Algorithmen und Datenstrukturen – eine Einführung

Bernd Ulmann

4. Dezember 2000

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das vorliegende Skript wurde für die einführende Veranstaltung „Algorithmen und Datenstrukturen“ der VWA, Wintersemester 2000/2001 erstellt. Es umfaßt in etwa die erste Hälfte der Veranstaltung sowie einige darüber hinaus gehende Punkte.

Kapitel 2

Grundlagen

Die Meinungen, was unter den Begriff des „Programmierens“ fällt, gehen im allgemeinen nicht weit auseinander – im Prinzip verbirgt sich hinter der Erstellung eines Programms, welches das Ergebnis des Programmierens darstellt, die Erzeugung einer Abfolge von Anweisungen an einen Computer, mit deren Hilfe eine Problemstellung gelöst wird. Zu klären sind nun eine ganze Reihe von Begriffen: Was für ein Computer? Welche Eigenschaften muß eine solche Abfolge von Anweisungen aufweisen, um ein Programm darzustellen? Sowie vieles mehr.

Um zunächst die Frage nach dem Computer zu klären, die auf den ersten Blick etwas anachronistisch wirkt, hat sich „der Computer“ doch – nicht zuletzt Dank der zunehmenden Popularisierung des Internet – mittlerweile bis in jeden nahezu letzten Winkel verbreitet. Das, was heutzutage gemeinhin als „Computer“ bezeichnet wird, ist ein Digitalcomputer oder Digitalrechner. Dieser Begriff ist eigentlich notwendig, um ihn von einer ganz anderen Klasse von Maschinen klar abzugrenzen, die allerdings in den letzten Jahrzehnten so rapide an Bedeutung verloren haben, daß eine solche Unterscheidung im allgemeinen als nicht mehr notwendig erachtet wird: Die Klasse der Analogrechner.

Was unterscheidet einen Digitalrechner von einem Analogrechner? Ein Analogrechner arbeitet nicht mit „diskreten“ Rechengrößen, um ein Problem zu lösen (solche diskreten Größen stellen beispielsweise die Elemente von Teilmengen der natürlichen Zahlen dar, die voneinander klar abgegrenzt sind – zwischen je zwei aufeinanderfolgenden natürlichen Zahlen findet keine weitere, dritte natürliche Zahl Platz), sondern verwendet analoge Größen, wie beispielsweise Spannungen oder Drehwinkel, etc. Analoge Größen lassen sich mit Hilfe natürlicher Zahlen $n \in \mathrm{N}$ oder auch rationaler Zahlen $q \in \mathrm{Q}$ nicht darstellen. Zwischen je zwei Zahlen aus diesen Mengen lassen sich beliebig viele weitere Zahlen einfügen – eine Eigenschaft, die den reellen Zahlen $r \in \mathrm{R}$ nicht eigen ist. Analogrechner arbeiten mit Rechengrößen, die in einer Teilmenge von R liegen, während Digitalrechner mit diskreten Größen auskommen, die entweder aus einer Teilmenge von N (bei einigen Maschinen sogar Teilmengen von Q) stammen beziehungsweise nur die beiden Werte 0 beziehungsweise 1 umfassen.

Da Analogrechner vollkommen anders „programmiert“ werden, als dies bei Digitalrechnern üblich ist (beispielsweise werden „Programme“ für elektronische Analogrechner im allgemeinen mit Hilfe von Kabeln auf einem Steckbrett verdrahtet, um eine Rechenschaltung zu erzeugen, die das Problem auf den Rechner abbildet und seine (näherungsweise) Lösung ermöglicht), soll auf sie im weiteren nicht eingegangen werden, vielmehr sei an dieser Stelle auf [7] und [18] verwiesen, die eine hervorragende Einführung in dieses geschichtlich ausgesprochen interessante Gebiet geben[1].

[1] Eine der einfachsten Formen eines Analogrechners findet sich in Gestalt von Rechenschiebern, mit deren Hilfe analoge Größen, nämlich Strecken, addiert werden können. Über den Umweg über Logarithmen können somit auch in eleganter Art und Weise Multiplikationen und Divisionen vollzogen werden.

Nachdem die Menge der zu betrachtenden Rechner auf die der Digitalrechner eingeschränkt wurde, bleibt die Frage, inwieweit sich ein solcher näher spezifizieren läßt (ungeachtet des Problems, daß sich hiermit ganze Bücher füllen ließen, soll hier eine kurze Darstellung versucht werden, soweit sie für das folgende von Interesse und Nutzen ist).

Wie bereits erwähnt wurde, arbeitet ein Digitalrechner mit diskreten Rechengrößen – im allgemeinen nur mit den beiden grundlegenden Werten oder Ziffern 0 und 1, d.h. mit Zahlen zur Basis 2 statt zur Basis 10, wie sie dem gewohnten Dezimalsystem eigen ist[2], so daß zunächst die gebräuchlichen Zahlensysteme eingehender betrachtet werden sollen.

2.1 Zahlendarstellung

Da die Lösung von Problemen – gleich ob mit Hilfe eines Programmes auf einem Computer oder gesundem Menschenverstand, mit Hilfe von Erfahrung oder Experiment – im allgemeinen die Verwendung von Zahlen als Grundlage voraussetzt, stellt der vorliegende Abschnitt einige Fakten und Verfahren zur Darstellung und Verarbeitung von Zahlen vor.

Entsprechend der täglichen Gewohnheit werden im weiteren Verlauf nur Zahlen betrachtet, die in Form eines sogenannten „Stellenwertsystems“ notiert sind. Hierunter fallen alle Zahlensysteme, in denen eine Ziffer in Abhängigkeit ihrer Position einen unterschiedlichen Wert besitzt, wie es beispielsweise bei der Dezimalzahl 111 der Fall ist, in der die rechte 1 den Wert Eins repräsentiert, während die mittlere für den Wert Zehn und die linke Ziffer für Einhundert stehen, obwohl in allen drei Fällen ein und dieselbe Ziffer, jedoch an unterschiedlichen Positionen verwendet wird.

Ein Beispiel für ein Zahlensystem, das kein Stellenwertsystem darstellt, sind römische Zahlen – steht beispielsweise V für den Wert Fünf, entspricht VIII dem Zahlenwert Acht, die aus der Summe Fünf plus Drei gebildet wird. Da derlei Zahlensysteme eine große Reihe von Nachteilen aufweisen, von denen der schwerwiegendste sicherlich ist, daß sich nicht ohne weiteres einfache Rechenvorschriften angeben lassen, mit deren Hilfe beispielsweise Additionen, Subtraktionen oder gar Multiplikationen und Divisionen durchgeführt werden können, wird auf sie im folgenden nicht nur nicht weiter eingegangen werden, vielmehr sind sie im allgemeinen nicht mehr im Gebrauch. (Einen hervorragenden Überblick über Zahlsysteme im allgemeinen gibt beispielsweise [12].)

Wie bereits kurz erwähnt, ist das Kennzeichen eines Stellenwertsystems die Tatsache, daß ein und dasselbe Symbol, im speziellen also eine Ziffer, unterschiedliche Werte repräsentiert, die durch ihre Position innerhalb der Zahl bestimmt werden[3]. Die Zahl 1234 setzt sich also folgendermaßen zusammen:

$$\begin{aligned} 1234 &= 1 \cdot 1000 + 2 \cdot 100 + 3 \cdot 10 + 1 \cdot 1 \\ &= 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 \end{aligned}$$

Jede einzelne Ziffer wird also mit der Wertigkeit ihrer Position multipliziert, bevor die Summe über alle diese Produkte gebildet wird. Hierzu sind nun zwei Dinge von essentieller Bedeutung: Zum einen muß ein Verfahren existieren, eine Stelle zu kennzeichnen, die keinen Wert repräsentiert, um beispielsweise die Zahl Einhundertunddrei notieren zu können, zum anderen tritt in obiger Darstellung der Begriff der „Basis“ eines Zahlensystems in Erscheinung.

[2] Es soll nicht verschwiegen werden, daß eine Reihe – durchaus einflußreicher – Computer-Architekturen im Dezimalsystem arbeiteten. Da dies heutzutage allerdings ausgesprochen selten wurde, wird auf diese Maschinenklasse im folgenden nicht weiter eingegangen.

[3] Wichtig ist hierbei, daß einzig und allein die Position der Ziffern innerhalb der Zahl für die Kennzeichnung ihres Wertes dient, nicht jedoch das Umfeld der Ziffer, wie es bei römischen Zahlen beispielsweise der Fall ist.

Zunächst zur Darstellung einer quasi leeren Stelle (ohne eine solche Möglichkeit ist ein Stellenwertsystem nicht denkbar) – hierfür hat sich das Zeichen „0“ eingebürgert, das im allgemeinen als „Null“ bezeichnet wird und eine interessante Entstehungsgeschichte aufweist. Erst durch die Einführung eines solchen solchen Spezialsymbols, das als Ziffer gerade die Abwesenheit einer Ziffer ein einer Stelle kennzeichnet, erlaubte die Entwicklung eines Stellenwertsystems. Der geschichtliche Hintergrund der Null an sich findet sich beispielsweise in [13]. Mit Hilfe dieser Ziffer läßt sich nun eine Zahl, wie die bereits erwähnte Einhundertunddrei folgendermaßen schreiben:

$$103 = 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0.$$

An den vorangegangenen Beispielen fällt die Häufung von Zehnerpotenzen auf, die ihren Grund darin hat, daß in einem Dezimalsystem zehn verschiedene Ziffern zur Verfügung stehen, so daß jede Stelle einer Zahl zehn verschiedene Werte annehmen kann, bevor ein Übertrag auf die nachfolgende Stelle vorgenommen werden muß. Wird mit der Zahl 0 begonnen, ergibt sich durch Addition von 1 zunächst eine 1, der nach weiterer Addition einer 1 die 2 folgt, usf., bis nach Auftreten der 9 keine weitere Ziffer zur Verfügung steht, um den elften Zahlenwert auszudrücken[4], so daß von einstelligen Zahlen nun zu einer zweitstelligen Zahl übergegangen werden muß, um das erstmalige Überschreiten des Wertebereichs der zehn Ziffern anzuzeigen. Aus 9 und Addition einer 1 ergibt sich also die Zahl 10. Die Ziffer 1 an der linken Stelle gibt an, daß die rechte Stelle einen Überlauf erzeugt hat, entsprechend ergibt sich aus 19 und Addition einer 1 die Zahl 20, usw.

Da jede einzelne Stelle einer Zahl im Dezimalsystem zehn verschiedene Werte annehmen kann, lassen sich mit Hilfe einer einzigen Stelle auch nur zehn verschiedene Zahlen darstellen, mit zwei Ziffern bereits 100, mit dreien 1000, ... Die Hinzunahme einer Stelle erweitert also die Anzahl darstellbarer Zahlen um einen Faktor zehn, der aus der Anzahl verschiedener zur Verfügung stehender Ziffern resultiert. Mit Hilfe einer neunstelligen Zahl lassen sich folglich zehn Milliarden verschiedene Zahlen darstellen – von 0 (der Kurzform für 000000000) bis hin zu 999999999.

Die Anzahl verschiedener Ziffern in einem Zahlensystem wird als „Basis“ bezeichnet und ist im Dezimalsystem gleich 10 (wie der Name bereits nahelegt). Nun stellt sich die Frage, welche, beziehungsweise wie viele Ziffern ein minimales Zahlensystem enthalten muß, um ein Stellenwertsystem konstruieren zu können. Klar ist, daß es zumindest ein Symbol für die Abwesenheit einer Ziffer enthalten muß – der Einfachheit halber wird hierfür im allgemeinen die 0 gewählt. Mit Hilfe dieser einzelnen Ziffer läßt sich nun jedoch nur eine Zahl in Form eines Stellenwertsystems darstellen – nämlich die Zahl Null selbst[5].

Außer der Ziffer 0 muß also noch eine weitere Ziffer vorhanden sein, um ein Stellenwertsystem zu konstruieren, beispielsweise die Ziffer 1. Ein solches Zahlensystem, das auf einem Stellenwertsystem mit lediglich zwei Ziffern beruht, wird als „binäres“ Zahlensystem bezeichnet und hat in erster Linie durch die weite Verbreitung von Computern einen unnachahmlichen Aufschwung genommen.

Im folgenden seien einige Beispiele zur Darstellung von Zahlen im Binärsystem gegeben: Aus der Binärzahl 0 entsteht durch Addition einer 1 die Zahl 1. Wird zu dieser Zahl nun eins addiert, erfolgt bereits ein Übertrag an die nächste Stelle der Zahl, da der Vorrat von zwei Ziffern in der ersten Stelle bereits ausgenutzt wurde, es ergibt sich mithin die Zahl 10, aus der sich durch weitere Additionen von 1 die Werte 11, 100, 101, 110, 111, 1000, usf. ergeben.

Die Umwandlung von Binärzahlen in das (bedauerlicherweise) gewohntere Dezimalsystem kann nach folgendem Schema vollzogen werden:

[4] Es handelt sich wirklich um den elften Zahlenwert, nicht um den zehnten, da die Null als erste Zahl verwendet wurde!

[5] Die Möglichkeit, beispielsweise Null durch 0, 1 durch 00, zwei durch 000, usw. auszudrücken, ist nicht zulässig, da es sich hierbei um kein Stellenwertsystem handelt.

Binär	Umrechnung	Dezimal
0	$0 \cdot 2^0$	0
1	$1 \cdot 2^0$	1
10	$1 \cdot 2^1 + 0 \cdot 2^0$	2
11	$1 \cdot 2^1 + 1 \cdot 2^0$	3
100	$1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$	4
101	$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$	5
110	$1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$	6
111	$1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$	7
1000	$1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$	8
1001	$1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$	9
...	...	

Entdeckt wurde dieses Zahlensystem zuerst im Jahre 1616 von John Napier[6] (der achte Gutsherr von Merchiston, 1550–1617, bestattet in der St. Cuthbert's Church in Edinburgh), danach von Gottfried Wilhelm Leibniz (1646–1716, siehe [10]), der hierfür den Begriff des „dyadischen Systems" prägte. Weiterhin wird das Zahlensystem zur Basis 2 oftmals auch als Dualsystem bezeichnet.

Die Vorteile eines Zahlensystems zur Basis zwei bei der Konstruktion rechnender Maschinen liegen klar auf der Hand: Es ist wesentlich leichter, Schalt- und Speicherelemente, unabhängig davon, ob mechanischer, elektrischer oder elektronischer Natur, zu entwickeln, mit deren Hilfe sich zwischen nur zwei Zuständen unterscheiden läßt, als entsprechende Vorrichtungen für ein höheres Zahlensystem, beispielsweise zur Basis zehn, zu implementieren.

Der erste, der das binäre Zahlensystem zum maschinellen Rechnen einsetzte, war Konrad Zuse in seinen Maschinen der Z-Reihe (siehe hierzu [33], [23] und [1]). Was aufgrund seiner Beschränktheit auf nur zwei anstelle der gewohnten zehn Ziffern unhandlich und eingeschränkt wirkt, erweist sich spätestens bei der Automatisierung von Rechenvorgängen als genialer Kunstgriff[7].

Allgemein läßt sich also über Stellenwertsysteme zu einer gegebenen Basis $b \in \mathrm{N}$, $b > 1$, feststellen, daß der Wert w einer aus $k+1$, $k \in \mathrm{N}_0$ Ziffern z_k, die jeweils b verschiedene Werte annehmen können, bestehenden Zahl gleich

$$w = \sum_{i=0}^{k} z_i b^i \tag{2.1}$$

ist, wobei z_0 der rechten Ziffer, z_k analog hierzu der linken Ziffer entspricht.

Die rechte Ziffer eine Zahl stellt stets die niederwertigste Ziffer dar und wird im Falle des Binärsystems auch als „LSB" (kurz für Least-Significant-Bit) bezeichnet. Entsprechend heißt die rechte Ziffer einer solchen Zahl „MSB" für Most-Significant-Bit, wobei jede Ziffer einer Zahl im Dualsystem als „Bit" bezeichnet wird.

2.1.1 Beispiel

Nachdem nun die grundlegenden Eigenschaften eines Stellenwertsystems dargestellt wurden, zeigt der folgende Abschnitt zwei Lösungsmöglichkeiten für das Problem, aus einer Folge von Ziffern den Wert der zugehörigen Zahl in ihrem Stellenwertsystem zu bestimmen. Obwohl die Lösung dieser Aufgabenstellung weitestgehend umgangssprachlich beschrieben wird, handelt es sich doch um ein Programm, beziehungsweise um einen

[6] Siehe [13], S. 142, S. 215.

[7] Abgesehen davon, daß das binäre Zahlensystem auch im täglichen Leben durchaus seine Vorteile mit sich brächte – ließe sich doch unter Zuhilfenahme von zehn Fingern nicht nur bis zehn, wie bei herkömmlicher Zählweise gelangen, sondern von 0 bis Eintausenddreiundzwanzig, da sich mit zehn Fingern, von denen jeder zwei verschiedene Zustände annehmen kann (0 beziehungsweise 1) insgesamt $2^{10} = 1024$ verschiedene Zahlen darstellen lassen – von 0 bis 1023.

sogenannten „Algorithmus“, einen Begriff, auf den in Abschnitt 2.2 näher eingegangen wird.

Als Beispiel diene zunächst die Ziffernfolge 3, 1, 4, 1, 5 im Dezimalsystem, deren Wert $w \in \mathrm{N}$ sich entsprechend (2.1) zu

$$w = 3 \cdot 10^4 + 1 \cdot 10^3 + 4 \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0$$

ergibt. Soll ein solcher Wert berechnet werden, scheint es zunächst gleichgültig zu sein, ob mit der Berechnung ausgehend von der rechten oder der linken Ziffer begonnen wird.

2.1.1.1 Erster Ansatz

Allgemein läßt sich nun folgende Berechnungsvorschrift zur Bestimmung des Wertes einer Ziffernfolge zu einer Basis b angeben, hierbei stehen w für den zu berechnenden Wert, i für die Stelle der gerade betrachteten Ziffer, wobei die rechte Stelle dem Wert $i = 0$ entspricht, z_j für den Wert der Ziffer an der j-ten Stelle der Ziffernfolge und k für die Nummer der linken Ziffer, da es sich um eine $k+1$-stellige Zifferfolge handeln soll:

1. Setze w auf den Wert Null[8].
2. $i := 0$.
3. Addiere zu w den Wert, der sich aus $z_i b^i$ ergibt.
4. Erhöhe i um eins.
5. Falls $i \leq k$ gilt, springe zu Punkt 3.

Die Operationen „Addiere zu...“ beziehungsweise „Erhöhe um den Wert...“ werden im allgemeinen in einer besonderen Form notiert, die auf den ersten Blick dem mathematischen Empfinden widerspricht. Beispielsweise wird „Erhöhe i um eins“ kurz als

$$i := i + 1$$

notiert. Zu beachten ist hierbei, daß es sich bei der Operation $:=$ nicht um das gewohnte $=$, sondern vielmehr um eine „Zuweisungsoperation“ handelt, so daß i der Wert zugewiesen wird, der sich aus der Addition von i und dem Wert 1 ergibt, i wird also um eins erhöht[9].

Die vorliegende Berechnungsvorschrift addiert nun – beginnend bei der linken Stelle (d.h. der 0-ten Stelle) – jeweils die sich aus der Multiplikation $z_k b^k$ ergebenden Werte zum angestrebten Endresultat auf. Die Berechnung endet, sobald i den Wert k überschreitet, d.h. sobald alle Ziffern der Ziffernfolge verarbeitet wurden.

Vom Standpunkt der Problemstellung aus betrachtet, ist diese Vorschrift sicherlich zulässig; sie berechnet für eine beliebige Ziffernfolge mit $k+1$, $k \in \mathrm{N}$ Ziffern den Wert der Zahl, der diese Ziffernfolge in einem Stellenwertsystem zur Basis $b \in \mathrm{N}$, $b > 1$, entspricht.

Handelte es sich beim Vorgang des Programmierens um einen rein technischen Vorgang, könnte das Beispiel hiermit abgeschlossen werden, da die Aufgabenstellung sicherlich erfüllt ist. Diese weit verbreitete Ansicht ist jedoch nicht zutreffend – Programmieren ist im allgemeinen kein mechanischer Vorgang, sondern vielmehr eine Kunstform, die nicht nur rein technischen, sondern auch und unter Umständen vielmehr ästhetischen Gesichtspunkten verpflichtet ist, wie [4], S. vii (eine übrigens ausgesprochen lesenswerte Lektüre!), treffend beschreibt[10]:

[8] Kurz $w := 0$, wobei der Operator $:=$ im Gegensatz zu einem einfachen $=$ die Zuweisung eines Wertes repräsentiert.

[9] Diese Operation wird im allgemeinen auch als „Inkrement“ bezeichnet. Wird der Wert einer Variablen um eins erniedrigt, spricht man entsprechend von einem „Dekrement“.

[10] Eine Sichtweise, die bereits der Titel von [14], [15], [16] hoffähig gemacht hat.

> Programming is a fascinating and challenging subject. Unfortunately, it is rarely presented as such. [...] When approached in this way, programming is indeed a dull activity!

Betrachtet man das Programm von einem übergeordneten Standpunkt, in diesem Fall entweder im Hinblick auf seine Ästhetik, wobei sich über diesen Begriff – wie in jedem anderen Zusammenhang – sicherlich streiten läßt, oder im Hinblick auf seine Effizienz, fällt zumindest ein Punkt auf: Zur Berechnung des Wertes einer jeden Stelle, von denen es $k+1$ Stück gibt, ist die Auswertung eines Ausdruckes der Form $z_k b^k$ notwendig. Hierbei ist z_k bekannt, b^k muß jedoch für jede Stelle erneut berechnet werden. Ausführlicher läßt also zunächst schreiben (wobei m als Multiplikator Verwendung findet, also dem Wert b^k entsprechen wird; j ist eine Hilfsvariable):

1. $w := 0$, $i := 0$.
2. (a) $m := 1$.
   (b) Springe zu 3, falls $i = 0$ gilt.
   (c) $j := i$.
   (d) $m := mb$; $j := j - 1$.
   (e) Springe zu 2d, falls $j > 0$ gilt.
3. $w := w + z_i m$; $i := i + 1$.
4. Springe zu 2, falls $i \leq k$.

In dieser Darstellung wird der Aufwand zur Berechnung von b^k erstmals wirklich deutlich – hinter der einfachen Notation b^i verbirgt sich in Wahrheit eine Folge von Multiplikationen, um b^k als Produkt $b \cdot b \cdot b \cdots b$ darzustellen – weder eine ausgesprochen elegante, noch eine besonders effiziente Methode, den gesuchten Wert für w zu berechnen.

Zur Berechnung der Exponenten für mehrstellige Zahlen sind bei dieser Berechnungsvorschrift folgende Anzahlen von Multiplikationen notwendig:

Anzahl Ziffern	Anzahl Multiplikationen
1	0
2	1
3	3
4	6
5	10
6	15
7	21
8	28
...	...

Die erste Ziffer benötigt Null Multiplikationen zur Berechnung des zugehörigen Multiplikators $b^0 = 1$, die zweite eine, die dritte zwei, die vierte drei, usf. Allgemein ergibt sich ein Wert von

$$n = \sum_{i=0}^{k} i = \begin{cases} 0, & \text{falls } k = 0, \\ \frac{k(k+1)}{2} & \text{sonst.} \end{cases}$$

für die Anzahl notwendiger Multiplikationen zur Berechnung der Multiplikatoren m bei Anwendung obiger Rechenvorschrift auf eine $k+1$-stellige Ziffernfolge. Für die Gesamtzahl an Multiplikationen ergibt sich also

$$n_{gesamt} = \sum_{i=0}^{k} i = \begin{cases} 1, & \text{falls } k = 0, \\ \frac{k(k+1)}{2} + k + 1 & \text{sonst.} \end{cases}$$

2.1.1.2 Zweite Variante

Die Notwendigkeit, in jedem Schritt des vorigen Beispiels eine Potenz der Form b^k durch eine Folge von Multiplikationen berechnen zu müssen, rührt nicht zuletzt aus der zu Anfang vollzogenen Überlegung her, mit der Summation über die einzelnen Teilausdrücke von rechts beginnend, zu arbeiten.

Die folgende Berechnungsvorschrift ist wesentlich eleganter als die in Abschnitt 2.1.1.1 angegebene und vermeidet die wiederholte Berechnung von Potenzen b^k:

1. $w = 0$, $i = k$.
2. $w := 10w + z_i$; $i := i - 1$.
3. Springe zu 2, falls $i \geq 0$ gilt.

Mit $k = 4$ und der Ziffernfolge 3, 1, 4, 1, 5 aus Abschnitt 2.1.1 läuft nun folgende Berechnung ab: Zunächst wird $w := z_k = 3$ gesetzt, da die Multiplikation $w := 10w$ mit $w = 0$ im ersten Schritt ohne Auswirkungen bleibt, so daß nach dem ersten Durchlauf (eine solche Konstruktion, in der ein Teil einer Berechnungsvorschrift wiederholt – im allgemeinen abhängig von einer Bedingung – ausgeführt wird, wird als „Schleife" bezeichnet) $w = 3$ gilt. Da nun $i = 3 \geq 0$ gilt, erfolgt ein weiterer Schleifendurchlauf: Es wird $w := 10w = 30$ berechnet. Zu diesem Wert wird nun 1 addiert, so daß sich w zu 31 ergibt. Die folgenden Schritte liefern nun jeweils die Zwischenergebnisse $w = 314$, $w = 3141$ und $w = 31415$, womit auch das Schleifenende erreicht ist, da nun $i = -1 < 0$ gilt.

Benötigte die Rechenvorschrift aus Abschnitt 2.1.1.1 für die Berechnung des Wertes einer $k + 1$-stelligen Ziffernfolge (es sei der Einfachheit halber $k > 0$ angenommen) insgesamt $\frac{k(k+1)}{2} + k$ Multiplikationen, so sind es bei der eben dargestellten Variante nur mehr $k + 1$ Multiplikationen!

Hinsichtlich des notwendigen Berechnungsaufwandes weist also die erste gegebene Berechnungsvorschrift ein etwa quadratisches Verhalten auf, das aus dem Term $\frac{k(k+1)}{2}$ resultiert, während die zweite Variante sich linear mit k verhält. Solche Betrachtungen spielen bei Berechnungsvorschriften, die nur auf kleine Datenmengen angewandt werden, im allgemeinen keine große praktische Rolle – anders stellt sich die Situation jedoch dar, sobald die Menge zu verarbeitender Daten anwächst. Für kurze Ziffernfolgen (ohne den Begriff „kurz" in diesem Zusammenhang näher untersuchen zu wollen) ist es sicherlich unerheblich, welche der beiden vorgestellten Varianten der Vorschrift zum Einsatz kommen, für $k = 7$ benötigt Ansatz 1 insgesamt $28 + 8 = 36$ Multiplikationen, während die verbesserte, zweite Version mit 8 auskommt. Wächst k jedoch weiter an, macht sich die Form des Berechnungsaufwandes stärker bemerkbar: Für $k = 1000$ ergeben sich im linearen, zweiten Fall insgesamt nur 1001 notwendige Multiplikationen, während die erste Variante bereits $\frac{1000 \cdot 1001}{2} + 1001 = 501501$ Multiplikationen benötigt.

Solche Komplexitätsbetrachtungen von Algorithmen werden im weiteren Verlauf des Skriptes nicht behandelt, es sei auf [25] verwiesen. Hier werden auch die mathematischen Grundlagen ausgebaut, die zur Abschätzung solcher Komplexitäten notwendig sind. Im Fall von Berechnungsvorschriften, die nur auf kleine Datenmengen angewandt werden, sind solche Betrachtungen, wie bereits erwähnt, im allgemeinen nicht zwingend notwendig, Sedgewick ([25], S. 23) spricht hier gar von „überoptimierten Programmen" – eine Sichtweise, der man sich nicht zwingend anschließen muß – wichtig ist nicht zuletzt die Zufriedenheit des Programmierers mit dem erdachten und erzeugten Programmcode, die unter Umständen Hand in Hand geht mit Erwägungen, wie den oben angerissenen.

Das Hornerschema Die der vorangegangenen Variante zugrundeliegende Idee basiert auf einer Anwendung des sogenannten Hornerschemas, mit dessen Hilfe im allge-

meinen Polynome entsprechend

$$z_i b^i + z_{i-1} b^{i-1} + \ldots + z_1 b^1 + z_0 b^0 = (\ldots((z_i b + z_{i-1})\, b + z_{i-2})\, b + \ldots)\, b + z_0 \quad (2.2)$$

ausgewertet werden können. Kennzeichnend hierfür ist, daß die einzelnen Potenzen b^i des Multiplikators nicht in jedem Schritt erneut ausgewertet werden müssen, sondern im Verlauf der gesamten Rechnung nur einmal bis zur höchsten auftretenden Potenz gerechnet wird, während alle benötigten, kleineren Potenzen zu geeigneten Zeitpunkten „abgefangen“ werden.

Im folgenden ist die Berechnung des dezimalen Zahlenwertes w der Ziffernfolge 3, 1, 4, 1, 5 nach beiden Varianten dargestellt, hierbei sei $z_0 = 5, z_1 = 1, z_2 = 4, z_3 = 1, z_4 = 3$. Zunächst wird die Summe $\sum_{i=4}^{0} z_i 10^i$ direkt ausgewertet, danach durch stückweise Berechnung der auftretenden Potenzen entsprechend dem Hornerschema aus Gleichung (2.2):

$$\begin{aligned} w &= \sum_{i=4}^{0} z_i 10^i \\ &= 3 \cdot 10^4 + 1 \cdot 10^3 + 4 \cdot 10^2 + 1 \cdot 10^1 + 5 \cdot 10^0 \\ &= (((3 \cdot 10 + 1) \cdot 10 + 4) \cdot 10 + 1) \cdot 10 + 5 \end{aligned}$$

2.1.2 Vorzeichenbehaftete Zahlen

Ein Stellenwertsystem zu einer Basis $b \in \mathrm{N}$, $b > 1$, mit $k+1$-stelligen Zahlen, $k \in \mathrm{N}_0$ und Ziffern z_i, $i \in \mathrm{N}, 0 \leq i \leq k$, wie es in den vorangegangenen Abschnitten vorgestellt wurde, ist hervorragend zur Darstellung positiver Zahlenwerte geeignet, da der Wert einer solchen Zahl sich bekanntlich zu

$$w = \sum_{i=0}^{k} z_i b^i$$

ergibt. Hierbei nicht berücksichtigt wurden bislang negative Zahlen (was nichts mit den Schwierigkeiten zu tun hat, an denen die Anerkennung negativer Zahlen als reale Objekte jahrhundertelang scheiterte) – eine durchaus einschneidende Einschränkung, sind diese doch Voraussetzung für Subtraktion und letztlich auch Division, etc.

Im Grunde ließe sich ein Stellenwertsystem ohne weiteres um ein Vorzeichen erweitern, so daß der dezimale Wert -12 in einem vorzeichenbehafteten dualen Zahlensystem als -1100 darstellbar wäre – ein Weg, der allerdings in der Mehrzahl aller Fälle aufgrund einer Reihe praktischer Erwägungen nicht beschritten wird.

Zunächst sei ein Beispiel zur Berechnung von Addition und Subtraktion im Dezimalsystem gegeben, das im Anschluß um ein konkretes Beispiel im Binärsystem erweitert wird, um den Hauptnachteil eines explizit vorangestellten Vorzeichens bei der Darstellung und Verarbeitung vorzeichenbehafteter Zahlen darzustellen.

Zunächst wird der Fall der Addition anhand des dezimalen Beispiels $17+9$ behandelt – im gewohnten Dezimalsystem ergibt sich sofort:

$$\begin{array}{rcc} & 1 & 7 \\ + & & 9 \\ \hline & 2 & 8 \end{array}$$

Binär betrachtet entspricht dies

$$\begin{array}{rccccc} & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 1 & 0 & 0 & 1 \\ \hline & 1 & 1 & 0 & 1 & 0 \end{array}$$

– kein unerwartetes Ergebnis. Wendet man die gewohnten Additionsregeln des Dezimalsystems, in dem ein Übertrag entsteht, sobald zur höchsten verfügbaren Ziffer, der 9, ein postivier Wert addiert wird, um das Überschreiten des Ziffernbereiches der betreffenden Stelle kenntlich zu machen, analog auf ein Zahlensystem mit nur zwei Ziffern, 0 und 1, an, werden die Rechenregeln schnell deutlich. Beginnend von rechts wird ziffernweise addiert – zunächst ergibt sich aus der Addition $1+1$ der Wert 0, da 1 bereits die höchste Ziffer in diesem Zahlensystem ist. Um den erfolgten Überlauf dieser Stelle auszuweisen, erfolgt ein Übertrag auf die links folgende Stelle, so daß sich dort die Addition $0+0+1$ mit Resultat 1 ergibt. Entsprechend gilt $0+0=0$ beziehungsweise $0+1=1+0=1$.

Die Subtraktion zweier Zahlen verläuft ebenfalls in vergleichsweise gewohnten Bahnen – es sei der Ausdruck $17-9$ auszuwerten – in dezimaler Schreibweise ergibt sich hierfür:

$$\begin{array}{rr} & 1\ 7 \\ - & 9 \\ \hline & 8 \end{array}$$

Ein Ergebnis, das mit den Erwartungen sicherlich übereinstimmt – zunächst wird von 7 der Wert 9 subtrahiert, was zu einer Art „negativen Übertrages“ führt, der die führende 1 zu Null werden läßt. Ein solcher „borgender“ Übertrag wird im englischen Sprachraum auch als „Borrow“ bezeichnet, da er einen Wert von der linken Seite entfernt, während für einen normalen Übertrag, der den Wert links von ihm vergrössert, der Begriff „Carry“ üblich ist.

Binär könnte die entsprechende Rechnung wie folgt durchgeführt werden:

$$\begin{array}{rrrrrr} & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 1 & 0 & 0 & 1 \\ \hline & 0 & 1 & 0 & 0 & 0 \end{array}$$

Von rechts beginnend werden nun bitweise Subtraktionen vorgenommen, wobei die Subtraktion einer 1 von einer 1 den Wert Null zurückläßt – ist von 0 eine 1 zu subtrahieren, tritt wieder ein borgender Übertrag auf, der dazu führt, daß der links von ihm stehende Wert um eins vermindert wird, usf.

2.1.2.1 Subtraktion durch Addition

Der Punkt, auf den die beiden obigen Beispiele hinauslaufen, ist die – auf den ersten Blick selbstverständlich und geradezu trivial anmutende – Feststellung, daß für die Durchführung von Additionen und Subtraktionen zwei verschiedene Vorgehensweisen, d.h. Berechnungsvorschriften beziehungsweise Algorithmen (siehe Abschnitt 2.2), notwendig sind, wie sie auch in der Schule (bedauerlicherweise) im allgemeinen vermittelt werden. Die Addition bedient sich positiver Überträge von rechts nach links, um Überschreitungen des Ziffernbereiches einzelner Stellen mitzuprotokollieren, während die Subtraktion einen analogen Mechanismus, jedoch mit „borgenden“ Überträgen zum Einsatz bringt, um entsprechend einer Unterschreitung des Ziffernbereiches Rechnung zu tragen.

Für eine Automatisierung von Rechenvorgängen hätte ein solches Vorgehen nun zur Folge, daß unterschiedliche Mechanismen, welcher Art auch immer, implementiert werden müßten, um sowohl Additionen als auch Subtraktionen möglich zu machen. Im Hinblick auf eine sicherlich stets angestrebte Aufwandsminimierung solcher Rechenmaschinen eine nur eingeschränkt wünschenswerte Vorgehensweise.

Prinzipiell können alle Subtraktionen durch Additionen ausgeführt werden, wobei der Begriff des „Komplements“ benötigt wird, mit dessen Hilfe die Berechnung wirklicher Subtraktionen überflüssig wird. Zunächst ein dezimales Beispiel, um die Idee des Komplements dazustellen:

Zu berechnen sei $911-17$ – in herkömmlicher Subtraktionstechnik ergibt sich hierbei:

$$\begin{array}{rcccc} & & 9 & 1 & 1 \\ - & & & 1 & 7 \\ \hline & & 8 & 9 & 4 \end{array}$$

Angenommen, die Differenz $983 = 1000 - 17$ sei bekannt (eine Annahme, die vor allem im Hinblick auf das Binärsystem durchaus zulässig ist, da hier eine ausgesprochen einfache Möglichkeit zur Verfügung steht, solche speziellen Differenzen zu berechnen), ließe sich auch folgende Addition durchführen:

$$\begin{array}{rcccc} & & 9 & 1 & 1 \\ + & & 9 & 8 & 3 \\ \hline & 1 & 8 & 9 & 4 \end{array}$$

Unter Fortlassung der führenden 1 in der Tausenderstelle ergibt sich hieraus der Wert 894, der genau dem Ergebnis der direkten Subtraktion $911 - 17$ entspricht, was auch naheliegend ist, da das Unterdrücken der führenden Eins einer impliziten Subktraktion des Wertes 1000 entspricht – genau der Eintausend, die zu Beginn in Form von $1000-17$ zuviel addiert wurde.

Auf den ersten Blick scheint dieser Trick, eine Subtraktion auf eine Addition abzubilden, vergleichsweise witzlos – es wird zu Beginn der Rechnung 1000 addiert, um gleich im Anschluß hieran wieder subtrahiert zu werden, im Grunde liegt hier eine vollkommen überflüssige Operation vor, wäre da nicht ein gewichtiger Vorteil des beschriebenen Verfahrens – hiermit können alle Subtraktionen auf Additionen zurückgeführt werden, sofern vorausgesetzt werden kann, daß die Berechnung von Werten wie $983 = 1000-17$ auf andere, einfachere Art und Weise möglich ist.

Ein weiteres Beispiel – diesmal im Bereich der binären Zahlen – soll diesen Punkt näher darstellen. Es sei (binär) $10111 - 01011$ ohne explizite Subtraktion zu errechnen. Zunächst stellt sich also die Frage, wie $100000 - 01011$[11] ermittelt werden kann, ohne eine explizite Subtraktion auszuführen, die schließlich vermieden werden soll. Hierbei führt folgende Überlegung zum Ziel:

Sollen zwei $k+1$-stellige Zahlen m (der Minuend) und s (der Subtrahend), k wie zuvor, voneinander subtrahiert werden, wobei eine eventuell kürzere Zahl links mit Nullen ergänzt wird, so ist zunächst der Subtrahend s von der $k+2$-stelligen Zahl der Form $c = 10\ldots0$ zu subtrahieren, um das Ergebnis $\overline{s}$ dieser Berechnung auf den eigentlichen Minuenden m aufzuaddieren. Das solcherart erhaltene Ergebnis muß nun lediglich noch um den Wert c vermindert werden, was sich auf einfachste Weise durch Unterdrücken der führenden 1 erzielen läßt.

Notwendig ist nun ein einfaches, subtraktionsfreies Verfahren zur Berechnung von $\overline{s} = c - s$, wobei c eine $k+2$-stellige Zahl, bestehend aus einer führenden 1 und $k+1$ nachfolgenden Nullen darstellt: Es ist also im vorliegenden Beispiel

$$\begin{array}{rcccccc} c = & 1 & 0 & 0 & 0 & 0 & 0 \\ s = - & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline \overline{s} = & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

zu berechnen. Es fällt auf, daß sich $\overline{s}$ aus s ergibt, indem zunächst jede einzelne Stelle von s, jedes Bit, umgekehrt, „invertiert“ wird, wobei am Ende noch der Wert 1 addiert werden muß. Der Wert s', der sich aus der Umkehrung der einzelnen Ziffern von s ergibt, entspricht hierbei dem Ergebnis der Subtraktion $s' = c' - s$ mit $k+1$-stelligem $c' = 1\ldots1$, da hierbei stets entweder eine 0 oder eine 1 (die Ziffern von s) von 1 abgezogen wird, so daß faktisch eine Invertierung der Ziffern von s vollzogen wird, d.h. jede 1 wird zu einer 0 und umgekehrt. Zieht man nun in Betracht, daß $\overline{s} = s' + 1$ gilt, ergibt sich die noch fehlende Addition.

[11] Entsprechend der Differenz $983 = 1000 - 17$ im vorangegangenen Beispiel.

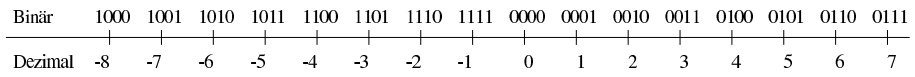

Abbildung 2.1: Zweierkomplement am Zahlenstrahl

Die Zahl $\overline{s}$ wird als „Zweierkomplement“ von s bezeichnet.

Die Bezeichnung Zweierkomplement rührt von der Basis des Binärsystems her, die zwei ist. Entsprechend wird der Wert $983 = 1000 - 17$ als „Zehnerkomplement“ von 017 bezeichnet (allerdings ist dessen Berechnung ein wenig komplexer als die des Zweierkomplements, da die Invertierung einer Ziffer im Zweiersystem wesentlich einfacher ist als im Zehnersystem). Ausführliche mathematische Hintergründe und Betrachtungen zu Zahlensystemen zu beliebiger Basis (insbesondere auch der Basis 2) finden sich in [11] und [26].

Da nun die Addition des Komplements $\overline{s}$ einer Zahl s zu einem Wert m der Subtraktion der Zahl s von m entspricht, bietet es sich an, diese Zahl $\overline{s}$ als negativen Wert von s aufzufassen, was Abbildung 2.1 anhand eines Zahlenstrahles verdeutlicht.

Wie zu erkennen ist, wird die Null auch im Zweierkomplement durch eine Folge von Nullen dargestellt, die Darstellung der positiven Zahlen erfolgt wie gewohnt, so entspricht beispielsweise 0100 dem Dezimalwert 4, etc. Entsprechend dem oben gesagten, entspricht dezimal -4 im Zweiersystem als Zweierkomplementzahl betrachtet dem Wert 1100, der sich aus Umkehren der Ziffern in 0100 und anschließender Addition von 1 (und den daraus resultierenden Überträgen) ergibt.

Wichtigster Punkt hierbei ist die Asymmetrie der Binärzahlen im Zweierkomplement – auf der positiven Seite des Zahlenstrahls kann mit Hilfe vierstelliger Zahlen nur der Dezimalwert 7 erreicht werden, während auf der negativen Achse die -8 noch enthalten ist. Abbildung 2.1 zeigt anschaulich, daß negative Zahlen in Zweierkomplementdarstellung an einer führenden 1 erkennbar sind, während positive Zahlen von einer 0 eingeleitet werden. Die linke Ziffer einer Zahl in dieser Darstellung kann also als Vorzeichen angesehen werden, das in seinen Auswirkungen jedoch weit über das einfache Vorzeichen, wie es im allgemeinen Gebrauch des Dezimalsystems beispielsweise sich eingebürgert hat, hinausgeht, da sich positive und negative Zahlen in Komplementdarstellung nicht nur in ihrem Vorzeichen sondern in allen Ziffern unterscheiden.

Zu bemerken ist in diesem Zusammenhang, daß es natürlich auch ein einfaches Zweiersystem gibt, in dem der Unterschied zwischen einer $+4_{10} = 0100_2$ (die Basis des jeweils verwendeten Zahlensystems ist hier zur besseren Lesbarkeit tiefgestellt) und einer $-4_{10} = 1100_2$ nur im Vorzeichen begründet liegt. Dies bringt aber unweigerlich den Nachteil mit sich, daß Subtraktion und Addition wieder unterschiedliche, voneinander getrennte Operationen darstellen, für die jeweils gesonderte Mechanismen vorgesehen werden müssen.

Solche Darstellungen, in denen sich negative von positiven Zahlen ausschließlich anhand des Vorzeichens unterscheiden, werden im englischen Sprachraum als „Sign Magnitude Representation“ bezeichnet. Darstellungen, die dem behandelten Zweierkomplement entsprechen, in dem negative Zahlen als Komplement dargestellt werden, werden im allgemeinen als „Radix Complement Representation“ bezeichnet.

2.1.2.1.1 Bemerkung zum Einerkomplement Neben den beiden behandelten Verfahren zur Darstellung positiver und negativer Zahlen, d.h. der herkömmlichen Darstellung mit Hilfe eines expliziten Vorzeichens, das allein über Positivität oder Negativität des nachfolgenden Wertes entscheidet, beziehungsweise der Darstellung in Form beispielsweise des Zweierkomplementes im Binärsystem (analog hierzu das Zehnerkomplement im Dezimalsystem), existiert ein weiteres Zahlensystem, mit dessen Hilfe zum

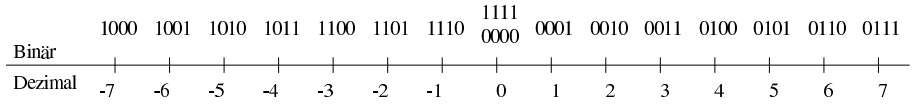

Abbildung 2.2: Einerkomplement am Zahlenstrahl

einen negative Zahlen dargestellt werden können, zum anderen aber auch Subtraktionen auf Additionen zurückgeführt werden können, das im folgenden ausschließlich im Zweiersystem betrachtet wird:

Ausschlaggebend für die Entwicklung des folgenden Mechanismus war die Bedingung, für die Berechnung des Zweierkomplementes $\overline{s}$ einer Zahl s nicht nur die einzelnen Bits von s umkehren zu müssen, sondern darüberhinaus im Anschluß hieran den Wert 1 addieren zu müssen, so daß eine einfache Subtraktion nicht auf lediglich eine, sondern vielmehr zwei Additionen abgebildet wird, was im Hinblick auf eine möglichst rasche Operationsausführung nicht unbedingt wünschenswert erscheint.

Die einfachste Art und Weise, eine Subtraktion auf eine, statt auf zwei Additionen zurückzuführen, besteht in der Verwenden des sogenannten „Einerkomplements" $\hat{s}$ einer Binärzahl s, das sich durch einfache Umkehrung der einzelnen Ziffern von s ergibt. In Bezug auf das Zweierkomplement $\overline{s}$ gilt also $\hat{s} = \overline{s} - 1$. Abbildung 2.2 zeigt die Verteilung von Zahlen in Einerkomplementdarstellung am Zahlenstrahl, entsprechend Figur 2.1. Gilt $\overline{s} = c - s$ mit $c = 10\ldots0$, $k+2$-stellig, so ist entsprechend $\hat{s} = c' - s$ mit $c' = 1\ldots1$, $k+1$-stellig, da $c = c' + 1$ gilt. Die Bildung des Einerkomplements verzichtet also auf die korrigierende Addition einer 1, wie sie bei der Bildung des Zweierkomplements vorgenommen wird.

Dieser Vorteil des Einerkomplements, der Verzicht auf die abschließende Korrekturaddition und der damit einhergehende Geschwindigkeitvorteil wird (ausgesprochen) teuer erkauft durch die Tatsache, daß Zahlen in Einerkomplementdarstellung symmetrisch auf dem Zahlenstrahl angeordnet sind, so daß es zwei verschiedene Repräsentationen des Wertes Null geben muß! In Einerkomplementdarstellung existieren sowohl $+0$, als auch -0, die als zwei getrennte Werte behandelt werden müssen, beispielsweise ist auf Systemen, die im Einerkomplement rechnen[12] eine Abfrage der Form $i = 0?$ nicht ohne weiteres möglich, vielmehr muß ($i = +0$ oder $i = -0$) geprüft werden, was den praktischen Umgang mit dieser Form der Zahlendarstellung nicht unwesentlich erschwert.

Im Lauf der Zeit wurden ausgesprochen ausgefeilte Verfahren entwickelt, um Additionen in einer Zeitspanne durchführen zu können, die den Mehraufwand, wie er bei der Berechnung des Zweierkomplements einer Zahl entsteht, erträglich werden ließen, so daß das Einerkomplement heutzutage in keiner (dem Autor bekannten) kommerziellen Computerarchitektur mehr zum Einsatz kommt und Geschichte geworden ist.

Die Einerkomplementdarstellung wird im Englischen zumeist als „Reduced Radix Complement Representation" bezeichnet, was aus der Reduktion von c auf $c' = c - 1$ herrührt.

2.1.3 Gleitkommazahlen

In den vorangegangenen Abschnitten wurden bislang lediglich positive, später auch vorzeichenbehaftete, ganze Zahlen, d.h. Zahlen, die auf eine Teilmenge der Menge der natürlichen Zahlen N abbildbar sind, betrachtet. Für eine Vielzahl von Berechnungen

[12] Alle Maschinen, die im Einerkomplement arbeiten, haben mittlerweile – zumindest teilweise – ihren Platz in Museen eingenommen. Zu den berühmtesten Architekturen, die Gebrauch von diesem Zahlensystem machten, gehören sicherlich die Maschinen der CDC6600- und CDC76600-Reihe der Control-Dara-Corporation, die von Seymour Cray designed wurden und zu ihrer Zeit die schnellsten Digitalrechner der Welt waren – ein Monopol, das Seymour Crays Maschinen über einen langen Zeitraum, einige Jahrzehnte, immer wieder innehatten.

und Betrachtungen ist eine solche Einschränkung durchaus hinzunehmen, hierunter fällt jedoch im allgemeinen nicht der große (und wichtige) Bereich des sogenannten „scientific computing“. Im Verlauf von Berechnungen im Bereich des naturwissenschaftlichen Rechnens treten oftmals Zahlen in grundlegend unterschiedlichen Größenordnungen auf; innerhalb eines Ausdruckes können Zahlen, die im Bereich von Millionsteln liegen, mit Zahlen, die Bereich der Trilliarden existeren, miteinander verknüpft und in Beziehung gesetzt werden.

Betrachtet man sich die Darstellung ganzer Zahlen in einem Zahlensystem zu einer Basis $b \in \mathrm{N}$, $b > 1$, so lassen sich mit $k+1$-stelligen Zahlen, $k \in \mathrm{N}_0$, allgemein betrachtet b^{k+1} verschiedene Werte repräsentieren. Mit Hilfe sechzehnstelliger Binärzahlen, können folglich $2^{16} = 65536$ verschiedene Zahlen dargestellt werden. Um Werte zwischen Null und etwa vier Milliarden[13] darstellen zu können, sind bereits 32 Bit, d.h. 32-stellige Zahlen notwendig, da $2^{32} = 4294967296$ gilt.

Hierbei werden jedoch nur ganze Zahlen dargestellt, d.h. der dezimale Wert 1.5 ist bereits nicht mehr ohne Kunstgriffe darstellbar. Natürlich ließe sich eine solche Darstellung erreichen, indem der zur Verfügung stehende Zahlenbereich verdoppelt und entsprechend vereinbart wird, daß hiermit Zahlen im Abstand von 0.5 anstelle einfacher natürlicher Zahlen repräsentiert werden sollen. Dieses Vorgehen hat jedoch seine Grenzen – die Länge der benötigten Zahlen für solche Darstellungen schnellt ausgesprochen schnell in die Höhe, sobald die Lücken zwischen den einzelnen darzustellenden Zahlen kleiner werden. Sollen Zahlen in Abständen von $\frac{1}{8}$ dargestellt werden, müssen bereits acht mal mehr Zahlen verwendet werden, die Darstellung erfordert also drei zusätzliche Bit (da $2^3 = 8$).

Eine Darstellung nicht ganzzahliger Werte durch Abbildung auf einfache, ganzzahlige Zahlen ist also kein brauchbarer Lösungsansatz. Vielmehr muß ein gänzlich neuer Weg beschritten werden, der bei der Verwendung eines (vorzugsweise naturwissenschaftlichen, am liebsten HP-) Taschenrechners deutlich wird. Da sich hier sehr große Werte aufgrund der beschränkten Ziffernanzahl der Maschine selbst, nicht mehr vollständig darstellen lassen, wird folgender Kunstgriff angewandt: Eine Zahl wird in Form einer Mantisse und eines Exponenten dargestellt, die beide eine endliche Länge aufweisen. Die Mantisse entspricht dem „normalisierten“ Anteil der darzustellenden Zahl, während der Exponent für die anschließend notwendige Verschiebung des Kommas sorgt.

Folgendes Beispiel mag diese Vorgehensweise verdeutlichen: In Mantissen/Exponenten-Schreibweise kann dezimal $x = 123456789$ bei Beschränkung auf beispielsweise 5 Mantissenstellen als $x' = 1.2345 \cdot 10^8$ dargestellt werden. Größenordnungsmäßig entspricht x' durchaus dem eigentlich Wert x, beide liegen im Milliardenbereich, der Fehler $x - x'$ beträgt in diesem Beispiel zwar 6789, was jedoch bei einer Zahl im Milliardenbereich nur unwesentlich ins Gewicht fällt.

Auch hier ist folglich abzuwägen – der Vorteil einer Darstellbarkeit von Zahlen stark schwankender Größenordnung mit Hilfe einer Mantisse und eines Exponenten, wird durch den Nachteil einer geringeren Genauigkeit erkauft. Im allgemeinen wird dieser Tausch eingegangen – mit allen Konsequenzen, die dies nach sich zieht, so ist beispielsweise bei der Implementation numerischer Berechnungsvorschriften mit Hilfe von Zahlen, die mit Hilfe längenbeschränkter Mantissen und Exponenten repräsentiert werden, im Vorfeld genau zu untersuchen, wie sich die zwangsläufig auftretenden Fehler (Darstellungsfehler, Rundungsfehler, usf.) auf das erwartete Ergebnis beziehungsweise die Praktikabilität der Berechnungsvorschrift selbst auswirken[14].

Zahlen, die mit Hilfe einer Mantisse und eines Exponenten jeweils gegebener, fester Länge dargestellt werden, heißen „Gleitkommazahlen“ beziehungsweise (im englischen Umfeld) „Floatingpoint Numbers“. Die „normale“ Form einer solchen Zahlendarstellung hat die Gestalt

[13] Beziehungsweise zwischen ca. -2 Milliarden und +2 Milliarden, sofern Zahlen im Zweierkomplement betrachtet werden.

[14] Siehe hierzu unter anderem [21], S. 23 ff. und [29], S. 1 ff.

VZM	Mantisse	VZE	Exponent

, wobei VZM das Vorzeichen der Mantisse repräsentiert, um auch negative Werte darstellen zu können, während VZE für das Vorzeichen des Exponenten steht, um Zahlen kleiner 1 darstellen zu können.

Hauptpunkt dieser kurzen Darstellung ist die Feststellung, daß im allgemeinen zwischen ganzen Zahlen, d.h. „Festpunktzahlen", wie sie in den vorangegangenen Abschnitten behandelt wurden, und Zahlen im Gleitkommaformat streng unterschieden werden muß. Erstere ermöglichen eine exakte Darstellung von Werten, die jedoch nur ganzen Zahlen entsprechen und nur verhältnismäßig wenige Größenordnungen überstreichen können, während letztere die Repräsentation von Zahlen stark wechselnder Größenordnungen (mit Hilfe des Exponenten) ermöglichen, hierbei jedoch eine gewisse, in vielen Fällen nicht zu vernachlässigende Ungenauigkeit mit sich bringen.

2.2 Algorithmen

In den vorangegangenen Abschnitten war des öfteren von „Berechnungsvorschriften" die Rede – ein Begriff, der nur selten fällt, da das, was in diesem Fall hiermit umschrieben wurde, im allgemeinen als „Algorithmus" bezeichnet wird.

Das Wort Algorithmus leitet sich von Muhammad Ibn al-Hwarizmi her[15], der im Jahre 825 ein bahnbrechendes Werk mit dem Titel „Al-gabr wa'l muqabalah" über Arithmetik verfasste[16] ([13], S. 102, 141).

Sedgewick, [25], S. 22, beschreibt einen Algorithmus wie folgt:

> Der Begriff *Algorithmus* wird in der Informatik verwendet, um ein Verfahren zur Lösung eines Problems zu beschreiben, das für eine Realisierung in Form eines Programms geeignet ist. Algorithmen sind der »Stoff« der Informatik: Sie sind zentraler Untersuchungsgegenstand in vielen, wenn nicht den meisten Bereichen dieses Fachgebiets.

Ein Algorithmus weist im allgemeinen eine Reihe von Eigenschaften auf, die sich in statische und dynamische Eigenschaften unterteilen:

Statische Eigenschaften:

Eindeutigkeit: Ein Algorithmus darf keine widersprüchlichen Aussagen liefern.

Parametrisierbarkeit: Algorithmen lösen Problem*klassen*, wobei die Auswahl eines bestimmten Teilproblems durch Parametrisierung des Algorithmus erfolgt[17].

Finitheit: Der gesamte Algorithmus kann in endlicher Länge beschrieben werden.

Dynamische Eigenschaften:

Ausführbarkeit: Ein Algorithmus muß prinzipiell ausführbar sein, darf also beispielsweise keine unerfüllbaren Anforderungen stellen.

Terminierung: Zur Berechnung des Ergebnis dürfen nur endlich viele Schritte verwandt werden, d.h. der Algorithmus muß nach einer endlichen Zeitspanne (die nichtsdestotrotz in ihrer Länge beliebig sein darf) terminieren.

[15] Dessen Name mittellateinisch Algorismi umschrieben wurde (siehe [13], S. 78).

[16] Aus dessen Titel sich im übrigen das heute gebräuchliche Wort „Algebra" herleitet.

[17] Beispielsweise löst der Algorithmus aus Abschnitt 2.1.1.2 die Problemklasse, den Wert einer beliebigen Ziffernfolge, die in einem Stellenwertsystem zur Basis b betrachtet wird, zu bestimmen. Durch Vorgabe der Werte (Parametern) 3, 1, 4, 1, 5 wird der Algorithmus auf die Lösung des Teilproblems, den Wert der Ziffernfolge 3, 1, 4, 1, 5 zu einer – ebenfalls zu spezifizierenden – Basis zu errechnen, parametrisiert.

Determiniertheit: Ein Algorithmus muß unter gleichen Ausgangsbedingungen stets gleiche Ergebnisse liefern.

Determinismus: Zu jedem Zeitpunkt kann innerhalb eines Algorithmus nur eine Möglichkeit zur Fortsetzung existieren.

Zusätzlich zu diesen grundlegenden Bedingungen, die ein Algorithmus erfüllen muß, werden in der Praxis oftmals eine Reihe zusätzlicher – teilweise nicht minder schwerwiegende – Forderungen gestellt, die im folgenden exemplarisch aufgezählt werden:

Effizienz: Hierunter ist ganz allgemein ein sparsamer Umgang mit Ressourcen, wie beispielsweise Hauptspeicherbedarf, Ausführungszeit, Plattenplatzbedarf, usf. zu verstehen[18].

Erweiterbarkeit/Änderbarkeit: Hierunter ist die Anpaßbarkeit eines Algorithmus an modifizierte Anforderungen zu verstehen, die im allgemeinen wenig Schwierigkeiten bereiten sollte.

Portabilität: Algorithmen sollten im allgemeinen (wobei auch hierbei Ausnahmen zulässig sind) möglichst wenig auf Maschinenspezifika einer bestimmten Rechnerarchitektur oder Spracheigenschaften einer bestimmten Programmiersprache eingehen oder sie gar zwingend voraussetzen, um eine leichte Portierbarkeit in eine andere Umgebung zu unterstützen.

Stabilität: Algorithmen sollten eine möglichst hohe Toleranz gegenüber Fehlersituationen aufweisen, sowie über eine ausgefeilte Fehlerbehandlung verfügen[19].

Wie bereits in den Abschnitten 2.1.1.1 und 2.1.1.2 deutlich wurde, lassen sich Algorithmen auf die unterschiedlichsten Arten und Weisen notieren – die folgenden Varianten sind hierbei am häufigsten anzutreffen:

- Umgangssprachliche Notation
- Pseudocode
- Beispielimplementation in einer Programmiersprache
- Flußdiagramme (siehe Abbildung 2.3)
- Struktogramme (sogenannte Nassi-Shneidermann-Diagramme, siehe Figur 2.4)

Die Formulierung des Beispielalgorithmus zur Auswertung von Ziffernfolgen erfolgte in Form von Pseudocode – unter Pseudocode fallen prinzipiell alle Darstellungen einer Berechnungsvorschrift, die in einer eingeschränkten, aber dennoch freien, an mathematische Gewohnheiten angelehnten Form erfolgen. Die Grenzen zwischen umgangssprachlicher Notation und Darstellung in Form von Pseudocode sind hierbei als relativ fließend zu betrachten. Anders natürlich im Fall einer Beispielimplementation in einer

[18] Meßverfahren, um das Verhalten von Algorithmen und Maschinen im Hinblick auf Effizienzfragen zu untersuchen, werden im allgemeinen als „Benchmarks“ bezeichnet. In diesem Zusammenhang sei die Frage erlaubt, ob der Begriff „überoptimierter Programme“, wie er von Sedgewick, [25], S. 23, verwendet wird, überhaupt zulässig ist.

[19] Zum Stichwort Fehlerbehandlung kursiert eine Geschichte aus den frühen siebziger Jahren, als auf einer Konferenz über das Betriebssystem Multics gesprochen wurde, dessen Entwickler darauf hinwiesen, daß etwas 30 Prozent des gesamten Quellcodes ausschließlich zur Fehlerbehandlung vorgesehen seien. Als daraufhin Dennis M. Ritchie – einer der Entwickler sowohl der Programmiersprache `C` als auch des Betriebssystems UNIX, das sich zu dieser Zeit gerade in den ersten Versionen fand, auf das Thema Fehlerbehandlung in UNIX angesprochen wurde, meinte er: „... You see – we have this single routine, called panic...“ Anstelle einer ausgefeilten Fehlerbehandlungsstrategie, über die beispielsweise das wesentlich ältere Multics verfügte, stellte UNIX zu dieser Zeit lediglich eine Routine `panic` zur Verfügung, um im Falle schwerer Fehler mit einer Ausgabe wie `Kernel panic, core dumped` zu crashen.

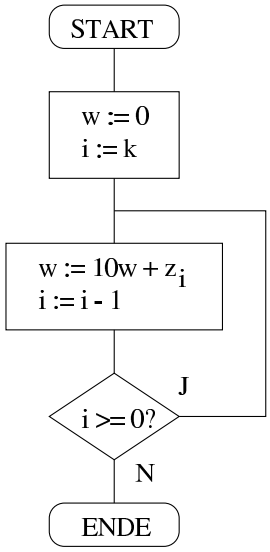

Abbildung 2.3: Flußdiagrammdarstellung der Ziffernfolgenumwandlung

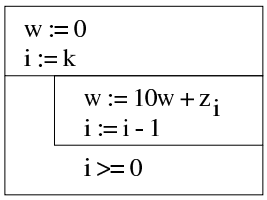

Abbildung 2.4: Struktogrammdarstellung der Ziffernfolgenumwandlung

wirklichen Programmiersprache: Hier können nur die sprachlichen Mittel der eingesetzten Programmiersprache zur Formulierung verwendet werden – die Abgrenzung zu umgangssprachlicher Darstellung oder einer Pseudocodedarstellung fällt leicht.

Flußdiagramme und Struktogramme sind Methoden, den Ablauf eines Algorithmus mit graphischen Hilfsmitteln zu veranschaulichen, wobei Struktogramme zeitlich weit nach den Flußdiagrammen aufkamen und im Gegensatz zu diesen einen bestimmten Programmierstiel (die sogenannte strukturierte Programmierung, hierauf wird in Kapitel 2.2.1 eingegangen werden) nahelegen.

Abbildung 2.3 zeigt die Darstellung des Algorithmus aus Abschnitt 2.1.1.2 in Form eines Flußdiagramms, während Figur 2.4 denselben Sachverhalt mit Hilfe eines Struktogrammes darstellt.

2.2.1 Strukturierte Programmierung

In den vorangegangenen Zeilen klang die strukturierte Programmierung in der Bezeichnung des Struktogrammes bereits an – ein Begriff, zu dem im folgenden eine kurze Charakterisierung gegeben werden soll, ohne zu sehr in die Tiefe zu gehen (im weiteren Verlauf des Skriptes finden sich an einer Reihe von Stellen entsprechende Anmerkungen und Beispiele zu diesem Thema).

Zur Darstellung der Grundidee der strukturierten Programmierung ist es zweckmäßig, quasi ihr Gegenteil in Form der etwas flapsig als „Spaghetti-Programmierung“ bezeichneten Vorgehensweise vorzustellen. Abbildung 2.3 mag einen ersten, vagen Eindruck von

der Haupteigenschaft eines solchen unstrukturierten Programmablaufes geben. Zentrales Element des implementierten Algorithmus zur Berechnung des Wertes einer gegebenen Ziffernfolge in einem Zahlensystem zu einer Basis b ist die Schleife, innerhalb derer die Summe über die einzelnen Ziffern, multipliziert mit ihrer jeweiligen Stellenwertigkeit, gebildet wird.

Während die Schleife als solche in der Darstellung gemäß Figur 2.4 klar zu Tage tritt, fristet sie in ihrer Flußdiagrammrepräsentation ein eher unscheinbares Dasein. Daß es sich hierbei um eine Schleife handelt, wird erst deutlich, wenn der Verlauf des Flußdiagramms näher in Augenschein genommen wird, während das Struktugramm keine Zweifel daran läßt, wann die Schleife beispielsweise beendet wird beziehungsweise was der Schleifenkörper beinhaltet.

Diese unterschiedliche Darstellung tritt nun nicht nur bei der graphischen Repräsentation von Algorithmen auf, sondern kann prinzipiell auch in (beinahe) beliebigen Programmiersprachen formuliert werden. Eine unstrukturierte Implementation, die dem obigen Flußdiagramm nahe steht, würde die Schleife in Form eines Sprunges, wie er im übrigen auch in der umgangssprachlichen Formulierung des Algorithmus in Abschnitt 2.1.1.2 auftritt, formulieren, was bei einem derart kleinen Programm sicherlich noch keine größeren Schwierigkeiten mit sich bringt. Der Hauptnachteil eines unverhüllten Sprunges ist die Tatsache, daß beim Lesen des Programmcodes an der Stelle des Sprunges selbst unmittelbar ersichtlich ist, wo sein Ziel liegt. Andererseits kann jedoch nicht mehr ohne weiteres bestimmt werden, ob eine bestimmte Stelle eines Programmes vielleicht das Ziel eines (unentdeckten?) Sprunges ist. Das „Wohin“ eines Sprunges ist im allgemeinen also weitaus klarer als das entsprechende „Woher“.

Der strukturierten Programmierung liegt nun in erster Linie die Überlegung zugrunde, daß fast alle denkbaren Anwendungen direkter Sprünge auch mit Hilfe spezieller Sprachkonstruktionen formuliert werden können, die jeweils nur für einen bestimmten Zweck einsetzbar sind. So bietet eine strukturierte Programmiersprache beispielsweise ein Konstrukt zur Implementation einer Schleife wie der in Abbildung 2.4 an.

Die folgenden beiden Beispielimplementationen der Zahlenumwandlung in Form zweier FORTRAN-Fragmente (ohne auf diese Sprache näher eingehen zu wollen) mögen diese unterschiedlichen Darstellungsformen verdeutlichen. Zunächst eine direkte Implementation des in Abbildung 2.3 dargestellten Algorithmus:

```
      W = 0
      I = K
10    W = 10 * W + Z (I)
      I = I - 1
      IF (I .GE. 0) GOTO 10
      ...
```

Hier ist nicht auf den ersten Blick klar, daß es sich im Grunde nur um eine einfache Schleife handelt, die verlassen wird, sobald `I` einen Wert kleiner Null enthält[20]. Anders bei Verwendung expliziter Sprachkonstrukte (wie sie ein modernes FORTRAN zur Verfügung stellt) zur Formulierung der Schleife, wie folgendes Beispiel zeigt:

```
      W = 0
      I = K
      DO WHILE (I .GE. 0)
        W = 10 * W + Z (I)
        I = I - 1
      END DO
      ...
```

[20] Die FORTRAN-typische Notation `.GE.` steht für Greater-Equal und vergleicht zwei Werte miteinander.

Nicht allein durch die augenfällige Einrückung der Schleife, beziehungsweise des „Schleifenkörpers“ (eine solche Einrückung wäre im Grunde bei der ersten Variante auch möglich) wird der Schleifenrumpf deutlich hervorgehoben. Der Hauptvorteil der Formulierung einer Schleife mit Hilfe einer `DO WHILE`-Anweisung wie in diesem Beispiel hat den großen Vorteil, daß hiermit *nur* eine Schleife gebildet werden kann – sowohl bei Lesen des einleitenden `DO WHILE` als auch des abschließenden `END DO` wird sofort deutlich, woher der Programmfluß kommt beziehungsweise wohin er strebt.

Ein direktes `GOTO` kann zu allen möglichen und unmöglichen Zwecken eingesetzt werden, ein `DO WHILE` ist nur zur Bildung von Schleifen verwendbar – eben diese eigentliche Einschränkung bildet die Grundlage der Stärke solcher strukturierten Sprachmittel. Bei ihrem Auftreten ist ohne jede weitere Überlegung oder Analyse des Quelltextes klar, um was für eine Struktur innerhalb des Programmes es sich handelt, eine While-Schleife, eine Endlosschleife, eine bedingte Ausführung von Instruktionen, etc.

Zu Beginn der siebziger Jahre wurde zum ersten Mal deutlich, daß die Folgekosten einer unstrukturierten Programmierweise die Kosten der eigentlichen Programmierung schnell überholen, da die Wartung solcher Programme unnötig erschwert, wenn nicht gar zum Teil unmöglich gemacht wird[21]. Im allgemeinen lohnt sich der Aufwand einer strukturierten Heransgehensweise bei der Entwicklung und Implementation von Programmen – auch kleinen – bereits nach kurzer Zeit.

Nachdem nun bereits zwei Beispiele in eine realen Programmiersprache aufgetreten sind, ist es an der Zeit, mehr zum Thema Programmiersprachen im allgemeinen zu sagen, was den folgenden Abschnitten zufällt:

[21] Auf einem Seminar über Parallelprogrammierung – unter anderem unter `FORTRAN` –, an dem der Autor vor einigen Jahren teilnahm, wurde die Geschichte einer durchaus großen Firma erzählt, die vor der Aufgabe stand, ein bestehendes `FORTRAN`-Programm aus den späten sechziger Jahren, das Strömungsanalysen durchführte, auf eine neue Maschine zu portieren und dabei in größerem Umfange zu modifizieren. Nach einigen Monaten wurde klar, daß dies eine Aufgabe war, die ohne Hilfe des ursprünglichen Programmierers unlösbar war, da die Arbeitsweise des Programmes nicht nachzuvollziehen war. Als sich dann herausstellte, daß dieser Programmierer einige Zeit vorher bei einem Autounfall ums Leben kam, wurde das Projekt zugunsten einer völligen Neuentwicklung gestoppt.

Kapitel 3

Programmiersprachen

3.1 Grundsätzliches

Programmiersprachen stellen eines der interessantesten Gebiete der Informatik dar – in kaum einem anderen Bereich, die Domäne der Computerarchitektur ausgenommen, existieren so viele unterschiedliche Ideen, Ansätze und Sichtweisen. In kaum einem anderen Teilgebiet können persönliche Vorlieben in einem Maße Verbreitung und Anwendung finden, wie in Programmiersprachen. Wer eine Programmiersprache entwickelt, die Anklang findet und in immer weiterem Rahmen genutzt wird, hat, wie kaum ein anderer, die Möglichkeit (aber auch die Verantwortung, die sich hieraus ergibt), das Denken anderer in bestimmte Bahnen zu lenken, manchen Ideen und Sichtweisen Vorschub zu leisten, usf.

Der erste große Unterschied zwischen Programmiersprachen und natürlichen Sprachen, wie dem Deutschen oder dem Englischen, ist die Forderung, daß Programmiersprachen „präzise" sein müssen. Mehrdeutigkeiten sind nicht tolerabel, was sich allein aus den in Abschnitt 2.2 genannten Anforderungen an einen Algorithmus ergibt. Sowohl die Notation einer Programmiersprache, ihre „Syntax", als auch die Bedeutung der einzelnen Konstruktionen, die in einer gegebenen Programmiersprache möglich sind, die „Semantik", müssen eindeutig festgelegt sein.

Eine hervorragende Einführung in das Gebiet der Programmiersprachen, die sowohl die wichtigsten Konzepte und Sichtweisen vorstellt, als auch auf die Interna der jeweiligen Sprachen eingeht, findet sich in [24].

Im Gegensatz zu natürlichen Sprachen, die – zumindest nach Kenntnisstand des Autors – im großen und ganzen derselben Idee folgen, Dinge der realen und der Gedankenwelt mit Hilfe von Aneinenanderreihungen von Wörtern, die in bestimmtem Zusammenhang zueinander stehen, zu beschreiben, finden sich bei Programmiersprachen einige Beschreibungskonzepte, grundlegend unterschiedlicher Form, die sich in ihrer gesamten Sicht der Dinge voneinander unterscheiden: Hiervon seien die wichtigsten Sprachfamilien kurz genannt (auch hierbei sie bemerkt, daß die Grenzen zwischen den einzelnen Gruppen – wie so oft – fließend sind):

Imperative Programmiersprachen: In dieser Familie finden sich eine Reihe der am weitesten verbreiteten Programmiersprachen, wie `FORTRAN`, `C`, `BASIC`, `PASCAL`, `APL` und viele andere. Diese Sprachklasse wird auch als „von-Neumann Sprachfamilie" bezeichnet – alle Sprachen in ihr zeichnen sich dadurch aus, daß mit ihrer Hilfe Anweisungen gegeben werden, wie bestimmte Werte im Speicher einer Maschine zu ändern sind, was auch der Grund für die Bezeichnung „Berechnung durch Seiteneffekte" ist[1].

[1] Sprachen, die ohne solche Seiteneffekte auskommen, gehören der Familie der funktionalen Programmiersprachen an. Diese basieren auf der Idee von Ausdrücken, die Werte besitzen, anstelle von

Funktionale Programmiersprachen: Sprachen dieser Klasse beruhen auf der Idee der rekursiven Definierbarkeit von Funktionen – eine Idee, die eng mit dem sogenannten (von Alonzo Church entwickelten) „Lambda-Kalkül“[2] verbunden ist, beziehungsweise hierauf beruht. Prominentester Vertreter einer Sprache dieser Klasse ist `LISP`(siehe beispielsweise [20], [31] oder [28]).

Prädikative Programmiersprachen: Solche Sprachen beruhen auf einer prädikatenlogischen Sichtweise, unter ihnen findet sich als vielleicht prominentester Vertreter unter anderem `PROLOG` ([3]).

Datenflußprogrammiersprachen: Mit Hilfe von Sprachen dieser Familie können Algorithmen beschrieben werden, die zur Ausführung auf einem Datenflußrechner bestimmt sind, beziehungsweise einer Datenflußsichtweise entspringen. Datenflußrechner sind Spezialarchitekturen, in denen nicht Operationen in einer, vom Programm vorgegebenen Sequenz auf Werte angewandt werden, sondern das Vorhandensein von Werten die automatische Ausführung von Operationen auslöst. Für weitere Informationen zu dieser Klasse von Maschinen sei beispielsweise auf [6], S. 282 ff. resp. [19] für eine Originalarbeit verwiesen.

Objektorientierte Programmiersprachen: Sprachen dieser Klasse favorisieren eine objektorientierte Weltsicht (siehe hierzu unter anderem [27]) und sind in den vergangenen Jahren stark favorisiert worden. Bedeutendste Vertreter dieser Familie sind beispielsweise `C++`, `JAVA` und andere. Die Wurzeln dieser Entwicklung reichen bis zu `SIMULA-67` (siehe [22]) und damit in die späten sechziger Jahre zurück.

3.1.1 Maschinen- und Assemblersprachen

Für die folgenden Abschnitte ist ein kleiner Exkurs in die Computerarchitektur notwendig, um die nötigsten Grundbegriffe für Maschinen- und Assemblersprachen darzustellen. Der Einfachheit halber wird im weiteren stets von einer von-Neumann-Architektur, wie sie in Abschnitt 3.1.1.1 dargestellt wird, ausgegangen (für im Bereich der Computerarchitektur interessierte sei an dieser Stelle auf [6] beziehungsweise vor allem auf [8] verwiesen, die einen erschöpfenden Überblick des Gebietes bieten).

3.1.1.1 Die von Neumann-Architektur

Im Jahr 1944 begann John von Neumann, angeregt durch den Rechner ENIAC, sich mit Fragen des automatisierten Rechnens zu befassen – eine Tätigkeit, aus der das bahnbrechende Paper „Preliminary Discussion of the Logical Design of an Electronic Computing Instrument“ (siehe [2]) hervorging, in dem die Struktur der EDVAC-Maschine[3] vorgeschlagen wurde. EDVAC wurde zu einem der ersten praktisch einsetzbaren, speicherprogrammierbaren Computer und somit zum Stammvater einer Reihe von Maschinen, denen gemeinsam die sogenannte von Neumann-Architektur ist. Abbildung 3.1 zeigt nach [6], S. 33, die Grundstruktur einer solchen Maschine.

[6], S. 35, formuliert das Charakteristische der von Neumann-Maschine wie folgt: „Das von Neumann-Konzept des minimalen Hardwareaufwandes ist im Sinne einer von Saint-Exupery stammenden Definition das schlechthin Vollkommene: Es kann nichts davon weggelassen werden.“ Treffender läßt sich diese Form einer Computer-Architektur nicht beschreiben – gerade diese Einfachheit war es, die in der Frühzeit der elektronischen Rechner dazu führte, daß fast alle Maschinen der Anfangszeit diesem Architekturschema zumindest verpflichtet waren.

Die grundlegenden Einheiten einer solchen Maschine sind im einzelnen:

Anweisungen, die Werte beziehungsweise Speicherzellen modifizieren.

[2] Siehe hierzu [9], S. 207ff. beziehungsweise [20], S. 56, 177.

[3] *E*lectronic-*D*iscrete-*V*ariable-*A*utomatic-*C*omputer.

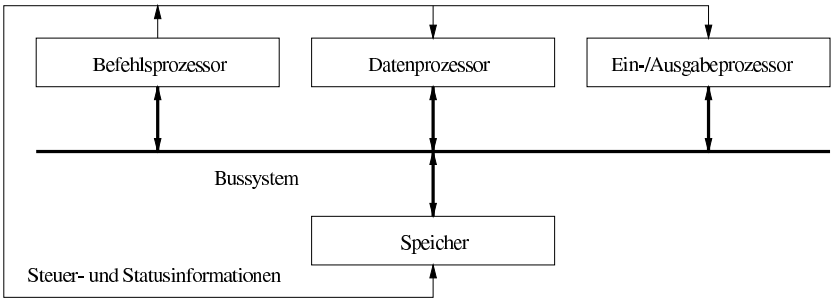

Abbildung 3.1: Eine einfache von Neumann-Maschine

- Der *Befehlsprozessor*, der die aus dem Speicher der Maschine gelesenen Befehle interpretiert und zur Ausführung an den
- *Datenprozessor* weitergibt, der für die eigentliche Verarbeitung von Daten zuständig ist.
- Dem *Ein-/Ausgabeprozessor* obliegen alle notwendigen Tätigkeiten, um mit externen Einheiten, wie beispielsweise Magnetband- oder -trommellaufwerken, kommunizieren zu können.
- Der *Speicher* dient nicht allein der Speicherung der zu verarbeitenden Daten sondern enthält auch das auszuführende Programm – ein Merkmal, das für die von Neumann-Architektur charakteristisch ist (es gibt Computerarchitekturen, die eine Trennung von Daten und Programm vorsehen, beispielsweise die sogenannte Harvard-Architektur).
- Zu nennen bleibt noch das *Bussystem*, über das alle Einheiten der Maschine miteinander in Verbindung stehen und Daten austauschen.

3.1.1.2 Ein IBM 650 Maschinenprogramm

Im folgenden wird ein kleines Programm für eine Maschine, die längst ihren Platz in den Museen gefunden hat, die berühmte IBM 650[4], entwickelt, mit dessen Hilfe die Summe zweier Werte, die in zwei Speicherzellen zur Verfügung gestellt werden, berechnet und in einer dritten Speicherzelle abgelegt wird[5].

Die IBM 650, die als Beispiel der folgenden Ausführungen dient, ist eine klassische Implementation eines von Neumann-Rechners. Zentrales Speicherelement hier ist eine Magnettrommel, d.h. eine schnell um ihre Achse rotierende Trommel, deren Wandung mit einer magnetisierbaren Schicht umgeben wurde. Entlang dieser rotierenden Trommel ist nun eine Reihe von Schreib-/Leseköpfen angeordnet, die, ähnlich einem Tonband, Muster unterschiedlicher Magnetisierung auf die Außenwand der Trommel schreiben beziehungsweise von ihr lesen können, während die Trommel selbst rotiert. Bild 3.2 zeigt schematisch die Konstruktion eines Magnettrommelspeichers.

Jeder der (im allgemeinen – auch hier existierten Ausnahmen) festmontierten Köpfe korrespondiert mit einer sogenannten „Spur“ auf der Magnettrommel; jede solche Spur kann eine bestimmte, feste Anzahl Daten (im Normalfall sogenannten „Wörter“ beziehungsweise „Maschinenwörter“) speichern. Besitzt eine Magnettrommel beispielsweise 40 Spuren, entsprechend 40 verschiedenen Schreib-/Leseköpfen, die jeweils 1000 Wörter fassen, so lassen sich auf der gesamten Trommel 40000 Maschinenwörter speichern.

[4] Eine wunderschöne Anlage dieser Baureihe steht beispielsweise im Deutschen Museum in München.
[5] Eingehendere Informationen zur IBM 650, insbesondere ihrer Programmierung, finden sich in [32].

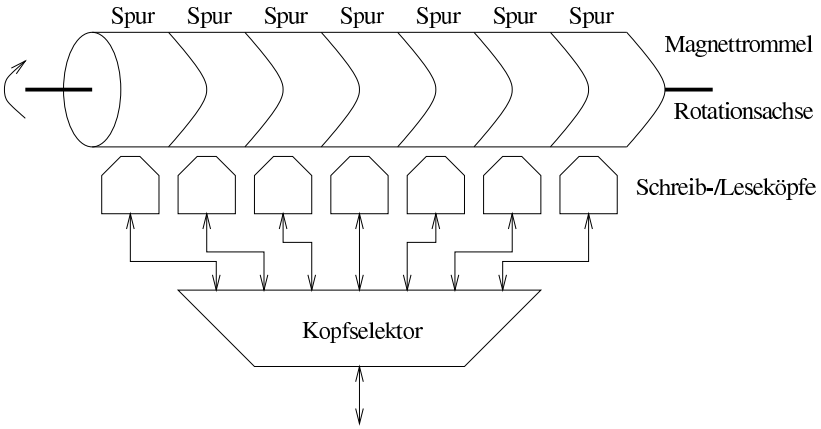

Abbildung 3.2: Ein Magnettrommelspeicher

Der Hauptvorteil der Magnettrommel als Speichermedium ist ihr (damals) vergleichsweise niedriger Preis – allerdings geht dieser Vorteil Hand in Hand mit einem verhältnismäßig schwerwiegenden Nachteil: Um eine bestimmte „Speicherzelle" auf der Trommel ansprechen, d.h. lesen oder schreiben zu können, muß nicht nur der entsprechende Kopf der Spur ausgewählt werden, auf der das gesuchte Wort liegt, vielmehr muß abgewartet werden, bis die gewünschte Speicherzelle durch die Rotation der Trommel unter dem Kopf vorbeieilt, um auf sie zugreifen zu können.

Eine solche Magnettrommel stellt den Hauptspeicher der IBM 650 dar – in diesem Falle handelte es sich um eine Hochgeschwindigkeitstrommel, die mit 12500 Umdrehungen pro Minute rotierte und 2000 Speicherzellen zur Verfügung stellte, wobei jede Zelle einen zehnstelligen Dezimalwert aufzunehmen im Stande war[6].

Im folgenden soll nun die Gleichung

$$x = a + b \tag{3.1}$$

für bekanntes, d.h. vorgegebenes a und b gelöst werden. Hierzu sei vorausgesetzt, daß die Speicherzellen 0100 und 0101 die Werte für a resp. b bereits enthalten; in Zelle 0102 ist das Ergebnis der Addition, x, abzulegen. Die folgende Speicherbelegungsliste zeigt ein Maschinenprogramm zur Berechnung der Addition für den Fall $a = 12$ und $b = 34$:

Speicherzelle	Inhalt
0100	0000000012
0101	0000000034
0102	??????????
0103	60 0100 0104
0104	10 0101 0105
0105	21 0102 ...

Wie bereits erwähnt, dienen die Speicherzellen 0100, 0101 und 0102 der Aufnahme der Rechenvariablen a, b und x entsprechend Gleichung 3.1, während die drei folgenden Speicherzellen die notwendigen Anweisung an den Computer enthalten, um die eigentliche Addition vornehmen zu können. Die ungewohnte Schreibweise der „Instruktionswerte" in diesen Speicherzellen rührt aus der Befehlsstruktur der IBM 650 her. Die beiden linken Ziffern beschreiben die auszuführende Instruktion, so entspricht etwa der

[6] Hier findet sich zugleich ein hervorragendes Beispiel eines Computers, der nicht binär, sondern im Dezimalsystem rechnet.

Wert 60 in diesen Stellen der Instruktion *Reset and Add upper* – hiermit wird einer der sogenannten „Akkumulatoren“ der Maschine, in denen im allgemeinen die eigentlichen Berechnungen stattfinden[7], auf Null gesetzt und anschließend um den Wert, der sich in Speicherzelle 0100 findet, d.h. um a erhöht. Nach dieser ersten Instruktion befindet sich in diesem Akkumulator der Wert von a. Die Instruktion 10 0101 0105 in Speicherzelle 0104 addiert nun den Inhalt der Speicherzelle 0101 zu diesem Akkumulator, so daß an dieser Stelle bereits die Addition $a + b$ ausgeführt wurde. Im letzten Schritt muß der solchermaßen erhaltene Wert noch in Speicherzelle 0102 abgespeichert werden, was durch die Instruktion 21 0102 ... in Speicherzelle 105 vollzogen wird.

Das Befehlsformat der IBM 650 dürfte nach diesem Beispiel zu einem großen Teil offenliegen – die ersten beiden Ziffern einer Instruktion kennzeichnen die Instruktion selbst, während die folgenden vier Ziffern die Speicherzelle enthalten, mit deren Daten operiert werden soll. Unerwähnt geblieben ist bislang jedoch die Funktion der letzten vier Ziffern, in denen sich scheinbar stets die Nummer der nächsten Speicherstelle findet, was auch tatsächlich ihre Funktion ist. Die letzten vier Ziffern einer jeden Instruktion der IBM 650 geben die Nummer der Speicherzelle an, aus der die nächste Instruktion gelesen werden soll – ein Mechanismus, der auf den ersten Blick unnötig kompliziert erscheint, vor allem, da das vorliegende Beispiel stets die nächsthöhere Adresse als Folgeadresse verwendet. Im Prinzip könnte auf dieses Feld verzichtet werden und immer automatisch die nächste Speicherzelle gelesen werden, sobald eine Instruktion vollständig abgearbeitet wurde.

Der Grund für diese Form der „Adressierung“ der jeweils nachfolgenden Instruktion in den letzten vier Stellen einer jeden Instruktion liegt in der Form des verwendeten Hauptspeichermediums der IBM 650 begründet. Da hier eine Magnettrommel als zentrales Speicherelement zum Einsatz kommt, liegt allen Speicherzugriffen zunächst der unabänderliche Rhythmus der Trommelrotation zugrunde. Angenommen, der Rechner führt die in Speicherzelle 0103 enthaltene Instruktion aus. In der Zeit, die allein zur Dekodierung der eigentlichen Instruktion benötigt wird, wurde die Trommel ein Stück weit unter den Schreib-/Leseköpfen hinwegbewegt, so daß nach erfolger Bearbeitung der Instruktion im allgemeinen nicht die direkt folgende Speicherzelle 0104 unter den Köpfen zu liegen kommt. In diesem Fall muß unter Umständen eine volle Umdrehung der Trommel abgewartet werden, bis die gewünste Speicherzelle 0104 gelesen werden kann, um die nächste Instruktion ausführen zu können[8].

Im Idealfall sorgte ein Programmierer der damaligen Zeit dafür, daß die Instruktionen und Daten eines auszuführenden Programmes nicht in konsekutiven Speicherzellen zu liegen kamen, sondern im Bestfall so geschickt auf den einzelnen Spuren und Speicherstellen der Magnettrommel verteilt wurden, daß zwischen zwei Zugriffen auf die Trommel möglichst wenig Wartezeit aufgewendet werden mußte. Eine realistischere Variante des obigen Programms hätte also beispielsweise folgende Gestalt:

Speicherzelle	Inhalt
0100	0000000012
0253	0000000034
0783	??????????
0167	60 0100 0387
0387	10 0253 0634
0634	21 0783 ...

Dieses Programm führt dieselben Operationen wie das vorangegangene aus, greift

[7] In diesem Fall der obere Akkumulator, was jedoch für die folgenden Ausführungen ohne Belang ist.

[8] Nicht berücksichtigt ist hierbei, daß dieselbe Problematik natürlich auch für den Zugriff auf Speicherzellen gilt, die Daten enthalten – allein die Verwendung der Speicherzelle 0100 in der Instruktion, die in Speicherzelle 0103 liegt, ist aus Performancesicht nicht günstig, da diese Speicherzelle vermutlich vergleichsweise weit entfernt liegt, so daß der Zugriff auf sie mit einer nicht zu vernachlässigenden Wartezeit verbunden ist.

für die Variablen a, b und x jedoch nicht auf die Speicherzellen 0100, 0101 und 0102, sondern auf 0100, 0253 und 0783 zu. Entsprechend liegen auch die einzelnen Instruktionen nicht in direkt aufeinanderfolgenden Speicherzellen, sondern wurden so auf die Trommel verteilt, daß zwischen zwei aufeinanderfolgenden Zugriffen möglichst wenig Wartezeit verstreicht, da sich die Trommel im Idealfall nach jeder Instruktion bereits in der Nähe der nachfolgenden Instruktion bzw. verwendeten Speicherzelle befindet.

Eine solche „direkte" Programmierung eines Computers wird Programmierung in „Maschinensprache" genannt, da hier direkt vom Programmierer die einzelnen Ziffernfolgen, wie sie für die Darstellung der Instruktionen eines Programmes benötigt werden, erstellt werden müssen – dies ist (im allgemeinen – auch hier, wie immer, finden sich Ausnahmen, beispielsweise im Bereich mikroprogrammierter Maschinen) die direkteste Form, einen Computer zu programmieren. Von Vorteil bei dieser Variante der Programmierung ist in erster Linie eben diese Nähe zur Maschine, welche – bei hinreichender Erfahrung – die Erstellung ausgesprochen effizienten Codes ermöglicht.

Nachteilig hingegen ist jedoch die Tatsache, daß eine solche direkte Programmierung in Maschinensprache nicht nur sehr zeitaufwendig (nicht nur bei der eigentlichen Codierung, sondern auch beim Training der Programmierer) ist, sondern vor allem sehr fehleranfällig. Aus diesem Grunde wurden in den frühen fünfziger und sechziger Jahren vermehrt sogenannte „Assembler" entwickelt, die einen ersten Schritt zur einfacheren Programmierung von Computer darstellten.

3.1.1.3 Ein Assemblerprogramm für die IBM 650

Der folgende Abschnitt zeigt die Formulierung des Beispielprogrammes aus Abschnitt 3.1.1.2 mit Hilfe eines sogenannten „Assemblers". Hierbei handelt es sich um ein Programm, das den Programmierer von einem Großteil der Tätigkeiten, wie sie bei der direkten Programmierung in Maschinensprache auftreten, befreit. Hierunter fällt beispielsweise die Möglichkeit, anstelle direkter Instruktionscodes, wie beispielsweise `10 ???? ????` für eine Addition, leichter zu merkende Abkürzungen, sogenannte „Mnemonics" zu verwenden. Die durch `10 ???? ????` spezifizierte Addition ließe sich in obigem Beispiel unter Verwendung eines Assemblers[9] in Form der „symbolischen" Instruktion `AUP` (kurz für `Add-UPper`) notieren – sicherlich leichter zu erinnern als die einleitende Ziffernfolge `10` der eigentlichen Maschineninstruktion.

Ein Assembler leistet im allgemeinen jedoch mehr als nur die Möglichkeit, symbolische Instruktionsbezeichnungen verwenden zu können. Eine der Hauptaufgaben eines solchen Programmes besteht darin, den Programmierer von der Verwaltung der Speicherzellen zu entlasten. Mit Hilfe eines Assemblers ist es nicht notwendig, von vornherein festzulegen, daß die Werte für a, b und x aus Gleichung 3.1 in den Speicherzellen 0100, 0253 und 0783 zu liegen kommen sollen. Vielmehr bietet er die Möglichkeit, sogenannte „symbolische Adressen" zu verwenden: Hiermit können Speicherzellen „Namen" zugeordnet werden, unter denen sie im weiteren Verlauf des Programmes angesprochen werden können. So ließe sich der Speicherzelle, in der der Wert der Variablen a abgelegt werden soll, der Name (die symbolische Bezeichnung) `A` zuordnen, um im weiteren Verlauf anstelle ihrer wahren Adresse nur diesen Namen spezifizieren zu müssen, was die Les- und vor allem Wartbarkeit solcher Programme ungemein verbessert. Ein solcher Name für eine Speicherzelle wird auch als „Label" bezeichnet.

Das Beispiel aus Abschnitt 3.1.1.2 ließe sich mit Hilfe von SOAP nun folgendermaßen formulieren:

[9] Zu Zeiten der IBM 650 hieß der verwendete Assembler *SOAP*, die Kurzform für „Symbolic Optimal Assembly Program".

Label	Operation	D-Adresse	I-Adresse
	SYN	A	0100
	SYN	B	0253
	SYN	X	0783
	RAU	A	
	AUP	B	
	STU	X	????

Das (in diesem Beispiel unbenutzte) Feld `Label` dient zur Aufnahme symbolischer Bezeichnungen für die Speicherzellen, in denen die zugeordnete Instruktion abgelegt ist, um beispielsweise Sprünge zu bestimmten Instruktionen zu ermöglichen.

Das Operationsfeld enthält hierbei jeweils den Namen der entsprechenden SOAP-Operation, so steht beispielsweise `STU` für STore-Uper. Anweisungen wie `RAU`, `AUP` oder `STU` werden vom Assembler automatisch in die entsprechenden Ziffernfolgen zur Steuerung des Computers übersetzt. Im Gegensatz zu solchen „wirklichen" Instruktionen existieren sogenannte „Pseudoinstruktionen", wie beispielsweise `SYN`, die nicht direkt zu ausführbarem Code führen, sondern stattdessen der Steuerung des Assemblers dienen. So kann mit Hilfe der Pseudoinstruktion `SYN` ein Synonym für eine Speicherzelle vereinbart werden – in obigem Beispiel werden hiermit drei Synonyme `A`, `B` und `X` für die Speicherzellen 0100, 0253 und 0783 vereinbart.

Die Spalte `D-Adresse` enthält Adressangaben, mit deren Hilfe die bei der Ausführung einer Instruktion zu verwendenden (Daten-)Speicherzellen adressiert werden, während in der Spalte `I-Adresse` bei Bedarf eine explizite Folgeadresse spezifiziert werden kann, falls nicht einfach mit der folgenden Instruktion fortgefahren werden soll. Zu beachten ist in diesem Zusammenhang, daß bereits der SOAP-Assembler in der Lage war, die Instruktionen eines Programmes automatisch auf der Magnettrommel in einer Form zu verteilen, die eine (nahezu) minimale Ausführungszeit garantierte. Dies hatte zur Folge, daß Instruktionen, die in einem in der SOAP-Assemblersprache formulierten Programm direkt aufeinander folgten, nicht notwendigerweise auch auf der Magnettrommel, d.h. im Hauptspeicher, aufeinander folgen mußten – dies war vielmehr die Ausnahme. Die Erleichterung, die dies für den Vorgang des Programmierens bedeutete, kann nicht überschätzt werden – gerade die geschickte Vergabe von Speicheradressen war ein Vorgang, der nur mit ausgesprochen viel Übung und Erfahrung aufgeführt werden konnte und in den meisten Fällen zu Code führte, der trotz allem im Hinblick auf notwendige Wartezeiten hinter von SOAP generiertem Code zurückstand.

Im allgemeinen liest ein Assembler das geschriebene Assemblerprogramm, um es in einer Reihe von Schritten (einfache Assembler benötigen nur einen einzigen Schritt) in entsprechenden Maschinencode zu verwandeln, wobei alle notwendigen Adressberechnungen, unter Umständen die Verarbeitung sogenannter „Makros" und eventuell auch Optimierungen vorgenommen werden.

Ein Programm wie das oben formulierte ist sicherlich wesentlich leichter lesbar als seine direkte Repräsentation in Maschinensprache, was im Lauf nur weniger Jahre dazu führte, daß ein Großteil des Codes für Computer nur noch mit Hilfe von Assemblern geschrieben wurde, die im Verlauf dieser Entwicklung zu mächtigen Werkzeugen entwickelt wurden, deren Möglichkeiten weit über das, was hier am Beispiel SOAP angedeutet wurde, hinausgehen.

Der Hauptvorteil einer direkten Programmierung eines Computers in Maschinen- beziehungsweise Assemblersprache liegt vor allem in den hierdurch erzielbaren – unter Umständen extrem niedrigen – Ausführungszeiten, was zur Folge hat, daß auch heutzutage ein gewisser, wenn auch ausgesprochen kleiner Teil an Programmen nach wie vor in Assembler codiert wird. Hierunter fallen im allgemeinen in erster Linie zeitkritische Anwendungen, wie zum Beispiel Gerätetreiber, Eventhandler und anderes, aber auch Laderoutinen, etc.

3.1.2 Compilersprachen

Wie bereits erwähnt, besteht die Hauptarbeit eines Assemblers in der Übersetzung von Code, dessen Instruktionen in Form sogenannter Mnemonics notiert werden, in von der Zielmaschine direkt ausführbaren Maschinencode, wobei der Programmierer von mühsamen und fehleranfälligen Adressrechnungen und Optimierungsaufgaben im allgemeinen weitgehend entlastet wird.

Trotz dieser Hilfestellungen, die ein Assembler zu geben in der Lage ist, bleibt die Schwierigkeit, ein Programm auf einem sehr maschinennahen Niveau formulieren zu müssen – im Grunde genommen können nur die Instruktionen Anwendung finden, die von der Zielarchitektur direkt ausgeführt werden können, auch ist eine detaillierte Kenntnis der genauen Struktur des zu programmierenden Rechners notwendig (hierunter fallen beispielsweise Dinge wie der obere (upper) und untere (lower) Akkumulator der IBM 650, ganz allgemein jedoch auch die Form der Zahlenrepräsentation, Wortlänge, Rechengenauigkeit, usf.

Im allgemeinen gilt, daß ein Assembler (oder Maschinenprogramm), das für eine bestimmte Zielhardware entwickelt wurde, nur auf dieser ablauffähig ist – eine Portierung auf eine andere Maschinenarchitektur läuft in der Mehrzahl der Fälle auf eine Neuimplementation hinaus, ein Vorgehen, das im allgemeinen zu zeitaufwendig ist, um wirklich in Betracht gezogen zu werden.

Aus diesen Gründen kam sehr frühr die Idee auf, sogenannte „Hochsprachen“ zu schaffen, mit deren Hilfe Algorithmen möglichst maschinenunabhängig formuliert werden können, so daß sich verhältnismäßig gut wartbare Programme ergeben, während gleichzeitig die Entwicklungszeit verkürzt wird, da zum einen keine dermaßen tief reichende Kenntnis der Maschinenarchitektur seitens der Programmierer vorausgesetzt werden muß, andererseits die Formulierung des Algorithmus auf einem Niveau erfolgen kann, das im allgemeinen dem Problem näher als dem Computer steht. So wäre es beispielsweise wünschenswert, in einem Programm den Ausdruck `X = A + B` niederschreiben zu können, ohne Punkten wie der Anzahl und Größe der vorhandenen Akkumulatoren (Register) oder der Speicherbelegung etc. Aufmerksamkeit schenken zu müssen. Im Idealfall sollte ein „Übersetzerprogramm“, das aus solchem Code letztlich wieder Maschinenprogramme erzeugt, in der Lage sein, eine gewisse Funktionsweise beispielsweise der Addition + unabhängig von den wirklichen Gegebenheiten der Zielarchitektur garantieren zu können. Ist beispielsweise die zu programmierende Maschine nicht in der Lage, eine Addition der geforderten Genauigkeit auszuführen, sollte vom Übersetzer automatisch Maschinencode erzeugt werden, der eine entsprechende Funktionalität zur Verfügung stellt, ohne daß hierfür unter Umständen notwendige Zusatzoperationen für den Programmierer direkt sichtbar werden.

Solche Übersetzerprogramme werden im allgemeinen als „Compiler“ bezeichnet. Wie ein Assembler lesen sie ein Programm, das in Form eines sogenannten „Quellcodes“ vorliegt, ein und erzeugen im Verlauf mehrer Verarbeitungsschritte ein diesem Programm semantisch äquivalentes Maschinenprogramm, das auf der Zielmaschine ausgeführt werden kann.

Eine der ersten solchen höheren Programmiersprachen war `FORTRAN`, dessen Wurzeln bis in die späten fünfziger Jahre zurückreichen; der Name `FORTRAN` – die Kurzform für `FOR`mula `TRAN`Slation – deutet das geplante Haupteinsatzgebiet dieser Sprache bereits an: Mit ihr sollte in erster Linie ein Werkezug geschaffen werden, um Formeln einfach und schnell zur Berechnung auf einem Computer formulieren zu können. So konnte bereits in frühen `FORTRAN`-Varianten folgendes Programm formuliert werden[10]:

```
ACCEPT TAPE 2, A, B
X = A + B
```

[10] Hierbei handelt es sich um 1620 `FORTRAN` – einen frühen `FORTRAN`-Dialekt, der auf dem Kleinrechner IBM 1620 zum Einsatz kam (siehe hierzu [17], Kapitel 15).

```
      PUNCH TAPE 3, X
```

In Erweiterung der Beispiele aus den Abschnitten 3.1.1.2 und 3.1.1.3, in denen die Werte für A und B vorgegeben waren und zu Beginn des Programmes in reservierte Speicherzellen geschrieben wurden, erlaubt dieses kurze FORTRAN-Programm bereits das Einlesen von Werten für A und B sowie die Ausgabe von Ergebnissen. Das Kommando ACCEPT TAPE liest vom Lochstreifenleser Nummer 2 die gewünschten Werte für die beiden Variablen, deren Summe in der nächsten Zeile durch X = A + B berechnet wird. Die Ausgabe der Summe erfolgt mit Hilfe des Ausgabekommandos PUNCH TAPE auf den Lochstreifenstanzer Nummer 3.

Aus heutiger Sicht mögen (nicht zuletzt) die Ein-/Ausgabemöglichkeiten, die dieses FORTRAN-Programm verwendet, rudimentär sein, zur damaligen Zeit (d.h. zu Beginn der sechziger Jahre) war es jedoch ein wirklicher Fortschritt, daß einfach Werte von Lochstreifen oder Magnetband gelesen und auch auf diese Medien geschrieben werden konnten, ohne jedes Mal erneut entsprechende Assemblerroutinen schreiben zu müssen, um überhaupt einen Zugriff auf solche I/O-Geräte zu ermöglichen. Ein vergleichbares Maschinenprogramm wäre ungleich länger und würde ein Vielfaches an Entwicklungs- und Testzeit erfordern.

FORTRAN ist nicht nur eine der ältesten Hochsprachen, sondern auch eine der am weitverbreitetsten und langlebigsten – was sicherlich nicht zuletzt daran liegt, daß Programme, die in dieser Sprache formuliert wurden, in der Mehrzahl aller Fälle ausgesprochen leicht von einer Maschine zu einer anderen portiert werden können. Entsprechend seiner ursprünglichen Zielsetzung, die Formulierung mathematisch orientierter Algorithmen zu vereinfachen, findet FORTRAN bis heute seine größte Verbreitung im Bereich des wissenschaftlich-technischen Rechnens. Vor allem im Bereich des Supercomputings wird FORTRAN – trotz seiner mittlerweile teilweise etwas archaisch anmutenden Wurzeln – nach wie vor in großem Maße eingesetzt, da die meisten Supercomputerhersteller hervorragende FORTRAN-Compiler in ihrem Programm haben, die ausgesprochen guten Maschinencode für ihre jeweiligen Spezialarchitekturen erzeugen.

3.1.2.1 Programmentwicklung mit Compilersprachen

Im allgemeinen besteht die Entwicklung eines Programmes unter Zuhilfenahme einer compilierten Hochsprache, wie beispielsweise FORTRAN oder auch C, aus einer Reihe von Schritten, die im folgenden kurz dargestellt werden:

- Zunächst ist der zu implementierende Algorithmus zu entwickeln – ein Vorgang, der im Prinzip für alle Sprachen der gleiche ist.
- Ausgehend von diesem Algorithmus wird nun ein Programm in der gewählten Hochsprache, beispielsweise C, formuliert.
- Dieses Programm wird nun mit Hilfe eines Compilers in Maschinencode übersetzt, der auf der Zielmaschine direkt ausgeführt werden kann.

Im allgemeinen deckt der eigentliche Übersetzungsschritt unter Umständen Codierfehler innerhalb des Quellcodes auf, die der Nachbearbeitung bedürfen, bevor der Übersetzungsvorgang erfolgreich abgeschlossen werden kann. Die Programmentwicklung mit Hilfe von Compilersprachen stellt in den meisten Fällen einen Zyklus dar – ausgehend von einer ersten Version des Quellcodes werden mehrere Iterationen durchlaufen, innerhalb derer das Programm von ersten Fehlern bereinigt wird, bis zum ersten Mal Maschinencode erzeugt werden kann. Dieser wird auf dem Zielcomputer zur Ausführung gebracht, wobei sich meist weitere Fehler zeigen, die zu einem großen Teil auf Planungsfehler bei der Erstellung des Algorithmus selbst oder auf Fehler bei der Implementation des Algorithmus in der gewählten Hochsprache zurückzuführen sind. Ausgehend von den hierbei

erkannten Fehlern kann nun die nächste Version des Programmes erstellt werden, wobei unter Umständen neue Codier-, Planungs- oder Implementationsfehler erzeugt werden, deren Beseitigung wieder iterativ erfolgt.

Ein wichtiger Punkt im Zusammenhang mit Compilersprachen ist der Vorgang des sogenanten „Bindens“ oder auch „Linkens“, der im folgenden kurz dargestellt wird:

3.1.2.1.1 Linken Im allgemeinen benötigt ein in einer Hochsprache formuliertes Programm zusätzliche Routinen, um beispielsweise auf Ein-/Ausgabegeräte zugreifen zu können, wie dies im vorangegangenen Beispiel mit Hilfe der Statements `ACCEPT` beziehungsweise `PUNCH` vollzogen wurde. Solche Operationen, aber auch komplexe mathematische Funktionen und vieles mehr fällt nicht in den eigentlichen Umfang einer Hochsprache, da hierbei stark maschinenabhängiger Code benötigt wird. Beispielsweise erfolgt der Zugriff auf einen IBM Lochstreifenstanzer vollkommen anders als auf ein entsprechendes Gerät eines anderen Herstellers. Dennoch sollen solche Unterschiede nach Möglichkeit vor dem Programmierer verborgen werden, um nicht unnötig von der Implementation des eigentlich Algorithmus auf letztlich nebensächliche Maschinenspezifika abzulenken.

Aus diesem Grunde hat es sich eingebürgert, daß ein Betriebssystem in den meisten Fällen eine Sammlung vorgefertigter Routinen mitbringt, mit deren Hilfe die verschiedensten Ein-/Ausgabeoperationen, etc. durchgeführt werden können. Eine solche Routinensammlung wird meist als „Bibliothek“ (auch „Library“) bezeichnet – ohne sie, beziehungsweise die in ihr enthaltenen Routinen, ist ein Programm in der Regel nicht lauffähig, da ihm jegliche Möglichkeit zur Interaktion mit seiner Umgebung (sowie vieles mehr) fehlt.

An die erfolgreiche Erzeugung von Maschinencode aus einem im Quelltext vorliegenden Programm mit Hilfe eines Compilers schließt sich in den allermeisten Fällen der Prozess des Linkens an, der das generierte Maschinenprogramm mit allen benötigten Bibliotheksfunktionen vereinigt und so erst ein wirklich ausführbares Programm schafft.

3.1.3 Interpretersprachen

Einen grundsätzlich anderen Weg als Compilersprachen gehen die sogenannten Interpretersprachen, die ebenfalls die Möglichkeit bieten, ein Programm weitgehend unabhängig von den tatsächlichen Maschinengegebenheiten formulieren zu können, aber darauf beruhen, nicht ein einziges Mal, wie dies bei Compilersprachen der Fall ist, ausführbaren Maschinencode zu erzeugen, sondern das betreffende Hochsprachenprogramm quasi im laufenden Betrieb zu analysieren (interpretieren) und stückweise zur Ausführung zu bringen. An die Stelle des Compilers tritt bei Interpretersprachen der sogenannte „Interpreter“, der das auszuführende Programm liest und jede Anweisung im Moment ihres Auftretens in eine entsprechende Folge von Aktionen übersetzt.

Erzeugt ein Compiler ausgehend von einem Programm im Quelltext ein hierzu äquvalentes Maschinenprogramm, das anschließend jederzeit ausgeführt werden kann, ist ein in einer Interpretersprache geschriebenes Programm ohne den Interpreter selbst nicht ausführbar, da es von diesem zum Zeitpunkt der Ausführung erst interpretiert wird!

3.1.4 Kurzer Vergleich Compiler-/Interpretersprachen

Beide vorgestellten Möglichkeiten, einen Algorithmus in einer Hochsprache zu formulieren und auszuführen – zum einen unter Zuhilenahme eines Compilers, zum anderen mit Hile eines Interpreters, weisen spezifische Vor- und Nachteile auf, die eine einfache Entscheidung für eine Variante im allgemeinen schwer fallen lassen und im folgenden zumindest exemplarisch dargestellt werden sollen:

Vorteile von Compilersprachen: Der Hauptvorteil einer Compilersprache ist sicherlich in der zumeist ausgesprochen niedrigen Ausführungszeit des erzeugten Maschinencodes zu sehen, da der meist ausgesprochen aufwendige Übersetzungsvorgang prinzipiell nur ein einziges Mal ablaufen muß und zudem die Möglichkeit besteht (da das gesamte Programm dem Compiler zum Übersetzungszeitpunkt bekannt ist) in großem Maße Optimierungsfunktionen ausführen zu können, um die Qualität des erzeugten Codes hinsichtlich seiner Ausführungszeit (oder seines Platzbedarfes, etc.) weiter zu steigern.

Nachteile von Compilersprachen: Die Entwicklung korrekter (oder zumindest nahezu korrekter) Programme unter Einsatz von Compilersprachen ist aufgrund der quasi verzögerten Ausführung etwas erschwert, da der Aufwand von Änderungen vergleichsweise hoch ist, muß doch (wieder nur im allgemeinen) der modifizierte Quelltext erneut übersetzt (und gelinkt) werden, bevor er für einen weiteren Testlauf ausgeführt werden kann. Hierbei sind die Auswirkungen von Änderungen für den Programmierer nicht in einer derart direkten Art und Weise ersichtlich, wie dies bei Interpretersprachen der Fall ist.

Vorteile von Interpretersprachen: Größter Vorteil der Interpretersprachen ist die Nähe zum Programmierer – am Quelltext vorgenommene Änderungen wirken sich im allgemeinen sofort aus und sind somit leichter abzuschätzen und durchzuführen.

Nachteile von Interpretersprachen: Diesem Vorteil steht jedoch als größter Nachteil die vergleichsweise ungünstige Ausführungszeit solcher interpretierten Programme gegenüber, da der Vorgang des Übersetzens Zeile für Zeile (in konkreten Implementation solcher Sprachen mag dies nicht wörtlich gelten) stattfindet – gegebenenfalls wieder und wieder, falls beispielsweise eine Schleife im Quelltext verarbeitet werden muß.

Diese – einander widersprechenden – Vor- und Nachteile der beiden genannten Sprachfamilien führten vor einigen Jahren dazu, Sprachen zu propagieren, für die sowohl ein Interpreter als auch ein Compiler existierte, so daß während der Programmentwicklung auf die Hilfe des Interpreters zurückgegriffen werden konnte, um, sobald das Programm hinreichend stabil und korrekt war, mit einem abschließenden Compilerlauf direkt ausführbaren Maschinencode zu erzeugen und auf diese Art und Weise die Vorteile beider Ansätze miteinander zu vereinigen.

Durch die mittlerweile erreichte Leistungsfähigkeit auch kleiner, erschwinglicher Computer (beispielsweise unter LINUX) schwand der Nachteil hoher Zykluszeiten bei der Entwicklung von Programmen mit Hilfe von Compilersprachen zusehends dahin, während auf der anderen Seite die Kluft hinsichtlich der Ausführungszeiten von interpretiertem und compiliertem Code nahezu konstant blieb, so daß mittlerweile in der Mehrzahl aller Fälle ausschließlich auf die Hilfe eines Compilers bei der Programmentwicklung zurückgegriffen wird.

Literaturverzeichnis

[1] Jürgen Alex, „Wege und Irrwege des Konrad Zuse“, *Spektrum der Wissenschaft*, 1/1997, S. 78ff.

[2] Arthur W. Burks, Herman H. Goldstine, John von Neumann, „Preliminary Discussion of the Logical Design of an Electronic Computing Instrument“, in A. H. Taub (ed.), *Collected Works of John von Neumann*, Vol. 5, Maxmillan, New York, 1963, pp 34...79.

[3] W. F. Clocksin, C.S. Mellish, *Programming in Prolog*, Third, Revised and Extended Edition, Springer-Verlag, 1987.

[4] Edward Cohen, *Programm in the 1990s, An Introduction to the Calculus of Programs*, Spinger-Verlag, New York, 1990.

[5] Dr. Dobb's Journal, C-*Tools*, Markt&Technik Verlag, 1986.

[6] W. K. Giloi, *Rechnerarchitektur*, Springer-Verlag, 1981.

[7] J. Heinhold, U. Kulisch, *Analogrechnen*, Bibliographisches Institut, Mannheim/Wien/Zürich, 1969.

[8] John L. Hennessy, David A. Patterson, *CCOMPUTER ARCHITECTURE A QUANTITATIVE APPROACH*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[9] Hans Hermes, *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit – Einführung in die Theorie der rekursiven Funktionen*, Springer-Verlag, Dritte Auflage, 1978.

[10] Kurt Huber, *Leibniz, Der Philosoph der universalen Harmonie*, R. Piper Gmbh & Co. KG, München, 1989.

[11] Kai Hwang, *Computer Arithmetic, PRINCIPLES, ARCHITECTURE, AND DESIGN*, John Wiley & Sons, Inc., 1979.

[12] Georges Ifrah, *Universalgeschichte der Zahlen*, Campus Verlag, Frankfurt / New York, 1981.

[13] Robert Kaplan, *Die Geschichte der Null*, Campus Verlag, Frankfurt / New York, 1999.

[14] Donald E. Knuth, *The Art of Computer Programming*, Volume 1, „Fundamental Algorithms“, Second Edition, Addison-Wesley, 1997.

[15] Donald E. Knuth, *The Art of Computer Programming*, Volume 2, „Seminumerical Algorithms“, Second Edition, Addison-Wesley, 1997.

[16] Donald E. Knuth, *The Art of Computer Programming*, Volume 3, „Sorting and Searching“, Second Edition, Addison-Wesley, 1997.

[17] Daniel N. Leeson, Donald L. Dimitry, *BASIC PROGRAMMING CONCEPTS AND THE IBM 1620 COMPUTER*, Holt, Rinehart and Winston, Inc., 1962.

[18] Oskar Mahrenholz, *Analogrechnen in Maschinenbau und Mechanik*, Bibliographisches Institut, Mannheim/Zürich, 1968.

[19] David P. Misunas, *A Computer Architecture for Data-Flow Computation*, Laboratory for Computer Science, Massachusetts Insitute of Technology, MIT/LCS/TM-100, 1978.

[20] Dieter Müller, *LISP, Eine elementare Einführung in die Programmierung nichtnumerischer Aufgaben*, Bibliographisches Intitut, Mannheim/Wien/Zürich, 1985.

[21] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, *Numerical Recipes in Pascal*, Cambridge University Press, 1989.

[22] Helmut Rohlfing, *SIMULA, Eine Einführung*, Bibliographisches Institut Mannheim/Wien/Zürich, 1973.

[23] R. Rojas, *Die Rechenmaschinen von Konrad Zuse*, Springer-Verlag, 1998.

[24] Michael L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann Publishers, San Franciso, California, 2000.

[25] Robert Sedgewick, *Algorithmen in C*, Addison-Wesley, 1998.

[26] Herbert S. Sobel, *introduction to digital computer design*, Addison-Wesley Publishing Company, 1970.

[27] Jag Sodhi, Prince Sodhi, *Object-Oriented Methods for Software Development*, McGraw-Hill, 1996.

[28] Guy L. Steele jr., *COMMON LISP – THE LANGUAGE*, Digital Equipment Corporation, 1984.

[29] Josef Stoer, *Numerische Mathematik 1*, Fünfte Auflage, Springer-Verlag, 1989.

[30] Bernd Ulmann, *Algorithmen, Datenstrukturen und Programmieren – in* `C` *– für Katzen*, Skript für die Vorlesung „Algorithmen und Datenstrukturen“ an der VWA, WS1999, WS2000.

[31] Patrick Henry Winston, Berthold Klaus Horn, *LISP*, Addison-Wesley, 1987.

[32] Marshal H. Wrubel, *A Primer of Programming for Digital Computers*, McGraw-Hill Book company, Inc., 1959.

[33] Konrad Zuse, *Der Computer – Mein Lebenswerk*, Springer Verlag, 2. Auflage, 1986.