5 Vom Programm zum Computer

5.1 Einleitung

In den Kapiteln 2 und 4 haben Sie einen Eindruck davon bekommen, wie man Algorithmen und Programme entwirft, um ein vorgegebenes Problem mit Hilfe eines Computers zu lösen, sofern die Algorithmen überhaupt existieren (Kapitel 3). Dabei sind wir stets davon ausgegangen, daß man Computer bauen kann, die die erstellten Programme verarbeiten. In diesem Kapitel werden wir zeigen, wie Computer aufgebaut sein müssen, die Programme der Programmiersprache PRO verarbeiten können.

5.2 Entfernen höherer Sprachelemente aus einem Programm

Am einfachsten wäre es, direkt eine Maschine für PRO durch elektronische Schaltungen zu realisieren. Solch eine Maschine wäre jedoch außerordentlich kompliziert und unübersichtlich und ließe sich nur schwer geänderten Verhältnissen (anderen Programmiersprachen) anpassen. Man geht daher schrittweise vor, indem man die Sprachelemente von PRO nach und nach auf immer einfachere Elemente zurückführt, bis man schließlich ein Niveau erreicht hat, das als gegeben betrachtet werden kann (z.B. das Niveau elektronischer Schaltungen). Einen solchen *Reduktionsschritt* wollen wir beispielhaft im folgenden vorstellen.

Betrachten Sie hierzu ein Problem und einen zugehörigen Lösungsalgorithmus.
Problem: Es wird eine positive Zahl n und anschließend eine beliebig lange Folge von positiven ganzen Zahlen eingelesen. In zwei Zählern größer und kleiner werden die Zahlen aufsummiert, die größer bzw. kleiner gleich der Zahl n sind. Wird eine Null eingegeben, so stoppt das Programm und gibt die Werte von größer und kleiner aus.
Das Lösungsverfahren ist einfach. Die PRO-Notation lautet:

```
def x, n, größer, kleiner: Zahl
größer←0
kleiner←0
lies(n)
lies(x)
solange x≠0 tue
    wenn x≤n dann
        kleiner←kleiner+x
    sonst
        größer←größer+x
    ende
    lies(x)
ende
```

```
zeige(größer)
zeige(kleiner).
```

Offenbar sind die Sprachelemente zu komplex, um sie unmittelbar einer Maschine zu übertragen. Man sucht daher "einfachere" (= leichter zu realisierende) Sprachelemente, mit deren Hilfe man die komplexeren Sprachelemente präzise beschreiben kann. Hierbei kann man sich von der menschlichen Vorgehensweise leiten lassen. Man gehe z.B. genauso vor, wie man einem Kind mit geringem Wortschatz die Bedeutung unbekannter Wörter erklärt.

Beginnen wir mit den *Daten*. Zahlen notiert man als Mensch auf Zetteln, in Büchern oder Karteien. Die geplante Maschine muß also über ein vergleichbares "Gedächtnis" verfügen. Dieses wollen wir *Speicher* nennen. Der Speicher besteht für obiges Beispielprogramm aus mindestens vier einzelnen Behältern (man nennt sie *Speicherzellen* oder kurz Zellen), in denen jeweils eine Zahl aufgeschrieben wird. Jeder Behälter speichert den Wert eines der vier Bezeichner x, n, größer und kleiner. Natürlich muß man jede Zelle einzeln "ansprechen" können, d.h., man muß ihren Inhalt herauskopieren können, um ihn z.B. mit der Zahl, die im Behälter n steht, zu vergleichen. Hierzu gibt man jeder Zelle einen Namen, der die Zelle eindeutig identifiziert. Im allgemeinen verwendet man hierfür Zahlen 0, 1, 2, ..., die man *Adressen* nennt. Diese Zahlen schreibt man in Zeichnungen unmittelbar neben die zugehörige Zelle. Der Speicher besitzt also eine Struktur gemäß Abb. 1. Den Inhalt einer Speicherzelle mit dem Bezeichner x bezeichnen wir mit «x».

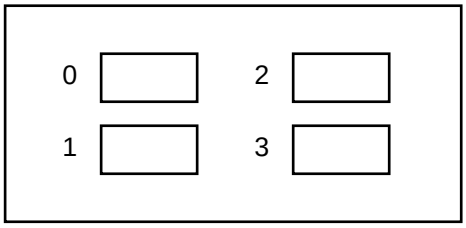

Abb. 1: Speicher mit 4 Speicherzellen, die von 0 bis 3 durchnumeriert sind.

Soweit die Daten. Nun zu den *Anweisungen*. Es muß elementare Anweisungen und Vergleiche sowie Sprachelemente für die Konstruktoren geben. An elementaren Anweisungen werden benötigt (siehe obiges Programm):

- Einlesen einer Zahl in eine Speicherzelle.
- Transportiere in eine Speicherzelle die Summe des Inhalts zweier Speicherzellen.
- Transportiere in eine Speicherzelle eine Konstante.
- Führe den Vergleich "≠" mit dem Inhalt einer Speicherzelle und einer Konstante aus.

- Führe den Vergleich "≤" mit dem Inhalt zweier Speicherzellen aus.
- Ausgeben einer Speicherzelle.

Diese Anweisungen lassen sich auf zwei Grundformen zurückführen:

a) Wenn x eine Speicherzelle ist und A ein Ausdruck, dann darf man den Wert von A in die Speicherzelle x schreiben; der ursprüngliche Inhalt der Speicherzelle x wird hierdurch vernichtet. Schreibweise: x←A.
b) Wenn x und y Speicherzellen sind, dann darf man mit ihren Inhalten «x» bzw. «y» einen der Vergleiche "≤", "≥", "=", "<", ">" und "≠" durchführen, sofern diese Vergleiche überhaupt möglich sind. Schreibweise z.B.: x≤y.
c) Wenn x eine Speicherzelle ist, dann darf man einen Wert einlesen, der anschließend der neue Wert «x» dieser Speicherzelle ist. Schreibweise: get(x).
d) Den Wert einer Speicherzelle x darf man ausgeben. Schreibweise: print(x).

Beachten Sie, daß der Name x abhängig von der Anweisung verschieden zu interpretieren ist: In "x←0" ist die Speicherzelle mit dem Namen x gemeint, die einen neuen Inhalt erhält; in "x=y" ist der Inhalt von x gemeint, der nicht verändert, sondern nur zum Vergleich herangezogen wird.

Kommen wir nun zu den Anweisungskonstruktoren. In unserem Beispiel liegen drei vor: die Hintereinanderausführung von Anweisungen (*Konkatenation*, *Sequenz*), die *bedingte Anweisung* und die *bedingte Schleife*. Da man die Anweisungen hintereinander aufschreibt, kann diese normale Anordnung auch zugleich als Ausführungsreihenfolge aufgefaßt werden. Dies gilt auch für die Anweisungen, die im Inneren einer Schleife oder in den beiden Zweigen einer bedingten Anweisung stehen; nur jeweils bei der letzten dieser Anweisungen muß die normale Anordnung durchbrochen werden, und man muß an einer anderen Stelle die Bearbeitung fortsetzen. Man sagt: Man muß aus der normalen Abarbeitungsreihenfolge "herausspringen" können (sog. **Sprung**). Eine Möglichkeit besteht darin, daß man einen Pfeil ans Ende der Anweisung setzt, die als letzte in der normalen Reihenfolge abgewickelt wurde: Der Pfeil zeigt dann auf die Anweisung, die als nächste abzuarbeiten ist. In unserem Beispiel:

PRO-Formulierung	neue Formulierung
größer←0	größer←0
kleiner←0	kleiner←0
lies(n)	get(n)
lies(x)	get(x)
solange x≠0 tue	berechne, ob x≠0 ist
wenn x≤n dann	falls das nicht zutrifft, dann weiter bei
kleiner←kleiner+x	berechne, ob x≤n ist
sonst	falls das zutrifft, dann kleiner←kleiner+x und weiter bei

```
        größer←größer+x          größer←größer+x
    ende
    lies(x)                  ->  get(x)
ende
zeige(größer)                ->  print(größer)
zeige(kleiner).                  print(kleiner).
```

Hierdurch wurde der Algorithmus in erster Näherung *linearisiert*, d.h., er besitzt nun keine verschachtelte Gestalt mehr, sondern seine Anweisungen stehen in einer linearen Reihung hintereinander.

Nur die Pfeile stören noch. Um den ersten zu beseitigen, könnte man z.B. schreiben: Setze die Bearbeitung an der Stelle "print(größer)" fort. In großen Programmen muß man dann eventuell lange suchen, oder man weiß nicht, wo man weitermachen soll, falls eine solche Anweisung mehrfach im Programm vorkommt. Um diese Probleme zu vermeiden, kennzeichnet man die Stelle, an der die Berechnung fortgesetzt werden soll, mit einem eindeutigen Bezeichner (einer sog. **Marke**). Dieser Bezeichner ist frei wählbar, und er wird durch einen Doppelpunkt von der gekennzeichneten Anweisung getrennt. (Allerdings darf man als Marken keine Bezeichner verwenden, die schon anderweitig als Marke, Variable oder Prozedur benutzt werden.) Wählen wir als Bezeichner die Worte ende, next und schleife, dann geht obiges Programm über in die Formulierung:

```
            größer←0
            kleiner←0
            get(n)
            get(x)
schleife:   berechne, ob x≠0 ist
            falls das nicht zutrifft, dann weiter bei ende
            berechne, ob x≤n ist
            falls das zutrifft, dann kleiner←kleiner+x
            weiter bei next
            größer←größer+x
next:       get(x)
            weiter bei schleife
ende:       print(größer)
            print(kleiner).
```

Dieses Programmstück besteht nun nur noch aus

- Auswertungen von Bedingungen,
- Wertzuweisungen und
- bedingten oder unbedingten Sprüngen zu Marken ("weiter bei ..."),

die aneinandergereiht sind.

Eine Maschine, die diesen Algorithmus durchführen soll, muß über einen Speicher von vier Zellen verfügen, sie muß arithmetische Ausdrücke ausrechnen können, sie muß die

Inhalte von Speicherzellen miteinander vergleichen können, und sie muß sowohl eine Zelle im Speicher als auch eine markierte Stelle im Programm finden können.
Wir haben nun das ursprüngliche PRO-Programm auf ein semantisch äquivalentes, aber einfacher aufgebautes Programm zurückgeführt. Jemand, der die zuletzt angegebene Formulierung des Algorithmus bearbeiten muß, braucht nicht so viel zu wissen wie jemand, dem man die PRO-Version zur Bearbeitung überträgt. Der Algorithmus bedient sich jetzt primitiverer Elementarvorschriften und Sprachelemente. Es ist zu erwarten, daß man zu dieser primitiveren Formulierung leichter eine Maschine konstruieren kann als zu der ursprünglichen.

Skizzieren wir einmal eine solche Maschine. Sie muß über den bereits erwähnten **Speicher**, über ein **Rechenwerk** für die arithmetischen Ausdrücke und über ein **Steuerwerk** verfügen, welches das Programm in der richtigen Reihenfolge abarbeitet. Zusätzlich muß sie einen **Programmzettel** (einen Speicher für Anweisungsfolgen) besitzen. Hinzu kommt meist eine **Ein-/Ausgabeeinheit**, die den Transfer von Daten von und zu den Ein-/Ausgabegeräten steuert (z.B. Bildschirm, Tastatur). Man erhält die Struktur aus Abb. 2. Meist verwendet man für den Datenspeicher und den Programmzettel ein und denselben Speicher, d.h., man reserviert einen Bereich des Speichers für das Programm und einen anderen Bereich für die Daten. In diesem Fall erhält man die Struktur aus Abb. 3.

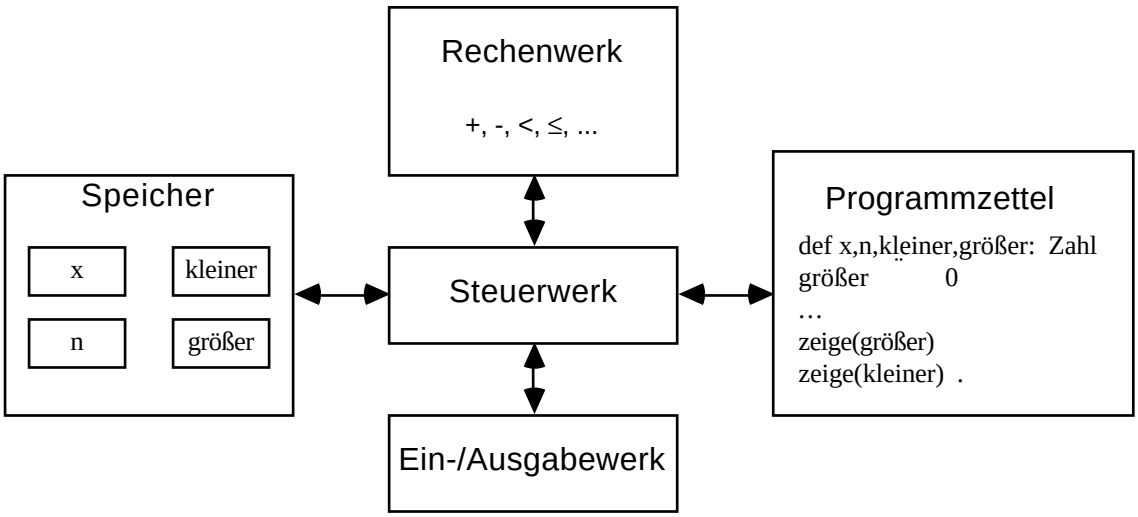

Abb. 2: Grobstruktur einer Maschine mit getrenntem Daten- und Programmspeicher

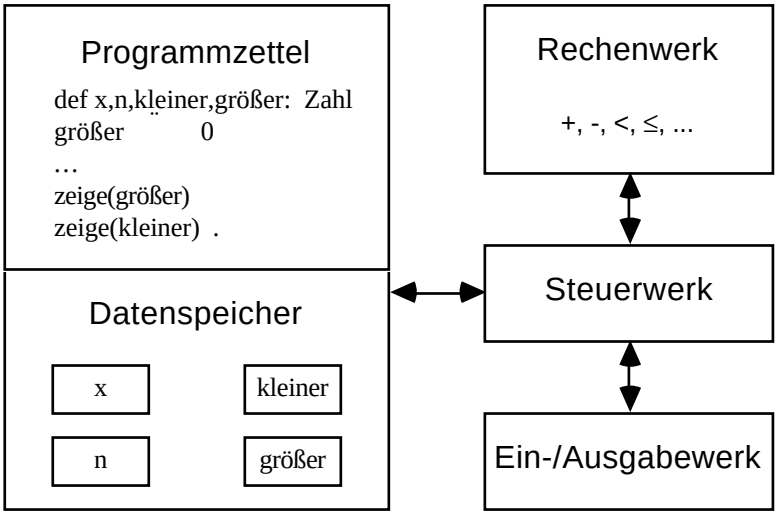

Abb. 3: Grobstruktur einer Maschine mit gemeinsamem Daten- und Programmspeicher

Man kann sich überlegen, daß man alle Sprachelemente von PRO mit den drei (evtl. mit einer Marke versehenen) primitiven Grundbefehlen "Vergleich", "Wertzuweisung" und "bedingter/unbedingter Sprung zu einer Marke im Programm" simulieren kann, wobei man als Konstruktoren nur die Konkatenation benötigt. Während die normalen Konstruktoren "Schleife" und "bedingte Anweisung" relativ leicht in die primitiveren Befehle übertragen werden können, bereiten Prozeduraufrufe Schwierigkeiten. Üblicherweise löst man dieses Problem mit Hilfe einer besonderen Datenstruktur, dem sog. **Keller**, den wir später kennenlernen.

5.3 Verfeinerung der Maschine

Als erstes vereinfachen wir das **Rechenwerk**. Bisher sind wir davon ausgegangen, daß es beliebig lange arithmetische Ausdrücke auswerten kann. Dies läßt sich aber nicht realisieren, da das Rechenwerk als physikalisches Gerät nur von fester endlicher Größe sein und nur endlich viele Operanden gleichzeitig verknüpfen kann. Folglich müssen wir versuchen, Ausdrücke in kleinere Teile zu zerlegen, die man einzeln berechnet und aus denen man dann den Wert des Gesamtausdrucks ermittelt.

Dies ist in der Tat stets möglich, wobei man sich auf Teile beschränken kann, die nur einen einzigen Operator enthalten. Betrachten wir zum Beispiel die Wertzuweisung

x←a+27∗(a+b)-c∗d.

Wir führen Hilfsbezeichner h1, h2, h3 und h4 ein und berechnen:

(*) h1←a+b;
h2←27∗h1;

h3←a+h2;
h4←c∗d;
x←h3-h4.

Offenbar berechnet diese Folge von Wertzuweisungen genau das gleiche wie die obige Wertzuweisung. Man hätte sogar einige Hilfsbezeichner einsparen können, denn auch die folgende Folge von Wertzuweisungen leistet das gewünschte:

h1←a+b;
h1←27∗h1;
h1←a+h1;
x←c∗d;
x←h1-x.

Wichtig ist: Jede Wertzuweisung der Form x←Ausdruck läßt sich durch eine Folge von Wertzuweisungen der Form u←v oder u←v op w ersetzen, wobei u ein Bezeichner, v und w Bezeichner oder Konstanten sind und op genau einen Operator (z.B. +, -, *, /) bezeichnet. Den Spezialfall der einstelligen Operatoren (z.B. unäres Minus) kann man entweder durch Zuweisungen der Form u←op w berücksichtigen, oder man kann den einstelligen Operator durch geeignete zweistellige Operatoren ersetzen; zum Beispiel kann man u←0-w für u← -w schreiben.

Es ist leicht einzusehen, daß man jeden arithmetischen Ausdruck durch solche Folgen berechnen kann, wenn man folgende Regeln beachtet:

- Jeder Operator erhält eine Priorität, und wenn rechts und links von einem Operanden je ein Operator steht, dann ist zunächst der mit der höheren Priorität zu bearbeiten. Die übliche Regel "Punktrechnung vor Strichrechnung" fällt hierunter. Man gibt den Operatoren ∗ und / eine höhere Priorität als + und -.
- Stehen rechts und links von einem Operanden zwei Operatoren gleicher Priorität, dann ist zunächst der links stehende Operator auszuwerten.
- Klammern gehen vor.

Ein Ausdruck wird nun von links nach rechts nach diesen Regeln abgearbeitet. Als Beispiel betrachte man den obigen Ausdruck

a+27*(a+b)-c∗d.

Durchläuft man diesen Ausdruck von links nach rechts, so kann man das erste Pluszeichen zunächst nicht auswerten, da rechts von dem hierbei beteiligten Operanden 27 ein Operator ∗ von höherer Priorität als + steht. ∗ läßt sich auch nicht unmittelbar auswerten, da die direkt folgende Klammer Vorrang hat. Innerhalb der Klammer steht nur ein Operator, der somit als erster auszuwerten ist. Kürzen wir das Ergebnis von a+b durch h1 ab, so haben wir aus dem ursprünglichen Ausdruck nun den Ausdruck

a+27∗h1-c∗d

erhalten. Auf diesen wenden wir die gleichen Überlegungen an und erkennen, daß das vordere ∗ nun berechnet werden muß; denn nach dem h1 folgt der Operator - von geringerer Priorität. Bezeichnen wir das Ergebnis von 27∗h1 mit h2, so erhalten wir den Ausdruck:

a+h2-c∗d.

Durchlaufen wir diesen Ausdruck von links nach rechts, so muß als nächstes a+h2 ausgerechnet werden, da rechts und links von h2 Operatoren gleicher Priorität stehen. Dieses Ergebnis sei h3. Im verbleibenden Ausdruck h3-c∗d ist erst die Multiplikation h4←c∗d und danach die Subtraktion h3-h4 auszuführen, womit die Berechnung des ursprünglichen Ausdrucks beendet ist. Faßt man die Bezeichner h1, h2, h3 und h4 als (Hilfs–)Bezeichner auf, dann kann man die gesamte Berechnung von

x←a+27∗(a+b)-c∗d

durch die obige Folge (*) von Wertzuweisungen realisieren.

Fassen wir zusammen: Jeder beliebige Ausdruck läßt sich durch eine Folge von Wertzuweisungen, in denen jeweils höchstens ein Operator vorkommt, berechnen. Mehr noch: Diese Folge spiegelt genau die korrekte Auswertung des Ausdrucks wieder, sie kann also verwendet werden, um die Bedeutung des Ausdrucks exakt zu definieren.

Folgerung: Das zu konstruierende Rechenwerk kann mit drei Speicherzellen auskommen: In einer Zelle A steht der erste Operand, in der nächsten Zelle B der zweite Operand (sofern der Operator zweistellig ist), und in der dritten Zelle E steht nach Durchführung der Operation das Ergebnis. Somit erhält das Rechenwerk die Struktur aus Abb. 4.

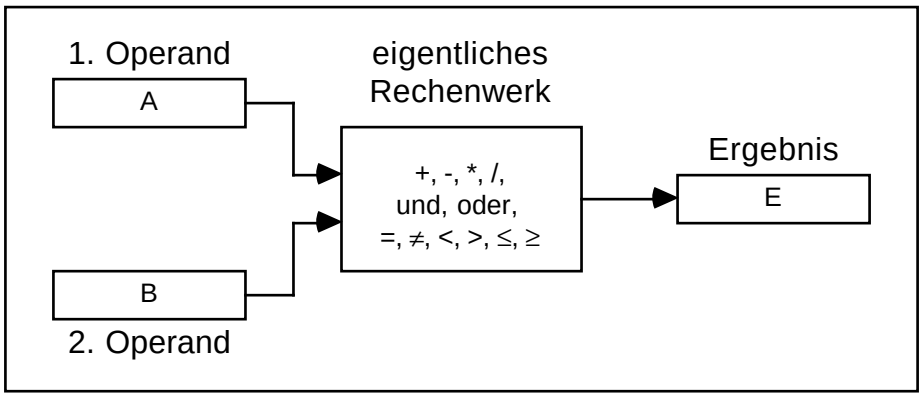

Abb. 4: Grobstruktur des Rechenwerks

Nun wenden wir uns dem **Steuerwerk** (auch **Leitwerk** genannt). Das Steuerwerk zusammen mit dem Rechenwerk (und eventuell einer Speichereinheit) bezeichnet man als **Zentraleinheit** eines Rechners oder als **CPU** (**c**entral **p**rocessing **u**nit) oder auch als (Zentral-)**Prozessor**. Das Steuerwerk muß über folgende Fähigkeiten verfügen:

- Es muß die jeweils abzuarbeitende Elementaranweisung, den sog. **Befehl**, vom Programmzettel lesen und seine Ausführung steuern können. Der gelesene Befehl wird zur Verarbeitung im sog. **Befehlsregister** des Steuerwerks gespeichert. Man spricht in diesem Zusammenhang statt von Anweisungen von Befehlen, weil es sich hierbei um Sprachelemente mit sehr begrenzter Wirkung handelt.
- Es muß den nächsten Befehl ermitteln können, entweder den nächsten Befehl in der Reihenfolge auf dem Programmzettel oder einen mit einer Marke versehenen Befehl, der angesprungen wird. Hierzu numeriert man alle Befehle des Programms mit 1,2,3 usw. durch. Das Steuerwerk besitzt eine spezielle Speicherzelle, den **Befehlszähler**, der die jeweilige Nummer des aktuell bearbeiteten Befehls enthält.
- Es muß die Operanden vom Speicher zum Rechenwerk und von dort die Ergebnisse zurück in den Speicher transportieren lassen können (**Lesen** und **Schreiben** des Speichers).
- Es muß die Speicherzellen, in denen sich die benötigten Operanden befinden, ermitteln können (**Adreßberechnung**).

Somit bekommt das Steuerwerk die Struktur aus Abb. 5

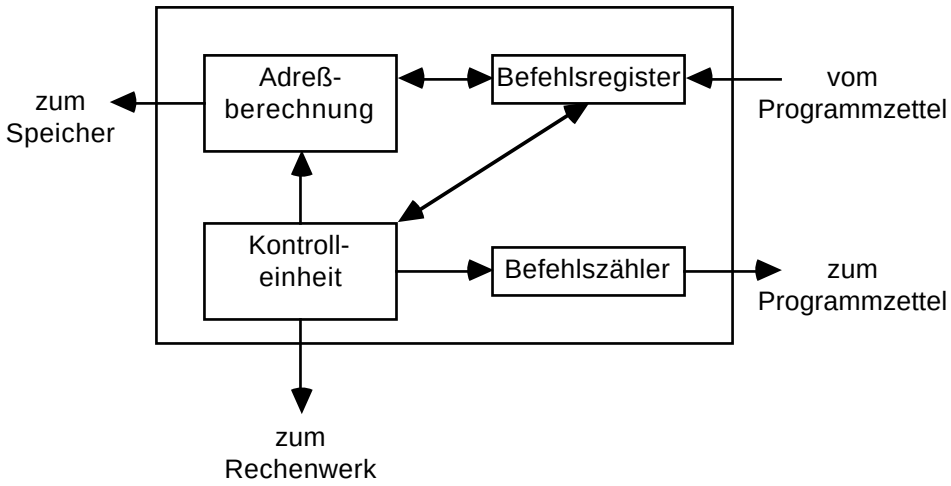

Abb. 5: Struktur des Steuerwerks

Speicherzellen, die einem speziellen Zweck dienen, bezeichnet man in der Informatik als **Register**. Da die in Abb. 5 aufgeführten Zellen jeweils für spezielle Aufgaben vor-

gesehen sind, wurde sofort die Bezeichnung Register verwendet. Man benennt dann die einzelnen Speicherzellen wie oben nach ihrem Verwendungszweck, also z.B. Befehlsregister.

Betrachten wir schließlich den **Speicher**. Er besteht aus einer Menge von Speicherzellen. Jede Speicherzelle muß man ansprechen können. Man kann dies vergleichen mit den Häusern in einer Stadt: Jedes Haus besitzt eine eigene Adresse, unter der man es erreichen kann. Daher sagt man auch beim Speicher: Jede Speicherzelle besitzt eine **Adresse**, unter der man sie ansprechen kann. In der Praxis sind diese Adressen meistens natürliche Zahlen, d.h., man numeriert die Speicherzellen mit 0,1,2, ... durch und verwendet diese Nummern als Adressen. Statt Zahlen kann man aber auch andere eindeutige Benennungen, z.B. Bezeichner, benutzen. Der Speicher erhält zwei Verbindungen zur Umwelt: Eine spezielle Speicherzelle, in die die Adresse der gewünschten Speicherzelle geschrieben werden kann, und eine Speicherzelle, in der der Wert steht, der aus der gewünschten Speicherzelle gelesen wurde oder der in die gewünschte Speicherzelle geschrieben werden soll. Da diese beiden Speicherzellen wieder einem speziellen Zweck dienen, bezeichnet man sie den Konventionen entsprechend als Register und nennt sie **Speicheradreßregister** (abgekürzt SAR) und **Speicherpufferregister** (abgekürzt SPR).
Weiterhin muß man den Speicher von außen "anstoßen" (ihn zu einer Tätigkeit anregen) können, wobei die Signale "lies" und "schreib" bereits ausreichen. "schreib" soll bewirken, daß der Speicher den Wert, der im SPR steht, in diejenige Speicherzelle überträgt, deren Adresse in SAR angegeben ist; und "lies" bewirkt, daß der Inhalt der Speicherzelle, deren Adresse im SAR angegeben ist, in das SPR kopiert wird (Abb. 6).

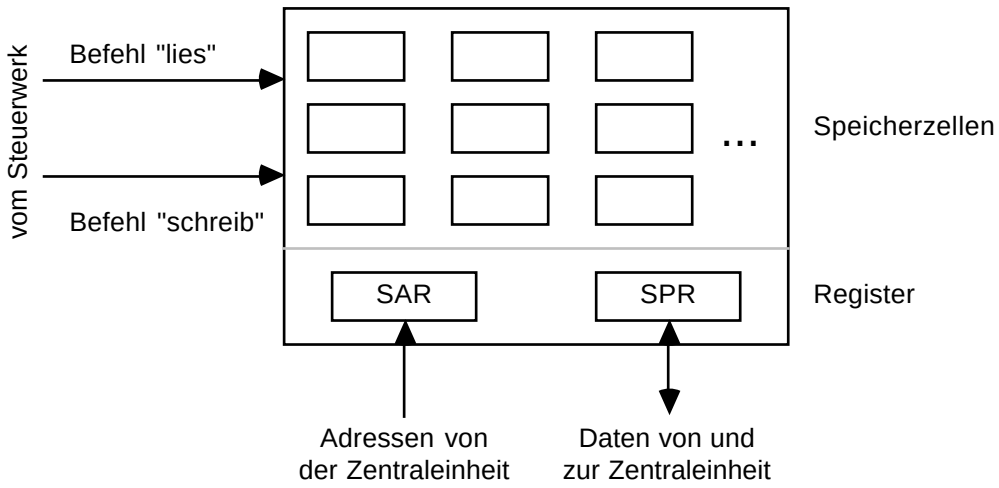

Abb. 6: Struktur des Speichers

Diese Struktur hat nun Einfluß auf die Wertzuweisungen. Die Wertzuweisung x←y muß bezogen auf die obige Struktur des Speichers jetzt durch folgende Folge von Einzelschritten ausgewertet werden:

- Berechne die Adresse der Speicherzelle, die zur Variablen y gehört.
- Schreibe diese Adresse ins Register SAR.
- Stoße den Speicher durch "lies" an (anschließend steht der Wert von y im SPR).
- Berechne die Adresse der Speicherzelle, die zu x gehört.
- Schreibe diese Adresse ins Register SAR.
- Stoße den Speicher durch "schreib" an (hierdurch wird der Inhalt von SPR in die zu x gehörende Speicherzelle geschrieben).

5.4 Eine einfache maschinennahe Sprache

Wir konnten die komplexen Anweisungen einer höheren Programmiersprache schrittweise durch Folgen einfacher Befehle simulieren. Diese Befehle bestanden aus sehr einfachen Wertzuweisungen und aus bedingten und unbedingten Sprüngen im Programm. Die Befehle können mit Marken versehen werden. Die einfachen Wertzuweisungen kann man nun noch weiter aufspalten, indem man sich auf Transportbefehle zwischen Registern, auf die Befehle "lies" und "schreib" und auf das Anstoßen des Rechenwerks beschränkt. Alle jetzt noch zulässigen Befehle werden im folgenden mit ihrer umgangssprachlichen Bedeutung aufgelistet, hierbei sind die Transportbefehle, die den Zuweisungen vergleichbar sind, durch einen Pfeil "→" dargestellt. Beachten Sie, daß bei Transportbefehlen der Form R→Q vom Register R in das Register Q der Wert im Register R unverändert bleibt. Statt "Transportieren" könnte man daher besser von "Kopieren" sprechen, jedoch hat sich diese Bezeichnung nicht eingebürgert:

read	Transportiere den Inhalt der Speicherzelle, deren Adresse in SAR angegeben ist, nach SPR
write	Transportiere den Inhalt von SPR in die Speicherzelle, deren Adresse in SAR angegeben ist
SPR→A	Transportiere den Inhalt des SPR in das Register A des Rechenwerks
SPR→B	Transportiere den Inhalt des SPR in das Register B des Rechenwerks
A→SAR	Transportiere den Inhalt von Register A des Rechenwerks nach SAR
B→SAR	Transportiere den Inhalt von Register B des Rechenwerks nach SAR
E→SAR	Transportiere den Inhalt von Register E des Rechenwerks nach SAR
A→SPR	Transportiere den Inhalt von Register A des Rechenwerks nach SPR
B→SPR	Transportiere den Inhalt von Register Bdes Rechenwerks nach SPR
E→SPR	Transportiere den Inhalt von Register E des Rechenwerks nach SPR
A→B	Transportiere den Inhalt von Register A des Rechenwerks in Register B
B→A	Transportiere den Inhalt von Register B des Rechenwerks in Register A
E→A	Transportiere den Inhalt von Register E des Rechenwerks in Register A
n→A	Transportiere die ganzzahlige Konstante n in Register A
n→B	Transportiere die ganzzahlige Konstante n in das Register B

add	Addiere A und B und schreibe das Ergebnis nach E, die Inhalte von A und B bleiben erhalten
sub	Subtrahiere B von A und schreibe das Ergebnis nach E, die Inhalte von A und B bleiben erhalten
mult	Multipliziere A und B und schreibe das Ergebnis nach E, die Inhalte von A und B bleiben erhalten
div	Dividiere A durch B und schreibe das Ergebnis nach E, die Inhalte von A und B bleiben erhalten
less	Ermittle «A»<«B» und schreibe das Ergebnis (wahr oder falsch) nach E
greater	Ermittle «A»>«B» und schreibe das Ergebnis (wahr oder falsch) nach E
lesseq	Ermittle «A»≤«B» und schreibe das Ergebnis (wahr oder falsch) nach E
greatereq	Ermittle «A»≥«B» und schreibe das Ergebnis (wahr oder falsch) nach E
equal	Ermittle «A»=«B» und schreibe das Ergebnis (wahr oder falsch) nach E
unequal	Ermittle «A»≠«B» und schreibe das Ergebnis (wahr oder falsch) nach E
goto M	Setze die Verarbeitung bei der mit M markierten Anweisung fort
iftruegoto M	Setze die Verarbeitung bei dem mit M markierten Befehl fort, sofern in E der Wert wahr steht, anderenfalls gehe zum nächsten Befehl über
get	Eingabe eines Datums in das Register A
print	Drucke den Inhalt des Registers A
print "..."	Drucke den Text zwischen den Anführungszeichen
stop	Ende der Berechnung.

Eine maschinennahe Programmiersprache, die mindestens eine vergleichbare Ausdruckskraft besitzt wie die obige, bezeichnet man als **Assemblersprache**. Wir bezeichnen die obige Assemblersprache ab sofort mit **ASS**. Ein **Assemblerprogramm** ist eine Folge solcher Befehle, die jeweils evtl. mit einer Marke versehen sind. Weiterhin dürfen keine Bereiche überschritten werden, d.h., man darf nicht auf Speicherzellen zugreifen, die es nicht gibt, oder zu Marken springen, die nicht im Programm enthalten sind. Unzulässig ist ebenfalls die Anwendung arithmetischer Operationen oder der Vergleiche <, >, ≤ und ≥, wenn die beteiligten Operanden keine Zahlen sind.
Aus Resultaten der Theoretischen Informatik geht hervor: Wenn der Speicher unbegrenzt wäre, dann gibt es zu jedem PRO-Programm ein semantisch äquivalentes ASS-Programm.

Mit den Befehlen von ASS lassen sich die Grundprinzipien des maschinennahen Programmierens erläutern.
Prinzipiell sollte ein Algorithmus zunächst in einer höheren Sprache (etwa PRO) ausformuliert werden. Anschließend erstellt man einen *Speicherbelegungsplan*, d.h., man legt genau fest, welche Bezeichner mit welchen Speicherzellen identifiziert werden. Danach schreibt man den Algorithmus in die maschinennahe Sprache um. Damit man spätere Korrekturen vornehmen oder das Programm anpassen kann, sind der Speicherbelegungsplan und der ursprüngliche Algorithmus zusammen mit dem Assemblerprogramm zu dokumentieren; insbesondere soll das maschinennahe Programm klare Bezüge (in Form von Kommentaren) zum ursprünglichen Algorithmus enthalten.

Anschließend optimiert man in der Regel das Assemblerprogramm, indem man z.B. überflüssige Zwischenspeicherungen entfernt oder einzelne Berechnungsfolgen durch äquivalente Programmstücke ersetzt. Das Ergebnis ist i.a. ein ziemlich unverständliches Programm, das ohne eine gute Dokumentation nicht mehr veränderbar ist.

Beispiel: Wir wollen das Problem "Einlesen einer Zahl n und einer Zahlenfolge und Aufsummieren der Zahlen >n bzw. ≤n" in ASS formulieren. Hierzu verwenden wir das vereinfachte Programm aus Abschnitt 5.2. Um das Programm in ein Assemblerprogramm umzusetzen, müssen wir zunächst für die einzelnen Variablen Speicherzellen vergeben. Wir einigen uns auf folgenden Speicherbelegungsplan:

Variable	Adresse der Speicherzelle
größer	10
kleiner	11
n	12
x	13

Nun können wir das PRO-Programm umsetzen; rechts herausgerückt steht die jeweilige Anweisung des vereinfachten Programms, die bis zu dieser Stelle durch die maschinennahen Befehle realisiert wurde. Man beachte, daß sich in den Befehlen zwischen den Marken schleife und next der Wert von x ununterbrochen im Register A befindet:

```
            10→A
            A→SAR
            0→A
            A→SPR
            write          größer←0
            11→A
            A→SAR
            write          kleiner←0
            12→A
            A→SAR
            get
            A→SPR
            write          get(n)
            13→A
            A→SAR
            get
            A→SPR
            write          get(x)
schleife:   0→B
            equal          x=0?
```

```
            iftruegoto ende
            12→B
            B→SAR
            read
            SPR→B
            greater          x>n?
            iftruegoto       sonst
            11→B
            B→SAR
            read
            SPR→B
            add
            E→SPR
            write            kleiner←kleiner+x
            goto next
sonst:      10→B
            B→SAR
            read
            SPR→B
            add
            E→SPR
            write            größer←größer+x
next:       13→A
            A→SAR
            get
            A→SPR
            write            get(x)
            goto schleife
ende:       10→A
            A→SAR
            read
            SPR→A
            print            print(größer)
            11→A
            A→SAR
            read
            SPR→A
            print            print(kleiner)
            stop.
```

5.5 Das Ebenenmodell

In den vorigen Abschnitten dieses Kapitels haben wir einen Rahmen entworfen, um die Elemente der Programmiersprache PRO durch eine Programmiersprache einfacherer

Bauart aber gleicher Leistungsfähigkeit (die *Assemblersprache* ASS) zu beschreiben. Verfügen wir über einen Computer C, der ASS-Programme verarbeiten kann, so besitzen wir damit auch gleichzeitig eine Maschine für PRO. Jedes PRO-Programm muß nur gemäß der skizzierten Vorschrift in ein semantisch äquivalentes Programm der Sprache ASS transformiert werden. Man kann sich überlegen, daß jede einzelne Transformation – das Entfernen von Konstruktoren, das Vereinfachen arithmetischer Ausdrücke usw. – selbst wieder durch einen Algorithmus beschrieben also maschinell durchgeführt werden kann. Wandelt man diesen Algorithmus schließlich noch in ein ASS-Programm um, so kann man ihn auf C ausführen. Damit besitzen wir ein Programm, welches beliebige PRO-Programme einliest und semantisch äquivalente ASS-Programme ausgibt. Solch ein Programm bezeichnet man als *Übersetzer*.

Formalisierung: Für ein Programm P bezeichne wie üblich f_P die Funktion, die das Programm berechnet (f_P ist also die Semantik von P). Ein Übersetzer U von der Sprache A in die Sprache B berechnet dann eine Funktion (Abb. 7)

f_U: {P | P ist Programm der Sprache A}$\rightarrow$
{P | P ist Programm der Sprache B} mit
$f_U(P)=P'$, so daß gilt $f_P=f_{P'}$.

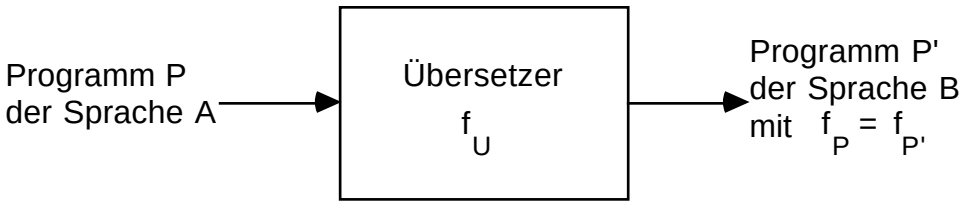

Abb. 7: Übersetzer von Sprache A nach Sprache B

Definition A:
Ein **Übersetzer** (oder **Compiler**) ist ein Programm, das Programme einer Programmiersprache in semantisch äquivalente Programme einer anderen Programmiersprache umwandelt.

Eine andere Methode, PRO-Programme auf C auszuführen, besteht darin, ein ASS-Programm zu schreiben, das ein PRO-Programm P Anweisung für Anweisung sowie zugehörige Eingabedaten x einliest, *interpretiert* und sofort durch eine geeignete Befehlsfolge in ASS ausführt. Die Ausgabe, die das ASS-Programm liefert, entspricht der Ausgabe, die P mit der Eingabe x liefern würde. Solch ein Programm bezeichnet man als *Interpreter*.

Formalisierung: Ein Interpreter J für die Sprache A berechnet eine Funktion (Abb. 8)

f_J: {P | P ist Programm der Sprache A}×{x | x ist mögliche Eingabe}→
{x | x ist mögliche Ausgabe} mit

$f_J(P,x)=f_P(x)=y$.

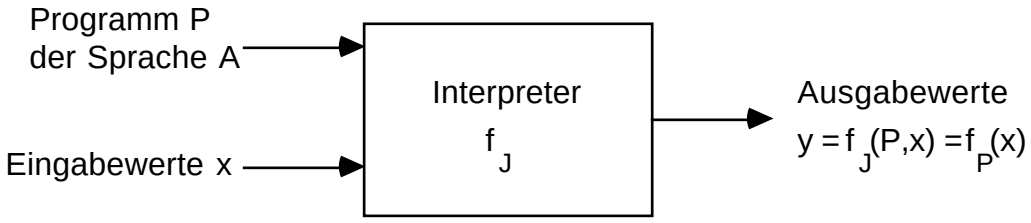

Abb. 8: Interpreter für die Sprache A

Definition B:

Ein Programm, das Programme einer anderen Programmiersprache einliest und sofort schrittweise ausführt, bezeichnet man als **Interpreter**.

Beispiel: Liest der Interpreter für PRO eine PRO-Anweisung der Form

x←x+y

ein, so kann er diese Anweisung sofort durch Ausführung der folgenden ASS-Befehlsfolge simulieren:

```
50→A
A→SAR
read
SPR→A
65→B
B→SAR
read
SPR→B
add
50→A
A→SAR
E→SPR
write.
```

Die Speicherzellen mit den Adressen 50 für x und 65 für y hat der Interpreter bereits reserviert, als er die PRO-Deklaration

def x,y: Zahl

gelesen hat.

Die gegebene Maschine C für ASS verhält sich also vermöge eines geeigneten Übersetzers oder Interpreters wie eine Maschine für PRO. Man sagt: Übersetzer bzw. Interpreter machen aus der *realen* Maschine für die Sprache ASS eine *virtuelle* (d.h. gedachte) Maschine für PRO.

Ein Ziel dieses Abschnitts war es, PRO-Programme auf einem Computer ausführen zu lassen. Die direkte Realisierung aller PRO-Elemente durch logische Schaltungen erschien wegen des großen Unterschieds der beiden Abstraktionsniveaus jedoch außerordentlich kompliziert. Wir haben daher eine Reihe von Zwischenniveaus (PRO→Sprache ohne Konstruktoren→Sprache mit vereinfachten arithmetischen Ausdrücken→ASS) definiert, um den Abstand zwischen PRO und logischen Schaltungen schrittweise zu verringern. In der Praxis haben sich sechs Abstraktionsebenen herausgebildet, die man alle zusammen als **Ebenenmodell der Rechnerarchitektur** bezeichnet (Abb. 9). Jede Ebene besitzt eine spezielle Form von Daten und zugehörigen Operationen, die nach oben hin immer mächtiger wird. Die Objekte und Strukturen einer Ebene beschreibt man stets mit den Elementen der unmittelbar darunterliegenden Ebene. Durch Übersetzer oder Interpreter werden Programme einer Ebene auf semantisch äquivalente Programme der unmittelbar darunterliegenden Ebene abgebildet.

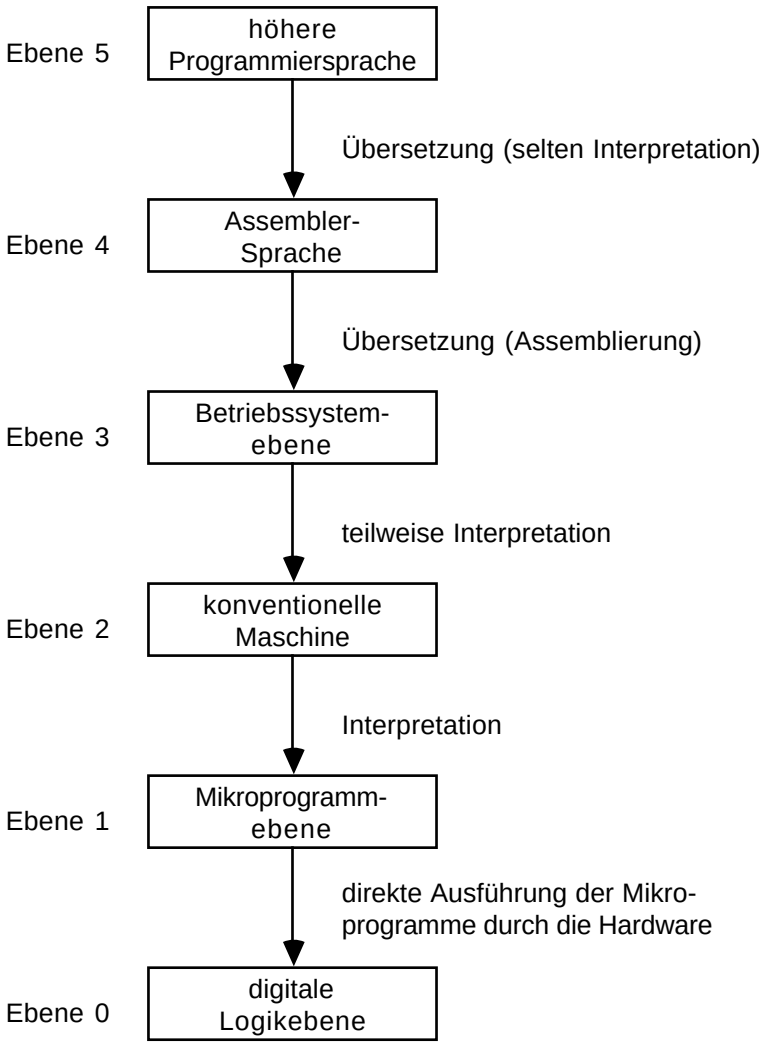

Abb. 9: Ebenenmodell

Zu den einzelnen Ebenen: Die operationalen Grundbausteine der *digitalen Logikebene* sind die sog. Gatter. Ein **Gatter** ist eine Funktionseinheit mit n Eingängen und m Ausgängen (Abb. 10). Als Datenobjekte sind nur zwei unterschiedliche Spannungswerte zugelassen. Den einen Spannungswert bezeichnet man mit 0, den anderen mit 1. **Digital** (dt. ziffernmäßig) bezeichnet man die Bausteine deshalb, weil alle Daten durch diskrete (d.h. 0 und 1), also nicht stetig veränderbare Werte dargestellt werden.

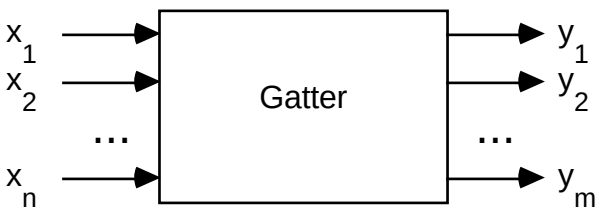

Abb. 10: Gatter

Ein Gatter berechnet eine Funktion

$$f: \{0,1\}^n \rightarrow \{0,1\}^m.$$

Beispiele für Gatter sind das UND-Gatter zur Realisierung der und-Funktion (Abb. 11) und das ODER-Gatter für die oder-Funktion. Das UND-Gatter realisiert z.B. die Funktion

$$f_{und}: \{0,1\}^2 \rightarrow \{0,1\} \text{ mit}$$

$$f_{und}(x,y) = \begin{cases} 0, \text{ falls } x=0 \text{ oder } y=0, \\ 1, \text{ falls } x=1 \textit{ und } y=1. \end{cases}$$

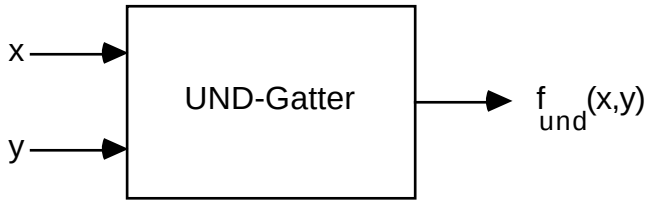

Abb. 11: UND-Gatter

Durch Zusammenschalten von Gattern, d.h. Verbinden der Ausgänge von Gattern mit den Eingängen anderer Gatter, kann man kompliziertere Funktionen aufbauen. Jedes Gatter besteht seinerseits aus noch einfacheren Bauelementen (z.B. Transistoren, Dioden und Widerstände). Unterhalb der Ebene 0 kann man sich also noch weitere Ebenen vorstellen, die jedoch nicht mehr zur Informatik sondern zur Elektrotechnik oder Physik gezählt werden. Daher wollen wir uns mit diesen Ebenen hier nicht beschäftigen. Aus Informatiksicht bilden Gatter also die einfachsten nicht weiter zerlegbaren Grundelemente eines Computers.

Während auf der Ebene der digitalen Logik noch keine informatikbezogenen Konzepte verwendet werden, besitzt die Ebene 1 bereits ein wesentliches informatiktypisches Merkmal, das (Mikro-)*Programm*.

Ein **Mikroprogramm** hat den Aufbau eines gewöhnlichen Programms, d.h., es ist eine Folge von einzelnen (Mikro-)Befehlen, die schrittweise abgearbeitet werden. Die Befehle sind noch sehr primitiv (daher Mikro). Typische Operationen auf dieser Ebene sind:

- Transportiere eine 0-1-Information von einer Stelle des Computers zu einer anderen.

- Prüfe, ob eine bestimmte Gatterleitung den Wert 1 besitzt, und führe ggf. eine Operation aus.
- Lege auf eine bestimmte Gatterleitung den Wert 0.

Auf der *Mikroprogrammebene* stehen üblicherweise etwa 20 verschiedene Mikrobefehle zur Verfügung. Mikroprogramme werden direkt durch die logischen Gatter ausgeführt. Jedes Programm jeder Programmiersprache wird letztendlich auf eine Folge solcher Mikrobefehle zurückgeführt.

Auf der Ebene 2 erhalten die recht unzusammenhängenden Objekte der Ebene 1 (Gatter und Gatterleitungen) eine Struktur. Man gruppiert z.B. bestimmte Gatter zu Registern und gibt ihnen Namen. Ebenso faßt man gewisse Folgen von Mikrobefehlen zu einem neuen Befehl zusammen. Als Ergebnis erhält man eine abstraktere Programmiersprache, die sog. **Maschinensprache**. Auch in dieser Sprache bestehen alle Elemente noch aus Folgen von Gruppen von 0 und 1. Ein Interpreter transformiert Programme in Maschinensprache in semantisch äquivalente Mikroprogramme.
Die *konventionelle Maschinenebene* ist üblicherweise die unterste Ebene, auf die man sich als "gewöhnlicher" (System-)Programmierer begibt. Die Mikroprogrammebene ist im allgemeinen dem Hersteller der Rechenanlage vorbehalten.

Auf der *Betriebssystemebene* verfügt man neben den meisten Sprachelementen der Ebene 2 zusätzlich noch über Anweisungen, die den "Komfort" der Rechenanlage erhöhen. Typisch sind auf dieser Ebene neue Anweisungen zur Speicherverwaltung, zur Verarbeitung mehrerer Programme gleichzeitig, zur Bearbeitung von Unterbrechungen, zur Kommunikation mit anderen Rechnern und zur Unterstützung der Ein- und Ausgabe. Während die bereits auf Ebene 2 vorhandenen Anweisungen der Ebene 3 direkt durch ein Mikroprogramm ausgeführt werden, werden die neu hinzugekommenen Anweisungen durch Operationen der Ebene 2 interpretiert. Auch auf der Betriebssystemebene bestehen alle Befehle und Daten noch aus 0-1-Folgen.

Von der Ebene 4 an werden die Programmiersprachen allmählich *benutzerorientiert*. Anweisungen und Datenobjekte erhalten nun symbolische Bezeichnungen. Statt eines Befehls

01100111,

der bedeuten mag "Transportiere den Inhalt des Registers 1 in das Register 3", schreiben wir auf der Ebene 4 z.B.

MOVE R1,R3 oder R3←R1.

Die Sprache, die man durch diesen Übergang erhält, bezeichnet man als **Assemblersprache** oder kurz **Assembler**. Programme in Assembler werden durch einen Übersetzer, den sog. **Assemblierer**, in semantisch äquivalente Programme der Sprache von Ebene 3 übersetzt. Der Assemblierer ist ein Programm geschrieben in der Sprache der Ebene 3. Die fiktive Assemblersprache ASS ist dieser Ebene zuzuordnen.
Zu Beginn der Entwicklung von Computern war die Programmierung in Assembler noch recht bedeutsam und weit verbreitet. Seit längerem programmiert man jedoch zunehmend in sog. höheren Programmiersprachen.

Die fiktive Sprache PRO, die der Ebene 5 zuzurechnen ist, kennen Sie bereits aus Kapitel 4. Sprachen der Ebene 5 bezeichnet man als **höhere** oder **problemorientierte** Programmiersprachen. Sie werden üblicherweise mit Übersetzern, Interpretern oder einer Mischung aus beidem in Programme der Ebene 3 oder 4 übertragen. Zur Zeit gibt es über tausend verschiedene höhere Programmiersprachen, von denen ADA, ALGOL, BASIC, C, COBOL, FORTRAN, LISP, ML, MODULA-2, PASCAL, PROLOG und PL/I die bekanntesten sind. Sprachen der Ebenen 0 bis 4 nennt man **niedere** oder **maschinenorientierte Programmiersprachen**.

Ebenso wie unterhalb der Ebene 0 schließen sich auch oberhalb der Ebene 5 weitere Ebenen an, z.B. die Menge der Anwendungsprogramme als Ebene 6.

Bei den üblichen Rechnern sind die Ebene 0 durch Hardware, die Ebenen 2 bis 5 durch Software realisiert. Einen Sonderstatus zwischen Soft- und Hardware nimmt häufig die Ebene 1 ein: Die Software der Mikroprogrammebene wird meist vom Hersteller während der Fertigung in die elektronischen Bausteine eingebettet. Man bezeichnet diese Programme daher als **Firmware**, da sie zwar prinzipiell zur Software gehören, jedoch über einen längeren Zeitraum (oder immer) fest bleiben und nur mit Hilfsmitteln verändert werden können (also quasi "hard" sind).
Wohl gemerkt, diese Einteilung in Hard-, Firm- und Softwareebenen ist nicht zwingend; sie gilt nur für die üblichen Rechenanlagen. Jede Anweisung, die durch Software ausgeführt wird, kann man auch durch geeignete Zusammenschaltung von elektronischen Bauteilen in die Hardware verlagern. Umgekehrt kann auch jede Operation, die durch Hardware realisiert ist, durch geeignete Software simuliert werden. Welche Operationen man bei der Konstruktion eines Computers der Hardware, welche der Software bzw. Firmware überläßt, hängt von Randbedingungen ab, etwa von der gewünschten Verarbeitungsgeschwindigkeit, den Kosten, der Erstellungszeit, dem Grad der Parallelverarbeitung, der Qualität der Bauelemente usw.

Es gilt also:

Hardware und Software sind logisch äquivalent.

Fassen wir zusammen: Computersysteme werden von verschiedenen Ebenen aus betrachtet. Jede Ebene beschreibt das System auf einem anderen Abstraktionsniveau und unterscheidet sich von den anderen Niveaus durch die vorhandenen Grundoperationen und -objekte. Die Grundelemente jeder Ebene werden nur durch Kombination der Grundelemente der unmittelbar darunterliegenden Ebene gebildet.