Zeichenkästen

Ein Zeichenkasten $\mathbf{Box}(h, d, w, \mathbf{Chr}(ch))$ hat als Inhalt genau ein Zeichen *ch*. Dieses Zeichen wird durch den Befehl **Set** gedruckt. Dabei rückt die aktuelle Position um die Breite des Zeichens nach rechts, so daß die Invariante ohne besondere Vorkehrungen erfüllt ist.

$$code(\mathbf{Box}(h, d, w, \mathbf{Chr}(ch))) \;=\; \mathbf{Set}(ch)$$

Rechteckkästen

Ein Rechteckkasten $\mathbf{Box}(h, d, w, \mathbf{Black})$ beschreibt die Erzeugung eines schwarzen Rechtecks, das den ganzen Kasten ausfüllt, also Höhe $h + d$ und Breite w hat. Der Befehl **SetRule** erzeugt ein solches Rechteck, allerdings mit der linken unteren Ecke an der aktuellen Position. Die aktuelle Position muß also zunächst vom Referenzpunkt dorthin verschoben werden. Durch **SetRule** bewegt sie sich zur rechten unteren Ecke und muß also am Schluß wieder auf die Höhe des Referenzpunktes geschoben werden, damit die Invariante erfüllt ist.

$$code(\mathbf{Box}(h, d, w, \mathbf{Black})) \;=\; \mathbf{Down}(d);\; \mathbf{SetRule}(h + d, w);\; \mathbf{Down}(-d)$$

Horizontale Kombination

Wenn ein Kasten $b = \mathbf{Box}(h, d, w, \mathbf{HList}[b_1, \ldots, b_n])$ horizontal aus den *unverschobenen* Kästen b_1 bis b_n zusammengesetzt ist, dann liegen die Teilkästen ohne Zwischenraum so nebeneinander, daß der Referenzpunkt von b_1 mit dem von b übereinstimmt, der von b_2 sich in derselben Höhe am rechten Rand von b_1 befindet, usw.

Die Invariante der Übersetzung ist genau an diese Situation angepaßt, so daß das Ergebnis einfach $code(b_1); \ldots; code(b_n)$ ist.

Allgemein müssen wir aber mit vertikal verschobenen Kästen rechnen. Daher führen wir eine Hilfsfunktion $code_H$ für verschobene Kästen in einer horizontalen Liste ein und definieren

$$code(\mathbf{Box}(h,d,w,\mathbf{HList}[b'_1,\ldots,b'_n])) \;=\; code_H(b'_1); \ldots; code_H(b'_n),$$

wobei $b'_1, \ldots, b'_n$ verschobene Kästen sind, und

$$code_H(s \triangleright b) \;=\; \mathbf{Down}(s);\; code(b);\; \mathbf{Down}(-s).$$

Vertikale Kombination

Bei der vertikalen Kombination sind die einzelnen Teilkästen von oben nach unten übereinander angeordnet, so daß im unverschobenen Fall die Referenzpunkte genau übereinanderliegen. Für diese Anordnung ist unsere Invariante nicht besonders günstig. Daher führen wir eine Hilfsfunktion $code'$ ein, die eine andere Invariante erfüllt: Vor der Ausführung von $code'(b)$ befindet sich die aktuelle Position in der linken oberen Ecke von b und nachher in der linken unteren. Der von $code'(b)$ mit $b = \mathbf{Box}(h,d,w,c)$ erzeugte Code muß also zuerst die aktuelle Position von der linken oberen Ecke zum Referenzpunkt verschieben ($\mathbf{Down}(h)$). Dann folgt $code(b)$, das die Position zum rechten Rand verschiebt. Das wird durch $\mathbf{Right}(-w)$ rückgängig gemacht; danach ist die Position wieder am Referenzpunkt. Der Befehl $\mathbf{Down}(d)$ schiebt sie schließlich in die linke untere Ecke. Damit ergibt sich

$$code'(b) \;=\; \mathbf{Down}(h);\; code(b);\; \mathbf{Right}(-w);\; \mathbf{Down}(d)$$

für $b = \mathbf{Box}(h,d,w,c)$.

Für die horizontal verschobenen Kästen in einer vertikalen Liste brauchen wir eine weitere Hilfsfunktion $code'_V$:

$$code'_V(s \triangleright b) \;=\; \mathbf{Right}(s);\; code'(b);\; \mathbf{Right}(-s).$$

Für die vertikale Kombination selbst wird erst die aktuelle Position vom Referenzpunkt zur linken oberen Ecke verschoben ($\mathbf{Down}(-h)$), dann werden die Teilkästen von oben nach unten mit $code'$ abgearbeitet und am Schluß muß die Position von der linken unteren Ecke zum rechten Rand in Höhe des Referenzpunkts verschoben werden. Daher definieren wir:

$$\begin{aligned} &code(\mathbf{Box}(h,d,w,\mathbf{VList}[b'_1,\ldots,b'_n])) \;= \\ &\quad \mathbf{Down}(-h);\; code'_V(b'_1); \ldots; code'_V(b'_n);\; \mathbf{Down}(-d);\; \mathbf{Right}(w). \end{aligned}$$

Dabei sind $b'_1, \ldots, b'_n$ die eventuell horizontal verschobenen Teilkästen.

Optimierungen des erzeugten Codes

Der durch die oben definierte Codefunktion erzeugte Code kann noch Ineffizienzen in Gestalt unnötiger oder iterierter Bewegungsbefehle aufweisen. Im einzelnen dürfen die folgenden Verbesserungen vorgenommen werden:

1. Instruktionen **Right**(0) und **Down**(0) werden weggelassen.
2. Zwei aufeinanderfolgende Instruktionen **Right**(a) und **Right**(b) werden zusammengefaßt zu **Right**$(a+b)$. Zwei aufeinanderfolgende **Down**-Instruktionen werden genauso behandelt.
3. Um die eben beschriebenen Optimierungen zu ermöglichen, dürfen **Right**-Instruktionen mit direkt folgenden **Down**-Instruktionen vertauscht werden.

Anstatt wie hier beschrieben schlechten Code zu generieren, der nachträglich optimiert wird, ist es natürlich vorteilhaft, sofort optimierten Code zu erzeugen. Dazu kann man den Übersetzer so abwandeln, daß er die zu generierenden Bewegungen zuerst intern akkumuliert und erst dann sofort optimalen Code dafür erzeugt, wenn er eine **Set**- oder **SetRule**-Instruktion generieren muß.

9.5 Die Sprache *PostScript*

Da *PostScript* eine sehr mächtige und komplizierte Sprache ist, kann sie hier nicht vollständig beschrieben werden. Dazu gibt es ausführliche Handbücher [Ado90] und Benutzeranleitungen [Ado85].

Im folgenden werden zunächst einige einfache Konzepte von *PostScript* recht ausführlich eingeführt. Dann werden diese Konzepte in einem Beispiel benutzt, um das linksbündige Ausgeben von Zeilen zu programmieren. Danach geben wir einen Überblick über den Rest von *PostScript*, der keinerlei Anspruch auf Vollständigkeit erhebt, wobei wir zuerst die Graphikfähigkeiten von *PostScript* beleuchten und dann *PostScript* als Programmiersprache untersuchen. Zum Abschluß präsentieren wir ein längeres *PostScript*-Programm (Abbildung 9.12) und das von ihm erzeugte Bild (Abbildung 9.13).

Grundprinzip und Seitenfüllung

Wie bereits in Abschnitt 9.2 erläutert, besitzt die *PostScript*-Maschine einen Seitenraum mit einer aktuellen Position. Punkte im Seitenraum wie z. B. die aktuelle Position werden in einem geräteunabhängigen Koordinatensystem beschrieben, das während der Ausführung eines *PostScript*-Programms verändert werden kann.

Gewisse Befehle erzeugen *graphische Objekte* und legen sie so in den Seitenraum, daß ihr „Anfang“ an der aktuellen Position liegt. Dabei verschiebt sich die aktuelle Position zum „Ende“ des Objekts. In *PostScript* überdecken später erzeugte graphische Objekte die bereits vorhandenen.

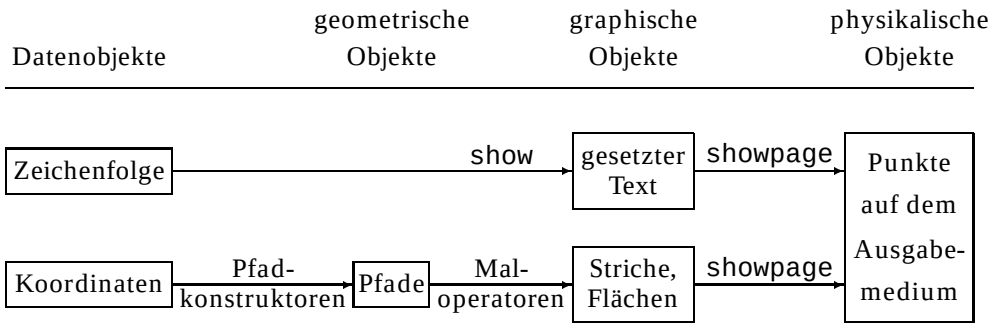

Abbildung 9.2: Grundprinzip von PostScript

Texte liegen zunächst als Zeichenfolgen vor, die als Datenobjekte vom *PostScript*-Programm manipuliert werden können. Der Operator `show` erzeugt aus einer Zeichenfolge unter Berücksichtigung des *aktuellen Fonts* und des *aktuellen Koordinatensystems* einen gesetzten Text als graphisches Objekt im Seitenraum.

Im Gegensatz zu Text entstehen Striche und gefärbte Flächen zweistufig (siehe Abbildung 9.2). Zunächst werden aus *PostScript*-Datenobjekten, die als Koordinaten dienen, *geometrische Objekte* erzeugt, die bereits vom aktuellen Koordinatensystem abhängig sind, aber noch nicht als graphische Objekte im Seitenraum liegen. Diese geometrischen Objekte sind idealisierte Strecken und Kurven ohne Breite, die in *PostScript Pfade* genannt werden. Erst durch sogenannte *Maloperatoren* entstehen aus den geometrischen Objekten graphische Objekte im Seitenraum. Der Maloperator `stroke` z. B. erzeugt abhängig von der *aktuellen Strichstärke* und anderen Parametern aus Pfaden Striche, die eine von 0 verschiedene Breite besitzen.

Wenn eine Seite im Seitenraum fertig zusammengestellt ist, kann die Seite durch den Befehl `showpage` auf Papier oder dem Bildschirm dargestellt werden. Dazu werden die graphischen Objekte im Seitenraum in physikalische Objekte im Ausgabemedium umgewandelt. Die noch geräteunabhängigen graphischen Objekte werden dabei so gut wie irgend möglich durch das Ausgabegerät dargestellt. Nach dem Drucken ist der Seitenraum wieder leer.

Der gesamte Weg von Datenobjekten im *PostScript*-Programm bis hin zu physikalischen Objekten im Ausgabemedium wird in Abbildung 9.2 veranschaulicht.

Der Operandenkeller

Zusätzlich zu dem Seitenraum besitzt die *PostScript*-Maschine eine Reihe von Kellern und eine Halde. Die Halde, in *PostScript* „global virtual memory“ genannt, dient zur Speicherung zusammengesetzter Datenobjekte. Der wichtigste Keller ist der *Operandenkeller*. Im Programm stehende Konstanten (Zahlen, Zeichen, Wahrheitswerte, Zeichenfolgen, Felder und Prozeduren) werden bei ihrem Abarbeiten

vorher	Operator	nachher	Bedeutung
	`5`	5	Konstanten werden gekellert
a	`pop`		löscht oberstes Kellerelement
a	`dup`	$a\ a$	verdoppelt oberstes Kellerelement
$a_n \ldots a_1\ n$	`copy`	$a_n \ldots a_1\ a_n \ldots a_1$	verdoppelt die n obersten Elemente
$a_n \ldots a_0\ n$	`index`	$a_n \ldots a_0\ a_n$	kopiert ein Element
$a\ b$	`exch`	$b\ a$	vertauscht die obersten zwei
$a_n \ldots a_1\ n\ j$	`roll`	$a_{j'} \ldots a_1\ a_n \ldots a_{j'+1}$	wobei $j' = j \bmod n$

Tabelle 9.1: PostScript-Operatoren für den Operandenkeller

jeweils auf diesem gekellert. Genau genommen liegen aber nur skalare Objekte wie z. B. Zahlen wirklich auf dem Operandenkeller. Bei zusammengesetzten Objekten liegt der eigentliche Wert in der Halde und nur ein Verweis darauf im Keller.

In Tabelle 9.1 sind einige Befehle zur expliziten Manipulation des Operandenkellers aufgelistet. Bei jedem Operator wird angegeben, wie die Kellerspitze vorher und nachher aussieht. Die aufgelisteten Operatoren arbeiten auf beliebigen Datenobjekten.

Abarbeitung von Operatoren

PostScript-Operatoren nehmen die ihnen zustehende Anzahl von Operanden aus dem Keller, operieren darauf und schreiben eventuelle Resultate auf den Keller. Das Lesen der Operanden vom Keller erfolgt bei fast allen Operatoren destruktiv, d. h. die Operanden werden vom Keller entfernt, bevor die Ergebnisse gekellert werden. Der Befehl `sub` z. B. erwartet zwei Zahlen an der Spitze des Kellers, entfernt sie von dort, berechnet ihre Differenz und kellert sie. Wenn also vor Ausführung von `sub` $9,7,4$ an der Spitze des Kellers stand, dann steht nachher $9,3$ dort und der Keller ist um eins kürzer geworden.

Da bei Ausführung des Operators die Operanden bereits im Keller stehen müssen, steht der Operator im *PostScript*-Programm hinter den Programmstücken, die die Operanden erzeugen und auf den Keller legen. Deshalb spricht man von *PostScript* als einer *Postfix*-Sprache.

Tabelle 9.2 enthält einige der vielen Instruktionen, die Berechnungen auf Zahlen und Wahrheitswerten durchführen.

Textausgabe

Bevor Text ausgegeben wird, sollte zuerst ein *aktueller Font* gesetzt werden. Dieser Font wird dann während der folgenden Textausgabe solange benutzt, bis ein

vorher	Operator	nachher	Kommentar
$a\ b$	`add`	$a+b$	Summe
$a\ b$	`sub`	$a-b$	Differenz
$a\ b$	`mul`	$a \cdot b$	Produkt
$a\ b$	`div`	a/b	Quotient
a	`abs`	$\|a\|$	Absolutbetrag
a	`ln`	$\ln a$	natürlicher Logarithmus
$a\ b$	`or`	$a \vee b$	Disjunktion von Wahrheitswerten
$a\ b$	`and`	$a \wedge b$	Konjunktion von Wahrheitswerten
$a\ b$	`eq`	$a = b$	liefert einen Wahrheitswert

Tabelle 9.2: Einige arithmetische und logische Operatoren

vorher	Operator	nachher	Bedeutung
Name	`findfont`	*Font*	sucht Font mit gegebenem Namen aus dem Fontverzeichnis heraus
Font Faktor	`scalefont`	*Font′*	skaliert Font mit Faktor
Font	`setfont`		erklärt Font zum aktuellen Font
	`currentfont`	*Font*	legt aktuellen Font auf Keller
Zeichenfolge	`show`		druckt Text im aktuellen Font

Tabelle 9.3: Fontwahl und Textausgabe in PostScript

anderer Font gesetzt wird. Die Befehle zur Auswahl eines Fonts und zum Setzen des aktuellen Fonts sind in Tabelle 9.3 aufgelistet. Ein typisches *PostScript*-Programmstück zum Setzen des aktuellen Fonts sieht also so aus:

```
/Times-Roman  findfont    15 scalefont    setfont
```

Dabei ist `/Times-Roman` der Name eines Fonts.

Um ein Stück Text in den Seitenraum zu legen, kann der Operator `show` benutzt werden. Er erwartet, daß das oberste Kellerelement eine Zeichenfolge ist, und setzt diese Zeichenfolge unter Benutzung des aktuellen Fonts als graphisches Objekt in den Seitenraum. Die Zeichenfolge wird so positioniert, daß der Referenzpunkt des ersten Zeichens an der aktuellen Position liegt. Während der Abarbeitung von `show` wird die aktuelle Position um die Länge der gesetzten Zeichenfolge nach rechts verschoben. Das Aussehen des gesetzten Textes und insbesondere seine Ori-

vorher	Operator	nachher	Bedeutung
x' y'	`moveto`		definiert aktuelle Position neu: $(x,y) := (x',y')$
a b	`rmoveto`		verschiebt Position: $(x,y) := (x+a,y+b)$
	`currentpoint`	x y	lädt aktuelle Position auf Keller

Tabelle 9.4: Aktuelle Position (x,y) in PostScript

entierung („horizontal nach rechts“) hängen dabei vom aktuellen Koordinatensystem ab.

In *PostScript* werden konstante Zeichenfolgen als Text in Klammern angegeben. Das Programmstück `(Hallo!)show` setzt also das Wort „Hallo!“ in den Seitenraum.

Verändern der aktuellen Position

Einige Operatoren, wie der im letzten Abschnitt eingeführte `show`-Operator, benutzen die aktuelle Position als implizites Argument und verändern sie als Seiteneffekt. Es gibt auch Operatoren, die die aktuelle Position explizit beeinflussen (siehe Tabelle 9.4). Wie man sieht, ist es nicht möglich, die horizontale und die vertikale Koordinate der aktuellen Position unabhängig voneinander zu verändern. Das führt manchmal zu Umständlichkeiten, wie wir bald an einem Beispiel sehen werden.

Variablen und Prozeduren

Um Daten dauerhafter und leichter zugreifbar als auf dem Operandenkeller zu speichern, gibt es in *PostScript* die Möglichkeit, Daten an Namen zu binden. Der Operator `def` erwartet auf dem Operandenkeller einen Namen und ein Datenobjekt. Er entfernt diese beiden Operanden vom Keller und führt die Bindung des Namens an das Datenobjekt durch. Von nun an kann auf das Datenobjekt durch Nennen des Namens zugegriffen werden.

Das Programmstück `/x 7 def` z. B. bindet den Namen `x` an die Zahl 7. Wenn danach der Name `x` im Programm angetroffen wird, dann wird die an `x` gebundene Zahl 7 auf den Keller geschoben.

Beachten Sie die beiden verschiedenen Formen, in der der Name `x` hier benutzt wird: Vorkommen von `x` hinter einem Schrägstrich, also als `/x`, werden nicht interpretiert; der Name `x` selbst wird auf den Keller geschoben. „Nackte“ Vorkommen von `x` (ohne Schrägstrich) werden dagegen interpretiert; wenn `x` an ein Datenobjekt gebunden ist, dann wird dieses Datenobjekt auf den Keller geschoben.

Wert von x	Kellerspitze	nächste Instruktion
7		`/x`
7	x	`x`
7	x 7	`inc` $\rightarrow$ `1`
7	x 7 1	`add`
7	x 8	`def`
8		

Tabelle 9.5: Abarbeitung einer PostScript-Instruktionsfolge

Außer an Datenobjekte können Namen auch an Prozedurobjekte, das sind *PostScript*-Programmstücke, gebunden werden. Wenn so ein Name im Programm ohne Schrägstrich vorkommt, dann wird das an ihn gebundene Programmstück ausgeführt. In *PostScript*-Programmen werden Prozedurobjekte als in geschweifte Klammern eingeschlossene Programmstücke geschrieben. Sie werden nicht sofort ausgeführt, sondern auf den Operandenkeller gelegt.

Eine Prozedur, die eine Zahl an der Spitze des Kellers um 1 erhöht, kann also wie folgt definiert werden:

```
/inc  { 1 add }  def
```

Ein Programmstück, das den an `x` gebundenen Wert um 1 erhöht, kann dann so aussehen:

```
/x  x inc  def
```

(zu lesen als $x := inc(x)$.) – In Tabelle 9.5 wird veranschaulicht, wie dieses Programmstück ausgeführt wird.

Prozeduren können in *PostScript* auch rekursiv definiert werden. Es ist möglich, ein Prozedurobjekt in ein Datenobjekt zu verwandeln, das seiner textuellen *PostScript*-Programmdarstellung entspricht. Dieses Datenobjekt kann modifiziert und anschließend wieder in ein Prozedurobjekt zurückverwandelt werden. Auf diese Weise ist es möglich, in *PostScript* selbstmodifizierende Programme zu schreiben.

Beispiel: Ausgabe von Zeilen

Um ein Beispiel für ein sinnvolles kleines *PostScript*-Programm zu erhalten, betrachten wir die folgende Aufgabe: Eine Reihe von Zeilen soll ausgegeben werden. Der Font ist bereits gesetzt. Alle Zeilen sollen an derselben horizontalen Position beginnen, die durch die Variable `Anfang` gegeben ist. Der vertikale Abstand zweier aufeinanderfolgender Zeilen ist durch die Variable `Abstand` gegeben. Die vertikale Position der ersten Zeile liegt an der Spitze des Operandenkellers. Nun sollen

zwei Prozeduren `Init` und `neueZeile` definiert werden, so daß die einzelnen Zeilen durch das folgende Programmschema ausgegeben werden können:

```
Init
(Inhalt der ersten Zeile)      show   neueZeile
(Inhalt der zweiten Zeile)     show   neueZeile  ...
```

Nach „`show`“ befindet sich die aktuelle Position am Ende des Textes der gerade ausgegebenen Zeile. Die Prozedur `neueZeile` muß sie zum Anfang der nächsten Zeile verschieben. Sie muß also die aktuelle Position von (x,y) an die Stelle $(\texttt{Anfang}, y - \texttt{Abstand})$ verschieben. Unglücklicherweise ist das eine horizontale absolute Bewegung (`moveto`) kombiniert mit einer vertikalen relativen Bewegung (`rmoveto`). Da es in *PostScript* keine Operatoren gibt, die diese Bewegungen unabhängig voneinander durchführen könnten, muß während der Durchführung von `neueZeile` die momentane vertikale Position ermittelt werden. Dazu gibt es zwei Möglichkeiten.

Eine Möglichkeit ist, sich die vertikale Position in einer Variablen `vPos` zu merken. Die vertikale Position der ersten Zeile liegt am Anfang an der Spitze des Kellers. Die Prozedur `Init` muß also diesen Wert an `vPos` binden und dann die aktuelle Position an den gewünschten Anfang der ersten Zeile setzen.

```
/Init {  /vPos exch def    Anfang vPos moveto  } def
```

Durch die Anweisung `/vPos` wird der Name `vPos` über den Zahlenwert der vertikalen Position auf den Keller gelegt. Da diese beiden Operanden für den folgenden Befehl `def` in der umgekehrten Reihenfolge auf dem Keller liegen müssen, wird der Operator `exch` benutzt, um sie die Plätze tauschen zu lassen.

Die Prozedur `neueZeile` verändert den Wert von `vPos` und setzt dann die aktuelle Position auf den Anfang der nächsten Zeile.

```
/neueZeile {  /vPos  vPos Abstand sub  def
              Anfang vPos moveto
} def
```

Die erste Programmzeile ist die *PostScript*-Version der Zuweisung

```
vPos := vPos - Abstand.
```

Die andere Lösungsmöglichkeit benutzt den Operator `currentpoint`, um die vertikale Position zu ermitteln. Die Prozedur `Init` muß also nur die aktuelle Position an den Anfang der ersten Zeile setzen.

```
/Init  { Anfang exch moveto }  def
```

Der gewünschte *y*-Wert liegt ja bereits auf dem Keller. Der durch `Anfang` gegebene gewünschte *x*-Wert wird darüber gelegt. Der Operator `exch` erzeugt die richtige Reihenfolge *x y* für das folgende `moveto`.

Die Prozedur `neueZeile` ermittelt die momentane vertikale Position und setzt dann die aktuelle Position auf den Anfang der nächsten Zeile.

```
/neueZeile  { Anfang   currentpoint exch pop
                       Abstand sub
              moveto
} def
```

Sei x' der Wert von `Anfang` und d der Wert von `Abstand`, und sei die aktuelle Position zu Beginn (x,y). Zunächst wird x' auf den Keller gelegt. Der Operator `currentpoint` lädt die aktuelle Position, so daß die Kellerspitze zu x',x,y wird. Nun wird das überflüssige x beseitigt. Durch `exch` ergibt sich x',y,x und `pop` liefert x',y. Durch `Abstand` verändert sich die Kellerspitze zu x',y,d, und `sub` macht daraus $x',y-d$, was durch `moveto` zur neuen aktuellen Position wird.

Graphik in *PostScript*

Im Gegensatz zu *dvi* besitzt *PostScript* sehr mächtige Graphikoperationen zum Erzeugen von Linien und Flächen im Seitenraum. Im folgenden werden wir diese Operationen besprechen, wobei die Beschreibung keinerlei Anspruch auf Vollständigkeit erhebt.

Pfade

Wie wir schon am Anfang erwähnt haben, werden in *PostScript* nichttextuelle graphische Objekte nicht direkt in den Seitenraum gelegt. Statt dessen wird zunächst ein geometrisches Objekt, ein sogenannter Pfad konstruiert. Ein Pfad ist ein idealisierter Linienzug in der durch den Seitenraum beschriebenen Ebene. Er besteht aus einzelnen Segmenten, die gerade oder gekrümmt sein können. Ein Pfad hat einen bestimmten Durchlaufsinn, d. h. seine Segmente haben eindeutig bestimmte Anfangs- und Endpunkte. Pfade dürfen sich beliebig selbst überschneiden und können aus mehreren unzusammenhängenden Teilen bestehen (siehe Abbildung 9.3). Wir werden hier zur Vereinfachung allerdings nur zusammenhängende Pfade betrachten.

Die *PostScript*-Maschine verwaltet einen *aktuellen Pfad*, der durch gewisse *Pfadkonstruktoren* verlängert werden kann. Dieser aktuelle Pfad befindet sich zwar koordinatenmäßig im Seitenraum, ist aber noch kein graphisches Objekt, das durch `showpage` sichtbar würde. Erst durch einen *Maloperator* wird aus dem Pfad ein graphisches Objekt im Seitenraum berechnet.

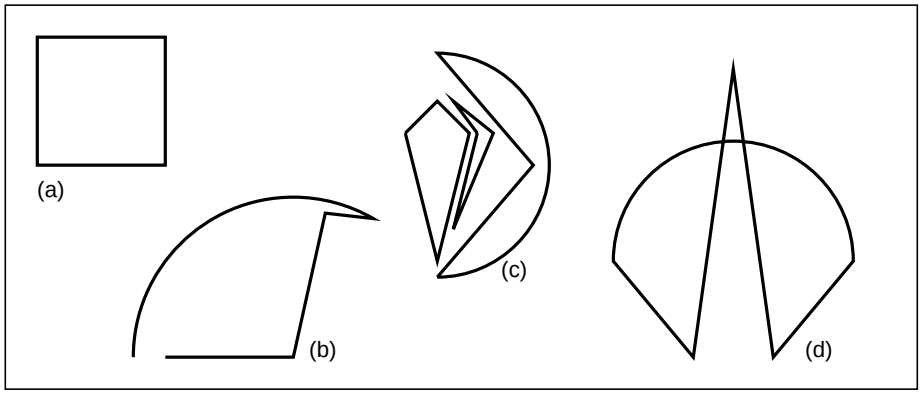

Abbildung 9.3: Mögliche Pfade

Pfadkonstruktoren

Einige der *PostScript*-Instruktionen zum Erzeugen eines Pfads sind in Tabelle 9.6 dargestellt. Alle Instruktionen außer `newpath` verlängern den aktuellen Pfad um ein neues Segment. Der Anfangspunkt des neuen Segments liegt immer an der aktuellen Position. Während der Ausführung der Instruktion wird die aktuelle Position immer zum Endpunkt des neuen Segments verschoben, was in Tabelle 9.6 nicht explizit erwähnt wird. Die Argumente der Pfadkonstruktoren beziehen sich natürlich auf das gerade aktuelle Koordinatensystem.

Mit Hilfe des Operators `charpath` ist es auch möglich, den aktuellen Pfad um ein Segment zu verlängern, das den Umrissen eines Zeichens entspricht. Damit verwischt sich der Unterschied zwischen Text und Graphik.

Maloperatoren

Wie schon oben erwähnt, hat der Aufbau eines aktuellen Pfads keinen direkten Einfluß auf den Seitenraum. Es gibt nun einige Operatoren, die den aktuellen Pfad in eine Veränderung des Zustands des Seitenraums umsetzen.

Wir betrachten hier nur die drei Operatoren `stroke`, `fill` und `clip`. Alle drei nehmen keine expliziten Operanden vom Operandenkeller und legen keine Ergebniswerte dorthin. Sie hängen jedoch von einigen impliziten Parametern ab und bewirken eine Veränderung des Zustands des Seitenraums. Alle drei haben die Eigenart, während der Abarbeitung den aktuellen Pfad zu löschen. Wie der aktuelle Pfad zur weiteren Verwendung gerettet werden kann, werden wir später sehen.

Der Operator `stroke` zeichnet den aktuellen Pfad in den Seitenraum, indem er für jedes Segment des aktuellen Pfads eine entsprechende Linie im Seitenraum erzeugt. Einige implizite Parameter bestimmen das exakte Aussehen dieser Linien. Dazu gehören die Breite, die Farbe, die Form der Enden (gerade abgeschnitten oder

Operanden	Instruktion	Bedeutung
	`newpath`	startet Konstruktion eines neuen Pfads
x' y'	`lineto`	zieht eine Strecke von der aktuellen Position zur Position (x',y')
r_x r_y	`rlineto`	zieht eine Strecke von der aktuellen Position (x,y) zur Position $(x+r_x,y+r_y)$
x' y' r α β	`arc`	zieht eine Strecke von der aktuellen Position zum Anfangspunkt des folgenden Kreisbogens und dann einen Kreisbogen um den Mittelpunkt (x',y') mit Radius r vom Anfangswinkel α bis zum Endwinkel β gegen den Uhrzeiger
	`closepath`	schließt den aktuellen Pfad durch eine Strecke von der aktuellen Position zurück zum Anfang

Tabelle 9.6: Einige Pfadkonstruktoren

abgerundet) und das Aussehen des Übergangs zwischen zwei benachbarten Segmenten des Pfads. All diese Parameter können durch gewisse Operatoren gesetzt und abgefragt werden.

Durch die Instruktion `fill` wird eine Fläche im Seitenraum gefärbt, die durch das Innere des aktuellen Pfads gegeben ist. Die Farbe der Fläche wird durch implizite Parameter bestimmt. Farbe ist hier in sehr allgemeinem Sinn zu verstehen. Es kann sich auch um eine Graustufe handeln oder ein Muster wie z. B. parallele Linien oder Karos. Zu beachten ist, daß jedes in den Seitenraum gesetzte Objekt alle Teile schon vorhandener Objekte, die es überlappt, vollständig überdeckt. Es ist also möglich, daß eine hellgraue Fläche eine dunkelgraue verdeckt. So können auch weiße Flächen, die an sich nicht sichtbar sind, dadurch erkennbar werden, daß sie Teile anderer Objekte verdecken. Das wird in Abbildung 9.4 demonstriert, die gleichzeitig einige der Graustufen zeigt, die durch den Befehl `setgray` gesetzt werden können.

Wie wird das Innere eines Pfads bestimmt? „Offene“ Pfade werden zunächst geschlossen. Mit Hilfe der *Nicht-Null-Drehregel* wird entschieden, ob ein Punkt innerhalb des aktuellen Pfads liegt. Man läßt von dem Punkt einen Strahl in einer beliebigen Richtung ausgehen. Ein Zähler wird erhöht für jedes Pfadstück, welches den Strahl von links nach rechts kreuzt und erniedrigt für jedes, welches ihn von rechts nach links kreuzt. Ist am Ende der Zählerstand Null, so ist der Punkt außerhalb, sonst innerhalb (siehe Abbildung 9.5). In dem pathologischen Fall, daß der Strahl mit einem Pfadsegment zusammenfällt, kann ein anderer Strahl gewählt werden.

Die *PostScript*-Maschine verwaltet auch eine *Seitenmaske*, das ist ein bestimmtes Teilgebiet des Seitenraums. Nur diejenigen Teile eines graphischen Objekts, die

Abbildung 9.4: Beispiele für Graustufen und gegenseitiges Verdecken

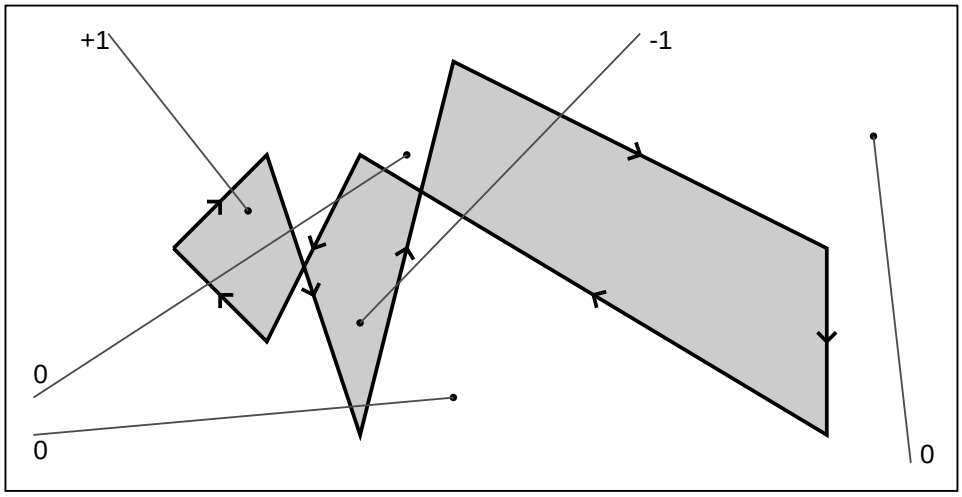

Abbildung 9.5: Die Nicht-Null-Drehregel

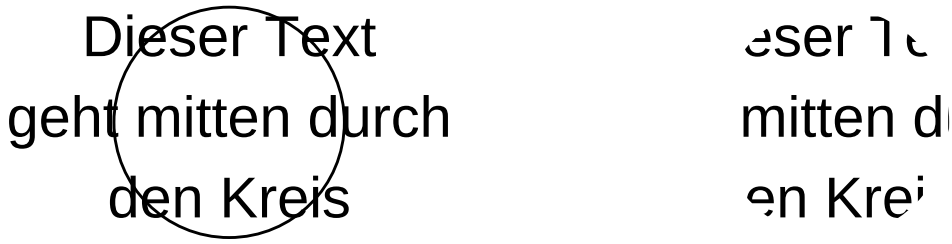

Abbildung 9.6: Der `clip`*-Operator*

bei seiner Erzeugung im Inneren der aktuellen Seitenmaske liegen, werden später durch `showpage` wirklich sichtbar gemacht. Der `clip`-Operator macht das Innere des aktuellen Pfads zur neuen aktuellen Seitenmaske. Während eine `fill`-Aktion nur die bereits im Seitenraum vorhandenen Objekte durch Überdeckung beeinflussen kann, kann eine `clip`-Aktion nur die Objekte beeinflussen, die nachher in den Seitenraum gelegt werden.

Die Wirkungsweise des `clip`-Operators wird in Abbildung 9.6 verdeutlicht. Auf der linken Seite wird ein kreisförmiger Pfad konstruiert und mit `stroke` gezeichnet. Danach wird der Text geschrieben. Auf der rechten Seite wird ebenfalls zuerst ein kreisförmiger Pfad konstruiert, dann aber mit Hilfe von `clip` zur Seitenmaske erklärt. Von dem anschließend geschriebenen Text werden nur diejenigen Teile sichtbar, die in das Innere des Kreises fallen.

Schließlich gibt es in *PostScript* noch den mächtigen Operator `pathforall`, der es erlaubt, den aktuellen Pfad zu durchlaufen und für jedes angetroffene Segment eine bestimmte Aktion durchzuführen. Unter Benutzung von `pathforall` können Prozeduren geschrieben werden, die die Länge eines Pfads berechnen oder eine Textzeile längs eines Pfads plazieren.

Koordinatensysteme

Zur Beschreibung von Positionen im Seitenraum wird ein geräteunabhängiges Benutzerkoordinatensystem herangezogen, das automatisch in das (meist davon verschiedene) Koordinatensystem des jeweiligen Ausgabegeräts transformiert wird.

Das Benutzerkoordinatensystem hat für jede Seite eine feste Voreinstellung: Der Ursprung $(0,0)$ ist in der linken unteren Ecke der Seite. Die *x*-Achse ist horizontal nach rechts und die *y*-Achse vertikal nach oben gerichtet. Beide Achsen sind mit $\frac{1}{72}$ Zoll skaliert. Das ist eine der verschiedenen möglichen Definitionen des „typographischen Punktes“ (siehe Unterabschnitt 2.1.2). Ein *PostScript*-Punkt heißt in TEX und LATEX `bp`; er ist etwas größer als der TEX-Punkt `pt`.

Das Benutzerkoordinatensystem kann durch die *PostScript*-Befehle, die in Tabelle 9.7 beschrieben sind, jederzeit verändert werden. Dazu kommt noch ein Ope-

Operanden	Instruktion	Bedeutung
r_x r_y	`translate`	verschiebt den Ursprung des Benutzerkoordinatensystems relativ um (r_x, r_y) (siehe Abbildung 9.7)
s_x s_y	`scale`	streckt x-Achse und y-Achse um s_x bzw. s_y (siehe Abbildung 9.8)
α	`rotate`	rotiert das Benutzerkoordinatensystem um α Grad (siehe Abbildung 9.9)

Tabelle 9.7: PostScript-Operatoren zur Manipulation des Koordinatensystems

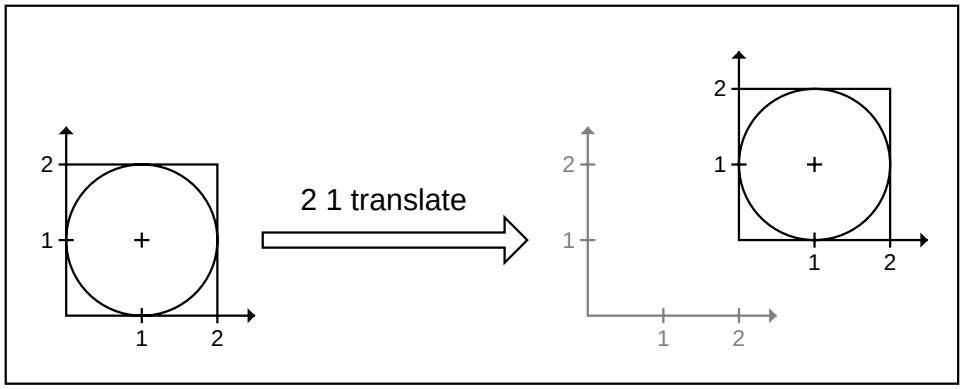

Abbildung 9.7: Der `translate`*-Operator*

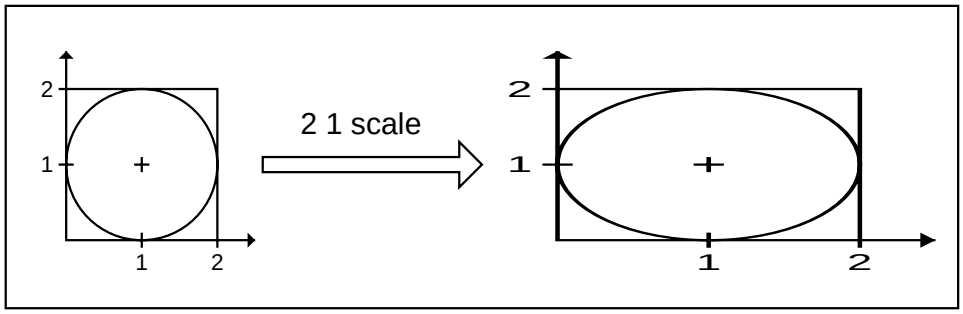

Abbildung 9.8: Der `scale`*-Operator*

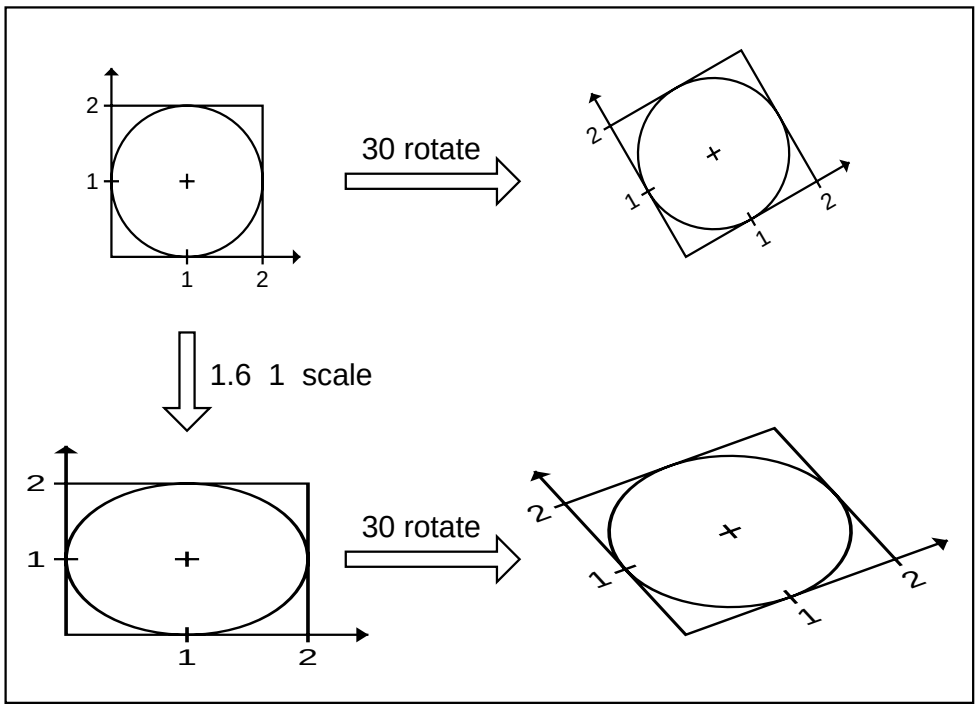

Abbildung 9.9: Der `rotate`*-Operator*

rator, der eine beliebige affine Transformation auf das Benutzerkoordinatensystem anwendet. Alle diese Transformationen beziehen sich auf das gerade gültige Koordinatensystem, d. h. eine Folge von Transformationen hat einen kumulativen Effekt.

Die durch α `rotate` ausgelöste Rotation bewirkt nicht immer das, was man intuitiv erwarten würde (siehe Abbildung 9.9 unten). Bei allen Koordinatensystemen verschiebt `90 rotate` den Punkt $(1,0)$ in den Punkt $(0,1)$. Wenn das gerade gültige System ungleichmäßig skalierte Achsen hat, erfolgt also die Rotation bezüglich des Gerätekoordinatensystems (d. h. der entstehenden Papierdarstellung) nicht auf Kreisbahnen, sondern auf elliptischen. Die entstehenden Verzerrungen bewirken, daß ein rechtwinkliges ungleichmäßig skaliertes System durch den Befehl `45 rotate` in ein System übergeht, in dem die Achsen gleich skaliert sind, aber nicht mehr rechtwinklig aufeinander stehen.

Eine Koordinatentransformation hat keine Auswirkung auf die Objekte, die bereits im Seitenraum liegen, oder auf den aktuellen Pfad. Die spätere Lage eines Objekts auf der gedruckten Seite richtet sich immer nach dem Koordinatensystem, das während der Konstruktion des Objekts gültig war. Betrachten Sie z. B. die folgende Prozedur:

```
/box { newpath
       0 0 moveto  1 0 lineto  1 1 lineto  0 1 lineto
       closepath   stroke
} def
```

Jeder Aufruf von `box` erzeugt ein viereckiges Objekt, dessen vier Ecken an den Punkten $(0,0)$, $(1,0)$, $(1,1)$ und $(0,1)$ des während des Aufrufs gültigen Koordinatensystems liegen. Ein Aufruf im vordefinierten System erzeugt also ein Quadrat der Seitenlänge $\frac{1}{72}$ Zoll, dessen linke untere Ecke an der linken unteren Ecke der Seite liegt. Aufrufe nach `translate`- und `scale`-Transformationen erzeugen seitenrandparallele Rechtecke. Nach der Rotation eines gleichmäßig skalierten Systems entstehen durch Aufrufe von `box` Quadrate in beliebiger Lage. Nach einer Rotation eines ungleichmäßig skalierten Systems können aufgrund der oben erwähnten Verzerrungen Parallelogramme entstehen.

Zu beachten ist, daß auch die vom `stroke`-Operator erzeugten Striche vom aktuellen Koordinatensystem abhängen. In dem ungleichmäßig skalierten System von Abbildung 9.8 sind z. B. die vertikalen Linien doppelt so breit wie die horizontalen.

Nicht nur gerade konstruierte graphische Objekte, sondern auch die durch `show` in den Seitenraum gelegten Texte richten sich nach dem aktuellen Koordinatensystem. Betrachten Sie dazu die Zahlen in Abbildung 9.8 und 9.9. Es ist also möglich, durch geeignete Koordinatentransformationen Text zu vergrößern (gleichmäßiges `scale`), zu dehnen (`scale` in einer Richtung), zu verzerren (nichtrechtwinkliges System) und in beliebige Richtungen laufen zu lassen (`rotate`). Rotierter Text wird in Abbildung 9.10 gezeigt. In dem *PostScript*-Programm für dieses Bild steht zwischen der Ausgabe der einzelnen Schriftzüge jeweils der Befehl `15 rotate`. Das Programm zu Abbildung 9.11 ist identisch zu dem von Abbildung 9.10 bis auf die Tatsache, daß ganz am Anfang der Befehl `1.4 0.7 scale` steht.

Der Graphikzustand

Wir haben mehrfach erwähnt, daß viele *PostScript*-Operatoren implizite Parameter lesen und / oder verändern. Diese impliziten Parameter werden von der *PostScript*-Maschine als Inhalte spezieller Register verwaltet. Sie bilden zusammen den *aktuellen Graphikzustand*. Die meisten Komponenten des aktuellen Graphikzustands können auch explizit gesetzt und abgefragt werden.

Die wichtigsten Komponenten des Graphikzustands sind:

- der aktuelle Pfad, der bereits ausführlich besprochen wurde.
- die aktuelle Position, (x,y), die durch `moveto` und `rmoveto` gesetzt und durch den operandenlosen Befehl `currentpoint` abgefragt werden kann.
- die Linienparameter, die unter anderem die Breite der von `stroke` erzeugten Linien regeln. Diese Breite kann mit dem Befehl `setlinewidth` gesetzt und mit `currentlinewidth` abgefragt werden.

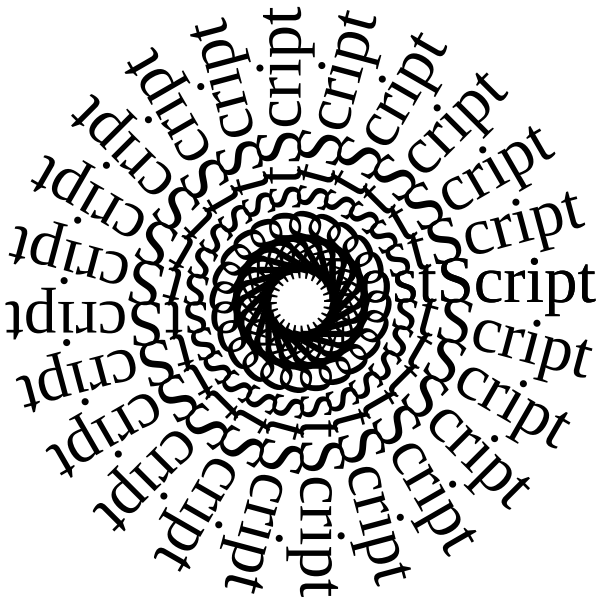

Abbildung 9.10: Rotierter Text

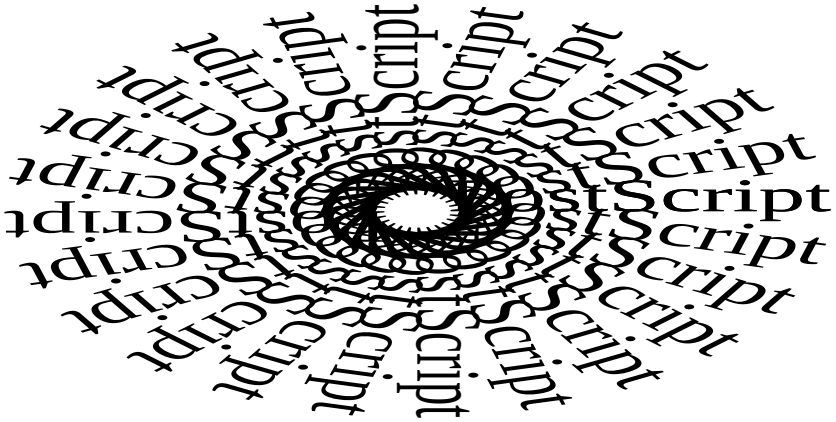

Abbildung 9.11: Skalierter und rotierter Text

- die Tintenfarbe, die vom `fill`-Operator benutzt wird. Dazu gehört insbesondere die verwendete Graustufe, die mit `setgray` gesetzt und mit `currentgray` abgefragt werden kann. Die Graustufe ist eine Zahl zwischen 0 (schwarz) und 1 (weiß) (siehe Abbildung 9.4). Der voreingestellte Wert ist 0.
- den Font, in dem der Operator `show` Texte setzt. Wie bereits erläutert, wird er mit `setfont` gesetzt. Der Operator `currentfont` schiebt den aktuellen Font auf den Keller.
- die Seitenmaske, das ist der Bereich, in dem gezeichnete Objekte wirklich sichtbar werden (siehe Abbildung 9.6). Der Operator `clip` macht, wie schon erwähnt, den aktuellen Pfad zum Rand der Seitenmaske. Der Operator `clippath` tut das Umgekehrte.
- die Transformationsmatrix, die die Übersetzung des jeweils aktuellen Benutzerkoordinatensystems in das Gerätekoordinatensystem beschreibt. Diese Matrix wird durch `translate`, `scale` und `rotate` implizit verändert. *PostScript* bietet die für Koordinatentransformationen benötigten Matrizen auch explizit als Datentyp mit einigen mathematischen Operationen wie Multiplikation und Invertierung an. Der Befehl `setmatrix` nimmt so eine Matrix vom Keller und macht sie zur aktuellen Transformationsmatrix, während `currentmatrix` die aktuelle Matrix auf den Keller schreibt.

Oft ist es günstig, zum Zeichnen eines graphischen Objekts den Graphikzustand zu verändern. Danach will man meist zu dem ursprünglichen Zustand zurückkehren. Daher besitzt die *PostScript*-Maschine auch einen *Graphikzustandskeller*. Der Befehl `gsave` kellert den aktuellen Graphikzustand, und `grestore` restauriert ihn wieder aus dem Graphikzustandskeller. So ist es auch möglich, den aktuellen Pfad, der ja durch `stroke` und `fill` gelöscht wird, mehrfach zu benutzen, z. B. um die Umrisse einer Figur schwarz zu zeichnen und das Innere grau zu färben.

PostScript als Programmiersprache

Zusätzlich zu der Fähigkeit, die Ausgabe von graphischen Objekten beschreiben zu können, besitzt *PostScript* eine Reihe von Mechanismen, die es zu einer vollgültigen Programmiersprache machen. Wir haben bereits gesehen, daß es möglich ist, in *PostScript* Prozeduren zu definieren, die auch rekursiv sein können. Hier wollen wir einige weitere Programmiersprachenfähigkeiten von *PostScript* betrachten.

Datenstrukturen

Wie schon mehrfach erwähnt, bietet *PostScript* außer einfachen Datenobjekten wie Zahlen und Wahrheitswerten auch eine Reihe von Typen zusammengesetzter Datenobjekte an, zu denen jeweils einige vordefinierte Operationen gehören. Die wichtigsten dieser Typen sind:

- *Felder* (Arrays): Ein Feld ist eine Zusammenfassung von anderen Datenobjekten beliebigen Typs. Es ist möglich, einzelne Feldelemente über ihre Nummer abzurufen oder gegen andere auszutauschen.
- *Zeichenfolgen* (Strings) sind spezialisierte Felder, die außer den allgemeinen Feldoperationen noch einige zusätzliche anbieten.
- *Matrizen* sind Felder aus sechs Zahlen, die einige Matrixoperationen unterstützen. Das aktuelle Koordinatensystem wird durch eine aktuelle Transformationsmatrix definiert.
- *Assoziativfelder* (Dictionaries) sind Felder aus Paaren, die aus einem Schlüssel (meist einem Namen) und einem Wert bestehen. Die Werte können über ihre Schlüssel abgefragt werden, neue Paare können eingetragen werden und der Wert in einem bereits existierenden Paar kann ausgetauscht werden.

Im Gegensatz zu einfachen Datenobjekten liegen die zusammengesetzten nicht auf dem Operandenkeller, sondern in der Halde. Auf dem Keller befinden sich nur Verweise auf die eigentlichen Objekte. Wenn also (in einer ungenauen Sprechweise) ein Feld oben auf dem Keller liegt, dann erzeugt der Befehl `dup` nicht eine Kopie dieses Feldes, sondern nur einen weiteren Verweis auf dasselbe Feld, was beim Verändern von Feldkomponenten offenbar wird.

Für jeden zusammengesetzten Datentyp gibt es in *PostScript* einen Operator, der ein Objekt dieses Typs in der Halde erzeugt und einen Verweis darauf oben auf den Keller legt. Diese Erzeugungsoperatoren nehmen meist einen Operanden vom Keller, der die Größe des zu erzeugenden Objekts angibt.

Nahezu alle Operatoren, die als Ergebnis ein zusammengesetztes Objekt liefern, erzeugen dieses Objekt nicht selbst, sondern benötigen unter ihren Operanden einen Verweis auf ein bereits existierendes Objekt, das sie mit ihrem Ergebniswert überschreiben. Der Operator `cvs` (convert to string) zum Beispiel hat zwei Operanden, eine Zahl und (einen Verweis auf) eine Zeichenfolge. Er überschreibt die Zeichenfolge mit der Dezimaldarstellung der Zahl und hinterläßt (einen Verweis auf) die geänderte Zeichenfolge auf dem Keller. Diese Vorgehensweise erlaubt einen speicherplatzsparenden Programmierstil, bei dem soweit möglich immer wieder dieselben Objekte benutzt werden, z. B. wenn eine ganze Folge von Zahlen gedruckt werden soll (was nur mit `cvs` gefolgt von `show` möglich ist).

Kontrollstrukturen

Als Programmiersprache besitzt *PostScript* auch Kontrollstrukturen für Verzweigungen und Schleifen (keine Sprünge). Anders als in den üblichen Programmiersprachen werden diese Kontrollstrukturen nicht durch eine spezielle Syntax dargestellt, sondern es gibt Kontrolloperatoren, die wie alle anderen Operatoren auch ihre Operanden vom Operandenkeller nehmen. Unter diesen Operanden sind bei Kontrolloperatoren gewöhnlich ein oder zwei Prozedurobjekte, die meist explizit als in geschweifte Klammern eingeschlossene Programmstücke angegeben werden. In Tabelle 9.8 werden die wichtigsten Kontrolloperatoren beschrieben.

Operanden	Operator	Bedeutung
b p	`if`	führt Programmstück *p* aus, wenn *b* = *true*, tut nichts im anderen Fall
b p_1 p_2	`ifelse`	führt p_1 aus, wenn *b* = *true*, und p_2, wenn *b* = *false*
n p	`repeat`	führt *p* *n*-mal aus
p	`loop`	führt *p* immer wieder aus
	`exit`	beendet die innerste aktive Schleife
a s e p	`for`	zählt vom Anfang *a* mit Schrittweite *s* bis zum Ende *e*, führt für jeden Zählerwert das Programmstück *p* aus, wobei vor Ausführung von *p* der aktuelle Zählerwert auf den Operandenkeller gelegt wird
f p	`forall`	führt *p* einmal für jedes Element des Feldes *f* aus, wobei dieses Element auf dem Operandenkeller liegt

Tabelle 9.8: Kontrolloperatoren in PostScript

Bindungen

Wir haben bereits gesehen, daß in *PostScript*-Programmen unter Benutzung des Operators `def` Namen an Daten- oder Prozedurobjekte gebunden werden können. Was bis jetzt verschwiegen wurde, ist, daß im Gegensatz zu den meisten modernen Programmiersprachen alle so vereinbarten Namensbindungen global sichtbar sind, wenn keine besonderen Vorkehrungen getroffen werden.

Um Namenslokalität zu ermöglichen, verwaltet die *PostScript*-Maschine einen *Bindungskeller* (dictionary stack). Die Einträge im Bindungskeller sind Assoziativfelder, d. h. Felder aus Paaren, die aus einem Namen und einem Wert bestehen. Der oberste Eintrag im Bindungskeller ist die *aktuelle Bindung*.

Die `def`-Instruktion nimmt einen Namen und einen Wert vom Operandenkeller und fügt sie in die aktuelle Bindung ein; entweder als neues Paar, wenn der Name noch ungebunden war, oder durch Überschreiben eines bereits bestehenden Paares.

Tritt ein Name angewandt im Programm auf, dann wird der Bindungskeller von oben nach unten durchsucht, bis der an den Namen gebundene Wert gefunden ist. Die Suche fängt also bei der aktuellen Bindung an, also mit der zuletzt hergestellten.

Beim Eintritt in eine Prozedur wird nicht automatisch eine neue Bindung angelegt. Wenn im Programm Lokalität gewünscht wird, können die Operatoren `begin` und `end` benutzt werden. Durch `begin` wird eine neue aktuelle Bindung auf den Bindungskeller geschoben und durch `end` wird sie wieder gelöscht. Der Bindungs-

wechsel erfolgt also rein dynamisch durch die Ausführung gewisser Operatoren. Die Zuordnung von angewandten zu definierenden Vorkommen von Namen kann also im allgemeinen nicht statisch durch Betrachten des Programmtexts ermittelt werden; *PostScript* benutzt im Gegensatz zu den meisten anderen Programmiersprachen die *dynamische Bindung* für Namen. Ein angewandtes Vorkommen eines Namens bezieht sich immer auf die letzte im Programmablauf für diesen Namen hergestellte Bindung. Bei *statischer Bindung* bezöge sich solch ein Vorkommen auf die Definition des Namens in dem kleinsten das Auftreten textuell umfassenden Gültigkeitsbereich, d. h. Programmstück zwischen `begin` und `end`.

In Wirklichkeit ist die Namensbindung in *PostScript* noch komplizierter als eben beschrieben. Bindungen sind nämlich nichts anderes als die im Abschnitt „Datenstrukturen“ eingeführten Assoziativfelder, also zusammengesetzte Datenobjekte, die auch explizit manipuliert werden können. Wie alle anderen zusammengesetzten Objekte liegen sie in der Halde, d. h. der Bindungskeller beinhaltet eigentlich nur Verweise auf Bindungen. Durch den Befehl `begin` wird nicht eine neue leere Bindung erschaffen. Stattdessen erwartet `begin` als Operanden auf dem Operandenkeller einen Verweis auf ein bereits existierendes Assoziativfeld, der auf den Bindungskeller geschoben wird. Dadurch wird eine Vielzahl von Programmiermöglichkeiten eröffnet. In einer Prozedur kann bei jedem Aufruf dieselbe Bindung benutzt werden, was bewirkt, daß lokale Variablen ihre Werte zwischen zwei Aufrufen behalten. Module können simuliert werden, indem einige Prozeduren zusammen mit einigen Variablen in eine Bindung gepackt werden, die an einen Namen gebunden wird. Sie kann von „Kunden“ des Moduls unter diesem Namen angesprochen und durch Installieren auf dem Bindungskeller benutzt werden.

Resümee der *PostScript*-Maschine und abschließendes Beispiel

Wie wir gesehen haben, benötigt die *PostScript*-Maschine mehrere Keller für verschiedene Arten von rekursiver Schachtelung. Diese Keller dienen dazu, verschiedene Arten der Rekursion in dem Anwendungsbereich zu unterstützen.

- Der Graphikzustandskeller wird benötigt, um Teilobjekte in ihrem eigenen Graphikzustand bearbeiten zu können.
- Mit Hilfe des Bindungskellers hat man die Möglichkeit, temporär die bestehenden Bindungen von Namen zu überschreiben.
- Da *PostScript* die Definition und den Aufruf von Prozeduren erlaubt, muß es einen Ausführungskeller für Prozeduraufrufe geben.
- Schließlich können *PostScript*-Programme auch noch rechnen und benötigen dafür den Operandenkeller.

Zusätzlich zu den Kellern hat die *PostScript*-Maschine noch eine Halde zur Abspeicherung nichtskalarer Objekte und solcher mit unbegrenzter Lebensdauer.

Die *PostScript*-Maschine hat also einen ziemlich komplexen Aufbau und einen mächtigen Befehlsvorrat. Der Befehlsvorrat ist weitaus mächtiger, als es für eine

```
/pt  { 72 div  2.54 mul }  def
% Umrechnen von pt in cm  (72 Punkte = 1 Zoll = 2.54 cm)
/centershow {   % erwartet einen String auf dem Keller.
  % Dieser wird zentriert um die aktuelle Position gesetzt.
  % Die Position verschiebt sich dabei zum Ende des Strings.
  dup   stringwidth pop    % L"ange des gesetzten Strings
          2 div neg        % x-Verschiebung zum Stringanfang
          0 rmoveto        % gehe zum Stringanfang
  show } def
/beschrift { % Keller: Beschriftung (string). Global: radius
  % Die y-Achse zeigt auf die zu beschriftende Stelle
  gsave   0 radius translate       % Ursprung auf Kreisrand,
                                   % x-Achse tangential
    newpath           % folgt: Dreieck am Kreisrand erzeugen
      -0.1 0 moveto   0 0.3 lineto   0.1 0 lineto  closepath
    fill              % mit Standardgrauwert schwarz
    0 0.5 moveto      % aktuelle Position in Textmitte
    % an der Kellerspitze liegt immer noch die Beschriftung!
    centershow        % druckt String zentriert an akt. Pos.
  grestore    % zur"uck zu System mit Ursprung in Kreismitte
} def                   % Was folgt, ist das Hauptprogramm
72 2.54 div dup scale   % Skalieren in Zentimetern
10.5 15 translate       % Ursprung in die Mitte
/radius 2 def           % Radius des Kreises (in cm)
gsave                   % es folgt Definition des Nordpfeils
  radius dup scale      % skaliert Achsen mit Kreisradius
  0.5 setgray           % legt Grauton f"ur Nordpfeil fest
  newpath  0 0.8 moveto  -0.4   0.3 lineto  -0.15 0.4 lineto
      -0.15 -0.8 lineto   0.15 -0.8 lineto   0.15 0.4 lineto
       0.4   0.3 lineto   closepath
  fill   % zeichnet Nordpfeil mit gew"ahltem Grauton
grestore % zur"uck zu cm-System und Standardgrauwert schwarz
2 pt  setlinewidth     % legt Liniendicke fest
newpath                % es folgt Definition des Kreisrings
  radius 0 moveto      % vermeidet Miterzeugen einer Strecke
  0 0 radius 0 360 arc        % Vollkreis von 0 bis 360 Grad
stroke         % zeichnet Kreis mit festgelegter Liniendicke
/Times-Roman findfont 14 pt scalefont setfont  % Fontsetzung
[(N) (NW) (W) (SW) (S) (SO) (O) (NO)] % ein Feld von Strings
   {       % wird von ‚forall' f"ur jeden String ausgef"uhrt
     beschrift   % Der jeweilige String liegt auf dem Keller
     45 rotate   % Die y-Achse soll auf die Stelle zeigen,
   } forall      %     die beschriftet werden soll
showpage         % druckt Inhalt des Seitenraums
```

Abbildung 9.12: PostScript-Programm für die Graphik in Abbildung 9.13

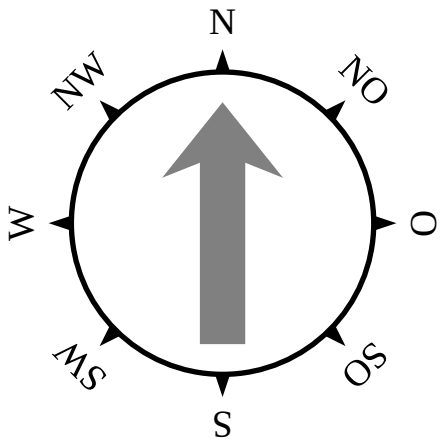

Abbildung 9.13: Beispiel einer PostScript-erzeugten Graphik

reine Zielsprache für Übersetzungen nötig wäre. Durch die funktionale Abstraktion und das explizite Manipulieren von Bindungen wird die *PostScript*-Maschine zu einer universellen (nicht einfach) programmierbaren Maschine.

In Abbildung 9.12 zeigen wir ein Beispiel für ein direkt in *PostScript* geschriebenes Programm, das viele der in diesem Kapitel eingeführten Operatoren verwendet. Alles, was rechts von einem Prozentzeichen steht, ist Kommentar. Die von dem Programm erzeugte Graphik wird in Abbildung 9.13 gezeigt.

9.6 Abstrakte Maschinen

Prozedurale Sprachen kann man als die Maschinensprachen von geeignet entworfenen abstrakten oder virtuellen Maschinen ansehen. Diese werden in der Informatik häufig eingesetzt, wenn Maschinen- und Geräteunabhängigkeit erreicht werden sollen. Der vorliegende Abschnitt beschreibt die Motivation für den Entwurf abstrakter Maschinen, charakterisiert sie und gibt einige Beispiele für ihre Architektur.

Motivation

Entwirft man ein Softwaresystem, etwa ein Textverarbeitungsprogramm oder einen Übersetzer für eine Programmiersprache, so soll dieses System im allgemeinen auf einer Vielzahl von Konfigurationen aus Prozessor und Peripheriegeräten ablauffähig sein. Damit möchte man die Marktchancen verbessern und den Systemerstellungsaufwand reduzieren. Deshalb möchte man die Realisierung im allgemeinen

unabhängig von dem verwendeten Rechnertyp und der Art der angesprochenen Peripheriegeräte vornehmen. Das heißt, daß in die Programme keine Größen „hineingebaut“ sein dürfen, die spezifisch für einen speziellen Rechnertyp oder ein spezielles Peripheriegerät sind.

Virtuelle Terminals

Betrachten wir die oben genannten Beipiele etwas näher. Ein Textverarbeitungsprogramm muß mit einem Terminal (Bildschirm und Tastatur) kommunizieren, vor dem der Benutzer sitzt. Leider stimmen verschiedene Terminaltypen selten in ihren Fähigkeiten und den Codes, die Tastendrücke an den Rechner schicken, sowie in der Ansteuerung von Funktionen des Bildschirms überein. Die gewünschte Geräteunabhängigkeit des Textverarbeitungssystems erreicht man durch eine *Virtualisierung*. Die Textverarbeitung kommuniziert nur mit *virtuellen* Geräten statt mit realen Geräten bestimmter Typen.

Dazu entwirft man ein abstraktes Bildschirmmodell, welches durch eine Menge von Eigenschaften und eine Menge von Attributen charakterisiert wird. Ein Paar von Attributen etwa bestimmt die Größe des Bildschirms, also die Zahl der Zeilen und der Spalten. Eine Eigenschaft ist es, ob das Terminal „rollen“ (scroll) kann oder nicht. Weiterhin gehören dazu die Belegung der Cursor-Tasten, der Funktionstasten und einiger anderer Tasten mit Zeichenkombinationen. Für jedes „reale“ Terminal müssen diese Eigenschaften und Attribute in einer Datenbank (unter Unix `termcap` bzw. `terminfo`) beschrieben werden. Die Anwendungen greifen zur Kommunikation mit dem Terminal auf diese Datenbank zu.

Nehmen wir einmal an, eine Anwendung belegt die Funktionstaste F1 mit einer bestimmten Funktion, z. B. dem Ruf nach Hilfe. Zwei Benutzer an zwei verschiedenen Terminals drücken die F1-Taste und bewirken damit, daß zwei verschiedene Zeichenkombinationen an den Rechner geschickt werden. Mittels der Einträge für diese beiden Terminaltypen in der entsprechenden Datenbank wird aber jedesmal erkannt, daß beide Zeichenkombinationen tatsächlich durch das Drücken der jeweiligen F1-Taste verursacht wurden. Die Anwendung stellt also jeweils fest, daß Hilfe verlangt wird.

Druckerunabhängigkeit

Ein Textverarbeitungsprogramm muß natürlich auch mit verschiedenen Druckertypen kommunizieren können. Der Ausdruck eines Dokuments sollte, im Rahmen der Gerätemöglichkeiten, auf allen Druckern möglichst gleich aussehen. Außerdem sollte wie oben die Schnittstelle zum Drucker virtualisiert sein; d. h. die Ausgabe des Textverarbeitungsprogramms sollte unabhängig vom verwendeten Drucker sein.

Sie haben im Abschnitt 9.1 schon eine solche (fast) geräteunabhängige Schnittstelle kennengelernt, nämlich Bitmatrizen. Dabei nimmt man an, daß der Drucker in

der Lage ist, ein Feld von Punkten (Pixel) zu drucken. Die einzige Funktion eines solchen Druckers wäre es, einen schwarzen Punkt an eine durch ein 1-Bit angegebene Stelle zu setzen. Das Textverarbeitungsprogramm schickt ihm also ein solches Pixelfeld für jede zu druckende Seite, und der Drucker druckt es brav auf's Papier.

Diese Vorgehensweise hat jedoch mehrere, im angesprochenen Abschnitt diskutierte Nachteile. Deshalb muß eine andere Art der Geräteunabhängigkeit gefunden werden. Drucker sind dazu „intelligenter" gemacht worden. Man hat ihnen einen eigenen Prozessor und eigenen Speicher gegeben. Auf diesem eingebauten Rechner ist eine *abstrakte Maschine* realisiert. Eine abstrakte Maschine ist einem *realen Rechner* ähnlich, wie Sie ihn bei Ihrem PC-Händler kaufen können. Sie hat also Speicher, Register und einen Prozessor mit einer Maschinensprache. Nur ist diese Maschine im allgemeinen einer speziellen Anwendung gewidmet, also als Allzweckprozessor schlecht zu gebrauchen, und nicht in Hardware, sondern in Software realisiert.

In unserem Fall bietet die abstrakte Maschine eine für diesen Zweck geeignete Menge von Funktionen an, wie z. B. „drucke ein a in Font f" oder „ziehe eine Linie von Position (x_1, y_1) nach Position (x_2, y_2)". Die Ausgabe des Textverarbeitungsprogramms ist dann ein Programm, welches solche Funktionen aufruft. Meist ist ein solches Programm, dessen Ausführung ein bestimmtes Druckbild produziert, erheblich kleiner als ein entsprechendes Pixelfeld. Dadurch wird die Kommunikation zwischen Rechner und Drucker stark reduziert.

Übersetzung von Programmiersprachen

Ein Übersetzer für eine höhere Programmiersprache *S* hat die Aufgabe, in *S* geschriebene Programme, sogenannte *Quellprogramme*, in *Zielprogramme* in der Maschinensprache eines Rechners zu übersetzen, auf dem die Programme zur Ausführung gebracht werden sollen. Verschiedene Rechnertypen enthalten oft verschiedene Prozessoren, die voneinander verschiedene Sprachen verstehen. Natürlich möchte man für möglichst viele verschiedene Prozessortypen Zielprogramme erzeugen können. Hat man n verschiedene Zielmaschinen im Auge, so werden n verschiedene *S*-Übersetzer fällig. Da *S* nicht die einzige höhere Programmiersprache auf der Welt ist – es gibt einige Hundert im Gebrauch –, so fiele für die Übersetzerbauer eine gewaltige Aufgabe an. Bei m zu implementierenden Programmiersprachen und n Zielrechnern müßten $m \times n$ Übersetzer geschrieben werden.

Auch hier ist der Entwurf und die Realisierung einer geeigneten abstrakten Maschine ein Hilfsmittel zur Reduktion des Aufwands. Man wird die abstrakte *S*-Maschine so entwerfen, daß die Übersetzung von *S* in die Sprache dieses Rechners leicht ist. Dann realisiert man den entsprechenden Übersetzer. Nun bleibt noch für jede betrachtete Zielmaschine *M* übrig, die Programme der abstrakten *S*-Maschine auf *M* zur Ausführung zu bringen. Dazu kann man z. B. die Register und die Speicherbereiche dieser abstrakten Maschine auf Register und / oder Speicher von *M* abbilden und ihre Befehle durch *M*-Programme realisieren. Das wird man in einer weit verbreiteten Programmiersprache machen, wodurch für jede weitere

Maschine nur Anpassungen und Neuübersetzungen der entsprechenden Programme durchgeführt werden müssen. Beim Entwurf der abstrakten Maschine sollte man also zusätzlich darauf achten, daß man die abstrakte Maschine leicht auf realen Rechnern implementieren kann.

Damit wird der Gesamtaufwand auf die Erstellung *eines* Übersetzers von der Quellsprache in die Sprache der abstrakten Maschine, eine Implementierung und *n* Anpassungen dieser abstrakten Maschine reduziert. Bei *m* Quellsprachen und *n* Zielrechnern braucht man also nur noch *m* Übersetzer und *n* Anpassungen.

Architektur abstrakter Maschinen

Wie schon oben gesagt, ähneln abstrakte Maschinen realen Maschinen. Sie besitzen einen oder mehrere Prozessoren mit einer Maschinensprache, Register, Speicher und Ein-/Ausgabegeräte. Die Register sind – im Gegensatz zu denen bei realen Maschinen – meist bestimmten Zwecken gewidmet, etwa als Zeiger in gewisse Speicherbereiche.

Der Speicher jedes Rechners dient dazu, Programme und Daten (Objekte) für Berechnungen aufzunehmen. An Objekten gibt es erst einmal *skalare Objekte*, das sind solche, die nicht in kleinere Unterobjekte aufgeteilt werden können. Dazu gehören Zahlen, Zeichen, Adressen und Wahrheitswerte. Weiter gibt es *zusammengesetzte Objekte* wie Felder (arrays), Listen und Verbunde (records). Mit Hilfe von Zeigern kann man Geflechte von Objekten aufbauen.

Der Speicher ist nicht ein homogener linearer Speicher oder in Bänke von solchen linearen Speichern aufgeteilt wie bei realen Rechnern, sondern in Bereiche mit unterschiedlicher Charakteristik gegliedert. Dabei spielt die *Lebensdauer* der abzuspeichernden Objekte die entscheidende Rolle. Die hier betrachteten Arten von Objekten werden während der Ausführung eines Programms irgendwann kreiert und beenden spätestens am Ende des Programmlaufs ihre Existenz. Das Zeitintervall von der Kreation bis zum Ableben nennt man die *Lebensdauer*. Die Semantik einer Programmiersprache legt fest, wie die Lebensdauern für verschiedene Klassen von Objekten sind. In blockstrukturierten Programmiersprachen etwa ist festgelegt, daß beim Eintritt in einen Block neue Inkarnationen für alle lokalen Variablen kreiert werden und daß sie beim Verlassen des Blocks wieder aufgegeben werden. Ebenso legt die Semantik von Pascal fest, daß dynamisch, d. h. durch *new*, kreierte Objekte bis zum Ende der Programmausführung leben.

Meist sind also die verschiedenen Speicherbereiche durch die Lebensdauereigenschaften der in ihnen gespeicherten Objekte charakterisiert. Diese Lebensdauereigenschaften bestimmen jeweils ihre Speicherverwaltung mit einer speziellen Belegungs- und Freigabestrategie. Dabei gibt es im wesentlichen zwei verschiedene Lebensdauerregeln. Es gibt Objekte mit *geschachtelter Lebensdauer*, wie die oben erwähnten lokalen Variablen in blockstrukturierten Sprachen. Sie besitzen ineinander geschachtelte oder disjunkte Intervalle als Lebensdauer. Später kreierte Objekte

beenden ihre Existenz nicht nach früher kreierten. Für Objekte mit solcher Lebensdauerregel bietet sich eine Abspeicherung auf einem *Keller* (stack) an; denn die weiter oben im Keller liegenden Objekte dürfen wegen der Speicherfreigabestrategie keine längere Lebensdauer haben als die weiter unten liegenden. Keller sind deshalb attraktiv, da man für sie eine sehr effiziente Belegungs- und Freigabestrategie benutzen kann.

Oft gibt es in einer abstrakten Maschine mehrere Keller. Das ist dann der Fall, wenn es verschiedene Klassen von Objekten gibt, deren Lebensdauern eventuell überlappen oder die sehr unterschiedliche Strukturen und Speicheranforderungen haben. Im ersten Fall kann man keinen gemeinsamen Keller benutzen; im zweiten erreicht man mit getrennten Kellern meist eine einfachere Speicherorganisation und eine größere Speicherökonomie.

Weiter gibt es meist Objekte, deren Lebensdauer sich von ihrer Kreation bis zum Ende der Programmausführung erstreckt, wie die oben erwähnten dynamisch kreierten Objekte in Pascal oder auch C. Dafür gibt es dann eine *Halde* (heap). Natürlich wird jede „reale“ Halde irgendwann einmal voll. In komfortablen abstrakten Maschinen wird dann eine Aufräumaktion eingeleitet, die feststellt, welche Haldenobjekte nicht mehr vom Programm erreichbar sind, weil es keine Referenz mehr auf diese Objekte gibt. Ihr Speicherplatz kann dann, obwohl sie gemäß der Semantik noch leben, wieder freigegeben werden. Dieser Prozeß heißt *Speicherbereinigung* (garbage collection).

Eventuell sollte sich der Benutzer einer abstrakten Maschine gar nicht um die Art der Abspeicherung seiner Objekte kümmern müssen. Dafür bietet die abstrakte Maschine einen Speicher auf höherem Abstraktionsniveau an, z. B. einen Listen- oder einen Baumspeicher oder ein Koordinatensystem mit Linienzügen. Der Benutzer der abstrakten Maschine kann dann solche Objekte kreieren, manipulieren, abspeichern und löschen, ohne sich um die Details der Darstellung im Speicher zu kümmern.

Die *Befehle* (Instruktionen) der abstrakten Maschine haben im allgemeinen einen oder mehrere Operanden und einen gewissen Effekt. Dieser Effekt kann in der Berechnung eines Ergebnisses bestehen, in einer Speicherbelegungs- oder Speicherfreigabeaktion, in der Konstruktion eines Objekts in einem Objektspeicher oder in einer Ein- oder Ausgabeaktion. Die Operanden können je nach der Architektur der Maschine und der Art des Befehls als Parameter im Befehl selbst stehen, über einen Adreßparameter im Speicher erreicht werden, mit ihrem Namen im Befehl erwähnt werden oder implizit für den Befehl festgelegt sein. Einige Beispiele für verschiedene Operandenarten sind im folgenden aufgelistet.

loadi R1 12 lädt die Konstante 12 in das Register R1.

load R1 12 lädt den Inhalt der Speicherzelle mit Adresse 12 in das Register R1.

add R1 R2 addiert die beiden Werte in den Registern R1 und R2 und speichert das Ergebnis in R1 ab.

push 12 verlängert den Keller um eine Zelle und lädt den Inhalt der Speicherzelle mit der Adresse 12 in die neue Zelle.

load lädt den Inhalt der Zelle, deren Adresse in der obersten Kellerzelle liegt, in die oberste Kellerzelle.

add entnimmt die Inhalte der beiden obersten Kellerzellen, verkürzt den Keller um diese, addiert die beiden (Zahlen-)Objekte und speichert das Ergebnis in einer neu angelegten obersten Zelle ab.

cons hd tl kreiert einen neuen Listenknoten im Listenspeicher mit einem Verweis auf den Kopf, hd, und den Rest, tl.

Haben die Befehle für die Arithmetik und die Adreßrechnung explizit oder implizit Register und/oder Speicheradressen als Operanden, so spricht man von *Registermaschinen*. Haben sie statt der Register implizit Kellerzellen als Operanden, so nennt man die Maschine eine *Kellermaschine*.

Die *dvi*-Maschine ist eine Registermaschine, da viele ihrer Befehle (implizit) Registeroperanden haben. Der Keller dient nur als Zwischenablage; es gibt keine Operationen auf den Inhalten von Kellerzellen. Die *PostScript*-Maschine dagegen ist eine Kellermaschine, da die meisten Befehle ihre Operanden vom Operandenkeller nehmen und eventuelle Ergebnisse dorthin schreiben.

Eine abstrakte Maschine hat wie jeder reale Rechner eine sogenannte *Hauptschleife*, die die Ausführung der Programme steuert. Ein spezielles Register, der *Befehlszähler*, zeigt jeweils auf die nächste auszuführende Instruktion im Programm. Er wird in der Hauptschleife um einen Befehl vorgerückt bzw. durch Sprünge oder Prozeduraufrufe explizit gesetzt. Die Hauptschleife sieht etwa folgendermaßen aus:

```
wiederhole:
    lade aktuellen Befehl;
    erhöhe Befehlszähler;
    führe aktuellen Befehl aus
```

Sie wird durch das Ausführen einer **stop**-Instruktion oder durch das Antreffen einer Fehlersituation beendet.