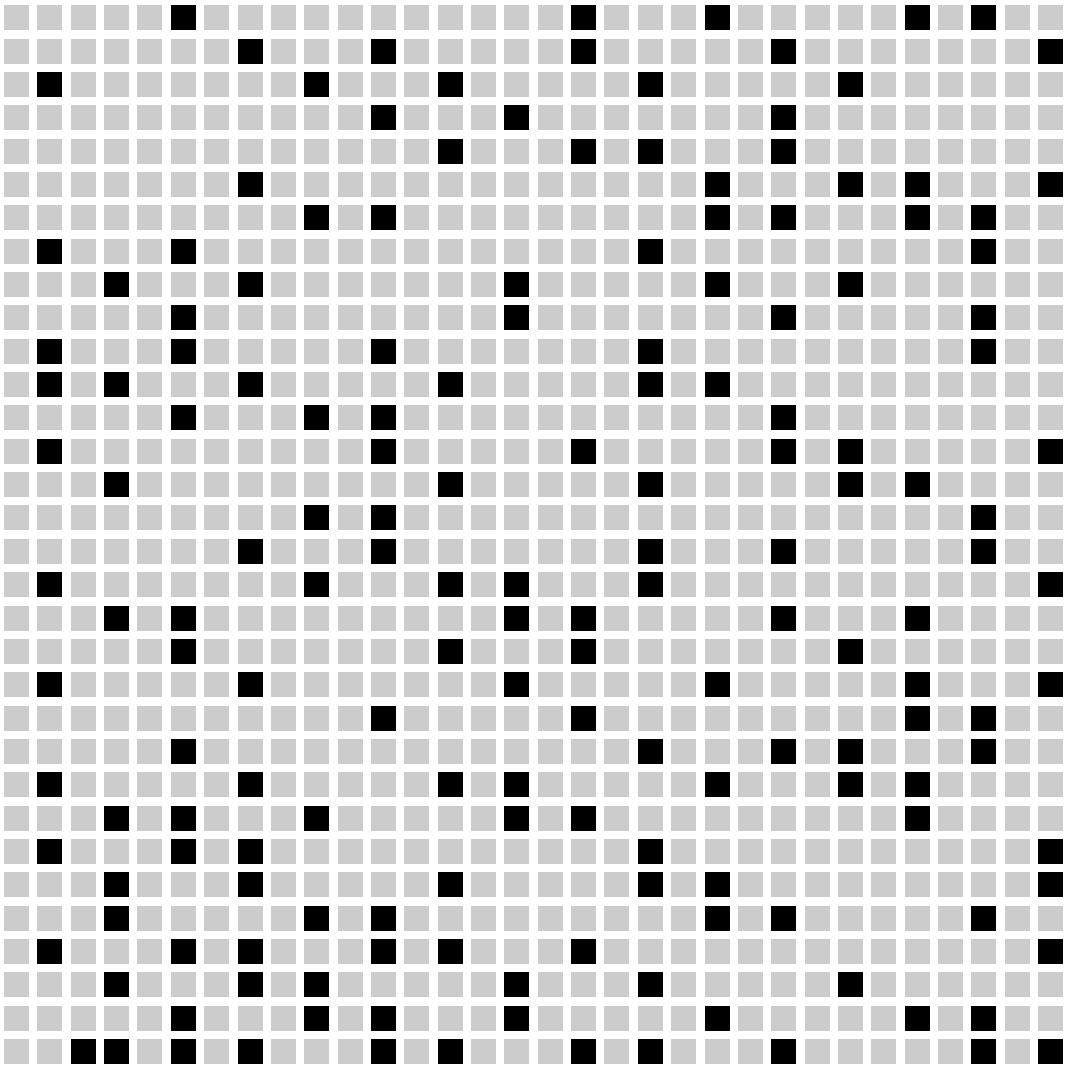


Imperative Programmierung

Sebastian Hack

Ich freue mich über Fehlerberichte an hack@cs.uni-saarland.de

Version bd946e3, Tue Jul 7 13:32:56 2020 +0200

Inhaltsverzeichnis

1 Arithmetik

Die meisten Rechner arbeiten digital (von lat. *digitus* = der Finger). Das heißt, dass eine Speicherzelle eine endliche Anzahl von Zuständen annehmen kann. In der Praxis ist eine Speicherzelle **binär**, das heißt, dass sie nur zwei Zustände annehmen kann: Spannung vorhanden, Spannung nicht vorhanden; 0 oder 1. In diesem Kapitel besprechen wir, wie man ganze Zahlen mithilfe von **Bits** repräsentiert und wie man die gewohnten Rechenoperationen (Addition, Subtraktion) durch Bitoperationen ausdrückt.

Wir diskutieren zuerst, wie man mittels **Stellenwertsystemen** die natürlichen Zahlen durch Folgen von Ziffern (aus einem endlichen Ziffernvorrat) darstellt. Im täglichen Leben verwenden wir zehn Ziffern, der Rechner verwendet zwei. Beim Programmieren ist es manchmal praktisch, sechzehn oder nur acht Ziffern zu verwenden. Eine binäre Ziffer nennt man **Bit**. Mit einer Folge von n Bits kann man dann 2^n Zahlen darstellen. Aus praktischen Gründen rechnet ein Prozessor immer mit Bitfolgen fest vorgegebener Länge. Dies rührt daher, dass die Schaltkreise, die die Bitfolgen verarbeiten eine feste Größe haben und sich nicht ändern lassen. Daher müssen wir verstehen, wie wir die gewohnten arithmetischen Operationen sinnvoll auf diese Zahlen fester Länge anpassen. Dies erreichen wir durch die **Moduloarithmetik**.

1.1 Stellenwertsysteme

Zahlen schreibt man gewöhnlich als Folge von **Ziffern**. Dadurch, dass man nur einen endlichen Vorrat von Ziffern hat, muss man Ziffern zu Folgen zusammensetzen, um jede natürliche Zahl darstellen zu können. Da es auf die Gestalt der Ziffer nicht ankommt, sondern allein entscheidend ist, dass man nur endlich viele Ziffern hat, wollen wir im Folgenden die Menge der Ziffern mit der Menge $\{0, \dots, \beta - 1\}$ gleichsetzen, wobei $\beta > 0$ die Anzahl der vorhandenen Ziffern ist und **Basis** des Stellenwertsystems heißt. In unserem Alltag ist $\beta = 10$, im Rechner ist $\beta = 2$ und die Ziffern heißen dort **Bits**. Im Folgenden kürzen wir die letzte Ziffer $\beta - 1$ mit α ab. Eine **Ziffernfolge** x der Länge n zur Basis β ist also ein Element der Menge $\{0, \dots, \alpha\}^n$. Zwei Ziffernfolgen $x = x_{n-1} \cdots x_0$ und $y = y_{m-1} \cdots y_0$ können zur Ziffernfolge

$$z = x \cdot y \coloneqq x_{n-1} \cdots x_0 \cdot y_{m-1} \cdots y_0$$

verkettet werden. Wie bei der Multiplikation lassen wir den Verkettungs-Operator ab und zu weg, um Bitfolgen kompakter zu schreiben. So schreiben wir für die Verkettung der Bitfolgen a und b dann ab anstatt $a \cdot b$. Des Weiteren schreiben wir a^n für $\underbrace{a \cdots a}_{n}$.

Man kann nun jeder Ziffernfolge $x_{n-1} \cdots x_0$ mit $x_i \in \{0, \ldots, \alpha\}$ eine natürliche Zahl zuordnen:

Definition 1.1 (Interpretation von Ziffernfolgen)**.**

$$\langle \cdot \rangle_\beta : \{0, \ldots, \alpha\}^+ \to \mathbb{N} \qquad x_{n-1} \cdots x_0 \mapsto \sum_{i=0}^{n-1} x_i \beta^i$$

⌟

x_{n-1} ist die Ziffer an der **höchstwertigen Stelle (engl. most significant digit)** und x_0 die Ziffer an der **niedrigstwertige Stelle (engl. least significant digit)**. Im Alltag notieren wir natürliche Zahlen mit Ziffernfolgen zur Basis 10:

$$\langle 123 \rangle_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

Im Rest des Kapitels werden wir die folgenden drei Lemmata noch häufig verwenden.

Lemma 1.1. *Sei x eine Ziffernfolge der Länge n und y eine Ziffernfolge der Länge m. Es gilt: $\langle x \cdot y \rangle_\beta = \langle x \rangle_\beta \times \beta^m + \langle y \rangle_\beta$*

Beweis. → Aufgabe 1.2 □

Lemma 1.2. *Für $n \geq 0$ gilt $\langle \alpha^n \rangle_\beta + 1 = \beta^n$*

Beweis. → Aufgabe 1.2 □

Lemma 1.3. *Sei x eine Ziffernfolge der Länge n. Dann gilt: $\langle x \rangle_\beta < \beta^n$*

Beweis. → Aufgabe 1.2 □

1.1.1 Surjektivität

Die Abbildung $\langle \cdot \rangle_\beta$ ist nur dann sinnvoll, wenn auch tatsächlich jeder natürlichen Zahl eine Ziffernfolge zugeordnet ist, sprich wenn $\langle \cdot \rangle_\beta$ surjektiv ist. Wir weisen die Surjektivität von $\langle \cdot \rangle_\beta$ nach, indem wir auf den Ziffernfolgen eine Nachfolgerfunktion $i : \{0, \ldots, \alpha\}^+ \to \{0, \ldots, \alpha\}^+$, ähnlich der Nachfolgerfunktion $s : \mathbb{N} \to \mathbb{N}$ der natürlichen Zahlen, definieren. Somit können wir dann zu jeder natürlichen Zahl n eine Ziffernfolge x konstruieren für die $\langle x \rangle_\beta = n$. Die Definition von $i(x)$ formalisiert die Addition mit 1, die man in der Schule gelernt hat.

Sei x zunächst eine Ziffernfolge mit n Stellen, die **ungleich** der Ziffernfolge $\alpha \cdots \alpha$ ist:

$$x = x_{n-1} \cdots x_{m+1} x_m \alpha^m \qquad \text{mit } x_m \neq \alpha \tag{1.1}$$

m gibt den höchsten Index an, ab dem alle niederwertigeren Stellen α sind. Hat die Zahl keine nachlaufenden α, so ist $m = 0$. Nun definieren wir den Nachfolger $i(x)$ für alle Bitfolgen x die (1.1) erfüllen:

$$i : x_{n-1} \cdots x_{m+1} x_m x_{m-1} \cdots x_0 \mapsto x_{n-1} \cdots x_{m+1} \cdot s(x_m) \cdot 0^m \tag{1.2}$$

Die Zahl $\langle 180 \rangle_{10}$ als Ziffernfolge zu verschiedenen Basen:

Basis	Ziffernfolge
2	10110100
8	264
16	b4

Die Menge A^+ enthält alle endlichen Tupel von Elementen aus A: $A^+ := \bigcup_{i=1}^{\infty} A^i$

wobei s die Nachfolgerfunktion der natürlichen Zahlen ist. i ist wohldefiniert, weil $s(x_m) \in \{1, \dots, \alpha\}$, da nach Voraussetzung $x_m \neq \alpha$. Auf allen Bitfolgen α^m, die nicht (1.1) erfüllen, definieren wir zusätzlich

$$i : \alpha^m \mapsto i(0\alpha^m) \tag{1.3}$$

Zeigen wir nun, dass i tatsächlich mit s korrespondiert.

Lemma 1.4. *Sei $x \in \{0, \dots, \alpha\}^n$. Dann ist $s(\langle x \rangle_\beta) = \langle i(x) \rangle_\beta$*

Beweis. Aufgrund von (1.3) können wir uns auf Ziffernfolgen beschränken, die (1.1) erfüllen. Sei also x eine Ziffernfolge ungleich α^m. Somit gibt es ein $m < n$ wie in (1.1), sprich $x_m \neq \alpha$ und alle niederwertigeren Stellen sind α:

$$\begin{aligned}
s(\langle x_{n-1} \cdots x_m \alpha^m \rangle_\beta) &= \langle x_{n-1} \cdots x_m \rangle_\beta \times \beta^m + \langle \alpha^m \rangle_\beta + 1 && \text{Lemma 1.1} \\
&= \langle x_{n-1} \cdots x_m \rangle_\beta \times \beta^m + \beta^m && \text{Lemma 1.2} \\
&= \left(\sum_{i=m}^{n-1} x_i \beta^i \right) + \beta^m \\
&= \left(\sum_{i=m+1}^{n-1} x_i \beta^i \right) + (x_m + 1) \times \beta^m \\
&= \langle i(x) \rangle_\beta && \square
\end{aligned}$$

1.1.2 Injektivität

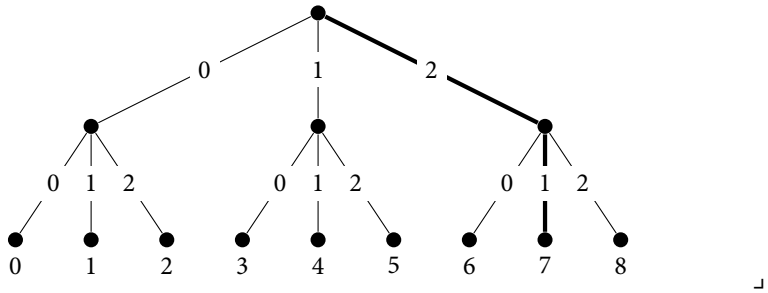
Die Abbildung $\langle \cdot \rangle_\beta$ ist **nicht** injektiv (eindeutig). Für jede Ziffernfolge x und für alle natürlichen Zahlen n, m mit $n - m > 0$ sind die Ziffernfolgen

$$0^n \cdot x \quad \text{und} \quad 0^m \cdot x$$

unterschiedlich, haben aber den gleichen Wert unter $\langle \cdot \rangle_\beta$. Verbietet man aber führende Nullen oder beschränkt man sich auf Ziffernfolge einer festen Länge n, so ist $\langle \cdot \rangle_\beta$ eindeutig. Man beachte, dass man zwei unterschiedlich lange Ziffernfolgen durch Nullerweiterung auf die selbe Länge bringen kann, ohne deren Wert zu ändern.

Dies kann man sich durch folgendes Bild klar machen: Mit n Ziffern kann man β^n unterschiedliche Ziffernfolgen bilden. Die höchstwertige Stelle unterteilt nun die Menge $\{0, \dots, \beta^n - 1\}$ in β gleichgroße Intervalle der Länge β^{n-1} und gibt an, in welchem dieser Teile die durch die Ziffernfolge kodierte Zahl liegt. Die nächste Ziffer unterteilt das so gewählte Intervall wieder in β gleich große Intervalle (der Länge β^{n-2}), usw.

Beispiel 1.1. Für $\beta = 3$, $n = 2$ ergibt sich folgender Baum. Hervorgehoben ist der Pfad mit $\langle 21 \rangle_3 = 7$.



Lemma 1.5. *Seien x, y zwei Ziffernfolgen der Länge n. Unterscheiden sich beide Ziffernfolgen mindestens in einer Stelle, so gilt $\langle x\rangle_\beta \neq \langle y\rangle_\beta$.*

Beweis. Sei k die größte Stelle, an denen sich x und y unterscheiden, sprich für $k \in \{0, \dots, n-1\}$ gilt $x_k \neq y_k$ und für alle $i > k$ gilt $x_i = y_i$. Sei ohne Beschränkung der Allgemeinheit $x_k > y_k$. Dann ist

$$\begin{aligned}
\langle x\rangle_\beta - \langle y\rangle_\beta &= (x_k - y_k)\beta^k + \sum_{i=0}^{k-1}(x_i - y_i)\beta^i \\
&\geq \beta^k + \sum_{i=0}^{k-1}(x_i - y_i)\beta^i && \text{nach Voraussetzung} \\
&= 1 + \sum_{i=0}^{k-1}\alpha\beta^i + \sum_{i=0}^{k-1}(x_i - y_i)\beta^i && \text{Lemma 1.2} \\
&= 1 + \sum_{i=0}^{k-1}\underbrace{\left(\alpha + x_i - y_i\right)}_{\geq 0 \text{ da } y_i \leq \alpha}\beta^i \\
&> 0 && \square
\end{aligned}$$

1.2 Moduloarithmetik und Dualzahlen

Intern rechnet ein Prozessor immer mit Bitfolgen einer konstanten Länge n. Gegenwärtig sind das meist 32 oder 64 Bits. Die herkömmlichen Rechenoperationen müssen also angepasst werden, um auf Ziffernfolgen konstanter Länge eine Bedeutung zu haben.

Betrachten wir zunächst die Addition und Subtraktion auf den ganzen Zahlen. Nehmen wir an, wir haben eine endliche Teilmenge von $\mathbb{Z}$ mit n Elementen. Es ist sinnvoll, dass die 0 in dieser Menge enthalten ist, da sie das (rechts-)neutrale Element bezüglich der Addition und Subtraktion ist. Die Addition $+'$ zweier Zahlen p, q dieser Menge sollte nun so definiert sein, dass sie mit der Addition auf den natürlichen Zahlen weitestgehend kompatibel ist, sprich $p + q < n \implies p +' q = p + q$. Des Weiteren sollen alle herkömmlichen Identitäten, wie z.B. $a + b - b = a$ weiterhin gelten. Eine Definition von $+'$, die das ermöglicht, ist die Addition **modulo** einer Zahl n. Hierbei setzt man mittels einer Äquivalenzrelation alle Zahlen gleich, die Vielfache von n auseinander liegen:

Definition 1.2 (Modulo)**.**

$$a \equiv b \mod n \quad :\Longleftrightarrow \quad \exists k \in \mathbb{Z} : a - b = k \cdot n$$

⌟

n ist ein Parameter der Relation. Wenn er sich aus dem Kontext ergibt, lassen wir ihn auch weg.

Die Äquivalenzrelation $\equiv$ verträgt sich nun mit der Addition:

Lemma 1.6. *Für alle* $p \equiv p' \mod n$ *und alle* $q \equiv q' \mod n$ *gilt* $p + q \equiv p' + q' \mod n$.

Beweis. Es existieren k, m, so dass $p = kn + p'$ und $q = mn + q'$. Also gilt: $p + q = p' + q' + (k + m)n$, sprich $p + q \equiv p' + q' \mod n$. □

⋮	⋮	⋮	⋮
8	9	10	11
4	5	6	7
0	1	2	3
−4	−3	−2	−1
−8	−7	−6	−5
⋮	⋮	⋮	⋮

Alle Zahlen einer Spalte sind äquivalent modulo 4. Jede Spalte ist eine Äquivalenzklasse bezüglich $\cdot \equiv \cdot \mod 4$.

Im Folgenden beschränken wir uns auf Stellenwertsysteme der Basis 2. Wir schreiben im Folgenden nur noch $\langle x \rangle$ für $\langle x \rangle_2$. Wir haben also nur zwei Ziffern aus der Menge $\mathbb{B} := \{0, 1\}$. Ein Element $b \in \mathbb{B}$ heißt ein **Bit**. Ein Element aus $\mathbb{B}^8$ heißt **Byte**. Das **Komplement** $\overline{b}$ eines Bits b ist definiert durch $\overline{b} := 1 - b$.

Darüber hinaus führen wir hier noch die drei assoziativen und kommutativen Operationen &, |, ^ auf Bits ein, mit denen wir später die Addition von Bitfolgen implementieren werden. Wir definieren sie durch die Angabe von Wertetabellen. In Hardware können sie mit einfachen Transistorschaltungen realisiert werden und bilden somit die Basis jedes Mikroprozessors. Das Bit b_{n-1} einer Bitfolge $b = b_{n-1} \dots b_0$ heißt **most significant bit** (MSB) und das Bit b_0 heißt **least significant bit** (LSB). Um Bitfolgen kompakt zu schreiben, stellt man sie häufig als Hexadezimalzahl (eine Ziffernfolge zur Basis 16) dar. Hierbei entsprechen vier Bit der Bitfolge einer Ziffer der Hexadezimalzahl. Für die Ziffern $10, \dots, 15$ verwendet man die Zeichen a, … , f. Zum Beispiel schreibt man:

		and	or	xor
a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

$$\langle 0001100111111010 \rangle_2 \quad = \quad \langle 19\text{fa} \rangle_{16}$$

In vielen Programmiersprachen werden Hexadezimalzahlen durch das Voranstellen eines Präfixes kenntlich gemacht, zum Beispiel `0x19fa` in C-artigen Sprachen und `$19fa` in Pascal.

1.3 Die Addition

Analog zum Additionsalgorithmus von Dezimalzahlen, den man im Alltag verwendet, definieren wir die Addition von zwei Bitfolgen. Betrachten wir zunächst die Summe $a + b$ zweier Bits a und b. Sie wird mit ^ gebildet: Die Summe ist genau dann 1, wenn exakt einer der Operanden 1 ist. Die Addition produziert einen Übertrag (engl. carry) $c := a \;\&\; b$ genau dann, wenn $a = b = 1$. Anstelle einzelner Bits möchte man jedoch meist zwei Bitfolgen $a_{n-1} \dots a_0$ und $b_{n-1} \dots b_0$ addieren. Hierzu muss man die Überträge fortschalten. Sprich, die Addition zweier Bits an Stelle i muss den Übertrag c_i, der potentiell von Stelle $i - 1$ kommt, berücksichtigen.

$$s_i := a_i \;\hat{}\; b_i \;\hat{}\; c_i \tag{1.4}$$

Dieser hat möglicherweise auch einen Einfluss auf den Übertrag, der an Stelle i produziert wird. Der Übertrag, der an i-ten Stelle entsteht ist genau dann 1, wenn mehr als eines der Bits a_i, b_i, c_i den Wert 1 hat:

$$c_{i+1} \coloneqq (a_i \,\&\, b_i) \mid (a_i \,\&\, c_i) \mid (b_i \,\&\, c_i) \tag{1.5}$$

Das folgende Lemma zeigt, dass die Bitfolge $c_{i+1}s_i$ tatsächlich die Summe der i-ten Stelle und des Übertrags, der in die i-te Stelle fließt, darstellt[1].

Lemma 1.7 (Addition einer Stelle)**.** $a_i + b_i + c_i = \langle c_{i+1}s_i \rangle$

Beweis. Durch eine Wertetabelle:

a_i	b_i	c_i	$a_i + b_i + c_i$	c_{i+1}	s_i	$\langle c_{i+1}s_i \rangle$
0	0	0	0	0	0	0
1	0	0	1	0	1	1
0	1	0	1	0	1	1
1	1	0	2	1	0	2
0	0	1	1	0	1	1
1	0	1	2	1	0	2
0	1	1	2	1	0	2
1	1	1	3	1	1	3

□

Wir nehmen an, dass kein Übertrag zur untersten Stelle gelangt, und setzen daher $c_0 \coloneqq 0$. Die Summe $z \in \mathbb{B}^{n+1}$ der Bitfolgen $a, b \in \mathbb{B}^n$ ist dann

Definition 1.3. $a + b \coloneqq c_n s_{n-1} \dots s_0$ ⌟

Zeigen wir nun, dass die Addition auf Bitfolgen mit der Addition der entsprechenden natürlichen Zahlen koinzidiert.

Satz 1.8. $\langle a + b \rangle = \langle a \rangle + \langle b \rangle$ *für* $c_0 = 0$.

Beweis. Durch Induktion über die Länge der Bitfolge.

1. Basisfall $n = 1$: Direkt durch Lemma 1.7.
2. Induktionsschritt $n - 1 \to n$. Die Induktionshypothese ist:

$$\langle c_{n-1}s_{n-2} \cdots s_0 \rangle = \langle a_{n-2} \cdots a_0 \rangle + \langle b_{n-2} \cdots b_0 \rangle$$

$$\begin{aligned}
\langle a \rangle + \langle b \rangle &= a_{n-1}2^{n-1} + \langle a_{n-2} \cdots a_0 \rangle + b_{n-1}2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle && \text{Lemma 1.1} \\
&= (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle a_{n-2} \cdots a_0 \rangle + \langle b_{n-2} \cdots b_0 \rangle \\
&= (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle c_{n-1}s_{n-2} \cdots s_0 \rangle && \text{IH} \\
&= (a_{n-1} + b_{n-1} + c_{n-1}) \cdot 2^{n-1} + \langle s_{n-2} \cdots s_0 \rangle && \text{Lemma 1.1} \\
&= \langle c_n s_{n-1} \rangle \cdot 2^{n-1} + \langle s_{n-2} \cdots s_0 \rangle && \text{Lemma 1.7} \\
&= \langle c_n s_{n-1} \cdots s_0 \rangle && \text{Lemma 1.1}
\end{aligned}$$

Ist der Übertrag der letzten Stelle 1, so benötigt man $n + 1$ Bits um die Summe darzustellen. Da man im Rechner aber nur Speicherzellen konstanter Länge hat, können diese Summen nicht (immer) exakt dargestellt werden. Daher implementiert ein Rechner nicht die Arithmetik der ganzen Zahlen sondern Moduloarithmetik:

[1] Folgendes Beispiel illustriert Lemma 1.7:

$$\begin{array}{rccccc}
a = & & 1 & 0 & 1 & 1 \\
b = & & 0 & 1 & 1 & 0 \\
c = & 1 & 1 & 1 & 0 & 0 \\
\hline
s = & 0 & 0 & 0 & 0 & 1
\end{array}$$

Die Ziffern im gestrichelten Kasten stellen die rechte Seite Lemma 1.7 dar und die Ziffern im anderen Kasten, die der linken.

Lemma 1.9. *Seien $a, b \in \mathbb{B}^n$ und $c_n s_{n-1} \dots s_0 = a + b$. Dann ist $\langle s_{n-1} \cdots s_0 \rangle \equiv \langle a \rangle + \langle b \rangle \mod 2^n$.*

Beweis.

$$\begin{aligned} \langle a \rangle + \langle b \rangle &= \langle c_n s_{n-1} \cdots s_0 \rangle && \text{Satz 1.8} \\ &= c_n \cdot 2^n + \langle s_{n-1} \cdots s_0 \rangle && \text{Lemma 1.1} \\ &\equiv \langle s_{n-1} \cdots s_0 \rangle \mod 2^n && \square \end{aligned}$$

Um deutlich zu machen, dass wir nur die unteren n Bits des Additionsergebnisses meinen, schreiben wir im Folgenden oft $a +_n b$ oder lassen den Index n weg, wenn aus dem Kontext klar wird, dass das Übertragsbit der höchstwertigen Stelle nicht berücksichtigt wird.

Die Subtraktion modulo 2^n definieren wir über die Addition: Denn es existiert für jede Bitfolge $b \in \mathbb{B}^n$ eine Bitfolge $b^* \in \mathbb{B}^n$, so dass $-\langle b \rangle \equiv \langle b^* \rangle \mod 2^n$ und somit $\langle b \rangle + \langle b^* \rangle \equiv 0 \mod 2^n$. b^* liefert das folgende Lemma:

Lemma 1.10. $\langle b \rangle + \langle \overline{b} +_n 1 \rangle \equiv 0 \mod 2^n$

Beweis.

$$2^n = 2^n + \langle b \rangle - \langle b \rangle = \langle b \rangle + \underbrace{2^n - \langle b \rangle}_{\langle b^* \rangle}$$

$$\begin{aligned} &= \langle b \rangle + 1 + \sum_0^{n-1} 2^i - \langle b \rangle && \text{Lemma 1.2} \\ &= \langle b \rangle + 1 + \sum_0^{n-1} (1 - b_i) 2^i && \text{Definition 1.1} \\ &= \langle b \rangle + 1 + \langle \overline{b} \rangle \\ &\equiv \langle b \rangle + \langle 1 + \overline{b} \rangle \mod 2^n && \text{Lemma 1.9} \end{aligned}$$

Da $\equiv$ transitiv ist (→ Aufgabe 1.8) und $0 \equiv 2^n \mod 2^n$, gilt die Behauptung. □

Somit definieren wir die Subtraktion von Bitfolgen fester Länge durch die Addition:

Definition 1.4 (Subtraktion auf Bitfolgen)**.** $a - b \coloneqq a + \overline{b} + 1$ ⌟

1.4 Ganze Zahlen

Bislang ordnen wir jeder Bitfolge eine Zahl größer oder gleich Null zu. Um auch negative Zahlen darstellen zu können, führen wir eine neue Zuordnung von Bitfolgen zu den ganzen Zahlen ein:

Definition 1.5 (Vorzeichenbehaftete Interpretation von Bitfolgen, two's complement number)**.**

$$[\cdot] : \mathbb{B}^n \to \mathbb{Z} \qquad [b_{n-1} b_{n-2} \cdots b_0] \mapsto -b_{n-1} \cdot 2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle$$ ⌟

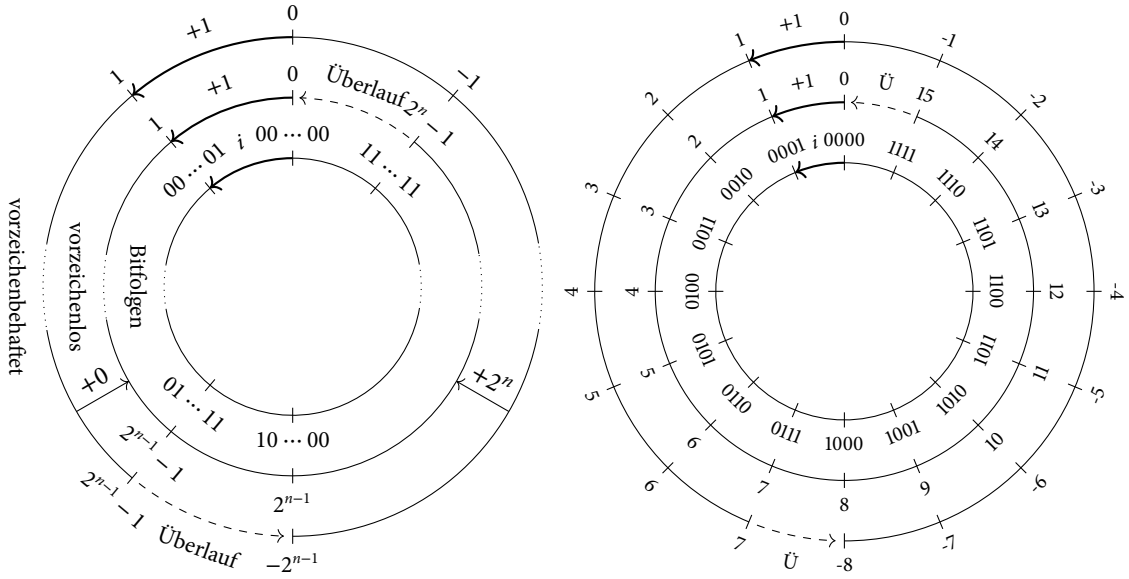


Abbildung 1.1: Zuordnung von vorzeichenlosen/-behafteten Zahlen zu Bitfolgen

Abbildung 1.1 zeigt den Unterschied der beiden Zuordnungen. Schön an $[\cdot]$ ist, dass die Addition auf Bitfolgen, wie oben definiert, weiter funktioniert, da $[b]$ und $\langle b \rangle$ äquivalent modulo 2^n sind:

Lemma 1.11. $[b] \equiv \langle b \rangle \mod 2^n$

Beweis.

$$\begin{aligned}\langle b \rangle - [b] &= b_{n-1} \cdot 2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle + b_{n-1} \cdot 2^{n-1} - \langle b_{n-2} \cdots b_0 \rangle \\ &= b_{n-1} \cdot 2^n\end{aligned}$$

□

Aus Lemma 1.6 und Lemma 1.11 folgt, dass der Schaltkreis zum Addieren von vorzeichenlosen und vorzeichenbehafteten Zahlen derselbe ist. Es ist jedoch nicht jede Operation auf vorzeichenlosen und vorzeichenbehafteten Zahlen gleich. Betrachten wir die Vergleichsoperation $\leq$ und zwei Bitfolgen der Länge 3. Es gilt

$$\langle 001 \rangle = 1 \leq 6 = \langle 110 \rangle \qquad \text{aber} \qquad [001] = 1 \not\leq -2 = [110]$$

Will man zwei Bitfolgen vergleichen, so muss man also entscheiden, ob man sie vorzeichenlos oder vorzeichenbehaftet **interpretieren** will. Daher haben Prozessoren zwei verschiedene Befehle für $\leq$, aber nur einen für die Addition (→ Aufgabe 1.6).

Manche Prozessoren haben auch nur einen Vergleichsbefehl, der dann aber sowohl das Ergebnis des vorzeichenlosen als auch des vorzeichenbehafteten Vergleiches produziert.

1.4.1 Bereichserweiterung

Es kommt häufig vor, dass man kürzere Bitfolgen in längere umwandeln will. Beispielsweise arbeitet ein Prozessor intern mit einer gewissen **Wortbreite** von w Bits (in der Praxis ist $8 \leq w \leq 64$). Lädt man nun beispielsweise ein Byte b in einen Prozessor mit $w = 32$, so muss das Byte b

zu einer Bitfolge $b' \in \mathbb{B}^{32}$ umgewandelt werden. Der Programmierer hat natürlich das Interesse, dass b' derselben Zahl entspricht wie b, sprich, dass entweder $[b] = [b']$ oder $\langle b \rangle = \langle b' \rangle$. Abhängig davon, ob b und b' vorzeichenlos oder vorzeichenbehaftet interpretiert werden sollen, müssen die „neuen" Bits 8–31 in b' entsprechend gesetzt werden. Betrachten wir Abbildung 1.2, in der alle Bitfolgen der Länge zwei und drei mit den entsprechenden vorzeichenlosen/-behafteten Zahlen aufgeführt sind.

Abbildung 1.2: Alle Bitfolgen der Längen zwei und drei mit den entsprechend zugeordneten Zahlen

$b \in \mathbb{B}^3$	$[b]$	$\langle b \rangle$	$b' \in \mathbb{B}^2$	$[b]$	$\langle b \rangle$
000	0	0	00	0	0
001	1	1	01	1	1
010	2	2	10	−2	2
011	3	3	11	−1	3
100	−4	4			
101	−3	5			
110	−2	6			
111	−1	7			

Will man nun eine Bitfolge $b' \in \mathbb{B}^2$ in eine Bitfolge $b \in \mathbb{B}^3$ umwandeln, so dass $\langle b \rangle = \langle b' \rangle$, so ist $b = 0 \cdot b'$. Ist man nun aber daran interessiert, dass $[b] = [b']$, so reicht eine Erweiterung mit 0 nicht aus. Nach Abbildung 1.2 ist

$$[10] = -2 = [110] \neq [010]$$

Es gilt:

Lemma 1.12. $[b_{n-1} \cdot b_{n-1} \dots b_0] = [b_{n-1} \dots b_0]$

Beweis.

$$\begin{aligned}
[b_{n-1} \cdot b_{n-1} \dots b_0] &= \langle b_{n-1} \dots b_0 \rangle - b_{n-1} \cdot 2^n && \text{Definition 1.5} \\
&= \langle b_{n-2} \dots b_0 \rangle + b_{n-1} \cdot 2^{n-1} - b_{n-1} \cdot 2^n && \text{Definition 1.1} \\
&= \langle b_{n-2} \dots b_0 \rangle - b_{n-1} \cdot 2^{n-1} \\
&= [b] && \text{Definition 1.5} \quad \square
\end{aligned}$$

Wir verwenden die folgenden beiden Operationen, um die Bereichserweiterung auszudrücken:

$$\begin{aligned}
\texttt{zext}_m^n : \mathbb{B}^m &\to \mathbb{B}^n \\
b &\mapsto 0^{n-m} \cdot b
\end{aligned}$$

„Zero-Extension"

$$\begin{aligned}
\texttt{sext}_m^n : \mathbb{B}^m &\to \mathbb{B}^n \\
b &\mapsto b_{m-1}^{n-m} \cdot b
\end{aligned}$$

„Sign-Extension"

1.4.2 Überlauf

Sei $f : \mathbb{B}^n \to \mathbb{Z}$ eine Funktion, die einer Bitfolge eine ganze Zahl zuordnet, wie beispielsweise die vorzeichenlose Interpretation $\langle \cdot \rangle_n$ oder die vorzeichenbehaftete Interpretation $[\cdot]_n$. Seien $a, b \in \mathbb{B}^n$ zwei Bitfolgen der Länge n.

Definition 1.6 (Überlauf)**.** Gilt $f(a) + f(b) \neq f(a +_n b)$, so ist die Addition von a und b bezüglich der Interpretation f übergelaufen. ⌟

So ist beispielsweise

$$\begin{aligned} \langle 011\rangle + \langle 110\rangle = 9 \quad &\neq \quad \langle 011 +_3 110\rangle = 1 \\ [011] + [001] = 4 \quad &\neq \quad [011 +_3 001] = -4 \end{aligned}$$

Interessant ist nun die Frage, wie wir bei einer Addition zweier Bitfolgen a, b feststellen, dass ein Überlauf eingetreten ist. Das hängt nun davon ab, ob wir die Bitfolgen vorzeichenlos oder vorzeichenbehaftet interpretieren:

Lemma 1.13. *$\langle a\rangle + \langle b\rangle \neq \langle a +_n b\rangle$ genau dann wenn $c_n = 1$.*

Beweis. Ein Überlauf tritt genau dann auf, wenn $\langle a +_n b\rangle \neq \langle a\rangle + \langle b\rangle$, sprich, wenn $\langle a\rangle + \langle b\rangle \geq 2^n$. Betrachten wir das Ergebnis der Addition $a + b = c_n s_{n-1} \cdots s_0$. Nach Lemma 1.3 ist $\langle s_{n-1} \cdots s_0\rangle < 2^n$, somit muss $c_n = 1$. □

Lemma 1.14. *$[a] + [b] \neq [a +_n b]$ genau dann wenn $c_n \neq c_{n-1}$.*

Beweis. Wir betrachten die um ein Bit vorzeichenerweiterten Bitfolgen

$$\begin{aligned} a' &= \mathtt{sext}_n^{n+1}\, a = a_{n-1} \cdot a_{n-1} \cdots a_0 \\ b' &= \mathtt{sext}_n^{n+1}\, b = b_{n-1} \cdot b_{n-1} \cdots b_0 \end{aligned}$$

Nun gilt:

$$\begin{aligned} [a] + [b] &= [a'] + [b'] && \text{Lemma 1.12} \\ &= -a_{n-1}2^n + \langle a\rangle - b_{n-1}2^n + \langle b\rangle && \text{Definition 1.5} \\ &= -(a_{n-1} + b_{n-1})2^n + \langle c_n s_{n-1} \cdots s_0\rangle && \text{Satz 1.8} \\ &= -(a_{n-1} + b_{n-1})2^n && \text{siehe Beweis Satz 1.8} \\ &\quad +(a_{n-1} + b_{n-1} + c_{n-1})2^{n-1} + \langle s_{n-2} \cdots s_0\rangle \\ &= -(a_{n-1} + b_{n-1} - c_{n-1})2^{n-1} + \langle s_{n-2} \cdots s_0\rangle \\ &= c_{n-1}2^n - (a_{n-1} + b_{n-1} + c_{n-1})2^{n-1} + \langle s_{n-2} \cdots s_0\rangle \\ &= c_{n-1}2^n - \langle c_n s_{n-1}\rangle 2^{n-1} + \langle s_{n-2} \cdots s_0\rangle \\ &= (c_{n-1} - c_n)2^n - \langle s_{n-1}\rangle 2^{n-1} + \langle s_{n-2} \cdots s_0\rangle \\ &= (c_{n-1} - c_n)2^n + [a +_n b] && \text{Definition 1.5} \end{aligned}$$

Ein Überlauf tritt also genau dann ein, wenn $c_{n-1} \neq c_n$. □

Prozessoren mit Statusregister (engl. flag register) haben zur Anzeige Überlaufs zwei Bits (C = carry und O = overflow) in diesem Register, die bei jeder Addition die Werte $C = c_n$ und $O = c_n \mathbin{\hat{}} c_{n-1}$ enthalten. Hat ein Prozessor kein Statusregister, so muss man sich die Werte selbst berechnen.

1.4.3 Shifts

Die Multiplikation einer Bitfolge mit Basis zwei ist einfach. Man rückt einfach alle Stellen eins „nach links“ und fügt eine 0 in der niedrigstwertigen Stelle ein. Dies nennt man auch einen Links-Shift:

$$\begin{aligned} \langle x_{n-1} \dots x_0\rangle \cdot 2 &= \left(\sum_0^{n-1} x_i 2^i\right) \cdot 2 = \sum_0^{n-1} x_i 2^{i+1} \\ &= \langle x_{n-1} x_{n-2} \dots x_0 0\rangle \equiv \langle x_{n-2} \dots x_0 0\rangle \mod 2^n \end{aligned}$$

Da die Wortbreite eines Prozessors konstant ist, wird durch das Einfügen der 0, das höchstwertige Bit verworfen. Wir definieren also

$$x_{n-1} \dots x_0 \ll k \coloneqq x_{n-1-k} \dots x_0 \cdot 0^k$$

um eine Bitfolge der Länge n zu erhalten.

Ähnlich geht man beim Dividieren durch zwei vor, man shiftet jedoch nach rechts. Hier wird die 0 jedoch an der höchstwertigen Stellen eingefügt und die niedrigstwertige Stelle verworfen:

$$\begin{aligned}\lfloor \langle x_{n-1} \dots x_0 \rangle / 2 \rfloor &= \left\lfloor \left(\sum_0^{n-1} x_i 2^i \right) / 2 \right\rfloor = \sum_1^{n-1} x_i 2^{i-1} \\ &= \langle 0 x_{n-1} x_{n-2} \dots x_1 \rangle\end{aligned}$$

Auch hier definieren wir eine Kurzschreibweise:

$$x_{n-1} \dots x_0 \gg k \coloneqq 0^k \cdot x_{n-1} \dots x_k$$

Bei vorzeichenbehaftet interpretierten Bitfolgen entspricht der Rechts-Shift jedoch nicht der Division durch zwei. Man benötigt einen speziellen Rechts-Shift, der das Vorzeichenbit repliziert:

$$\begin{aligned}\lfloor [x_{n-1} \dots x_0] / 2 \rfloor &= \left\lfloor \left(\left(\sum_0^{n-2} x_i 2^i \right) - x_{n-1} 2^{n-1} \right) / 2 \right\rfloor \\ &= \left(\sum_1^{n-2} x_i 2^{i-1} \right) - x_{n-1} 2^{n-2} \\ &= [x_{n-1} \dots x_1] \\ &= [x_{n-1} x_{n-1} x_{n-2} \dots x_1] \qquad \text{Lemma 1.12}\end{aligned}$$

Kurz schreiben wir den vorzeichenerhaltenden Rechts-Shift als

$$x_{n-1} \dots x_0 \overset{s}{\gg} k \coloneqq x_{n-1}^k \cdot x_{n-1} \dots x_k$$

Möchte man also vorzeichenbehaftete Zahlen durch zwei dividieren, muss beim Rechts-Shift das höchstwertige Bit erhalten bleiben. Man beachte, dass das Rechts-Shiften immer abrundet, wohingegen unsere Alltagsarithmetik immer zur null hin rundet.

Beispiel 1.2. $-7 = [1001]$ um eins vorzeichenbehaftet nach rechts geshiftet ist $[1100] = -4$. ⌟

Zwei Bitfolgen der Länge n werden durch eine Sequenz von Shifts und Additionen, wie in der Schularithmetik, zu einer Bitfolge der Länge $2n$ multipliziert. Meist werden die oberen n Bit aufgrund von Platzmangel (Speicherzellen/Register haben immer eine fixe Länge) verworfen.

1.5 Gleitkommazahlen

Wir haben nun gesehen, wie wir die Arithmetik der ganzen Zahlen in einem Rechner durch Manipulation von Bitfolgen implementieren können. Wie stellen wir aber Kommazahlen dar? Für viele Berechnungen,

insbesondere in der Numerik, man denke an Simulationen von physikalischen Prozessen, braucht man eine Näherung an die reellen Zahlen. Hierbei ist es wichtig, dass sowohl sehr kleine Zahlen (man könnte beispielsweise den Abstand zweier Atome speichern wollen) als auch sehr große Zahlen (man könnte aber auch den Abstand zweier Galaxien speichern wollen) darstellbar sind. Dies erreicht man durch Gleitkommazahlen[2] (engl. floating point numbers).

Wir stellen in diesem und dem folgenden Abschnitt einige Grundlagen der Gleitkomma-Arithmetik dar, können aber im Rahmen dieses Buches nur an der Oberfläche kratzen.[3]

Definition 1.7 (Gleitkomma-Zahlensystem)**.** Ein Gleitkomma-Zahlensystem $(\beta, p, e_{\min}, e_{\max})$ ist gegeben durch einen **Radix** $\beta \geq 2$, eine **Genauigkeit** $p \geq 2$, und zwei Schranken $e_{\min} < e_{\max}$ für den Bereich des Exponenten. ⌟

Definition 1.8 (Gleitkommazahl)**.** Eine Gleitkommazahl f in einem Gleitkomma-Zahlensystem $(\beta, p, e_{\min}, e_{\max})$ ist gegeben durch ein Paar

$$f = (M, e) \quad \text{mit } 0 \leq |M| < \beta^p \text{ und } e_{\min} \leq e \leq e_{\max}$$

Der Wert von f ist definiert als

$$M \cdot \beta^{e+1-p}$$ ⌟

Man kann eine Gleitkommazahl auch als Kommazahl interpretieren

$$\begin{aligned} M \cdot \beta^{e+1-p} &= \operatorname{sgn}(M) \cdot \langle m_0 \cdots m_{p-1} \rangle_\beta \cdot \beta^{e+1-p} \\ &= \operatorname{sgn}(M) \cdot \langle m_0, m_1 \cdots m_{p-1} \rangle \cdot \beta^e \quad \text{wobei } \langle m_0, m_1 \cdots m_{p-1} \rangle = \sum_{i=0}^{p-1} m_i \beta^{-i} \end{aligned}$$

wobei $m_0 \cdots m_{p-1}$ nun die Ziffernfolge ist, die zur Basis β den Wert $|M|$ besitzt. Der Name „Gleitkommazahl“ rührt daher, dass man durch die Wahl des Exponenten e das Komma durch die Mantisse „gleiten“ lassen kann.

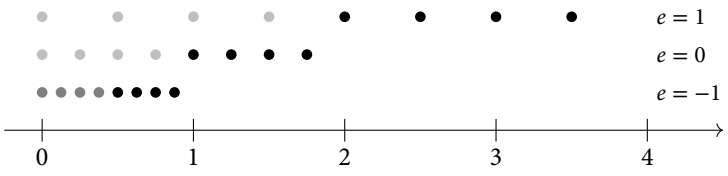


Abbildung 1.3: Die Abbildung zeigt alle Gleitkommazahlen gemäß Definition 1.8 für $\beta = 2, p = 3$, $e_{\min} = -1, e_{\max} = 1$. Die negativen Zahlen sind symmetrisch und werden aus Platzgründen nicht gezeigt. Die schwarzen Punkte entsprechen normalisierten Gleitkommazahlen.

Abbildung 1.3 zeigt alle Gleitkommazahlen des Systems $\beta = 2, p = 3$, $e_{\min} = -1, e_{\max} = 1$. Wie man der Abbildung entnehmen kann, werden manche Zahlen durch mehrere Gleitkommazahlen dargestellt. Zum Beispiel wird die Zahl 0,5 durch drei Gleitkommazahlen in diesem System dargestellt:

$$0{,}5 = \langle 100 \rangle \cdot 2^{-3} = \langle 010 \rangle \cdot 2^{-2} = \langle 001 \rangle \cdot 2^{-1}$$

[2] Konrad Zuses Rechner Z1 aus dem Jahre 1937 verwendete Gleitkommazahlen. Seine Darstellung der Gleitkommazahlen entspricht in den wesentlichen Punkten der Darstellung, die auch heute noch verwendet wird. Gemäß Definition 1.8 sprach Zuse sprach von semilogarithmischen Zahlen [Zus90].

[3] Dem interessierten Leser seien insbesondere das Gleitkomma-Kompendium von Muller u. a. [Mul+10], das Standard-Lehrbuch der numerischen Analyse von Higham [Hig02], und der Übersichtsartikel von Goldberg [Gol91] empfohlen.

Um diese Redundanz zu vermeiden, **normalisiert** man Gleitkommazahlen und fordert, dass $\beta^{p-1} \leq |M| < \beta^p$ und somit $m_0 \neq 0$. Die **normalisierten** Gleitkommazahlen werden in Abbildung 1.3 durch die schwarzen Punkte gekennzeichnet. Wie die Abbildung zeigt, gibt es keine zwei normalisierten Gleitkommazahlen, die denselben Wert repräsentieren.

Die kleinste normalisierte positive Gleitkommazahl ist

$$\langle 1 \underbrace{0 \cdots 0}_{p-1} \rangle \cdot \beta^{e_{\min}+1-p} = \beta^{p-1} \cdot \beta^{e_{\min}+1-p} = \beta^{e_{\min}}$$

und die größte Gleitkommazahl ist

$$\Omega := (\beta^p - 1) \cdot \beta^{e_{\max}+1-p} = \beta^{e_{\max}}(\beta - \beta^{1-p})$$

Lemma 1.15. *Die normalisierten Gleitkommazahlen zu einem Exponenten e teilen das Intervall $[\beta^e, \beta^{e+1}]$ in β^{p-1} gleichlange Intervalle der Länge β^{e+1-p}.*

Korollar 1.16 (Maschinen-Epsilon)**.** *Die Zahl β^{1-p} ist die kleinste Gleitkommazahl f für die gilt:* $1{,}0 + f > 1{,}0$. *Sie heißt **Maschinen-Epsilon** und wird mit ε bezeichnet.*

Durch die Normalisierung gehen allerdings auch Zahlen „verloren“: Zum Beispiel ist 0 keine normalisierte Gleitkommazahl. Auch die Zahlen

$$M \cdot \beta^{e_{\min}} \quad \text{für } 0 \leq |M| < \beta^{p-1}$$

können nicht als normalisierte Gleitkommazahlen dargestellt werden. Man nennt sie **subnormale** (engl. subnormal, denormal) Gleitkommazahlen. In Abbildung 1.3 sind sie durch dunkelgraue Punkte dargestellt.

Subnormale Zahlen sind hilfreich um abrupte Bereichsunterscheitungen zu verhindern. Betrachten wir die beiden kleinsten normalisierten Gleitkommazahlen. In Abbildung 1.3 sind dies die beiden schwarzen Punkte, die am nächsten bei der 0 sind. Ihre Differenz beträgt nach Lemma 1.15 $|\beta^{e_{\min}+1-p}|$. Diese Zahl ist (da $p \geq 2$) kleiner als die kleinste normalisierte Gleitkommazahl $\beta^{e_{\min}}$ und somit nicht mehr normalisiert darstellbar. Ohne subnormale Zahlen müsste man diese Differenz zur 0 runden; es käme zu einer **Bereichsunterschreitung** (engl. underflow). Das heißt, dass es zwei Gleitkommazahlen $a \neq b$ gäbe, deren Differenz 0 ist, was unschön ist. Nimmt man jedoch die subnormalen Zahlen hinzu, so gibt es keine zwei unterschiedlichen Gleitkommazahlen, deren Differenz 0 ist.

1.5.1 Der IEEE 754 Standard

Heutzutage werden fast ausschließlich Gleitkommazahlen gemäß des IEEE 754 Standards [08] verwendet. Dieser Standard definiert in seiner neuesten Version von 2008 die Arithmetik und das Binärformat von binären ($\beta = 2$) und dezimalen ($\beta = 10$) Gleitkomma-Systemen. In den meisten Programmiersprachen und Prozessoren kommen **binäre**

$e'_0 \cdots e'_{k-1}$	$\langle e'_0 \cdots e'_{k-1} \rangle$	$m_1 \cdots m_{p-1}$	Wert
$00 \cdots 00$	0	$0 \cdots 0$	$(-1)^s \cdot 0$
$00 \cdots 00$	0	$\neq 0 \cdots 0$	$(-1)^s \cdot \langle 0, m_1 \cdots m_{p-1} \rangle \cdot 2^{e_{\min}}$
$00 \cdots 01$	1		
$\vdots$	$\vdots$		$(-1)^s \cdot \langle 1, m_1 \cdots m_{p-1} \rangle \cdot 2^{e'-b}$
$11 \cdots 10$	$2^k - 2$		
$11 \cdots 11$	$2^k - 1$	$0 \cdots 0$	$(-1)^s \cdot \infty$
$11 \cdots 11$	$2^k - 1$	$\neq 0 \cdots 0$	NaN

Abbildung 1.4: Kodierung der Werte gemäß IEEE 754.

($\beta = 2$) Gleitkomma-Systeme zum Einsatz. Diese werden binär wie folgt gespeichert:

s	$e'_0 \cdots e'_{k-1}$	$m_1 \cdots m_{p-1}$

Das oberste Bit s speichert das Vorzeichen. Die Bits $m_1 \cdots m_{p-1}$ speichern die letzten $p-1$ Stellen der Mantisse. Für $\beta = 2$ wird m_0 nicht explizit gespeichert. Subnormale Zahlen werden durch einen speziellen Wert im Exponentenfeld gekennzeichnet (siehe unten). Die Bits $e'_0 \cdots e'_{k-1}$ speichern den Exponenten in einer „versetzten“ (engl. biased) Codierung: Anstatt die Bits von e direkt zu speichern, wird die Summe

$$e' = e + b \qquad \text{mit dem Bias } b = 2^{k-1} - 1 \tag{1.6}$$

gespeichert, wobei k die Anzahl der Bits des Exponenten ist. In der Praxis kommen meist zwei Gleitkomma-Systeme zum Einsatz:

Der Grund für das Verwenden der Bias ist, dass der Vergleich zweier Gleitkommazahlen dann mit dem Vergleich der Ganzzahl-Interpretation der Bitfolge $s \cdot e'_0 \cdots e'_k \cdot m_1 \cdots m_{p-1}$ implementiert werden kann.

Name	Größe (Bits)	p	$e_{\min}$	$e_{\max}$	Bits Exponent k	Bias b
`float`	32	24	-126	127	8	127
`double`	64	53	-1022	1023	11	1023

$e_{\min}$ und $e_{\max}$ sind so definiert, dass neben den Exponenten für normalisierte Zahlen noch zwei Werte für die Darstellung von subnormalen Zahlen und Spezialwerten verfügbar sind:

$$e_{\min} = -2^{k-1} + 2 \qquad e_{\max} = 2^{k-1} - 1$$

Nach (1.6) liegt der versetzte Exponent von normalisierten Zahlen dann im Intervall $[1, 2^k - 2]$. Für diese ist dann implizit $m_0 = 1$.

Ist $e' = 0$, so ist die Gleitkommazahl subnormal. Es gilt $m_0 = 0$ und $e = e_{\min}$. Ist $e' = 2^k - 1$, so ist der Wert der Gleitkommazahl entweder $\pm\infty$ oder NaN (engl. not a number). Die Zahlen $+\infty$ und ∞ um im Falle eine Überlaufs trotzdem ein Ergebnis produzieren zu können. So ist beispielsweise $\Omega + 1 = +\infty$.

Für nicht definierte Ergebnisse werden NaNs (NaN = not a number) verwendet. So ist beispielsweise $\sqrt{-1} = \text{NaN}$. Streng genommen unterscheidet man noch zwischen *quiet* und *signalling* NaNs. Letztere führen direkt zu einer Ausnahme (engl. exception, trap) während erstere

keine Ausnahme auslösen. Jede Rechenoperation produziert ein NaN als Ergebnis sobald einer ihrer Operanden ein NaN ist. Vergleichsoperationen liefern *falsch* sobald einer ihrer Operanden ein NaN ist. Somit ist insbesondere NaN ≠ NaN wahr.

Abbildung 1.4 fasst die Kodierung der verschiedenen Werte einer IEEE Gleitkommazahl zusammen.

Folgende Operationen sind ungültig (engl. invalid) und produzieren eine NaN.

$(+\infty) - (+\infty)$
$(-\infty) - (-\infty)$
$(+\infty) + (+\infty)$
$(-\infty) + (-\infty)$
$\pm 0 \cdot \pm\infty$
$\pm 0 \div \pm 0$
$\pm\infty \div \pm\infty$
$\sqrt{-x}$ für $x < 0$

1.5.2 Rundung

Da die Gleitkommazahlen nur einen endlichen Ausschnitt der rationalen Zahlen darstellen, kann man nicht jede rationale Zahl in einem gegebenen Gleitkomma-System exakt darstellen. Daher muss eine rationale Zahl, die in einem Gleitkomma-System nicht dargestellt werden kann, gerundet werden. Die Abbildung

$$rnd : \mathbb{Q} \rightharpoonup \mathbb{F}$$

heißt „Runden“ und ordnet jeder rationalen Zahl $-\Omega \leq q \leq \Omega$ die Gleitkommazahl des Gleitkomma-Systems $\mathbb{F}$ zu, der sie am nähsten liegt. Im IEEE 754 Standard werden zahlen außerhalb dieses Intervalls zu den Zahlen $-\infty$ bzw. $+\infty$ gerundet. Hat eine Zahl zwei gleichweit entfernte, nähste Gleitkommazahlen, so wird standardmäßig zu der gerundet, deren M geradzahlig ist (engl. round to nearest, ties to even).

Beispiel 1.3 (0,1)**.** Dem Dezimalbruch 0,1 entspricht der periodische Binärbruch $0{,}000\overline{1100}$ weswegen der Dezimalbruch 0,1 in keinem Gleitkomma-System der Basis 2 exakt dargestellt werden kann. Das schließt die IEEE 754 Zahlen, die in vielen Programmiersprachen verwendet werden, ein. Beispielsweise gilt in `float`:

$$rnd(0{,}1) = 0{,}100000001490116119385$$ ⌟

Beispiel 1.4. Betrachten wir nochmals das System aus Abbildung 1.3 und die Addition

$$1{,}00 \cdot 2^1 + 1{,}01 \cdot 2^0 \quad \text{(entspricht dezimal } 2 + 1{,}25\text{)}$$

Das exakte Ergebnis ist $1{,}101 \cdot 2^1$ (dezimal 3,25) was aber vier Bit Genauigkeit erfordert. Unser System hat aber nur drei Bit Genauigkeit und somit ist das Ergebnis darin nicht exakt darstellbar. ⌟

Bemerkung 1.1. Der Standard IEEE 754 unterstützt weitere Rundungsmodi: Round to $+\infty$ (immer aufrunden), round to $-\infty$ (immer abrunden), round to zero (betragsmäßig abrunden). Round to nearest ist der Standardmodus: In Beispiel 1.4 hat $1{,}101 \cdot 2^1$ die beiden Nachbarn $1{,}10 \cdot 2^1$ (3,0) und $1{,}11 \cdot 2^1$ (3,25). Round to nearest, ties to even rundet diese Zahl zu $1{,}10 \cdot 2^1$. Abbildung 1.5 illustriert die verschiedenen Rundungsmodi an einem Beispiel.

⌟

Betrachtet man eine Zahl $\hat{x}$ als Approximation (beispielsweise durch Rundung) einer anderen Zahl x, so verwendet man häufig den **relativen Fehler** um den Fehler der Approximation zu quantifizieren:

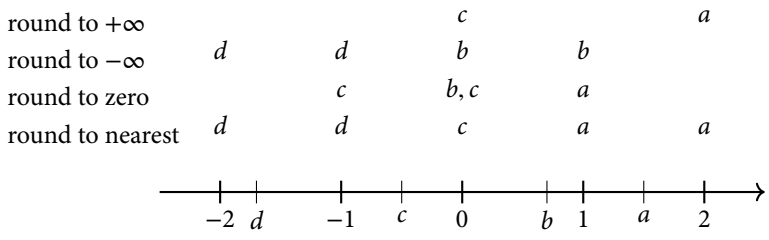


Abbildung 1.5: Beispiel für die verschiedenen Rundungsmodi.

Definition 1.9 (Relativer Fehler)**.** Der relative Fehler von $\hat{x}$ bezüglich x ist definiert durch $\left|\frac{\hat{x}-x}{x}\right|$. ⌟

Das folgende Lemma gibt eine alternative und kompaktere Art, relative Fehler darzustellen:

Lemma 1.17. *Für δ mit $\hat{x} = x(1+\delta)$ gilt, dass $|\delta|$ gleich dem relativen Fehler von $\hat{x}$ ist.*

Beweis. Einsetzen. □

Der Rundungsfehler bei round to nearest ist beschränkt durch die Zahl $u = \frac{1}{2}\beta^{1-p}$, die auch **unit roundoff** genannt wird.

Lemma 1.18. *Der relative Fehler von $rnd(x)$ ist kleiner oder gleich $\frac{1}{2}\beta^{1-p}$.*

Beweis. Es gilt $rnd(0) = 0$ und somit ist der relative Fehler von $rnd(0)$ gleich 0. Wir nehmen im Folgenden an, dass $x > 0$ ist. Für negative x gilt eine analoge Betrachtung. Wir schreiben die rationale Zahl x nun als

$$x = M \cdot \beta^{e-p} \qquad \text{mit } M \in \mathbb{Q} \text{ und } \beta^{p-1} \leq M < \beta^p$$

mit einem **rationales** M. Somit liegt x zwischen den beiden Gleitkommazahlen $y_1 = \lfloor M \rfloor \beta^{e-p}$ und $y_2 = \lceil M \rceil \beta^{e-p}$. Also ist $rnd(x) \in \{y_1, y_2\}$. Es gilt

$$|rnd(x) - x| \leq \frac{|y_2 - y_1|}{2} \leq \frac{\lceil M \rceil - \lfloor M \rfloor}{2} \beta^{e-p} = \frac{1}{2}\beta^{e-p}$$

Der relative Fehler ist dann also:

$$\left|\frac{rnd(x) - x}{x}\right| \leq \frac{\frac{1}{2}\beta^{e-p}}{M \cdot \beta^{e-p}} = \frac{1}{2M} \leq \frac{1}{2}\beta^{1-p}$$ □

Bemerkung 1.2. Bei Gleitkomma-Systemen der Basis 2 ist der unit roundoff gleich dem Maschinen-Epsilon. ⌟

Bemerkung 1.3. Die Arithmetik der IEEE 754 Gleitkommazahlen ist so definiert, dass zunächst exakt gerechnet wird und dann gerundet wird. Für zwei IEE 754 Gleitkommazahlen x, y gilt:

$$x \hat{\circ} y \coloneqq rnd(x \circ y) \quad \text{für } \circ \in \{+, -, \cdot, /\}$$

Wobei $\hat{\circ}$ die Gleitkomma-Variante des Operators $\circ$ der rationalen Zahlen darstellen soll. ⌟

1.6 Fallstricke bei der Verwendung von Gleitkommazahlen

In diesem Abschnitt besprechen wir einige praktische Fallstricke bei der Verwendung von Gleitkommazahlen.

1.6.1 Precision versus Accuracy

In der englischsprachigen Literatur wird häufig zwischen Accuracy und Precision unterschieden, was im deutschen eigentlich beides Genauigkeit bedeutet. Am ehesten kann man im Kontext der Gleitkomma-Arithmetik Accuracy vielleicht noch mit Exaktheit und Precision mit Genauigkeit übersetzen. Mit Accuracy meint den absoluten oder relativen Fehler einer approximierten Größe, also beispielsweise eine zu einer Gleitkommazahl gerundeten rationalen oder reellen Zahl. Man quantifiziert mit

Mit Precision meint man, wieviel Bits man zur Darstellung einer Gleitkommazahl verwendet. So hat `float` beispielsweise 24 Bit Präzision.

Leider sind die beiden Begriffe in der deutschsprachigen Literatur nicht mit eindeutigen Übersetzungen belegt. So wird Precision häufig mit Genauigkeit oder Präzision übersetzt, obwohl man in der Numerik mit Genauigkeit häufig accuracy meint.

1.6.2 Anhäufung von Rundungsfehlern

Betrachten wir folgendes C-Programm im Kontext von Beispiel 1.3.

```
float time = 0.0f;
for (int i = 0; i < 20000; i++)
    time += 0.1f;
```

Der Wert von `time` ist nach Beendigung der Schleife 1999,6588 und nicht wie vielleicht erhofft 2000. Somit hat `time` einen relativen Fehler von $1{,}7 \cdot 10^{-4}$, was signifikant höher ist, als der relative Fehler der Konstante `0.1f`, der nach Lemma 1.18 kleiner gleich dem Maschinen-Epsilon $2^{-24} \approx 6 \cdot 10^{-8}$ von `float` ist.

Hieraus folgt, dass man bei Berechnungen, in denen Dezimalbrüche eine zentrale Rolle spielen und exakte Ergebnisse wichtig sind (wie beispielsweise im Finanzwesen) keine binären Gleitkommazahlen verwenden sollte. Neuere Varianten des IEEE 754 Standards definieren daher auch basis-unabhängige Varianten und insbesondere ein Binärformat für dezimale Gleitkommazahlen.

Bemerkung 1.4. Eine ähnliche Konstellation führte zum Patriot Missile Disaster. Hierbei lief ein interner Timer der Raketensteuerung aufgrund der Anhäufung von Rundungsfehler aus dem Ruder, so dass die Radarsoftware die Position einer abzufangenden Rakete falsch einschätzte und die Patriot-Abfangrakete ihr Ziel verfehlte. ⌟

1.6.3 Unzureichende Präzision

Addiert oder subtrahiert man Zahlen aus unterschiedlichen Größenbereichen, so hängt es von der Präzision des Zahlensystems ab, ob das

Resultat exakt ist. Beispielsweise ist in `float`:

$$16777216 + 1 = 16777216$$

In diesem Beispiel reichen die verfügbaren binären Stellen (24) nicht aus, um das richtige Ergebnis 16777217 darzustellen. Je weiter die zu addierenden Zahlen auseinander liegen, desto höher muss die Präzision des Zahlsystems sein. Beispielsweise ist in `double` (53 binäre Nachkommastellen)

$$16777216 + 1 = 16777217$$

1.6.4 Katastrophale Auslöschung

Subtrahiert man zwei ähnlich große Zahlen, so gleichen sich die führenden Stellen bis zu einem gewissen Punkt und löschen sich bei der Subtraktion aus. Auslöschung kann gutartig oder katastrophal sein. Waren die voneinander subtrahierten Zahlen beide exakt (im Sinne von Accuracy), so ist es das Ergebnis auch. In diesem Falle spricht man von gutartiger Auslöschung. Ist jedoch mindestens eine der beiden Zahlen mit einem Fehler behaftet, besteht die Gefahr, dass bei der Subtraktion der relative Fehler der Differenz wesentlich höher ist, wie der der Subtrahenden.

Nehmen wir beispielsweise an, dass die Zahl 1000 durch die Zahl $x = 1002$ approximiert wird (beispielsweise weil bei vorigen Rechnungen Rundungsfehler aufgetreten sind). Sie hat somit einen relativen Fehler von 0,002. Die Differenz

$$1002 - 999 = 3$$

hat jedoch einen relativen Fehler von 2!

1.7 Aufgaben

Aufgabe 1.1 (Einige Stellenwertsysteme. ⋆⋆)**.** Wir betrachten folgende Stellenwertsysteme:

Name	Basis	Präfix	Ziffern
Binärsystem	2	0b	0,1
Oktalsystem	8	0	0,…,7
Hexadezimalsystem	16	0x	0,…,9,A,B,C,D,E,F

Die Ziffern sind in aufsteigender Reihenfolge ihrer Wertigkeit angegeben, z.B. hat die Ziffer F im Hexadezimalsystem die Wertigkeit 15.

Der Inhalt der Spalte Präfix wird einer Zahl vorangestellt um anzuzeigen, dass sie im jeweiligen Stellenwertsystem kodiert ist. So ist z.b. 013 eine Oktalzahl, 0x13 eine Hexadezimalzahl, und 13 eine Dezimalzahl.

1. Geben Sie die folgenden Dezimalzahlen in jedem der drei Stellenwertsysteme an. Rechnen Sie unbedingt von Hand!

   5 16 49 81 257 317 1721 4096 65536 1000000

2. Wie sind sie beim Umrechnen vorgegangen? Notieren Sie ihre Vorgehensweise als Anleitung für einen anderen Studenten!

3. Wir nehmen an, wir verfügen über eine Funktion $\mathit{ziff} : [0, 15] \to \{0, \ldots, 9, \mathrm{A}, \ldots, \mathrm{F}\}$ die zu einer Dezimalzahl im Intervall $[0, 15]$ die entsprechende Ziffer liefert. Vervollständigen Sie folgende rekursive Definition einer Funktion $\rangle\cdot\langle_b : \mathbb{N} \to \{0, \ldots, 9, \mathrm{A}, \ldots, \mathrm{F}\}^*$ die zu einer natürlichen Zahl die Darstellung in Basis b liefert. Beispielsweise soll $\rangle 257\langle_8 = 401$ gelten.

$$\rangle n\langle_b = \begin{cases} & \text{wenn} \\ & \text{sonst} \end{cases}$$

⌟

Aufgabe 1.2 (⋆⋆)**.** Beweisen Sie:

1. Sei x eine Ziffernfolge der Länge n und y eine Ziffernfolge der Länge m. Es gilt: $\langle x \cdot y \rangle_\beta = \langle x \rangle_\beta \times \beta^m + \langle y \rangle_\beta$
2. Zeigen Sie, dass für alle $n > 0$: $\langle \alpha^n \rangle + 1 = \beta^n$
3. Sei x eine Ziffernfolge der Länge n. Dann gilt: $\langle x \rangle_\beta < \beta^n$

⌟

Aufgabe 1.3 (Rechnen. ⋆)**.** Wir betrachten die Interpretation von Binärzahlen als vorzeichenlose und vorzeichenbehaftete Zahlen. Vervoll-

ständigen Sie folgende Tabelle:

$b \in \mathbb{B}^8$	Hex	$\langle b \rangle$	$[b]$
01101111			
	0xab		
		127	
			-128
11001001			
	0xff		
		64	
			127

⌟

Aufgabe 1.4 (and, or, xor. ⋆⋆)**.**

- Zeigen Sie mittels Wertetabelle, dass ˆ und | assoziativ sind.
- Welche der drei Operationen distribuieren miteinander? Sprich, für welche $\circ, \circ' \in \{\&, |, \hat{}\}$ gilt $a \circ (b \circ' c) = (a \circ b) \circ' (a \circ c)$.
- Drücken Sie $\overline{b}$ durch ˆ aus.

⌟

Aufgabe 1.5 (Das Vorzeichenbit. ⋆⋆)**.** Zeigen Sie, dass für alle Bitfolgen $b_{n-1} \cdots b_0 \in \mathbb{B}^n$ gilt:

$$b_{n-1} = 1 \quad \text{genau dann wenn} \quad [b] < 0$$

⌟

Aufgabe 1.6 (Vergleiche. ⋆⋆⋆)**.** In dieser Aufgabe untersuchen wir die Implementierung der Vergleichsoperationen für vorzeichenlose und vorzeichenbehaftete Bitfolgen.

1. Zeigen Sie: $\langle x \rangle < \langle y \rangle \iff c_n = 1$ für $c_n s_{n-1} \cdots s_0 = y - x$. Hierdurch erhalten wir eine Implementierung für den Vergleich $x \overset{u}{<} y$ zweier vorzeichenlos interpretierter Bitfolgen.
2. Geben Sie einen Implementierung des Vergleichs vorzeichenbehaftet interpretierter Bitfolgen $x \overset{s}{<} y$ an. Benutzen Sie hierzu $\overset{u}{<}$.

⌟

Aufgabe 1.7 (ASCII. ⋆⋆)**.** In dieser Aufgabe sollen sie verstehen, wie Zeichen gemäß der ASCII-Kodierung dargestellt werden. Der ASCII-Zeichensatz umfasst die folgenden 95 druckbaren Zeichen[4]

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

[4] Quelle: `http://de.wikipedia.org/wiki/ASCII`

und weitere 33 nicht druckbare Steuerzeichen. Insgesammt gibt es 128 ASCII-Zeichen. Wir fassen die Kodierung als Bijektion $\mathbb{A} \to [0, 127]$ zwischen der Menge der ASCII Zeichen und dem Intervall $[0, 127]$ auf. Überlicherweise stellt man ASCII-kodierte Texte als Bytefolgen dar, indem man jeweils ein Byte zur Kodierung eines Zeichens verwendet. Die die definierende Tabelle findet sich im Anhang des Skripts.

Einen Text, genannt Zeichenkette, stellt man als Reihung von Bytes dar, deren Wert dem ASCII-Wert des jeweilgen Buchstabens entspricht. Eine mögliche und gängige Art und Weise das Ende einer Zeichenkette zu kennzeichnen, ist, sie mit einem Byte mit dem Wert 0 abzuschließen. Eine so kodierte Zeichenkette nennt man **C string** oder ASCIIZ-Zeichenkette.

Geben Sie einen Ausdruck an, der …

1. abhängig von einer Zahl d im Intervall $[0, 9]$ den zugehörigen ASCII-Code liefert.
2. abhängig von der Position (a ist Position 0) p eines Buchstabens im Alphabet den ASCII-Code des zugehörigen Großbuchstabens liefert.
3. abhängig vom ASCII-Code eines Großbuchstabens den ASCII-Code des zugehörigen Kleinbuchstabens liefert.
4. ASCII-Code eines Buchstabens die Groß-/Kleinschreibung wechselt. Verwenden Sie weder Addition, Subtraktion noch Konditional.

⌟

Aufgabe 1.8 (⋆⋆)**.** Zeigen Sie, dass $\cdot \equiv \cdot \mod n$ eine Äquivalenzrelation ist. ⌟

Aufgabe 1.9 (Inverse und Absolutwerte. ⋆⋆⋆)**.**

- Neben 0^n gibt es in $\mathbb{B}^n$ noch eine weitere Bitfolge z mit $z +_n z = 0^n$. Welche? Welcher Zahl entspricht diese Bitfolge? Welche Bedeutung hat dieses Element für die Bildung des Absolutwertes?
- Zeigen Sie:

$$(b_{n-1} \dots b_{k+1} 10^k)^* = \overline{b}_{n-1} \dots \overline{b}_{k+1} 10^k \qquad \text{für alle } b \in \mathbb{B}^n$$

⌟

Aufgabe 1.10 (Überlauf. ⋆⋆⋆)**.**

1. Gegeben sind zwei Bitfolgen a und b der Länge n. Verwenden Sie Lemma 1.14 um folgende Aussagen zu zeigen:
   (a) $c_{n-1} = 1$ und $c_n = 0$ genau dann wenn $[a] \geq 0$ und $[b] \geq 0$ und $a +_n b$ überläuft.
   (b) $c_{n-1} = 0$ und $c_n = 1$ genau dann wenn $[a] < 0$ und $[b] < 0$ und $a +_n b$ überläuft.
   (c) Sind die Vorzeichen von $[a]$ und $[b]$ unterschiedlich, so kann die Addition nicht überlaufen.
2. Was ist der Unterschied zwischen Überlauf und Übertrag?

Aufgabe 1.11 (Vergleich. ⋆⋆⋆)**.** Betrachten Sie Lemma 1.14 und insbesondere den Kommentar zum Carry- und Overflow-Bit. Wie können Sie das Ergebnis des vorzeichenbehafteten Vergleichs $\lesssim$ und des vorzeichenlosen Vergleichs $\overset{u}{\lesssim}$ zweier Bitfolgen aus deren Subtraktion gewinnen? *Hinweis:* Nutzen sie Aufgabe 1.10 für den vorzeichenbehafteten Fall.

Aufgabe 1.12 (Shifts. ⋆⋆)**.**

1. Geben Sie einen Ausdruck von Bitoperationen an, der, gegeben eine Bitfolge $a_{n-1} \cdots a_0$ und $i < j < n$, die Bitfolge $a_j \dots a_i$ extrahiert, d.h. die Bitfolge $0^{n-(j-i+1)} a_j \dots a_i$ produziert.

2. Geben Sie einen Ausdruck an, der aus zwei Bitfolgen $a_{n-1} \dots a_0$, und $b_{n-1} \dots b_0$ sowie $i \in [0, n-1]$ mit die Bitfolge $a_{i-1} \dots a_0 b_{n-1} \dots b_i$ liefert. Verwenden die ausschließlich Bitoperationen auf Bitfolgen der Länge n.

3. Geben Sie jeweils einen Ausdruck an, der eine Bitfolge a mit 4, 5, und 31 multipliziert.

2 Maschinensprache

Wir beginnen mit einer sehr einfachen imperativen Programmiersprache, der Maschinensprache für den Befehlssatz der MIPS-Prozessorfamilie. Maschinensprache ist überschaubar und leicht erlernbar. Es gibt keine Abstraktionen, die Details der Programmausführung verbergen. Jede Zeile die wir schreiben, entspricht eins zu eins einem Prozessorbefehl: „What you see is what you execute". In der Praxis sind die Abstraktionen, die von imperativen Hochsprachen bereit gestellt werden, selbstverständlich hilfreich, da sie Arbeit sparen. Sie nehmen es dem Programmierer ab, sich um Details des Rechners auf dem das Programm laufen soll, zu kümmern, und ermöglichen die Entwicklung **portabler** Programme. Das sind Programme, die man auf Rechnern unterschiedlicher Prozessorfamilien ausführen kann. Dies ermöglicht ein **Übersetzer (engl. Compiler)**, der das in der Hochsprache geschriebene Programm nach Maschinensprache übersetzt.

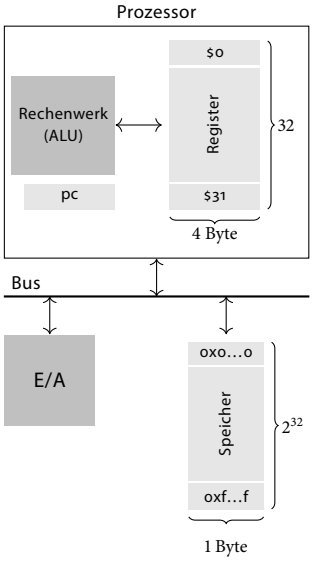
Durch die Beschäftigung mit Maschinensprache lernen wir, wie ein Prozessor arbeitet und welche Operationen er bereitstellt. Dieses Wissen hilft uns, besser nachvollziehen zu können, warum man imperative Sprachen (wie C und Java) braucht, und welche Probleme diese auf welche Weise lösen. Zu untersuchen, wie die Maschinensprach-Programme aussehen, die ein Hochsprachen-Programm implementieren, hilft uns einerseits, besser zu verstehen, welche Kosten (Laufzeit, Speicherverbrauch) ein Programm verursacht, aber auch wie potentielle Sicherheitslücken aussehen, wie man diese ausnutzt bzw. schließt.

2.1 Der von-Neumann Rechner

Die grundlegenden Konzepte des Rechneraufbaus sind seit 70 Jahren die gleichen. Erstmals ausführlich dokumentiert wurden sie von dem ungarisch-amerikanischen Mathematiker John von Neumann in seinem bahnbrechenden Bericht über den Rechner EDVAC [Neu93].

Der Entwurf hat sich bis heute kaum geändert. Ein Rechner besteht aus einem (oder auch mehreren) Prozessoren. Ein **Prozessor (engl. Central Processing Unit, CPU)** ist aber nichts anderes als ein Busteilnehmer, der über den Bus mit anderen Komponenten kommuniziert. Beispielsweise kann er Daten in den Speicher schreiben oder aus ihm lesen.

Der Bus dient als zentrales Kommunikationsmittel der Komponenten der Rechenanlage. Jede Komponente, auch der Prozessor, kann Anfragen an den Bus richten, beispielsweise: Lade 4 Bytes von Adresse `0xe0000000`. Dies meint nicht notwendigerweise eine Hauptspeicheradresse. Der **Bus-Controller** verwaltet eine Liste, die jeder Komponente einen Speicherbereich zuordnet. So könnte obige Adresse beispielsweise zu einem Puffer in einer im System vorhandenen Netzwerkkarte gehören. Nicht in jeder Bus-Kommunikation ist der Prozessor involviert. Beispielsweise kann das Betriebssystem jener Netzwerkkarte auftragen, 4 Kilobyte Daten über den Bus in den Hauptspeicher zu transferieren.



2.2 Der MIPS-Prozessor

Der Kern des MIPS Prozessors ist ein fünfstufiges **Fließband (engl. Pipeline)**, das die Abarbeitung der Befehle organisiert. Ein Befehl „durchwandert" mehrere (meistens alle) Stufen der Pipeline. Geht ein Befehl alle fünf Stufen, so vergehen fünf Taktzyklen, bis sein Ergebnis zurückgeschrieben ist. Allerdings arbeiten die Stufen parallel, so dass fünf unterschiedliche Befehle in fünf unterschiedlichen Stufen gleichzeitig ausgeführt werden. Kann in jedem Taktzyklus ein neuer Befehl in die Pipeline eingefügt werden, so steht nach fünf Takten Wartezeit in jedem weiteren Takt ein Ergebnis zur Verfügung. MIPS-Befehle unterteilen sich in drei Klassen:

Rechenbefehle beziehen ihre Operanden aus der **Registerbank (engl. register file)** und manchmal auch aus in das Befehlswort eingebauten Konstanten (engl. immediate). Die Registerbank ist ein kleiner Speicher auf dem Prozessor, in dem 31 Worte der **Breite** 32 Bit gespeichert werden können. Das Register 0 ist fest auf den Wert 0 verdrahtet; es zu beschreiben hat keinen Effekt. Die Operationen werden vom **Rechenwerk (engl. arithmetic logical unit (ALU))** durchgeführt. Die ALU implementiert die arithmetischen und bitweisen Operationen, die wir im letzten Kapitel kennengelernt haben. MIPS hat eine **Wortbreite** von 32 Bit, sprich, der Ausgang der ALU liefert ein 32-Bit Wort.

Speicherbefehle transferieren Werte zwischen Registerbank und Hauptspeicher. Hierbei wird die ALU verwendet um eine 4 Byte lange

Takt	Instruktion				
	1	2	3	4	5
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	ME	EX	ID	IF	
5	WB	ME	EX	ID	IF
6		WB	ME	EX	ID
7			WB	ME	EX
8				WB	ME
9					WB

Abbildung 2.1: Fließbandverarbeitung (Pipelining). Die Zeilen sind Prozessortakte, die Spalten entsprechen Befehlen. Anfangs ist die Pipeline leer. Die Ausführung des ersten Befehls beginnt: Er belegt die IF Stufe. Im nächsten Takt wird die ID Stufe des ersten und die IF Stufe des zweiten Befehls ausgeführt. In Takt 5 ist die Pipeline voll ausgelastet. Sie bearbeitet fünf Befehle gleichzeitig. Nach 5 Takten liegt das Ergebnis des ersten Befehls vor. Nach 9 Takten das Ergebnis aller 5 Befehle.

In der Praxis kann eine Pipeline-Stufe länger als einen Takt brauchen. Zum Beispiel muss MEM unter Umständen auf einen Speicherzugriff warten. In diesem Fall blockiert (stalls) das Fließband solange.

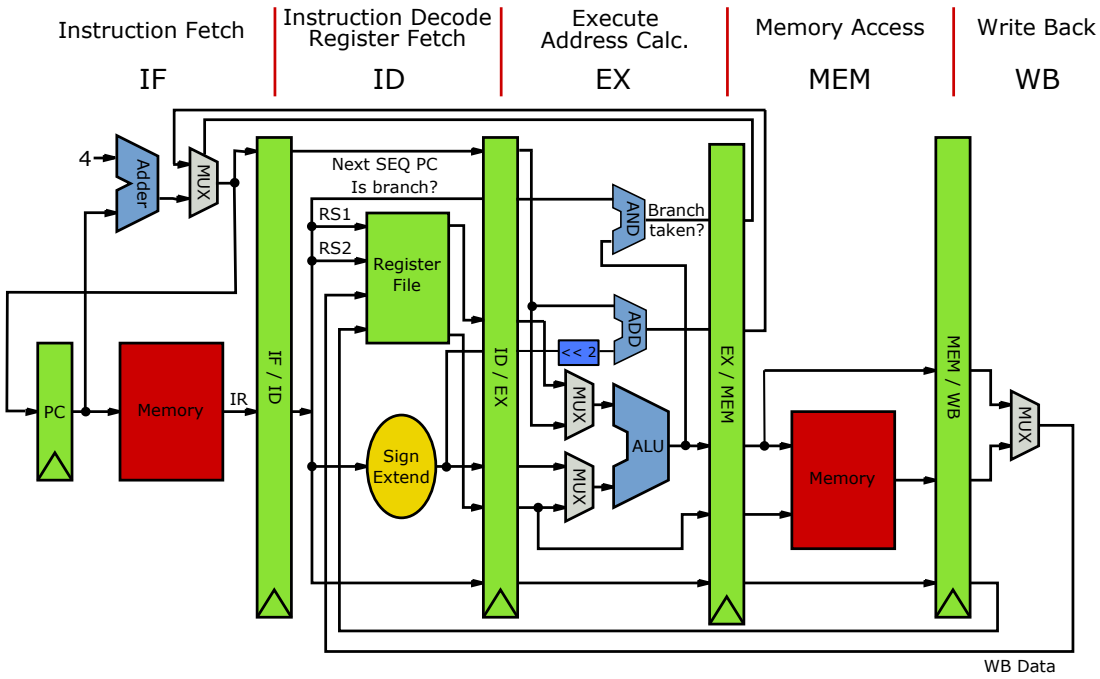


Abbildung 2.2: Die MIPS Pipeline. Die Abbildung basiert auf `https://en.wikipedia.org/wiki/MIPS_instruction_set#/media/File:MIPS_Architecture_(Pipelined).svg`

Adresse zu generieren, an der der Speicher dann gelesen/geschrieben wird. An jeder Adresse kann ein Byte gespeichert werden.

Sprungbefehle beeinflussen den Programmablauf. Der Befehlszeiger (Register PC (engl. PC = program counter)) enthält die Speicheradresse des nächsten abzuarbeitenden Befehls. Bei einem Rechen- oder Speicherbefehl wird PC automatisch um 4 erhöht, da jedes Befehlswort in MIPS 4 Byte lang ist. Mit einem Sprungbefehl kann man PC auf eine andere Adresse setzen um die Programmausführung an einer anderen Stelle fortzusetzen.

Die fünf Stufen der Befehlsabarbeitung sind:

1. Laden der Instruktion (IF)

   Das Register PC (engl. PC = program counter) enthält die Speicheradresse des nächsten abzuarbeitenden Befehls. Aus dem Speicher wird ein Befehlswort (bei MIPS 4 Bytes) von dieser Adresse geladen. Das geladene Befehlswort wird durch ein prozessor-internes Register (IR) an die nächste Stufe kommuniziert.

2. Dekodieren der Instruktion (ID)

   In dem 4 Byte langen Befehlswort ist kodiert, um was für einen Befehl es sich handelt und woher er seine Operanden bezieht. Betrachten wir beispielsweise das Befehlswort 0x00641021. Es gliedert sich in mehrere Felder, die der Prozessor nun interpretiert: Die Felder *opcode*

und *funct* geben an, dass …

$$0x00641021 = \underbrace{000000}_{opcode} \cdot \underbrace{00011}_{rs} \cdot \underbrace{00100}_{rt} \cdot \underbrace{00010}_{rd} \cdot \underbrace{00000}_{\text{unbenutzt}} \cdot \underbrace{100001}_{funct}$$

$$= \begin{array}{|c|c|c|c|c|c|} \hline 0 & 3 & 4 & 2 & 0 & 33 \\ \hline 6 & 5 & 5 & 5 & 5 & 6 \\ \end{array}$$

(a) der Befehl eine Addition ist (*opcode* = 0 und *funct* = 33).

(b) die Operanden aus den Registern, deren Nummern in den Feldern *rs* und *rt* stehen, zu lesen sind. Hier: *rs* = 3 und *rt* = 4.

(c) das Ergebnis in das Register dessen Nummer im Feld *rd* steht, geschrieben werden soll. Hier: *rd* = 2.

Bezieht ein Befehl seine Operanden aus der Registerbank, so werden die Inhalte der entsprechenden Register am Ausgang der Registerbank bereit gestellt und an die nächste Stufe weitergereicht. Manche MIPS-Befehle stellen den Wert eines Operanden direkt als 16 Bit lange, in das Befehlswort **eingebaute Konstante (engl. immediate)** zur Verfügung. Diese wird dann in dieser Stufe auf 32 Bit erweitert. Anhand des Opcodes werden Steuersignale erzeugt, die die Funktion der Schaltkreise in den folgenden Stufen steuern: Zum Beispiel: Ist ein Befehl ein Speicherbefehl oder ein Sprungbefehl? Welche Funktion soll die ALU (siehe unten) ausführen, wenn der Befehl ein Rechenbefehl ist?

3. Ausführen (EX)

   Hier werden die geladenen Operanden vom **Rechenwerk (engl. arithmetic logical unit (ALU))** verarbeitet und das Ergebnis wieder mittels eines Registers (EX/MEM) der nächsten Stufe zur Verfügung gestellt.

4. Speicherzugriff (MEM)

   Ist der bearbeitete Befehl ein Speicherbefehl (Laden oder Speichern), so wird nun mit der, in der vorigen Stufe berechneten Adresse, auf den Speicher zugegriffen.

5. Schreiben des Ergebnisses (WB)

   Hier wird das Ergebnis des Befehls in die Registerbank zurückgeschrieben.

2.3 Der Assembler

Da man die Befehle nicht von Hand in Zahlen kodieren will, verwendet man einen **Assembler**. Er übersetzt eine textuelle Darstellung, den **Assemblercode (engl. assembly code)** des Maschinencodes in die binäre Darstellung, den **Binärcode (engl. binary code)**. Hierzu liest der Assembler eine Assemblerdatei ein, die neben Befehlen für den Prozessor noch eine Beschreibung des Datensegments und Direktiven für den Assemblierungsvorgang selbst enthalten kann. Der Assembler verarbeitet diese Datei dann zu einer so genannten Binärdatei (auch oft Objektdatei, vom englischen **object code**). Diese enthält die binäre Darstellung der Befehle und Daten, sowie Informationen für den **Linker**, der mehrere solche Objektdateien zu einem ausführbaren Programm zusammenführt. Dieser Vorgang ist in Abbildung 2.3 dargestellt.

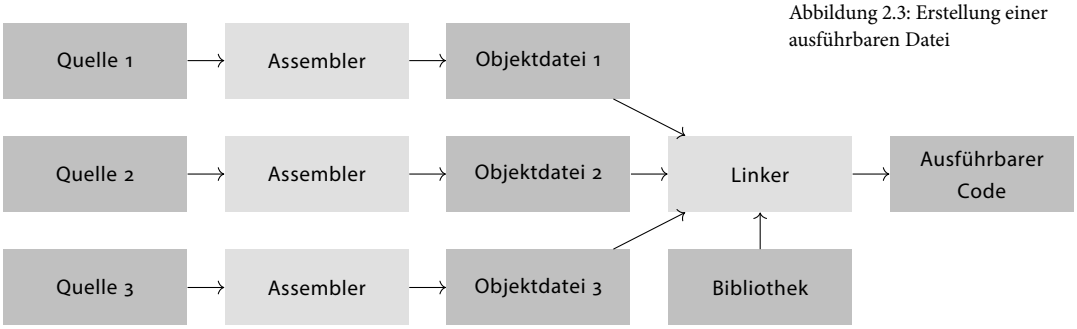


Abbildung 2.3: Erstellung einer ausführbaren Datei

Betrachten wir die Assemblierung an einem Beispiel. Die in Abbildung 2.4 (hexadezimal kodierten) Wörter sind, in der Interpretation der MIPS-Maschinensprache, ein Unterprogramm, das die Fakultät des Wertes berechnet, der zu Beginn in Register `$4` steht.

```
24020001 addiu $2 $0  1
04010002 bgez  $0 2
70441002 mul   $2 $2 $4
2484ffff addiu $4 $4 -1
1c80fffd bgtz  $4 -3
03e00008 jr    $31
```

Abbildung 2.4: Eine Folge von Wörtern (à 4 Bytes) in Hexadezimalschreibweise und ihre Disassemblierung

Für Menschen ist das kaum nachvollziehbar. Rechts neben der Hexadezimal-Darstellung des Maschinenwortes steht eine textuelle Darstellung des Befehls, die für Menschen leichter zu lesen ist. Jeder Befehl hat ein **Mnemonic**, einen Kurznamen, der seiner Bedeutung entspricht. Zahlen mit vorangestelltem Dollarzeichen bezeichnen Prozessorregister; Zahlen ohne Dollarzeichen sind Konstanten, die in den Befehlsstrom eingebaut sind (engl. Immediates). Wir sehen, dass jeder Befehl auf einem oder mehreren Registern operiert und manche Befehle auch Konstanten mit einbeziehen. Beispielsweise bildet der Befehl `addiu $2 $0 1` die Summe aus dem Inhalt des Registers `$0` und der Konstanten `1` und legt das Ergebnis in Register `$2` ab. Die Befehle `bgez` und `bgtz` sind Sprungbefehle. `bgtz` überprüft, ob der Inhalt des Registers (hier `$4`) größer 0 ist (gtz = greater than zero). Trifft dies zu, wird der Befehlszeiger *n* Befehle weiter gerückt (relativ zum nachfolgenden Befehl). Trifft dies nicht zu, hat der Befehl keine Wirkung. Bei Sprungbefehlen nennt die Konstante *n* auch **Versatz (engl. offset)**. Der Befehl `bgez` wirkt hier als unbedingter Sprung: Er vergleicht, ob der Inhalt des Registers (hier

$0) größer oder gleich 0 ist. Da $0 immer den Wert 0 enthält, ist diese Bedingung immer wahr und der Sprung wird immer durchgeführt.

Nun sind gerade solche Sprungbefehle auch in dieser Darstellung mühsam vom Menschen zu kodieren. Fügt man nach dem bgez beispielsweise einen Befehl ein, so muss der Versatz von bgtz von Hand angepasst werden. Die Konstante selbst muss durch Abzählen der Befehle ermittelt werden. Da dies mühselig ist, wird das vom Assembler übernommen. Hierzu bietet der Assembler sogenannte **Marken (engl. label)** an. Das Einsetzen der korrekten Werte übernimmt dann der Assembler. Der folgende Code zeigt das Unterprogramm fac in MIPS Assemblersprache. Aus ihm wurde der Binärcode in Abbildung 2.4 erzeugt. Die Sprungmarken sind durch Bezeichner gegeben, denen ein

```
    .text
    .globl fac
fac:
    li    $v0 1       # Lade Konstante 1 nach $v0
    b     check       # Verzweige nach check
loop:
    mul   $v0 $v0 $a0 # Multipliziere $v0 mit $a0 und speichere
                      # Ergebnis in $v0
    addiu $a0 $a0 -1  # Verringere $a0 um 1
check:
    bgtz  $a0 loop    # Verzweige nach loop, wenn $a0 > 0
    jr    $ra         # Kehre zum Aufrufer zurueck
```

Abbildung 2.5: Ein Unterprogramm, das die Fakultät $n!$ berechnet. Die Zahl n wird hierbei im Register $a0 erwartet.

Doppelpunkt folgt (hier im Beispiel fac, loop und check). Sie stehen für die Adresse des nachfolgenden Datums (hier ein Befehl). Der Assembler kann nun den passenden Versatz zum Sprungziel selbst berechnen.

Ein weiterer Unterschied ist die Verwendung von Registernamen. Der MIPS-Prozessor hat 32 allgemeine Register, die jeweils eine 32-Bit Zahl speichern können. Neben den 32 allgemeinen gibt es noch 3 Spezialregister, darunter der Befehlszeiger pc (program counter). Das erste der 32 allgemeinen Register ($0 oder $zero in der Assemblersprache) enthält immer den Wert 0. Schreiboperationen darauf haben keine Wirkung. Die anderen 31 haben in der Assemblersprache einen speziellen Namen (neben der bloßen Nummer), welcher der Rolle des Registers in der sogenannten Aufrufkonvention (später darüber mehr) geschuldet ist. Beispielsweise enthält das Register $4 das erste Argument bei einem Aufruf eines Unterprogramms. Daher hat es auch den Namen $a0 (argument 0). Abbildung 2.12 zeigt die verwendeten Registernamen vollständig.

Des Weiteren sind in obigem Beispiel noch zwei Assembler-Direktiven zu sehen:

.text erklärt dem Assembler, dass nun ausführbarer Code folgt. Die Markierung, ob es sich beim folgenden Inhalt um Code oder Daten handelt ist wichtig, da beim Laden des Programms durch das Be-

triebssystem Code und Daten an bestimmte, festgelegte Adressen geschrieben werden.

`.globl fac` macht die Marke `fac` für andere Übersetzungseinheiten sichtbar. Das heißt, dass der Linker in anderen Objektdateien, die Referenzen auf eine Marke `fac` enthalten, die Adresse des Befehls der in dieser Übersetzungseinheit mit `fac` markiert ist, einsetzt. In diesem Beispiel markiert `fac` den Beginn der Routine, die die Fakultät berechnet. Diese kann dann von anderen Übersetzungseinheiten gerufen wird. Die Marken `check` und `loop` sind nicht global. Ihre Namen können also in anderen Übersetzungseinheiten in anderem Kontext wiederverwendet werden, ohne dass es einen Einfluss auf diese Übersetzungseinheit hat.

Weiter fällt auf, dass die Mnemonics leicht abweichend sind. Der Befehl `li` (load immediate) ist eine **Pseudo-Instruktion** zum Laden einer beliebigen Konstante. Es ist ein Kurzname, den der Assembler bereitstellt, um dem Programmierer das Leben angenehmer zu machen. Je nach Wert der Konstante werden ein oder zwei Befehle erzeugt, da nicht jede Konstante in einem einzelnen Befehl kodiert werden kann.

Der Befehl `jr` (jump register) entnimmt dem angegebenen Register (hier `$31`) eine Adresse und setzt den Befehlszeiger darauf. Der Aufrufer des Unterprogramms legt per Konvention in diesem Register die Adresse ab, an die es an seinem Ende zurückkehren soll. Daher heißt das Register in der Assemblersprache `$ra` (englisch: return address).

Ein weiteres Beispiel ist der Befehl `b check`. in der Disassemblierung zuvor steht aber `bgez $0 2`. Die 2 ist der Versatz für die Sprungmarke `check`: Man sieht leicht, dass die Anweisung, die mit `check` markiert ist **in der Disassemblierung drei** Befehle weiter steht. Der Versatz wird relativ zur **nachfolgenden** Anweisung kodiert. `b` (branch) ist auch eine Pseudo-Instruktion, die unbedingt verzweigt. Dieser Befehl wird häufig benötigt, existiert jedoch im MIPS-Befehlssatz nicht. Es existieren bedingte Sprungbefehle (zum Beispiel `bgez`), die einen Registerinhalt auf 0 vergleichen und bei positivem Ergebnis verzweigen. Da das erste Register `$0` aber immer 0 enthält, bietet der Assembler `b marke` als Abkürzung für `bgez $0 marke` an.

Zuletzt geben wir noch ein Hauptprogramm an, das unsere Routine `fac` aufruft. Nehmen wir an, wir wollen die Fakultät von 10 auf der Konsole ausgeben.

```
    .text
    .globl main
main:
    li   $a0 10  # Lade 10 in Register $a0
    jal  fac     # Rufe Unterprogramm fac
    move $a0 $v0 # Ergebnis ist in Reg $v0 -> verschiebe nach $a0
    li   $v0 1   # Lade Nummer des Systemaufrufs ”print_int” in Reg $v0
    syscall      # Rufe das Betriebssystem
    li   $v0 10  # Lade Nummer des Systemaufrufs ”exit” in Reg $v0
    syscall      # Dieser Systemaufruf beendet das Programm
```

Die Marke `main` dient als Einsprungspunkt in das Programm. Ein Code für die Operation, die das Betriebssystem ausführen soll, wird zuvor im Register `$v0` abgelegt. Ein Wert von 1 bedeutet, dass das Betriebssystem die Zahl, die in Register `$a0` steht, auf der Konsole ausgeben soll. Der Systemaufruf 10 beendet das Programm. Der Befehl `syscall` ruft das Betriebssystem (Anhang B fasst die in den gängigen Simulatoren verfügbaren Systemaufrufe zusammen). `jal` (jump and link) setzt den Befehlszeiger auf die Adresse der angegebenen Marke und speichert zudem die Adresse des auf `jal` folgenden Befehls im Register `$ra`. Somit kann die aufgerufene Routine zurückkehren, sprich die Ausführung nach `jal` fortführen.

2.3.1 Adressierung

Wir haben bereits gesehen, dass man mittels Marken Adressen Namen geben kann. Ob die konkrete Adresse von Bedeutung ist, hängt vom Kontext der Marke ab. Beispielsweise verwenden bedingte Sprungbefehle in MIPS (wie `bne` und `beq`) **relative** Adressen. Das heißt, sie ändern den Befehlszeiger **relativ** zur aktuellen Position, indem ein Versatz auf ihn addiert wird. Verwendet ein Befehl relative Adressierung, so ist die **absolute** Adresse irrelevant. Sind das Sprungziel und alle Verwender der Marke in einer Übersetzungseinheit, so kann der Assembler den korrekten Versatz direkt eintragen. Alle Sprungbefehle in MIPS, die relativ adressieren, beginnen mit einem `b` für **branch** (verzweige).

Bei der **absoluten** Adressierung existiert kein Versatz, sondern eine konkrete Adresse. Alle MIPS Sprungbefehle, die mit `j` beginnen, verwenden absolute Adressen. Der Befehlszeiger wird einfach auf die gegebene Adresse gesetzt und nicht relativ inkrementiert. Hierbei ergibt sich das Problem, dass der Assembler zum Zeitpunkt der Assemblierung nicht immer weiß, an welcher Stelle das referenzierte Stück Code letztlich landen wird[1]. Manche Adressen stehen erst beim Zusammenbinden oder Laden des Programmes fest (siehe Abschnitt 2.8) und müssen dann eingesetzt werden.

Allen Adressierungsarten ist gemein, dass die Reichweite des adressierbaren Speichers begrenzt ist. So kann man bei MIPS beispielsweise 2^{15} Befehle nach oben und $2^{15} - 1$ Befehle nach unten verzweigen. Absolute Sprünge erlauben bei MIPS einen Umfang von 2^{26} Instruktionen (Dies entspricht einem Speicherbereich von 256 Megabyte). Sprungziele außerhalb dieses Bereichs können nicht direkt angesprungen werden und müssen anders implementiert werden: Beispielsweise durch mehrere hintereinander geschaltete Sprünge, oder durch das Laden der absoluten Adresse in ein Register und einen anschließenden **indirekten** Sprung. Indirekt bedeutet hier, dass das Sprungziel nicht als Konstante im Befehlsstrom liegt.

[1] Das Programm könnte sich aus vielen Übersetzungseinheiten (in Form von Bibliotheken) zusammensetzen, die alle **getrennt** voneinander assembliert werden.

2.4 Das Ausführungsprotokoll

Um ein Gefühl für eine Programmiersprache zu bekommen, ist es hilfreich, die Ausführung einiger Programme Schritt für Schritt nachzuvollziehen. Hierzu benutzen wir Ausführungsprotokolle. In imperativen Sprachen besteht die Ausführung eines Programms aus mehreren (potentiell unendlich vielen) Schritten. In jedem Schritt wird eine **Anweisung** (daher der Name imperativ) abgearbeitet. Eine Anweisung hat eine **Wirkung** (engl. effect). Ein Effekt ist entweder die Änderung des **Zustands** des Rechners oder eine Interaktion mit der Umgebung (engl. environment). Unter einer Zustandsänderung versteht man das Ändern des Inhalts eines Registers oder einer Speicherzelle. Unter der Umgebung versteht man die anderen Bestandteile des Rechners (zum Beispiel Tastatur, Bildschirm, Festplatte, etc.). Der Zugriff darauf wird meist vom Betriebssystem verwaltet[2]. In Kapitel 5 werden wir den Zustandsbegriff und die Programmausführung formal betrachten.

[2] In MIPS Assembler müssen wir hierzu entsprechende Systemaufrufe mittels `syscall` absetzen.

Ein Ausführungsprotokoll beschreibt den schrittweisen Ablauf eines Programms bezüglich eines **Startzustandes**. Betrachten wir nochmal das Programm aus Abbildung 2.4 und nehmen an, die Adresse des ersten Befehls ist `0x00400000`.

```
0x00400000: addiu $v0 $0 1
0x00400004: bgez  $0 2
0x00400008: mul   $v0 $v0 $a0
0x0040000c: addiu $a0 $a0 -1
0x00400010: bgtz  $a0 -3
0x00400014: jr    $ra
```

Das Ausführungsprotokoll ist eine Tabelle, die die Änderung des Zustands nach der Ausführung jedes Befehls kenntlich macht. Für jeden Teil des Zustands der von Interesse ist, gibt es im Ausführungsprotokoll eine Spalte. Eine Zeile gibt somit den relevanten Teil des Zustands vor der Ausführung der nächsten Anweisung an. Beachten Sie, dass die Adresse des als nächsten auszuführenden Befehls (Inhalt des Registers `$pc`) ein Teil des Effektes einer Befehlsausführung ist.

Abbildung 2.6: Ausführungsprotokoll des Beispielprogramms auf einem Zustand in dem das Register `$a0` den Wert 3 hat.

`$pc`	`$v0`	`$a0`
`0x004000000`	?	3
`0x004000004`	1	3
`0x004000010`	1	3
`0x004000008`	1	3
`0x00400000c`	3	3
`0x004000010`	3	2
`0x004000008`	3	2
`0x00400000c`	6	2
`0x004000010`	6	1
`0x004000008`	6	1
`0x00400000c`	6	1
`0x004000010`	6	0
`0x004000014`	6	0

Abbildung 2.6 zeigt ein Ausführungsprotokoll dieses Programms mit einem Startzustand, in dem das Register `$a0` den Wert 3 hat. Um nun den Zustand nach der Ausführung des Befehls zu ermitteln, müssen wir den Effekt des Befehls kennen. Anhang B spezifiziert die Effekte der für uns relevanten MIPS-Befehle. Zu Beginn steht der Befehlszeiger auf der Adresse `0x00400000`. Der Befehl an dieser Adresse `addiu $v0 $0 1` addiert auf den Inhalt des Registers `$0` den Wert 1 und speichert das Ergebnis in Register `$v0` Zusätzlich erhöht er den Befehlszeiger um 4.

Somit ist der nächste auszuführende Befehl an der Adresse `0x00400004`. `bgez $0 2` „springt“ $2 + 1$ Befehle weiter, wenn der Inhalt des Registers `$0` gleich 0 ist, was es immer ist. Es ist also ein unbedingter Sprung, der den Befehlszeiger 3 Befehle weiter setzt. Und so weiter und so fort. Wir brechen das Ausführungsprotoll beim Erreichen des Sprungbefehls `jr $ra`, der zur Adresse in `$ra` springen würde.

2.5 Das Datensegment

Neben dem Code enthalten viele Programme auch **statische** Daten. Statisch bedeutet hier, dass deren Umfang *vor* der Ausführung bekannt ist, also nicht zur Laufzeit des Programms variiert. Der Speicherplatz für die statischen Daten wird beim Assemblieren und Binden schon veranschlagt und in der Objektdatei vermerkt. Beim Laden des Programms wird er angefordert und bleibt bis zum Ende des Programms reserviert. Oft sind statische Daten schon bei Programmstart mit Werten belegt. Ein typisches Beispiel sind im Programm auftretende Zeichenketten oder Konstante größeren Umfangs (beispielsweise Reihungen von Zahlen). Zum Beispiel muss ein Programm, das die Zeichenkette `Hallo Welt` ausgibt, die Zeichenkette natürlich irgendwo enthalten.

In der Assemblerdatei können statische Daten nach der Direktive `.data` vereinbart werden. Zum reinen Anfordern von Platz (undefinierten Inhaltes) dient die Direktive `.space` *n*, wobei *n* die Anzahl der zu reservierenden Bytes ist. Um später auf die Adressen dieses Speichers Bezug nehmen zu können, können diese Bereiche mit Marken versehen werden. Beispielsweise legt

```
    .data
some_bytes:
    .space 1000
```

einen Bereich von 1000 Bytes undefinierten Inhaltes an, auf dessen Adresse mit der Marke `some_bytes` Bezug genommen werden kann. Eine Marke referenziert stets die Adresse der nächsten Vereinbarung, die Speicher belegt.

Zur Bequemlichkeit des Programmierers existieren weitere Direktiven, um das Anlegen von statischem Speicher mit vordefiniertem Inhalt zu erleichtern. Die Direktiven `.byte`, `.half` und `.word` legen Bytes, Halbwörter (MIPS-Slang für 2 Byte lange Ganzzahlen) und Wörter (4 Byte lange Ganzzahlen) an. `.ascii` *str* und `.asciiz` *str* legt ein Bereich an, der initial mit den Bytes gefüllt ist, die den ASCII-Werten der Zeichen der Zeichenkette *str* entsprechen. Im Falle von `.asciiz` wird noch ein Null-Byte angehängt. Diese Darstellung von Zeichenketten ist vor allem in der Programmiersprache C verbreitet. Zum Beispiel entspricht

```
    .data
hello:
    .asciiz "Hello World"
```

der Vereinbarung

```
    .data
hello:
    .byte 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20
    .byte 0x57, 0x6f, 0x72, 0x6c, 0x64, 0x00
```

Für den Zugriff der Daten ist auch die sogenannte **Ausrichtung (engl. Alignment)** wichtig. Die Ladebefehle des MIPS-Prozessors schreiben dies vor: Wird ein Wort (4 Bytes) geladen, so muss dessen Adresse

durch 4 teilbar sein. Wird ein Halbwort geladen, so muss dessen Adresse entsprechend durch 2 teilbar sein. Um im Datensegment eine gewisse Ausrichtung herzustellen, verwendet man die Direktive `.align` *n*. Dies lässt das nachfolgende Element im Datensegment an einer Adresse beginnen, die durch 2^n teilbar ist.

Zum Beispiel würde folgendes Programm

```
    .data
    .ascii "Hallo"
x:
    .byte 8, 0, 0, 0

    .text
    .globl main
main:
    lw $t0 x
```

zu einer Prozessorausnahme[3] beim Ausführen des Ladebefehls führen, da die Adresse von x, in diesem konkreten Beispiel `0x10000005`, nicht durch 4 teilbar ist. Mit korrekter Ausrichtung liest sich das Programm so:

```
    .data
    .ascii "Hallo"
    .align 2
x:
    .byte 8, 0, 0, 0

    .text
    .globl main
main:
    lw $t0 x
```

Es sei abschließend bemerkt, dass man in der Praxis die statischen Daten noch weiter unterteilt: Einerseits in konstante Daten, deren Wert nicht mehr verändert werden darf. Dies wird meist dadurch sichergestellt, dass diese in einen Adressbereich geladen werden, der dann vom Betriebssystem als nicht änderbar gesetzt wird. Das Einhalten dieser Einschränkung geschieht mit Hilfe des Prozessors, genauer der virtuellen Speicherverwaltung. Andererseits unterscheidet man bei den Daten, die initialisiert und änderbar sind, solche, die zu 0 oder zu anderen Werten initialisiert werden sollen. Erstere brauchen in der Objektdatei und der ausführbaren Datei keinen Platz zu belegen, da sie entsprechend markiert werden. Das Betriebssystem setzt dann beim Laden des Programms den Speicher an den entsprechenden Adressen auf 0.

[3] Der Prozessor kommt in einen Zustand, in der die Fortführung der Programmausführung nicht mehr möglich ist. Er unterbricht dann die Ausführung und springt an eine spezielle Fehlerbehandlungsroutine im Betriebssystem.

2.6 Zusammengesetzte Datentypen

Der grundlegende Datentyp, dessen Werte ein Prozessor verarbeitet, ist die Bitfolge fester Länge. Diese Bitfolgen werden verwendet um Zahlen und Adressen zu repräsentieren. Diese Datentypen nennen wir **Basistypen (engl. base type)**. In der Praxis hat man jedoch mit komplexer strukturierten Daten zu tun. Zum Beispiel besteht ein Bild aus Breite × Höhe Bildpunkten (engl. Pixel). Jeder Bildpunkt besteht wiederum aus einem Rot-, Grün- und Blau-Wert. Solche Daten nennen wir **zusammengesetzt** (engl. compound data type), da sie aus mehreren einzelnen Elementen bestehen. Wie in dem Beispiel zu sehen, können zusammengesetzte Daten aus weiteren zusammengesetzten Daten bestehen (Bild aus Bildpunkten, Bildpunkt aus RGB-Werten). Wir besprechen im Folgenden zwei zusammengesetzte Datentypen detaillierter: Die **Reihung (engl. array)** und den **Verbund (engl. struct)**.

2.6.1 Reihungen

Eine Reihung (manchmal auch **Puffer (engl. buffer)** genannt) umfasst viele gleichartig strukturierte Daten. Ein solches Datum heißt **Element** der Reihung. Die Elemente einer Reihung legt man hintereinander im Speicher ab. Da alle Elemente gleichartig strukturiert sind, belegt jedes Element gleich viel Speicherplatz. Die Adresse des ersten Elements heißt **Basisadresse (engl. base address)**. Seien im Folgenden: b die Basisadresse, s die Größe eines Elements und l die Anzahl der Elemente in der Reihung. Die Adresse des i-ten Elements der Reihung lässt sich leicht arithmetisch bestimmen:

$$\text{Adresse des } i\text{-ten Elements} = b + s \cdot i \tag{2.1}$$

Dies impliziert, dass das erste Element den Index 0 hat[4]. Das letzte Element hat dann die Adresse $b + s \cdot (l - 1)$. Es gibt unterschiedliche Techniken um die Länge einer Reihung zu repräsentieren:

- Die einfachste Art ist, die Anzahl der Elemente l zu speichern.
- Oft merkt man sich aber auch die Adresse der ersten Speicherzelle **hinter** dem letzten Element, genannt **Endadresse**

  $$e = b + s \cdot l. \tag{2.2}$$

  Die Motivation für diese Variante ist, dass man sehr häufig über Reihungen iteriert. Hierbei erhöht man meist die Basisadresse nach jeder Iteration um s Bytes. Ist sie bei e angelangt, wurden alle Elemente besucht und die Iteration kann abgebrochen werden (→ Beispiel 2.1).
- Man kann auch auf das explizite Speichern der Länge verzichten, indem man das Ende der Reihung durch ein speziellen Wert, der sonst in der Reihung nicht vorkommen darf, kenntlich macht (→ Beispiel 2.2).

Beispiel 2.1 (Endadresse)**.** Betrachten wir folgenden Code, der die Elemente einer Reihung von Wörtern (4 Byte Ganzzahlen) im Register `$v0` aufaddiert.

[4] Edsger Dijkstras Bemerkung [Dij82] macht deutlich, warum es sinnvoll ist, mit der 0 und nicht bei der 1 zu beginnen. Ein weiterer Vorteil davon, dem ersten Element einer Reihung den Index 0 zuzuweisen ist, dass man den Index direkt in der Adressarithmetik wie in (2.1) verwenden kann. Hätte das erste Element den Index 1, so müsste man (2.1) zu $b + s \cdot (i - 1)$ anpassen.

```
    # $a0 enthaelt die Basisadresse einer Reihung von Wörtern
    # $a1 enthaelt die Endadresse dieser Reihung
    li    $v0 0          # Initialisiere $v0 mit 0
    b     cmp            # Springe zum Test der Bedingung
loop:
    lw    $t0 0($a0)     # Lade ein Wort (4 Byte) von Adresse $a0
    addu  $v0 $v0 $t0    # Addiere den geladenen Wert zu $v0
    addiu $a0 $a0 4      # Erhöhe Basisadresse um 4
cmp:
    bne $a0 $a1 loop     # springe zu loop wenn $a0 != $a1
```

In jeder Iteration wird von der Basisadresse geladen und dann die Basisadresse um ein Wort (4 Byte) weiter gerückt. Ist nach dem letzten Weiterrücken die Basisadresse gleich der Endadresse, wird nicht mehr erneut iteriert.

Hätte man die Länge der Reihung nicht durch den Ende-Zeiger, sondern durch die Anzahl der Elemente repräsentiert, müsste man entweder zusätzlich einen Zähler in der Schleife hochzählen oder vor Beginn der Schleife die Endadresse aus der Länge nach (2.2) berechnen. ⌟

Beispiel 2.2 (Zeichenketten)**.** Betrachten wir eine Zeichenkette (engl. string). Nehmen wir an, wir kodieren Buchstaben gemäß des ASCII-Standards (→ Aufgabe 1.7). Jeder Buchstabe benötigt dann 7 Bit, wir können ihn also durch 1 Byte darstellen. Eine Zeichenkette ist nun eine Reihung von Bytes. Eine Möglichkeit die Länge der Zeichenkette zu kodieren ist, ein Nullbyte (ein Byte mit dem Wert 0) an die Zeichen der Zeichenkette anzuhängen. Dies wird zum Beispiel in der Sprache C so gehandhabt.

Die Zeichenkette selbst wird durch die Adresse z des ersten Elements, sprich des ersten Buchstabens identifiziert. Die Adresse des i-ten Buchstaben (wobei das erste Zeichen den Index 0 hat) ist $z + 1 \cdot i$, da jeder Buchstabe 1 Byte lang ist.

Zur Bestimmung der Länge der Zeichenkette muss man beginnend von der Adresse des ersten Zeichens das nächste Nullbyte suchen.

```
    b     load
begin:
    addiu $a0 $a0 1
load:
    lb    $t0 0($a0)
    bnez  $t0 begin
    # $a0 enthaelt nun die Adresse des ersten Nullbytes
```

⌟

Adresse	Inhalt
0x00000000	?
⋮	⋮
$z+0$	0x4D
$z+1$	0x49
$z+2$	0x50
$z+3$	0x53
$z+4$	0x00
⋮	⋮
0xFFFFFFFF	?

Speicherauszug an der Adresse z.

2.6.2 Verbunde

Ein Verbund besteht aus einer festen Zahl (möglicherweise) unterschiedlich strukturierter **Felder (engl. field)**. Im Unterschied zur Reihung ist die Zahl der Felder konstant, dafür können sie unterschiedliche Struktur haben. Jedes Feld hat einen konstanten Versatz zur Basisadresse des Verbunds.[5] Hierbei ist die Ausrichtung der Felder zu beachten: Zum

[5] Dies ist auch der Grund, warum es die Speicherbefehle von MIPS erlauben, die Adresse für den Speicherzugriff aus einem Registerinhalt (Basisadresse) und einer Konstante (Versatz) zu bilden. Hierdurch spart man ein Register (wenn man die Basisadresse nochmals benötigt) und eine Addition.

Beispiel muss ein MIPS Wort an einer Adresse stehen, die durch 4 teilbar ist (siehe unten).

Beispiel 2.3. Betrachten wir den Verbund „Person", der aus den Feldern Vorname, Nachname und Geburtsdatum besteht. Wir repräsentieren das Geburtsdatum wie folgt: Jeweils 1 Byte für Tag und Monat und 2 Byte für das Jahr. Für die Zeichenketten Name und Vorname haben wir nun zwei Möglichkeiten:

Komposition. Wir speichern die Buchstaben der Zeichenketten direkt im Verbund ab (Abbildung 2.7a). Dann müssen wir uns auf die Länge der Zeichenketten **konstant** festlegen, da die Felder eines Verbunds ja hintereinander im Speicher stehen und der Versatz jedes Feldes konstant sein muss.[6] Des weiteren können wir den Namen dann nicht wiederverwenden. Sprich, wenn wir beispielsweise eine Reihung von Personen verarbeiten und der Name „Meier" n-mal vorkommt, müssen wir auch n Kopien in den jeweiligen Verbunden anlegen.

[6] Man beachte, dass der Versatz ja als „Immediate" in den Befehlen kodiert ist.

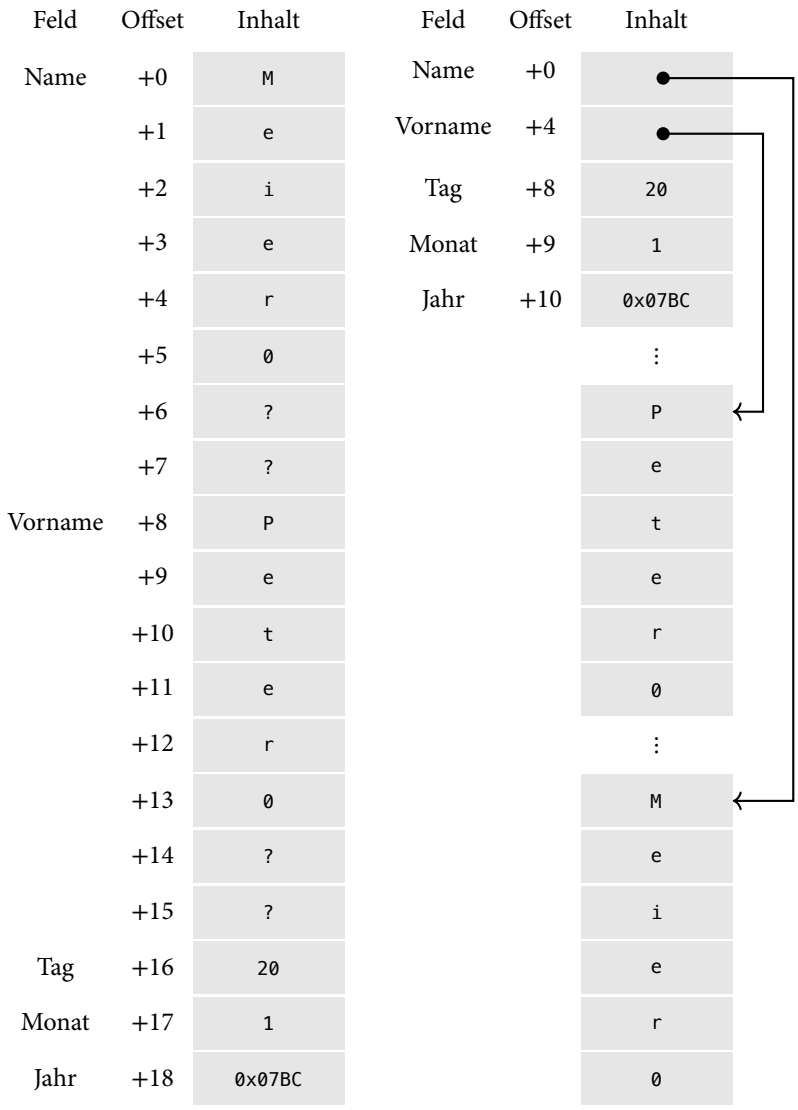
Aggregation. Statt die Zeichen der Zeichenketten direkt im Verbund abzulegen, speichern wir die Adresse der Zeichenkette im Verbund (Abbildung 2.7b). Eine Adresse hat immer die gleiche Länge (bei MIPS 4 Byte), so dass wir den Feldern unseres Verbunds konstante Offsets zuweisen können. Somit müssen wir uns nicht auf eine Länge der Zeichenkette festlegen. Des Weiteren können wir Zeichenketten „wiederverwenden", sprich, die Zeichenkette „Meier" muss nur einmal gespeichert werden. Die n „Meiers" speichern dann in ihrem Nachname-Feld einfach die Adresse der Zeichenkette „Meier". Allerdings müssen wir nun die Zeichenketten für Vor- und Nachnamen getrennt vom Verbund anlegen. ⌟

2.6.3 Ausrichtung (engl. alignment)

Wie bereits angedeutet, ist bei manchen Prozessoren, so auch MIPS, die Ausrichtung der Daten wichtig. Typischerweise kann ein Prozessor mit einem Speicherbefehl für ein n-Byte-Wort nur von einer Adresse laden/speichern, die durch n teilbar ist. Um diese Bedingung bei zusammengesetzten Datentypen zu erfüllen, ermittelt man (rekursiv[7]) den größten verwendeten Basistyp. Nehmen wir an, dieser Basistyp ist n Bytes groß, sprich ein Datum dieses Typs muss an einer durch n teilbaren Adresse liegen. Hat das Feld/Element nun einen Versatz, der durch n teilbar ist, so ist das Feld/Element n-ausgerichtet, wenn die Basisadresse auch n-ausgerichtet ist.

[7] wenn der zusammengesetzte Datentyp aus weiteren zusammengesetzten Datentypen besteht

Bei einem Verbund müssen also die Offsets der Felder so zugeteilt sein, dass jedes Offset durch n teilbar ist, wenn das entsprechende Feld n-ausgerichtet werden muss. Will man einen zusammengesetzten Datentyp in einer Reihung organisieren, so muss man dafür sorgen, dass seine Größe s auch durch n teilbar ist, damit (nach (2.1)) die Adresse jedes Elements der Reihung wieder durch n teilbar ist, wenn die Basisadresse durch n teilbar ist.



(a) Komposition

(b) Aggregation

Abbildung 2.7: Speicherauszüge eines Verbunds „Person“. In der linken Abbildung ist der Name und der Vorname direkt im Verbund abgelegt (Komposition). In der rechten Abbildung sind Name und Vorname aggregiert: Anstelle der Zeichenkette sind die Adressen der Zeichenketten hinterlegt. Hierzu muss die Basisadresse auch 2-ausgerichtet sein. Man beachte, dass das Feld „Jahr“ 2 Byte lang ist und die Zeiger in der rechten Abbildung 4 Byte lang sind. Um in der Abbildung Platz zu sparen, sind die normalerweise 1 Byte großen Speicherzellen für diese Felder zu größeren Blöcken zusammen gefasst.

2.6.4 Speicherallokation

Ist die Größe des Datums und dessen Anzahl **statisch** (also zur Übersetzungszeit) bekannt, so kann es im Datensegment (siehe Abschnitt 2.5 und Abschnitt 2.7) abgelegt werden. Dann wird beim Laden des Programms durch das Betriebssystem Speicher für das Datum reserviert und man kann mit einer Marke Bezug auf die Basisadresse nehmen.

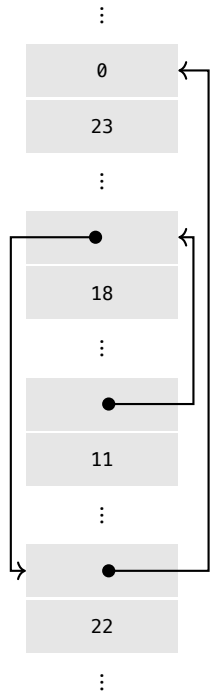
Oftmals entscheidet sich die Größe oder die Anzahl der Daten erst zur Laufzeit des Programms. Zum Beispiel könnte ein Bildverarbeitungsprogramm, nach Aufforderung durch den Benutzer, ein Bild laden. Die Größe des Bildes, und damit die Größe der anzulegenden Reihung, steht aber in der Bilddatei, die man statisch nicht kennt. Will man sich a priori (also statisch) nicht auf eine Obergrenze für die Größe dieser Reihung festlegen, so kann man die Reihung nicht im statischen Datensegment ablegen. Für dynamisch angeforderten Speicher ist die **Halde (engl. Heap)** vorgesehen. Der Speicher der Halde wird typischerweise, zur Laufzeit des Programms, von einer Speicherverwaltung (siehe Abschnitt 2.7) verwaltet. Das Programm kann sich bei dieser, durch einen Unterprogrammaufruf, noch unbelegten Speicher reservieren und wieder freigeben.

Beispiel 2.4 (Verkettete Listen)**.** Stellen wir uns folgendes Szenario vor: Man muss eine Liste von Daten speichern, weiß aber während des Anfügens der Elemente noch nicht, wie lange die Liste werden wird. Implementiert man die Liste mit einer Reihung, hat man das Problem, dass man nicht weiß, wieviel Speicher man für die Reihung anfordern soll. Man könnte einfach eine Größe n festlegen. Fügt man dann weniger als n Elemente an, verschwendet man Platz. Fügt man aber mehr als n Elemente an, so muss man (bei der Speicherverwaltung) eine neue Reihung anfordern, die Elemente aus der alten in die neue kopieren, und die alte Reihung wieder freigeben.

Je nach Anwendung kann es vorteilhafter sein, die Daten nicht in einer Reihung, sondern in einer verketteten Liste zu organisieren. In einer verketteten Liste wird jedes Element durch einen Verbund, der aus dem Element und der Adresse des nächsten Verbunds besteht, repräsentiert. So ein Listen-Element hat eine bekannte Größe und kann daher einzeln angefordert werden. Da jedes Listen-Element die Adresse des nächsten speichert, müssen die Elemente nicht (wie bei einer Reihung) im Speicher hintereinander liegen. Das Ende der Liste wird dadurch markiert, dass die Adresse des nächsten Elements auf 0 gesetzt wird.

Dies ist allerdings der große Nachteil einer verketteten Liste: Die Adresse des i-ten Elements kann nicht durch Adress-Arithmetik (wie in (2.1)) ermittelt werden, sondern durch das Verfolgen der Adressen der Listen-Elemente (engl. pointer chasing). Das Ermitteln der Adresse des i-ten Elements verursacht also i Speicherzugriffe.

Ein weiterer Nachteil, der in der Praxis eine große Rolle speilt, ist, dass die dadurch enstehende Sequenz von Speicherzugriffen **unregelmäßig** ist. Dies rührt daher, dass die Speicherverwaltung den angeforderten Speicher für die Listen-Elemente dort anlegt, wo noch Speicher frei ist[8].



Verkettete Liste der Zahlen 11, 18, 22, 23. Die Listen-Elemente liegen nicht notwendigerweise hintereinander oder aufsteigend sortiert im Speicher.

[8] Dies verursacht im Allgemeinen mehr Cache-Misses und nutzt die Zugriffsvorhersage des Prozessors nicht aus: Moderne Prozessoren haben zwischen den Registern und dem Hauptspeicher mehrere Pufferspeicher (Caches), um die lange Latenz von Hauptspeicherzugriffen zu vermeiden. Im Allgemeinen gilt, je größer ein Cache/Speicher, desto länger ist die Zugriffszeit. Bei modernen Systemen kann ein Cache-Miss, also ein Zugriff auf den Hauptspeicher, leicht mehrere hundert Taktzyklen dauern.

2.7 Das Laden des Programms und die Speicheraufteilung

Beim Laden des Programms durch das Betriebssystem werden Daten und der Code an die vom Betriebssystem vorgegebenen Adressen geladen. In unserem MIPS-Szenario beginnt der Code ab Adresse `0x400000` und die Daten ab Adresse `0x10000000`. Kommen dynamisch gebundene Bibliotheken zum Einsatz, muss das Betriebssystem beim Laden des Programms den Linker erneut starten um die entsprechenden Bibliotheken an das Programm zu binden. Dies soll uns hier jedoch nicht weiter kümmern. Der Bereich oberhalb der statischen Daten, sprich der Bereich beginnend ab Adresse `0x10000000` + n wobei n die Größe der statischen Daten ist, ist die sogenannte **Halde (engl. Heap)**. Sie kann vom Programm frei genutzt werden. Eventuell muss das Programm das Betriebssytem über den benutzten Bereich der Halde (in Form eines Pegelstands) unterrichten. Am Ende der Halde beginnt der Laufzeitkeller, mit dem die **Aufrufschachteln (engl. call/stack frames)** für Unterprogramm-Aufrufe verwaltet werden (siehe nächster Abschnitt). Halde und Keller laufen sich also entgegen. Berühren sich Halde und Keller, so hat das Programm keinen freien Speicher mehr und das Betriebssystem bricht es ab.

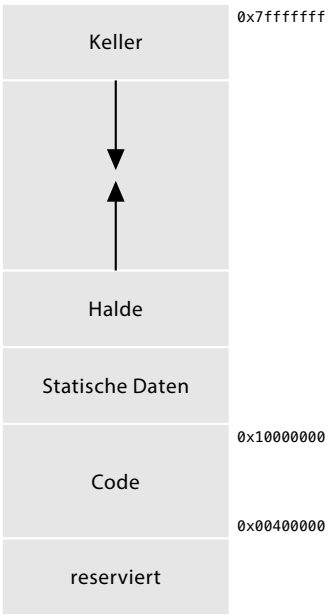


Abbildung 2.8: Aufteilung des 2^{32} Bytes großen Adressraums. Die Adressen sind für MIPS und das Betriebssystem spezifisch. Der Keller beginnt an der letzten Adresse und wächst der Halde entgegen.

2.7.1 Speicherverwaltung auf der Halde

Die detaillierte Verwaltung der Halde übernimmt das Laufzeitsystem der Programmiersprache, nicht das Betriebssystem. Das Laufzeitsystem der Sprache, das meist in einer Bibliothek mit dem Übersetzer ausgeliefert wird, verwaltet die belegten und freien Speicherbereiche auf der Halde. Herkömmlicherweise kann der Programmierer mittels Unterprogrammaufruf (in C zum Beispiel `malloc`) oder eigenem Sprachkonstrukt (in C++ und Java der Operator `new`) Speicher auf der Halde belegen. Die Freigabe des Speichers erfolgt entweder auch über Unterprogrammaufrufe in die Laufzeibibliothek (In C/C++ via `free` und `delete`) oder automatisch (wie in Java), wenn das Laufzeitsystem erkennt, dass ein einst angeforderter Speicherblock nicht mehr benötigt wird (da keine Referenzen mehr auf ihn existieren). Eine solche automatische Speicherbereinigung setzt eine starke Typisierung der Sprache voraus, da das Laufzeitsystem von jeder Variable klar sagen können muss, ob es sich um eine Referenz (Zeiger) handelt oder nicht. In unserer Assemblersprache haben wir überhaupt keine Verwaltung des Speichers auf der Halde; wir müssen selbst dafür sorgen, dass kein Unterprogramm Speicher auf der Halde überschreibt, der noch in Benutzung ist. Dies ist in größeren Programmen mit vielen Unterprogrammen, die teils nur in Form von Bibliotheken vorliegen (sprich, der Quelltext nicht vorhanden ist), extrem fehleranfällig und ohne zentralisierte Speicherverwaltung (à la `malloc`/`free` oder sogar automatischer Speicherbereinigung) fast nicht machbar.

2.8 Der Linker

Der Linker fügt mehrere Objektdateien zusammen. Hierbei können auch noch Bibliotheken von Dritten ins Spiel kommen (siehe Abbildung 2.3). Seine Hauptaufgabe ist die Auflösung der globalen Marken und das Einsetzen von absoluten Adressen. Wie oben erwähnt, kann in einer Objektdatei eine Marke referenziert werden, die in einer anderen Datei vereinbart wurde. Hierzu enthält jede Datei eine sogenannte **Symboltabelle (engl. symbol table)**, die alle globalen Marken (Symbole) auflistet, die die Objektdatei bereitstellt und referenziert. Der Linker ordnet nun den Code und die Daten der einzelnen Objektdateien hintereinander an. Bis auf Code und Daten aus sogenannten dynamisch gebunden Bibliotheken (engl. dynamic linked libraries) steht nun die Adresse jedes Befehls und jedes Datums fest. Der Code wird vom Linker dann so angepasst, dass an den entsprechenden Stellen die korrekten Marken referenziert werden. Dies kann bedeuten, dass er die Konstantenfelder in den Instruktionsworten umschreibt, oder gar mehrere Befehle einfügt, um entsprechende Adressen zu generieren.

Beispiel 2.5 (Linken von verschiedenen Objektdateien)**.** Betrachten wir folgende zwei Assemblerdateien:

main.asm

```
    .data
    .globl y
y:  .word 42
m:  .asciiz "Geben Sie eine Zahl ein: "
    .text
    .globl main
main:
    li    $v0 4
    la    $a0 m
    syscall
    li    $v0 5
    syscall
    move  $a0 $v0
    jal   add_y
    move  $a0 $v0
    li    $v0 1
    syscall
```

Symbol	Sichtbarkeit
y	definiert, global
m	definiert
main	definiert, global
add_y	benötigt

add.asm

```
    .text
    .globl add_y
add_y:
    lw   $v0 y
    addu $v0 $v0 $a0
    jr   $ra
```

Symbol	Sichtbarkeit
add_y	definiert, global
y	benötigt

⌟

2.9 Aufruf von Unterprogrammen

Sobald wir größere Programme schreiben, werden wir sie **modularisieren**. Das bedeutet, dass wir sie in viele Unterprogramme aufteilen, die wir in mehrere Übersetzungseinheiten gruppieren. Dies ist technisch nicht notwendig (man könnte alles auch in eine Riesendatei schreiben und auf Unterprogramme verzichten, was dann zum berüchtigten **Spaghetti-Code** führt). Jedoch sind solche Programme kaum zu überblicken. Man kann sie weder gut warten, gut testen, noch effizient Fehler darin finden. Insbesondere wenn mehrere Personen an einem Programm arbeiten, führt dies unweigerlich zur Katastrophe. Daher versuchen wir ein größeres Programm in kleinere Module zu unterteilen. Dies gilt natürlich für Programme in Hochsprachen aber erst recht auch für Programme in Maschinensprache. Damit die einzelnen Module zusammenspielen, brauchen wir eine klar definierte Regeln, die **systemweit** eingehalten werden. Wie wir sehen werden, betrifft dies vor allem die Verwendung der Registerbank.

Als einführendes Beispiel dient ein kleines Unterprogramm, das ein anderes aufruft:

```
    .text
foo:
    addu $t0 $a0 $a1
    jal  bar
    addu $v0 $v0 $t0
    jr   $ra
```

Nehmen wir an, das gerufene Unterprogramm `bar` ist in einer anderen Überseztungseinheit vereinbart, die wir nicht kennen. Wir wissen also nichts über die Routine `bar`. Wie können wir also davon ausgehen, dass im Register `$t0` nach dem Aufruf an `bar` immer noch der Wert steht, den wir zuvor dort abgelegt haben, sprich, dass `bar` das Register `$t0` nicht überschreibt? Da wir `bar` nicht kennen, müssen wir annehmen, dass es `$t0` selbst verwendet und den Wert, den wir in `$t0` abgelegt haben, überschreibt. Also müssen wir den Inhalt von `$t0` retten, bevor wir `bar` aufrufen. Dasselbe gilt auch für die Rücksprungadresse von `foo`, die im Register `$ra` liegt. Eine Möglichkeit, dies zu tun, wäre, den Inhalt von `$t0` an eine fest vereinbarte Speicherstelle im statischen Bereich des Datensegments zu schreiben. Dies funktioniert allerdings nicht für **rekursive** Funktionen: Wird eine Routine rekursiv gerufen (dies kann auch indirekt geschehen: Routine `f` ruft `g`, die dann wieder `f` ruft), so existieren mindestens zwei Instanzen der Routine gleichzeitig. Diese beiden Instanzen würden sich dann den einen Speicherplatz streitig machen, wie zu sehen in Abbildung 2.9. Ein zweiter Aufruf an `f` überschreibt das an Stelle `p` gesicherte `$t0`.

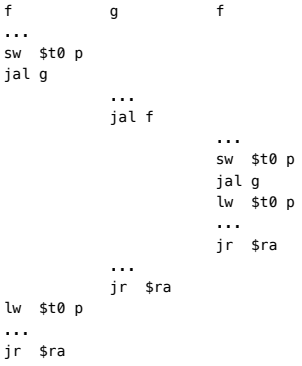


Abbildung 2.9: Rekursiver Aufruf von `f`

Wir müssen also für jeden Aufruf der Routine `foo` einen separaten Speicherplatz zur Sicherung von `$t0` bereitstellen. Da eine gerufene Routine immer vor ihrem Aufrufer zurückkehrt, können wir diesen Speicher als **Keller (engl. Stack)** organisieren. Sobald `foo` angesprungen wird, erzeugt es auf diesem Keller eine Region, genannt die **Aufrufschachtel (engl. stack frame)**, in der durch Aufrufe potentiell zerstörte Regis-

terinhalte abgelegt werden können. Dieses Anlegen geschieht durch Verringern des **Kellerpegels (engl. stack pointer)** `$sp`; der Keller wächst nach unten. Bevor `foo` zurückkehrt, wird die Aufrufschachtel beseitigt, indem der Kellerpegel auf den Stand bei Eintritt zurückgesetzt wird:

```
    .text
foo:
    addu $sp $sp -8   # Verringern des Kellerpegels, Anlegen der Schachtel
    addu $t0 $a0 $a1
    sw   $t0 0($sp)   # Sichern des Inhalts von $t0 in die Schachtel
    sw   $ra 4($sp)   # Sichern der Rücksprungadresse
    jal  bar
    lw   $ra 4($sp)   # Restaurieren der Rücksprungadresse
    lw   $t0 0($sp)   # Restaurieren des Werts von $t0
    addu $v0 $v0 $t0
    addu $sp $sp 8    # Zurücksetzen des Kellerpegels
    jr   $ra
```

Da der Kellerpegel sich innerhalb eines Unterprogramms auch weiter verändern kann (aufgrund weiterer Speicherallokation und Unterprogrammaufrufe), ist es manchmal zweckmäßig, ein eigenes Register für die Aufrufschachtel zu veranschlagen. Dies ist der sogenannte **Schachtelzeiger (engl. frame pointer)**, dessen Register im MIPS-Assembler `$fp` heißt. Während der Ausführung eines Unterprogramms markiert der Kellerpegel immer an den unteren Rand des Kellers (er zeigt auf das tiefste Element des Kellers). Die Aufrufschachtel enthält folgende Elemente (siehe auch Abbildung 2.10): Der Schachtelzeiger zeigt auf die erste Adresse oberhalb der Aufrufschachtel.

- An das Unterprogramm übergebene Werte (bei MIPS werden die ersten vier Argumente in den Registern `$a0`–`$a3` übergeben. Die restlichen Argumente werden auf dem Keller übergeben.
- Speicherplatz um Inhalte von Registern zu sichern, die …
  1. die Routine ändern möchte, wobei der Aufrufer aber erwartet, dass die Inhalte jener Register unverändert bleiben (man spricht hier auch von **callee-save** Registern, da sie der Aufgerufene sichern muss).
  2. potentiell von aufgerufenen Unterprogrammen zerstört würden. Hier spricht man von **caller-save** Registern, da der Aufrufer die Inhalte retten muss.
- Platz für weitere lokale Variablen des Unterprogramms, die nicht in Register passen, da zu wenig Register verfügbar sind, oder die Variablen zu groß sind oder adressierbar sein müssen.

Die Aufrufschachtel muss vom Aufrufer und vom Aufgerufenen explizit aufgebaut werden. Der Aufrufer legt unter Umständen Argumente für das Unterprogramm auf den Keller und der Aufgerufene benötigt unter Umständen Platz für die oben genannten Registerinhalte. Der Teil des Codes der des aufgerufenen Unterprogramms der die Aufrufschachtel

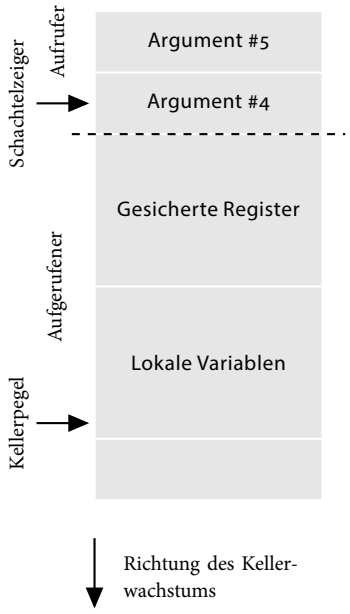


Abbildung 2.10: Aufrufschachtel (stack frame). In unseren Beispielen ändert sich die Größe des Kellerpegels immer konstant und unabhängig von der Eingabe. Ist das nicht der Fall, verwendet man zusätzlich zum Kellerpegel auch noch einen **Schachtelzeiger (engl. frame pointer)**, der dann das obere Ende des Kellers markiert.

auf- bzw. abbaut, wird **Prolog** bzw. **Epilog** genannt und sieht für den MIPS-Assembler wie folgt aus:

```
f:
    # Prolog
    addiu $sp $sp -platz  # reserviere Platz auf dem Keller
    ...
    # Epilog
    addiu $sp $sp platz   # freigeben des Platzes
    jr    $ra
```

2.9.1 Rekursion

Betrachten wir nochmal unser einführendes Beispiel, die Berechnung der Fakultät. Die in Abbildung 2.5 implementierte Version verwendet keine Funktionsaufrufe zu sich selbst, da sie **endrekursiv** ist. Dies lässt sich an den Rekursionsgleichungen erkennen:

$$f_r(n) \coloneqq \begin{cases} n \cdot f_r(n-1) & n \geq 1 \\ 1 & n < 1 \end{cases}$$

Herkömmliche, rekursive Definition.

$$f_e(n, r) \coloneqq \begin{cases} f_e(n-1, n \cdot r) & n \geq 1 \\ r & n < 1 \end{cases}$$

Endrekursive Version. (Entspricht Abbildung 2.5.)

Wir stellen nun hier beide Implementierungen gegenüber, um einen Eindruck von dem Mehraufwand zu bekommen, den Rekursion typischerweise erfordert. Die Tatsache, dass die Funktion f_e endrekursiv

```
fac:
    li     $v0 1
    blez   $a0 end
loop:




    mul    $v0 $v0 $a0
    addiu  $a0 $a0 -1
    bgtz   $a0 loop




end:
    jr     $ra
```

```
fac_rec:
    li     $v0 1
    blez   $a0 end

    addiu  $sp $sp -8
    sw     $ra 0($sp)
    sw     $a0 4($sp)

    addiu  $a0 $a0 -1
    jal    fac_rec
    lw     $a0 4($sp)
    mul    $v0 $v0 $a0
    lw     $ra 0($sp)
    addiu  $sp $sp 8
end:
    jr     $ra
```

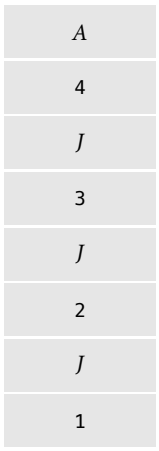
Abbildung 2.11: Rekursive und endrekursive Implementierung der Fakultät. Für die Rekursive Variante ist es notwendig, das Argument n über den rekursiven Aufruf zu speichern, was nicht ohne eine Aufrufschachtel geht.

ist, erkennt man daran, dass der Rückgabewert des rekursiven Aufrufs gleichzeitig auch der Rückgabewert der Funktion ist. Im Gegensatz zu f_r

muss der Rückgabewert muss also nicht weiter verrechnet werden. In der Implementierung der herkömmlichen Version f_r muss der Wert von n der rufenden Instanz über den rekursiven Aufruf von f_r gespeichert werden, um ihn dann mit dem Rückgabewert des rekursiven Aufrufs multiplizieren zu können.

Betrachtet man die Ausführung von `fac_rec` genauer und verfolgt den Zustand des Laufzeitkellers, so stellt man fest, dass der Code bis zur Instruktion `jal` nur Aufrufschachteln anlegt und die entsprechenden Werte von n auf den Keller legt.

Nach dem die letzte gerufene Instanz (mit $n = 0$) zurückkehrt, wird der Keller wieder abgebaut und die auf dem Keller abgelegten Werte werden aufmultipliziert.



Aufrufkeller nach vier rekursiven Aufrufen beginnend mit $n = 4$. Die Adresse *A* ist die Rückkehradresse im Unterprogramm des Aufrufers. *J* stellt die Adresse des Befehls der in `fac_rec` auf den Befehl `jal` folgt, dar.

2.9.2 Die Aufrufkonvention

Die Aufrufkonvention legt fest, welche Register ein aufgerufenes Unterprogramm ändern darf, ohne deren Wert sichern zu müssen. Solche Register werden **caller-save** Register genannt, da der Aufrufer sie, falls er ihren Inhalt noch benötigt, vor dem Aufruf sichern muss. Bei **callee-save** Registern kann der Aufrufer davon ausgehen, dass ihr Inhalt **nach** der Rückkehr vom Unterprogramm unverändert ist. Will der Aufgerufene sie modifizieren, so muss er sie sichern.

Aus Effizienzgründen übergibt man die Argumente an ein Unterprogramm auch in Registern. Bei MIPS werden hierzu die Register `$a0`–`$a3` verwendet. Hat ein Unterprogramm mehr als vier Parameter, so werden die Argumente ab dem vierten auf dem Keller übergeben. Das Resultat eines Unterprogramms wird im Register `$v0` abgelegt. Abbildung 2.12 zeigt alle MIPS Register und ihre Rolle in der Aufrufkonvention. Beispiel 2.6 diskutiert die Verwendung der Aufrufkonvention an einem konkreten Beispiel.

Abbildung 2.12: MIPS-Register und ihr üblicher Name und Verwendungszweck in der Aufrufkonvention. Die Spalte cs beschreibt, ob der Registerinhalt einen Funktionsaufruf unbeschadet übersteht, sprich, ob der Aufgerufene das Register sichern und restaurieren muss, wenn er es ändert (= callee-save).

Nummer	Name	save	Bedeutung
`$0`	`$zero`	—	Hat beim Lesen immer den Wert 0
`$1`	`$at`	—	Wird vom Assembler verwendet; darf nicht vom Programmierer verwendet werden
`$2`–`$3`	`$v0`–`$v1`	caller	Rückgabewerte eines Unterprogramms
`$4`–`$7`	`$a0`–`$a3`	caller	Erste vier Argumente eines Unterprogramms
`$8`–`$15`	`$t0`–`$t7`	caller	Temporäre Werte, Sichern durch Aufrufer
`$16`–`$23`	`$s0`–`$s7`	callee	Temporäre Werte, Sichern durch Aufgerufenen
`$24`–`$25`	`$t8`–`$t9`	caller	Temporäre Werte, Sichern durch Aufrufer
`$26`–`$27`	`$k0`–`$k1`	—	Reserviert für das Betriebssystem
`$28`	`$gp`	callee	global pointer. Zeigt in die Mitte des ersten 64K Blockes im Datensegment
`$29`	`$sp`	callee	stack pointer. Kellerpegel
`$30`	`$fp`	callee	frame pointer. Schachtelzeiger
`$31`	`$ra`	caller	return address. Rücksprungadresse

2.10 Beispiele

Beispiel 2.6 (Zahlenkonversion)**.** Als erstes Beispiel schreiben wir ein Programm, das eine 32-Bit Zahl (repräsentiert durch die Bitfolge in einem Register) in Hexadezimalschreibweise ausgibt. Ein 32-Bit Zahl hat hexadezimal acht Ziffern, da je vier Bit einer Stelle entsprechen. Wir müssen die acht 4-Bit Gruppen also nur hintereinander (von den höchstwertigen zu den niederwertigsten) ausgeben.

Betrachten wir zunächst das Ausgeben einer Zahl zwischen 0 und 15 (inklusive), also einer Zahl in deren Binärdarstellung alle Bits jenseits des vierten nicht gesetzt sind. [9] Sei nun also $n \in [0, 15]$. Um n auszugeben, muss der Zahlenwert in das entsprechende ASCII-Zeichen übersetzt werden:

$$0 \mapsto \texttt{0},\ \ldots,\ 9 \mapsto 9,\ 10 \mapsto \texttt{a},\ \ldots,\ 15 \mapsto \texttt{f}$$

Dies könnte man dadurch erreichen, dass man die Zahlwerte der ASCII-Zeichen in einer Tabelle der Länge 16 im Speicher ablegt und dann n als Index in diese Tabelle verwendet um den entsprechenden ASCII-Wert zu lesen:

```
    .data
ziffer_ascii:
    .byte 48, 49, 50, 51, 52, 53, 54, 55, 56, 57
    .byte 97, 98, 99, 100, 101, 102

    .text
    # ...
to_ascii:
    # $a0 enthält n
    lbu    $a0 ziffer_ascii($a0)
    # $a0 enthält nun den ASCII-Wert der Hexadezimal-Ziffer von n
```

Allerdings verrät uns ein Blick auf die ASCII-Tabelle (siehe Anhang C), dass man die Werte der ASCII-Zeichen leicht errechnen und man somit auf die Tabelle verzichten kann:

$$ascii(n) = \begin{cases} n + 48 & n < 10 \\ n + 87 & 10 \leq n < 16 \end{cases}$$

Dies setzen wir nun in die folgenden MIPS-Befehle um. Wir nehmen an, dass die Zahl $0 \leq n < 16$ im Register `$a0` abgelegt ist.

```
to_ascii:
    sltiu   $t2 $a0 10
    bnez    $t2 below10
    addiu   $a0 $a0 39
below10:
    addiu   $a0 $a0 48
```

Nach der Ausführung dieser Befehle enthält das Register `$a0` den ASCII-Wert der Zahl n Um das Programm ein wenig einfacher zu halten, haben wir ausgenutzt, dass für $n \geq 10$ gilt, dass $ascii(n) = ascii(n - 10) + 39$.[10]

[9] Wir wenden uns also zuerst einem Teilproblem zu, das wir separat lösen. Der Ansatz, eine Programmieraufgabe sukzessive in kleinere unabhängige Teilaufgaben aufzuteilen und getrennt zu bearbeiten, heißt **Schrittweise Verfeinerung** und wurde von Niklaus Wirth in einem berühmten Artikel 1971 vorgestellt. [Wir71]

Der Befehl `sltiu` legt hier eine 1 in `$t2`, wenn `$a0` $<$ 10. `bnez` verzweigt nach `below10` wenn `$t2` eine Zahl ungleich 0 (not equal to zero) enthält. Die 39 wird also nur aufaddiert, wenn `$a0` $\geq$ 10.

[10] Aufgabe 2.22 stellt die Frage, wie man `to_ascii` gänzlich ohne Sprung implementieren kann.

Um nun eine komplette 32-Bit Zahl hexadezimal auszugeben, müssen wir über die acht 4-Bit Gruppen des Registers iterieren und jede einzeln konvertieren und ausgeben:

`srlv` schiebt die Bits in `$t0` um `$t1` nach rechts und „füllt“ das Wort mit Nullen auf. Die untersten vier Bit von `$a0` enthalten dann *n*. `andi $a0 $a0 15` setzt alle Bits außer den untersten vier auf 0.

```
print_hex:
    move    $t0 $a0
    li      $v0 11
    li      $t1 32
head:
    beqz    $t1 end
    addiu   $t1 $t1 -4
    srlv    $a0 $t0 $t1
    andi    $a0 $a0 15
    # $a0 enthält nun die entsprechende Stelle.
    # Sie muss mit dem Code von oben in den ASCII-Wert konvertiert werden
    # Danach ist der ASCII-Code in $a0
    # Der Systemaufruf (Kennung 11 in $v0) gibt dann das Zeichen in $a0 aus.
    syscall
    b       head
end:
```

Wie fügen wir nun unsere Teillösungen zu einem Gesamtprogramm zusammen? Abbildung 2.13 zeigt drei Varianten:

1. `to_ascii` wird als Unterprogramm ausgeführt. Der Befehl `jal` setzt den Befehlszeiger auf die Adresse der Marke `to_ascii` schreibt aber zuvor die Adresse des auf ihn folgenden Befehls (hier `syscall`) in das Register `$ra`. `to_ascii` beendet sich dadurch, dass es an die Adresse springt, die in `$ra` steht.

2. Da `print_hex` der einzige Aufrufer von `to_ascii` ist, ist die Rückkehradresse konstant und kann daher direkt in den Code geschrieben werden anstatt sie mit dem Register `$ra` zu übergeben.

3. Der Code von `to_ascii` wird direkt in `print_hex` eingebaut

Die Varianten aus Abbildung 2.13 implementieren aber die Aufrufkonvention nicht. Das Unterprogramm `print_hex` geht beispielsweise davon aus, dass `to_ascii` die Register $`t0` und $`t1` nicht überschreibt, was der MIPS-Aufrufkonvention widerspricht. Soll das Unterprogramm `print_hex` die Aufrufkonvention erfüllen, so ist mehr Code nötig. Insbesondere müssen zwei Register (und $`ra`!) über den Aufruf zu `to_ascii` gerettet werden. Eine Möglichkeit `print_hex` konform zur Aufrufkonvention zu implementieren, zeigt Abbildung 2.14. Diese Implementierung verwendet die beiden callee-save Register $`s0` und $`s1` (→ Aufgabe 2.21).

```
to_ascii:
    sltiu   $t2 $a0 10
    bnez    $t2 below10
    addiu   $a0 $a0 39
below10:
    addiu   $a0 $a0 48
    jr      $ra
print_hex:
    move    $t0 $a0
    li      $v0 11
    li      $t1 32
head:
    beqz    $t1 end
    addiu   $t1 $t1 -4
    srlv    $a0 $t0 $t1
    andi    $a0 $a0 15
    jal     to_ascii




    syscall
    b       head
end:
```

```
to_ascii:
    sltiu   $t2 $a0 10
    bnez    $t2 below10
    addiu   $a0 $a0 39
below10:
    addiu   $a0 $a0 48
    b       print_digit
print_hex:
    move    $t0 $a0
    li      $v0 11
    li      $t1 32
head:
    beqz    $t1 end
    addiu   $t1 $t1 -4
    srlv    $a0 $t0 $t1
    andi    $a0 $a0 15
    b       to_ascii



print_digit:
    syscall
    b       head
end:
```

```
print_hex:
    move    $t0 $a0
    li      $v0 11
    li      $t1 32
head:
    beqz    $t1 end
    addiu   $t1 $t1 -4
    srlv    $a0 $t0 $t1
    andi    $a0 $a0 15
    sltiu   $t2 $a0 10
    bnez    $t2 below10
    addiu   $a0 $a0 39
below10:
    addiu   $a0 $a0 48
    syscall
    b       head
end:
```

Abbildung 2.13: Drei Varianten um die Teillösungen `to_ascii` und `print_hex` zusammen zu fügen. Erstens: Unterprogrammaufruf, Zweitens: Unbedingte Sprungbefehle. Drittens: Direkter Einbau von `to_ascii` in `print_hex`. Hat `to_ascii` noch andere Aufrufer, sind die letzten beiden Alternativen nur möglich, wenn der Code von `to_ascii` dupliziert wird.

```
print_hex:
    addiu $sp $sp -12
    sw    $ra 0($sp)
    sw    $s0 4($sp)
    sw    $s1 8($sp)

    move  $s1 $a0
    li    $s0 32
loop:
    addiu $s0 $s0 -4
    srlv  $a0 $s1 $s0
    andi  $a0 $a0 15
    jal   to_ascii
    move  $a0 $v0
    li    $v0 11
    syscall
    bnez  $s0 loop
    ...
```

```
    ...
    lw    $ra 0($sp)
    lw    $s0 4($sp)
    lw    $s1 8($sp)
    addiu $sp $sp 12
    jr    $ra
```

Abbildung 2.14: Eine Implementierung von `print_hex` die konform mit der Aufrufkonvention ist.

Beispiel 2.7 (Das Sieb des Eratosthenes)**.** Das Sieb des Eratosthenes ist ein Algorithmus zum Aufzählen aller Primzahlen, die kleiner sind als eine gegebene Zahl N. Hierzu nimmt man sich eine Tabelle mit Einträgen von 2 bis $N-1$ und streicht sukzessive Zahlen, die keine Primzahlen sind. Am Anfang ist kein Eintrag gestrichen. Man geht die Tabelleneinträge nun nacheinander ab. Trifft man auf einen noch nicht gestrichenen Eintrag q, streicht man die Einträge aller echter Vielfacher von q. Die Vielfachen jeder Zahl $i < q$ wurde bereits gestrichen. Die Zahl q hat demnach – außer der 1 – keine Teiler kleiner q. Sie ist eine Primzahl.

Das Aufzählen kann wie folgt beschleunigt werden: Betrachten wir die Primfaktorzerlegung eines Vielfachen von q: Alle Vielfachen von q, die Primfaktoren kleiner q haben, sind Vielfache kleinerer Primzahlen. Sie wurden also schon gestrichen. Die kleinste zu streichende Zahl, die noch nicht gestrichen ist, ist somit q^2. Entsprechend macht es keinen Sinn, Vielfache einer Zahl q' mit $q' \cdot q' > N$ zu streichen.

Interessant ist die Laufzeitanalyse des Verfahrens. Sei $M := \lfloor \sqrt{N} \rfloor$. Wir zählen also für jedes $1 < i < M$ alle Vielfachen größer gleich i^2 auf. Die Anzahl der durchgeführten Streichungen ist also[11]

$$\begin{aligned}\sum_{i=2}^{M}(N-i^2)/i &= N\sum_{i=2}^{M}\frac{1}{i} - \sum_{i=2}^{M} i \\ &< N\ln M - \frac{M(M+1)}{2} < N\ln N \\ &= O(N\ln N)\end{aligned}$$

[11] Wir nutzen hier aus, dass $1+\ln N$ eine obere Schranke der Reihe der harmomischen Zahlen $\sum_{i=1}^{N} 1/i$ ist. Dies kann man aus $\int_1^x dz/z = \ln x$ ableiten.

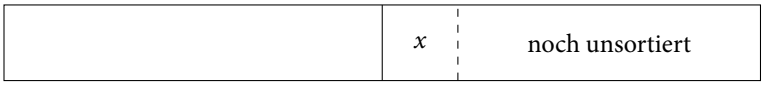
Dies setzen wir in folgendem Unterprogramm um

```
  .text
eratosthenes:
  .globl eratosthenes
  subu   $a2 $a1 $a0
  move   $t0 $a0

  # Fülle Reihung mit 0en
  b      zero_check
zero_loop:
  sb     $0 0($t0)
  addiu  $t0 $t0 1
zero_check:
  bne    $t0 $a1 zero_loop

  # Starte bei der 2
  li     $t0 2
era_loop:
  mul    $t9 $t0 $t0
  bge    $t9 $a2 end
  addu   $t1 $a0 $t0
  lb     $t1 ($t1)
  bnez   $t1 no_prime
```

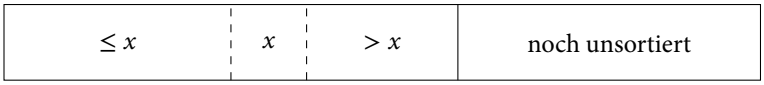
```
  # $t0 ist eine Primzahl
  # Streiche alle Vielfachen beginnend von $t0 * $t0
  # Setze $t1 auf die Adresse des Elements $t0 * $t0
  addu   $t1 $a0 $t9
  li     $t8 1
  b      mark_check
mark_loop:
  sb     $t8 ($t1)
  addu   $t1 $t1 $t0
mark_check:
  blt    $t1 $a1 mark_loop

no_prime:
  addiu  $t0 $t0 1
  b      era_loop

end:
  jr     $ra
```

⌟

Beispiel 2.8 (Sortieren durch Einfügen)**.** Sortieren durch Einfügen (engl. insertion sort) ist ein einfaches Sortierverfahren, das unserer alltäglichen Sortierpraxis nachempfunden ist. Beim Sortieren eines Blattes von Spielkarten teilt man das Blatt in einen sortierten Teil links und einen unsortierten Teil rechts auf. Zu Beginn besteht der sortierte Teil nur aus der Karte, die ganz links in der Hand liegt. Man nimmt nun eine Karte aus dem unsortierten Bereich (rechts)



und fügt sie sortiert in den linken Teil ein. Somit hat man den sortierten Teil um eine Karte vergrößert.



Man wiederholt dies, bis der unsortierte Teil leer ist.

Das Verfahren ist unabhängig vom **Typ** der zu sortierenden Elemente. Die einzige Voraussetzung ist, dass man die Elemente untereinander vergleichen kann. Wie dieser Vergleich implementiert ist, ist für das Sortierverfahren unerheblich. Der Einfachheit halber implementieren wir hier eine Variante, die 4 Byte Wörter vergleicht. Aufgabe 2.20 beschäftigt sich dann mit der **generischen** Variante.

```
    .text

    # $a0 Anfang der Reihung
    # $a1 Endadresse
insertsort:
```

```
    addiu   $t0 $a0 4
outer_loop:
    beq     $t0 $a1 outer_end
    move    $t1 $t0
inner_loop:
    beq     $t1 $a0 inner_end
    lw      $t2 ($t1)
    lw      $t3 -4($t1)
    slt     $t4 $t3 $t2
    bnez    $t4 inner_end
    sw      $t2 -4($t1)
    sw      $t3 ($t1)
    addiu   $t1 $t1 -4
    b       inner_loop
inner_end:
    addiu   $t0 $t0 4
    b       outer_loop
outer_end:
    jr      $ra
```

Ein Hauptprogramm, welches `insertsort` ruft und die sortierte Reihung ausgibt, sieht so aus:

```
    .data
data_begin:
    .word    10, 5, 2, 8, 7, 4, 6, 1
data_end:
space:
    .asciiz  " "
    .text
main:
    .globl   main
    la       $a0 data_begin
    la       $a1 data_end
    jal      insertsort
    move     $t0 $a0
print_loop:
    beq      $t0 $a1 print_end
    li       $v0 1
    lw       $a0 ($t0)
    syscall
    la       $a0 space
    li       $v0 4
    syscall
    addiu    $t0 $t0 4
    b        print_loop
print_end:
    li       $v0 10
    syscall
```

⌟

Beispiel 2.9 (Hase und Igel (engl. tortoise and hare))**.** Der Hase-und-Igel Algorithmus überprüft, ob eine verkettete Liste in einem Zyklus endet oder nicht. Das Besondere an diesem Algorithmus ist, dass er lineare Laufzeit hat und keinen zusätzlichen Speicher benötigt. Die Idee ist die Folgende: Man hält sich zwei Zeiger (Hase und Igel) mit der man die Liste abschreitet. In jeden Schritt schreitet der Hase zwei Listenelemente voran und der Igel nur eines. Die Liste ist zyklisch genau dann, wenn Hase und Igel sich treffen.

Betrachten wir eine verkettete Liste mit n Elementen, die einen Zyklus der Länge $l \leq n$ hat. Nach $n - l$ Schritten ist der Igel am Listenelement 0 des Zyklus angelangt. Dies ist das einzige Listenelement des Zyklus, auf das ein Zeiger zeigt, dessen Listenelement nicht Teil des Zyklus ist. Der Hase befindet sich dann an Listenelement d des Zyklus (→ Abbildung 2.15). In jedem Schritt erhöht sich der Abstand zwischen Hase und Igel um 1. Nach $l - d$ Schritten ist der Abstand genau l und Hase und Igel stehen auf demselben Feld. Ist die Liste also zyklisch, ermittelt der Algorithmus das $O(n)$ Zeit und $O(1)$ Platzverbrauch.

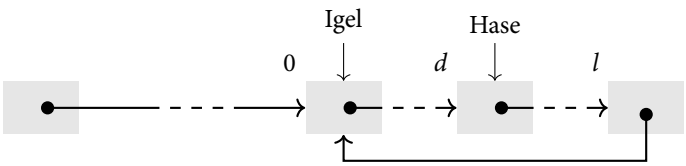


Abbildung 2.15: Der Zustand beim Betreten des Igels in den Zyklus

Im nicht-zyklischen Fall erreicht der Hase als erstes das Ziel. Es ist also wichtig, beim Hase-Zeiger immer zu testen, ob die Adresse des nächsten Elements 0 ist und somit das Ende erreicht ist.

In der folgenden Implementierung nehmen wir an, dass der Zeiger auf das nächste Listenelement an Offset 4 eines Listenelements liegt.

```
  .text
  .globl tortoise_and_hare

  # $a0: Adresse des ersten Listenelements
tortoise_and_hare:
  # $a0 Hase
  # $a1 Igel
  move  $a1 $a0
loop:
  beqz  $a0 not_cyclic
  lw  $a0 4($a0)
  beqz  $a0 not_cyclic
  lw  $a0 4($a0)
  lw  $a1 4($a1)
  beq $a0 $a1 cyclic
  b loop
cyclic:
  li  $v0 1
  jr  $ra
```

```
not_cyclic:
  li  $v0 0
  jr  $ra
```

Folgendes Hauptprogramm wendet den Algorithmus auf eine nicht-zyklische Liste im Datensegment an:

```
  .data

L1:
  .word 1
  .word L6
L2:
  .word 2
  .word L4
L3:
  .word 3
  .word L5
L4:
  .word 4
  .word L3
L5:
  .word 5
  .word 0
L6:
  .word 6
  .word L2

  .text
  .globl  main
main:
  la      $a0 L1
  jal     tortoise_and_hare
  move    $a0 $v0
  li      $v0 1
  syscall
  li      $v0 10
  syscall
```

⌟

Beispiel 2.10 (Horner-Schema)**.** Das Horner-Schema ist eine Technik, um ein Polynom

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n \tag{2.3}$$

an einer Stelle x auszuwerten. Sie kommt im Vergleich zur direkten Implementierung von (2.3) mit der Hälfte der Multiplikationen aus und beruht darauf, dass man in (2.3) die Variable x sukzessive ausklammert:

$$p(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_0 \tag{2.4}$$

oder rekursiv formuliert:

$$\begin{aligned} p_n(x) &= a_n \\ p_i(x) &= p_{i+1}(x) \cdot x + a_i \text{ für } 0 \leq i < n \end{aligned}$$

Das folgende MIPS Programm implementiert das Horner-Schema. Es setzt voraus, dass die Reihung mit den Koeffizienten mindestens einen Eintrag (a_0) hat. Dies ist vernünftig, denn ein „leeres“ Polynom gibt es nicht.

```
  .text
    .globl horner

  # $a0 Coefficients base
  # $a1 Coefficients end
  # $a2 val
horner:
  lw     $v0 ($a0)
  b      horner_head
horner_loop:
  lw     $t0 ($a0)
  mulu   $v0 $v0 $a2
  addu   $v0 $v0 $t0
horner_head:
  addiu  $a0 $a0 4
  bne    $a0 $a1 horner_loop
  jr     $ra
```

Folgendes Hauptprogramm verwendet `horner` um $\langle\cdot\rangle_2$ zu berechnen. Die Ziffernfolge ist als Reihung von Bytes im Datensegment angelegt.

```
  .data
coeff_begin:
  .word 1, 0, 1, 1, 1, 0, 0, 1
coeff_end:

  .text
main:
  .globl main
  la     $a0 coeff_begin
  la     $a1 coeff_end
  li     $a2 2
  jal    horner

  move   $a0 $v0
  li     $v0 1
  syscall
  li     $v0 10
  syscall
```

⌟

2.11 Aufgaben

Aufgabe 2.1 (Registernamen, Registernummern und Kodierungschiasmus. ⋆⋆)**.**

1. Wir können die Register in Assemblerbefehlen mit Namen oder Nummer referenzieren. Vervollständigen Sie folgende Tabelle:

Registernamen	**Registernummern**
`sltiu $t7 $a0 1`	
`addiu $v1 $zero 100`	
	`beq $4 $0 5`
`addu $sp $a2 $t8`	
	`jr $31`

   Hinweis: Wir verlangen nicht, dass Sie die Zuordnung von Registernamen auf Registernummern auswendig kennen.

2. Wir betrachten den Befehl `addiu $t9 $a0 100`. Nennen Sie Quell- und Zielregister. Was berechnet der Befehl? Geben Sie die Befehlskodierung in Binärdarstellung an. Sehen Sie dazu in der MIPS Befehlsreferenz [Pri95]nach, wie der Befehl kodiert wird. Markieren sie, welche Bits für Opcode, Quell- und Zielregister, und Konstante verwendent werden. In welcher Reihenfolge treten Quell- und Zielregister in den jeweiligen Darstellung auf?

⌟

Aufgabe 2.2 (Pseudoinstruktionen. ⋆⋆)**.** Der MIPS Assembler definiert eine Reihe von Pseudo-Instruktionen. Dies sind hilfreiche Instruktionen, die nicht in Hardware implementiert sind, aber leicht durch ein oder zwei existierende Befehle ausgedrückt werden können. Geben Sie Implementierungen der Pseudo-Instruktionen `blt`, `bgt`, `ble`, `neg`, `not`, `bge`, `li`, `la`, `lw`, `move`, `sge`, `sgt` an. ⌟

Aufgabe 2.3 (Speicherung und Interpretation binärcodierter Daten. ⋆)**.** Der folgende Speicherauszug eines MIPS Programms zeigt den Speicherinhalt ab Adresse `0x10000000`. In dieser Darstellung sind pro Zeile 4 MIPS-Wörter in Hexadezimal-Schreibweise dargestellt, insgesamt also 16 Bytes.

Adresse	Speicherinhalt			
`0x10000000`	`50322c20`	`7c2d703e`	`ffffffff`	`7fff3fff`
`0x10000010`	`8fffffff`	`01248888`	`00000000`	`00000000`

1. Welche Werte sind im Speicher abgelegt, wenn man folgende Interpretationen unterstellt. Beachten Sie die Endianess des MIPS-Prozessors.

Offset	Interpretation
0 – 7	8 ASCII-Zeichen
8 – 11	Ganz-Zahl: unsigned, 4 Bytes, durch 4 teilbare Adresse
12 – 13	Ganz-Zahl: signed, 2 Bytes, durch 2 teilbare Adresse
14	ASCII-Zeichen (1 Byte)
15	Ganz-Zahl: unsigned, 1 Byte
16 – 19	Ganz-Zahl: signed, 4 Bytes, durch 4 teilbare Adresse
20	Ganz-Zahl: signed, 1 Byte
21	ASCII-Zeichen (1 Byte)
22 – 23	Ganz-Zahl: unsigned, 2 Bytes, durch 2 teilbare Adresse

2. Notieren Sie die Datendefinitionen (`.data`), die nach dem Laden diese Speicherbelegung liefern!

   Überprüfen Sie die Wirkung Ihrer Datendefinitionen im MIPS Simulator.

⌟

Aufgabe 2.4 (Little und Big Endian. ⋆⋆)**.** In folgender Tabelle ist jeweils zu einer 32 Bit breite Zahl in Hexadezimaldarstellung die Darstellung im Speicher in Little und Big Endian angegeben. Vervollständigen Sie die Tabelle.

Hexadezimaldarstellung	**Little Endian**	**Big Endian**
`0x11223344`		
	`DE AD C0 DE`	
		`F0 05 BA 11`
`0xDECAFF01`		
	`C0 FF EE 00`	
		`BA 5E BA 11`

⌟

Aufgabe 2.5 (Ausführungsprotokoll für MIPS. ⋆⋆)**.** Nehmen Sie an, folgender Code wurde an Adresse 0x00400000 in den Speicher geladen. Wie groß ist ein einzelner Befehl? Notieren Sie zunächst am Anfang jeder Zeile die Adresse des jeweiligen Befehls, und markieren Sie Sprungziele mit einem Pfeil.

```
0x00400000    ori  $2  $0  1
              beq  $4  $0  3
              mul  $2  $2 $4
            addiu  $4  $4 -1
              beq  $0  $0 -4
               jr  $31
```

Vervollständigen Sie folgendes Ausführungsprotokoll. Um die Darstellung übersichtlich zu halten, tragen wir für Register, deren Wert unbestimmt ist, nichts ein.

Schritt	Register					Befehlszähler
	$0	$1	$2	$3	$4	*pc*
1.					4	0x00400000
2.			1		4	0x00400004

Wir rechtfertigen den ersten Schritt folgendermaßen: Der *pc* steht auf 0x00400000, demnach ist der aktuelle Befehl `ori $2 $0 1`. Da wir per Konvention das Zielregister in der Assemblerdarstellung immer ganz links, haben wir rt = 2, rs = 0 und imm = 1.

Der neue Inhalt der der Registerbank ergibt sich durch $r' = r[\text{rt} \mapsto f(r(\text{rs}), \text{imm})]$, wobei $f(s, i) = (s \mid \texttt{zext}_{16}^{32}(i))$, d.h. der Wert von $0 wird mit der Konstante 1 verodert und das Ergebnis in $2 abgelegt. Effektiv wird also die Konstante 1 nach $2 geladen.

Sinn dieser Aufgabe ist es, Sie darin zu trainieren die Semantikregeln schnell lesen und anwenden zu können - eine Fähigkeit die in der Klausur unerlässlich ist. Sie werden feststellen, dass Sie den zehnten Befehl deutlich schneller Ausführen können, als den ersten. Nach weiteren zehn Befehlen werden Sie vermutlich nicht mehr im Anhang nachsehen müssen.

⌟

Aufgabe 2.6 (Anordnung von Sprüngen in Schleifen. ⋆⋆)**.** Betrachten Sie die folgende Implementierung der Fakultätsfunktion und vergleichen Sie sie mit der in Abbildung 2.5.

```
    .text
    .globl fac
fac:
    li    $v0 1       # Lade Konstante 1 nach $v0
loop:
    blez  $a0 exit
    mul   $v0 $v0 $a0 # Multipliziere $v0 mit $a0 und speichere
                      # Ergebnis in $v0
    addiu $a0 $a0 -1  # Verringere $a0 um 1
    b     loop
exit:
    jr    $ra         # Kehre zum Aufrufer zurueck
```

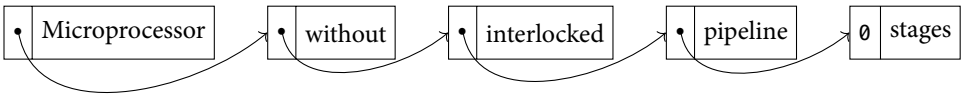
Fertigen Sie ein Ausführungsprotokoll beider Versionen an und vergleichen Sie die Anzahl der ausgeführten Befehle. Beachten Sie, dass ein bedingter Sprung auch als ausgeführter Befehl zählt, wenn er nicht verzweigt. Welche Variante ist bezüglich der Anzahl der auszuführenden Befehle vorzuziehen?

⌟

Aufgabe 2.7 (Listendarstellung. ⋆⋆⋆)**.** Wir stellen einfach verkettete Listen im Speicher dar. Ein Listenelement besteht aus der Adresse des Nachfolgerelements und den Nutzdaten. Wir verwenden die Adresse 0 um anzuzeigen, dass kein Nachfolgerelement existiert. Die Liste, die als Nutzdaten die Zeichenfolgen

Microprocessor, without, interlocked, pipeline, stages

enthält, sieht abstrakt dargestellt folgendermaßen aus:



Wir betrachten nun eine Darstellung der obigen Liste im Speicher. Dazu sehen wir uns den sogenannten Speicherauszug der entsprechenden Speicherregion an. Eine Zeile eines Speicherauszug besteht aus 3 Teilen: Links steht die Speicheradresse in Hexadezimaldarstellung, gefolgt von der Bytefolge der Länge 16, die an dieser Adresse im Speicher beginnt. Rechts steht eingegrenzt von `|` der Speicherinhalt interpretiert als ASCII-Zeichenfolge. Nicht druckbare Zeichen werden als `.` dargestellt.

```
10010000  14 00 01 10 4d 69 63 72  6f 70 72 6f 63 65 73 73  |....Microprocess|
10010010  6f 72 00 00 30 00 01 10  77 69 74 68 6f 75 74 00  |or..0...without.|
10010020  40 00 01 10 70 69 70 65  6c 69 6e 65 00 00 00 00  |@...pipeline....|
10010030  20 00 01 10 69 6e 74 65  72 6c 6f 63 6b 65 64 00  | ...interlocked.|
10010040  00 00 00 00 73 74 61 67  65 73 00 00 00 00 00 00  |....stages......|
10010050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

1. Geben Sie die Adresse des Bytes `4d` in der ersten Zeile an. Welchem Zeichen in der ASCII-Spalte entspricht es? Geben Sie zu jedem Element der Liste die Adresse im Speicherauszug an. Markieren Sie jetzt alle Bytes im Speicherauszug, die zur Darstellung einer Folgeelementadresse gehören. In welcher Endianness sind die Folgeelementadressen abgelegt?

2. Erklären Sie mithilfe von alignment zu welchem Listenelement das Byte mit der Adresse `0x10010013` gehört.

3. Sie sollen die im Speicherauszug gezeigte Liste mithilfe von MIPS-Assemblerdirektiven im Datensegment anlegen. Hinweis: Die folgende Vereinbarung legt ein Wort im Datensegment an, das seine eigene Adresse enthält:

   ```
     .data
   label:
     .word label
   ```

4. Schreiben Sie eine Unterprogramm, das gegeben die Adresse eines Listenelements die Zeichenketten der Elemente ausgibt. Testen Sie ihr Unterprogramm mit der Liste aus dem oben angelegten Datensegment.

⌟

Aufgabe 2.8 (Puffer kopieren. ⋆⋆)**.** Schreiben Sie ein MIPS-Programm `memcpy` dass eine gegebene Zahl an Bytes von einem Eingabepuffer in einen Ausgabepuffer kopiert. ⌟

Aufgabe 2.9 (★★★)**.** Schreiben sie ein MIPS-Programm, das eine Reihung von vorzeichenbehafteten 32 Bit breiten Zahlen in Dezimaldarstellung ausgibt. Ihr MIPS-Program soll die Adresse der ersten Zahl in Register `$a0`, und die Adresse direkt hinter der letzten Zahl in Register `$a1` erwarten. *Hinweis:* Suchen Sie zunächst nach einem `syscall` der die Ausgabe der Dezimaldarstellung vereinfacht. ⌟

Aufgabe 2.10 (Puffer ausgeben. ★★)**.** Schreiben Sie ein MIPS-Program, `println_range` dass eine gegebene Anzahl an Zeichen eines Puffers auf der Konsole ausgibt und schließlich einen Zeilenumbruch schreibt. ⌟

Aufgabe 2.11 (Bedingter Sprung (oder nicht). ★★)**.** Schreiben Sie ein Assemblerprogramm, das den Absolutwert eines 32-Bit-Wertes aus dem Datenbereich berechnet und ausgibt. ⌟

Aufgabe 2.12 (Eingabe/Ausgabe, ASCII und Schleifen in MIPS. ★★★)**.** Schreiben Sie ein Assemblerprogramm, das die Caesar-Verschlüsselung einer Zeichenkette mit einer Verschiebung von n berechnet und ausgibt. Bei der Caesar-Verschlüsselung wird jeder Buchstabe durch den Buchstaben ersetzt, der n Stellen weiter im Alphabet steht. Am Ende des Alphabets wird von vorne weitergezählt. Lesen Sie n und eine, bis zu 20 Zeichen lange, Zeichenkette vom Benutzer ein. Sie dürfen dabei annehmen, dass dieser nur die ASCII-Zeichen a-z, oder einen Zeilenumbruch (bei kürzerer Eingabe) eingibt. ⌟

Aufgabe 2.13 (Zahl zu Ziffer und zurück. ★★)**.** Wir betrachten die 36 Ziffern $0, \ldots, 9, A, \ldots, Z$ und stellen Sie als Zahlen im Interval $[0, 35]$ dar.

1. Schreiben Sie ein MIPS-Programm, das die Darstellung einer Ziffer in Register `$a0` erwartet, und die entsprechende Ziffer ausgibt. Beispielsweise soll ihr Programm die Ziffer A ausgeben, wenn `$a0` initial 10 war. *Hinweis:* Entweder Sie verwenden einen Sprungbefehl, oder sie benutzen das Datensegment geschickt.

2. Schreiben Sie ein MIPS-Programm, das den ASCII-Code einer Ziffer in Register `$a0` erwartet, und die Darstellung der entsprechenden Ziffer ausgibt. Beispielsweise soll ihr Programm 10 ausgeben, wenn `$a0` initial 65 war.

⌟

Aufgabe 2.14 (Schleifen vs. Rekursion am Beispiel des ggT. ★★)**.** Schreiben Sie zwei MIPS-Assemblerprogramme, die beide den *größten gemeinsamen Teiler* berechnen. Der $\mathrm{ggT}(a, b)$ zweier Zahlen $a, b \in \mathbb{N} \setminus \{0\}$ ist definiert als:

$$\mathrm{ggT}(a,b) = \begin{cases} a & \text{falls } a = b \\ \mathrm{ggT}(a-b,b) & \text{falls } a > b \\ \mathrm{ggT}(a,b-a) & \text{falls } b > a \end{cases}$$

Ein Beispiel:

$$\mathrm{ggT}(6,4) = \mathrm{ggT}(6-4,4) = \mathrm{ggT}(2,4-2) = \mathrm{ggT}(2,2) = 2$$

Das erste Programm (`ggt_rec`) soll den ggT rekursiv entsprechend obiger rekursiver Definition berechnen. Hierzu ruft sich das ggT-Unterprogramm selbst auf. Die zweite Variante (`ggt_iter`) soll den ggT ohne Unterprogrammaufruf mittels einer Schleife berechnen. Beachten Sie den zusätzlichen Aufwand (Aufsetzen der Aufrufschachtel, Register retten, etc.), die die rekursive Variante erzeugt. Die Parameter sollen in den Registern `$a0` und `$a1` übergeben werden, das Ergebnis in Register `$v0`.
Hinweis: Orientieren Sie sich an dem Beispielprogramm aus der Vorlesung, das die Fakultät berechnet. ⌟

Aufgabe 2.15 (Kodierung von MIPS-Befehlen. ⋆⋆⋆)**.** Dekodieren Sie die angegebene Folge von Befehlswörtern als ein MIPS Programm. Machen Sie sich an Hand der Regelschemata im Skript klar, was dieses Programm berechnet. Schreiben Sie eine `.data` Sektion und weiteren Code vor dieses Programm, so dass es eine sinnvolle Ausgabe produziert.

```
0085402a
15000001
00052021
34020001
0000000c
```

⌟

Aufgabe 2.16 (Laden in Register mit und ohne Vorzeichen-Erweiterung. ⋆)**.** Gegeben die folgende `.data`-Definition:

```
        .data
bytes:  .byte 0x7f, 0x8f
halves: .half 0x6655, 0xabcd
w:      .word 0x12345678
```

Geben Sie jeweils alle Befehle an, die durch Laden aus dem Datensegment die Werte 1.-6. im Register `$t0` produzieren. Achten Sie dabei auf Vorzeichenerweiterung und orientieren Sie sich an folgendem Beispiel:

```
0x12345678: lw $t0 w
```

1. `0x0000007f`
2. `0x0000008f`
3. `0xffffff8f`
4. `0x00006655`
5. `0x0000abcd`
6. `0xffffabcd`

⌟

Aufgabe 2.17 (Überlauf. ⋆⋆)**.** Schreiben Sie ein Programm, das die Werte der Register $a0$ und $a1$ addiert, und die Marke `overflow` anspringt, wenn ein Überlauf bezüglich der vorzeichenbehafteten Interpretation vorliegt. ⌟

Aufgabe 2.18 (Schachtel und Keller. ★★★)**.**

1. Beschreiben Sie die Aufrufkonvention in MIPS Assembler möglichst kurz. Können Sie am Registernamen ablesen, ob der Aufrufer das Register sichern muss?

2. In welchem Register steht die Adresse *sp* des Kellers? Ist die Adresse des nächsten freie Wortes im Keller höher oder niedriger als *sp*?

3. Betrachten Sie den folgenden Auszug eines MIPS-Unterprogramms `foo`. Nehmen Sie an, die Register `$t0`, `$t1`, `$s0` enthalten Werte, die nach dem Aufruf von `bar` noch verwendet werden. Setzen Sie eine Aufrufschachtel durch Einfügen geeigneter Befehle auf, sichern Sie genau die notwendigen Werte vor dem Aufruf, und stellen Sie sie nach dem Aufruf wieder her.

```
foo:
    # Code der $t0–$t1 und $s0 beschreibt
    jal bar
    # Code der $t0–$t1 und $s0 liest
    jr   $ra
```

⌟

Aufgabe 2.19 (Zeichenketten, Sprünge und Schleifen. ★★★)**.** Schreiben Sie ein Unterprogramm, dass die Anfangsposition einer Zeichenkette *a* innerhalb einer anderen Zeichenkette *b* berechnet. Eine Zeichenkette ist eine Reihung von Bytes, die mit der 0 abgeschlossen wird. Geben Sie in Register `$v0` -1 aus, wenn *a* nicht Teil von *b* ist, ansonsten den Index des Zeichens an dem *a* in *b* beginnt. *b* soll in Register `$a1` und *a* in Register `$a0` übergeben werden. Beachten Sie die Grenzfälle.

⌟

Aufgabe 2.20 (Generisches Sortieren durch Einfügen. ★★★★★)**.** Erweitern Sie die Insertion-Sort des Skriptes so, dass sie eine Reihung beliebiger Daten konstanter Länge sortieren kann, für die der Anwender eine Vergleichsfunktion angegeben hat. Ihre Implementierung soll vier Parameter nehmen:

`$a0`	Anfangsadresse der zu sortierenden Daten
`$a1`	Adresse des ersten Datums hinter der Reihung
`$a2`	Größe eines Datums
`$a3`	Adresse der Vergleichsfunktion

Die Vergleichsfunktion bekommt in Register `$a0` die Adresse des ersten und in Register `$a1` die Adresse des zweiten Datums. Auf der Webseite der Vorlesung finden Sie ein Hauptprogramm, dass die von Ihnen zu implementierende Funktion aufruft.
Hinweis: Gehen Sie wie folgt vor: Schreiben Sie für das Vertauschen der Daten ein Unterprogramm `exch`. Dieses wird wahrscheinlich ein Unterprogramm `memcpy` aufrufen wollen, dass *n* Bytes von Adresse *a* nach Adresse *b* bewegt.

⌟

Aufgabe 2.21 (Caller- vs. Callee-save. ⋆⋆)**.** Betrachten Sie die Implementierung von `print_hex` in Beispiel 2.6. Welchen Vorteil bringt es, die callee-save Register `$s0` und `$s1` anstelle von caller-save Register zu verwenden? ⌟

Aufgabe 2.22 (Ohne Sprung. ⋆⋆⋆)**.** Implementieren Sie das Unterprogramm `to_ascii` aus Beispiel 2.6. ohne Sprungbefehl (`jr` muss natürlich bleiben). ⌟

3 Einführung in C

Die Programmierung mit Assemblern ist mühselig, wie wir im vorigen Kapitel gesehen haben. Deshalb verwendet man bei der Software-Entwicklung höhere Programmiersprachen. Ein **Übersetzer (engl. compiler)** übersetzt dann das Programm der Hochsprache in die Maschinensprache und nimmt einem viele der Details, mit denen man sich in der Maschinensprach-Programmierung herumschlagen muss, ab. Dies sind vor allem:

- Assemblerprogramme sind nicht portabel. Sie sind immer im Befehlssatz einer Prozessorfamilie geschrieben. Soll das Programm auf einem anderen Prozessor laufen, muss es neu geschrieben werden. Schreibt man das Programm in einer Hochsprache und existiert ein Übersetzer für den neuen Prozessor, so kann das Programm einfach neu übersetzt werden.
- Man kann Speicherzellen keinen Namen geben, sondern sie nur durchnummerieren: Man muss Register verwenden und Feldern in Verbunden ein Offset zuweisen. Dies führt zu unleserlichen Programmen. Des Weiteren muss man entscheiden, wann man welche Werte in Registern hält, und wann sie in den Speicher geschrieben werden. Dies ist insbesondere problematisch, wenn die vorhandenen Register nicht ausreichen. Auch die Implementierung der Aufrufkonvention ist mühselig.
- Man kann seine Programme schlecht syntaktisch strukturieren. Insbesondere die uns Menschen vertraute Infixnotation von mathematischen Ausdrücken existiert in Maschinensprache nicht. Man muss einen komplizierteren Ausdruck von Hand serialisieren und selbst Zwischenergebnisse in Registern ablegen.
- Steuerstrukturen wie „Mache …solange Bedingung erfüllt ist“ oder „Wenn Bedingung gilt mache *A* ansonsten *B*“ müssen mühselig mit Sprunganweisungen implementiert werden.

Wir betrachten in diesem Kapitel die Programmiersprache C, die in allen obigen Punkten Abhilfe schafft. Der Anspruch von C ist nicht, eine besonders abstrakte Programmiersprache zu sein, sondern gerade so viel an Abstraktion zu bieten, wie man benötigt, um portable Programme schreiben zu können. C Programmierern ist es hierbei wichtig, dass der C Code in möglichst effizienten Maschinen-Code übersetzt wird, sprich die Abstraktionen, die C anbietet dürfen keinen „Overhead“

verursachen. Daher wird C von manchen Leuten auch „Hochsprachen-Assembler“ genannt. Des Weiteren führt dies zu einigen problematischen Spracheigenschaften, die wir in diesem Kapitel (→ Abschnitt 3.14) auch ansprechen werden. Diese sind Ursache für viele gravierende Sicherheitslücken.

C ist eine äußerst erfolgreiche Sprache. Alle modernen Betriebssysteme sind, wegen der Nähe von C zur Maschinensprache, in C geschrieben. Software, bei der Speicherverbrauch und Laufzeit wichtig sind, werden in C geschrieben. C kann ohne komplexes Laufzeitsystem ausgeführt werden und leicht in Maschinensprache übersetzt werden. Daher ist C bei eingebetteten Systemen die Standard-Sprache. Des Weiteren hat C viele andere erfolgreiche Sprachen (C++, C#, Java, JavaScript, etc.) beeinflusst.

3.1 Grundlagen

```
1  unsigned int fac(unsigned int n) {
2      unsigned int res;   /* Variablenvereinbarung */
3      res = 1;            /* Anweisung: Zuweisungsausdruck */
4      while (n > 1) {     /* Anweisung: Schleife */
5          res = res * n;  /* Anweisung: Zuweisungsausdruck */
6          n   = n - 1;    /* Anweisung: Zuweisungsausdruck */
7      }
8      return res;         /* Anweisung: Rückkehr zum Aufrufer */
9  }
10
11 int main() {
12     unsigned int res;
13     res = fac(3);
14     printf("%d\n", res);
15     return 0;
16 }
```

Abbildung 3.1: Ein C-Programm mit zwei Unterprogrammen, das die Fakultät von 3 berechnet und auf dem Bildschirm ausgibt.

3.1.1 Statische Eigenschaften

Betrachten wir unser erstes C-Programm in Abbildung 3.1. Wir diskutieren zunächst syntaktische Eigenschaften und Eigenschaften der statischen Semantik. Das sind Eigenschaften, die das Programm unabhängig von einer konkreten Eingabe hat, die also nicht einen Programmlauf, sondern die Struktur des Programms betreffen.

Jedes C-Programm besteht aus mindestens einer Datei. Jede einzelne Datei wird **Übersetzungseinheit (engl. translation unit)** genannt.

Jede Übersetzungseinheit besteht aus einer Liste an **Vereinbarungen (engl. declaration)**. Eine Vereinbarung gibt einem „Ding" einen Namen, den wir **Bezeichner (engl. identifier)** nennen. Alle nicht hervorgehobenen Buchstabenfolgen in Abbildung 3.1 sind Bezeichner. Die hervorgehobenen heißen **Schlüsselwörter (engl. key word)** und fungieren als Markierungen für spezifische syntaktische Konstrukte (z.B. **`while`**) oder sind vordefinierte Bezeichner (z.B. **`int`**). Bezeichner treten in zwei Weisen auf: **definierend** und **verwendend** (engl. defining and using occurrence). Unser Beispiel zeigt eine Übersetzungseinheit, in der zwei Unterprogramme (in C auch Funktionen genannt) `fac` und `main` vereinbart werden. Das Auftreten von `fac` in Zeile 1 ist definierend, da hier der Bezeichner `fac` mit dem Unterprogramm in Verbindung gebracht wird, und das in Zeile 12 ist verwendend, da sich `fac` hier auf das oben definierte Unterprogramm bezieht.

Betrachten wir zunächst das Unterprogramm `fac`. Die Vereinbarung des Unterprogramms in Zeile 1 sagt, dass `fac` einen Parameter `n` des Typs **`unsigned int`** und den Rückgabetyp **`unsigned int`** hat. Der Code des Unterprogramms, sein **Rumpf (engl. body)**, besteht aus einem **Block**: Eine von geschweiften Klammern eingeschlossene Folge von

Anweisungen (engl. statement, command)[1]. Einige Anweisungen werden durch Strichpunkte abgeschlossen. Die Strichpunkte sind lediglich Trennzeichen und tragen keine weitere Bedeutung. Der äußere Block unseres Beispielprogramms besteht aus den Anweisungen der Zeilen 2, 3, 4 und 8. (Die while-Schleife in Zeile 4 einhält einen weiteren Block, der in dem äußeren geschachtelt ist.)

Es gibt im wesentlichen vier Arten von Anweisungen: Variablenvereinbarungen, Ausdrücke, Blöcke und Konstrukte zur Ablaufsteuerung.

Die erste Anweisung des Rumpfes von `fac` ist eine **Variablenvereinbarung (engl. variable declaration)**. Sie besteht immer aus einem **Typ**, in diesem Fall **`unsigned int`**, und einem Bezeichner, hier: `res`. Den Bezeichner, der in einer Variablenvereinbarung auftritt, nennt man **Variable**[2]. Das Auftreten einer Variable in einer Vereinbarung ist definierend. Alle anderen Auftreten sind verwendend.

Der Bezeichner `n` in Zeile 1 ist auch eine Variable. Jedes Unterprogramm kann eine Liste von **Parametern** vereinbaren.

Jede Vereinbarung hat einen **Sichtbarkeitsbereich (engl. scope)**. Der Sichtbarkeitsbereich des definierten Bezeichners legt den Bereich des Programmtextes fest, in dem sich ein weiteres (verwendendes) Auftreten des Bezeichners auf diese Vereinbarung bezieht. So ist zum Beispiel die Variable `res` im gesamtem Rumpf (von ihrer Vereinbarung an) sichtbar, wodurch sich das Auftreten in der Anweisung `res = res * n;` auf diese Vereinbarung bezieht. Der Sichtbarkeitsbereich von **lokalen Variablen**, also solche, die innerhalb einer Funktion vereinbart werden, erstreckt sich von ihrer Vereinbarung bis ans Ende des innersten sie umschließenden Blockes[3].

3.1.2 Dynamische Eigenschaften

Wir diskutieren nun informell die dynamischen Eigenschaften, die Semantik von C. Das sind die Eigenschaften, die festlegen, was genau beim Ablauf eines C-Programms passiert. Das Ausführungsmodell von imperativen Sprachen (und somit auch von C) ist sehr eng an die Maschine angelehnt. Wie in der Einleitung zu diesem Kapitel diskutiert, bieten imperative Sprachen Abstraktionen für die grundlegenden Bausteine einer Maschinensprache, um ein Programmieren „nahe an der Hardware“ zu ermöglichen.

Charakteristisch für die imperative Programmierung ist das Konzept der **Anweisung** und das der **(imperativen) Variable**. Da der Begriff der Variable in der imperativen Programmierung von dem der Mathematik und der funktionalen Programmierung abweicht und gleichzeitig zentral für das Verständnis ist, diskutieren wir ihn in Abschnitt 3.3 ausführlich. Hier reicht es zunächst zu verstehen, dass eine Variable bei der Ausführung an einen Behälter gebunden wird, in dem man einen Wert eines Typen ablegen kann. So entsteht beispielsweise durch das Ausführen der Variablenvereinbarung in Zeile 2 ein neuer Behälter, der einen Wert des Typs **`unsigned`** aufnehmen kann.

Die Ausführung eines C-Programms beginnt immer in dem Unterprogramm, das den Namen `main` trägt. Die Ausführung unseres Programms beginnt also in Zeile 11 Wie die Befehle eines Maschinen-

[1] Anweisungen sind charakteristisch für imperative Programmiersprachen, daher auch der Name vom lateinischen *imperare* = befehlen.

[2] Der Begriff der Variablen wird von vielen Autoren unterschiedlich definiert. Wir folgen hier der gewohnten Bedeutung aus der Mathematik, wo man das Auftreten eines Buchstabens in einem Term Variable nennt. Gleichwohl *bedeuten* Variablen in imperativen Programmen etwas anderes; dazu mehr in Bemerkung 3.2.

[3] Da Blöcke geschachtelt werden können, können Variablenvereinbarungen auch überdeckt werden, wie in folgendem Beispiel zu sehen:

```
void foo(int x) {
    int y = 0;
    if (x > 0) {
        int y = 0;
        y += 1;
    }
    return y;
}
```

Hier bezieht sich das verwendende Auftreten von y innerhalb des ifs auf die Vereinbarung im if, nicht auf die in der zweiten Zeile.

sprach-Programms werden die Anweisungen eines Unterprogramms nacheinander abgearbeitet. Die erste Anweisung (in Zeile 12) legt einen neuen Behälter an in dem ein **`unsigned int`** abgelegt werden kann und bindet die Variable `res` daran. Danach wird das Unterprogramm `fac` aufgerufen (immer noch Zeile 12). Beim Aufruf entstehen neue Behälter für alle Parameter des aufgerufenen Unterprogramms. In diesem Fall einer für den Parameter `n`. Beim Aufruf werden die Ausdrücke in den runden Klammern nach dem Namen des zu rufenden Unterprogramms ausgewertet. Die entstandenen Werte heißen **Argumente** oder auch **tatsächliche Parameter**. Diese Werte werden in die, für die Parameter des Unterprogramms erstellten Behälter gelegt. In unserem Beispiel wird die `3` in den Behälter, der an `n` gebunden ist, gelegt. Danach fährt die Programmausführung im Unterprogramm fort.

Die Anweisungen eines Blockes werden nacheinander abgearbeitet. Die Ausführung des Unterprogramms beginnt also in Zeile 2, deren Bedeutung wir schon geklärt haben. Die Anweisung in Zeile 3 wertet den Ausdruck auf der rechten Seite des Zuweisungsoperators aus (die Konstante 1) und schreibt den errechneten Wert (hier die 1) in den Speicherplatz, der sich aus der linken Seite ergibt. Die **`while`**-Schleife wertet die sogenannte **Abbruchbedingung** (hier `n > 1`) aus. Wenn das Ergebnis ungleich 0 ist, führt sie die Anweisung ihres Rumpfes (hier der geschachtelte Block) aus. Diese beiden Schritte werden solange wiederholt, bis die Abbruchbedingung zu 0 auswertet. Innerhalb des Rumpfes der **`while`**-Schleife finden sich wieder zwei Zuweisungen, die, wie bei Zeile 2 beschrieben, ausgewertet werden.

Danach kehrt das Unterprogramm `fac` zu seinem Aufrufer zurück und übergibt diesem den Wert, der zu diesem Zeitpunkt in der Variablen `res` steht. `main` ruft dann noch das Unterprogramm `printf` auf, das hier zwei Argumente bekommt und bewirkt, dass das Ergebnis des Aufrufs an `fac` auf dem Bildschirm ausgegeben wird. Danach beendet `main` die Programmausführung durch das **`return`**.

3.2 Ausführungsprotokolle

Beim Programmieren ist es hilfreich, Programmteile im Kopf ausführen zu können, um die Bedeutung eines Programmteils schnell erfassen zu können. Hierzu muss man die Bedeutung der einzelnen Anweisungen des Programms genau verstehen. Dieses Verständnis übt man am besten dadurch ein, dass man zunächst das **Ausführungsprotokoll** kleinerer Programme auf Papier schreibt[4].

[4] Wir diskutieren die Semantik einer Teilmenge von C, und insbesondere den Zustandsbegriff, formal in Kapitel 5. Hier begnügen wir uns mit einer informellen Darstellung.

Aufruf von `main`

Zeile	`res`	Nebenwirkung
11		
12		
13	?	Aufruf von `fac` →
14	6	Ausgeben von `res`
15		Beenden

Aufruf von `fac`

Zeile	`n`	`res`	Nebenwirkung
1	3		
2			
3		?	
4		1	
5			
6		3	
7	2		
4			
5			
6		6	
7	1		
4			
8			Rückkehr mit Rückgabewert 6

Abbildung 3.2: Ausführungsprotokoll des Programms aus Abbildung 3.1.

Wir stellen das Ausführungsprotokoll tabellarisch dar. Jede Zeile der Tabelle entspricht der Abarbeitung einer Anweisung und stellt die Belegung der Variablen **vor Abarbeitung** der Anweisung dar. In der ersten Spalte vermerken wir die Zeilennummer der Anweisung, die ausgeführt wird[5]. Für jede Variablenvereinbarung legen wir eine weitere Spalte an, die wir mit dem Bezeichner der Vereinbarung betiteln. Der Einfachheit halber nehmen wir an, dass es keine sich überdeckenden Bezeichner gibt. Eine Anweisung kann verschiedene **Wirkungen** (engl. effect) auf den **Zustand** (engl. state) des Programms haben: Beispielsweise kann eine Variablenvereinbarung neue Behälter anlegen, das Ende eines Blockes existierende Behälter freigeben, eine Zuweisung Behälter auslesen oder beschreiben. Ändert sich der Inhalt einer Variable nicht, so übertragen wir den Inhalt nicht noch einmal. Initial ist der Inhalt eines Behälters undefiniert, was wir durch ein Fragezeichen darstellen. Neben den Änderungen an den Behälterinhalten kann eine Anweisungen „Nebenwirkungen“ haben, zum Beispiel den Aufruf eines Unterprogramms oder das Ausgeben von Text auf dem Bildschirm. Dies vermerken wir in einer Spalte mit dem Titel „Nebenwirkungen“.

[5] Natürlich können in einer Zeile des Programms auch mehrere Anweisungen stehen. Wir wollen hier aber der Einfachheit halber annehmen, dass das nicht der Fall ist.

Es ist zu beachten, dass jeder Aufruf eines Unterprogramms im Ausführungsprotokoll eine neue Tabelle erzeugt. In unserem Beispiel ist das der Aufruf an `main`, mit dem die Programmausführung beginnt und der darin enthaltene Aufruf an `fac`. Das erzeugt geschachtelte Ausführungs-

protokolle. Das Ausführungsprotokoll des gerufenen Unterprogramms tragen wir in die Spalte „Nebenwirkungen“ des aufrufenden Unterprogramms ein. Da Unterprogramme sich auch (mittelbar) selbst aufrufen können, kann es in einem Ausführungsprotokoll mehrere Tabellen des gleichen Unterprogramms geben. Es ist zu beachten, dass die Bindung der Variablen des Unterprogramms an Behälter **pro Aufruf** existiert.

3.3 Imperative Variablen und das Speichermodell von C

Berechnungen werden mit **Werten** durchgeführt. Beispielsweise ist `2` ein Wert. Ein **Typ** ist eine Menge von Werten. Man unterscheidet zwischen Basistypen (auch primitive Typen genannt) und zusammengesetzten Typen. Die Werte von Basistypen sind, im Gegensatz zu den Werten zusammengesetzter Typen, nicht aus anderen Werten zusammengesetzt. So ist `2` ein Wert des Basistyps **`int`** und `{ 3, 4 }` ein Wert des zusammengesetzten Typs **`struct { int a, b; }`**, der Paare von `int`s darstellt.

Zur Laufzeit des Programms werden Werte in **Behältern**[6] gespeichert. Jeder Behälter hat eine **Größe** (in Bytes), eine **Adresse** und eine **Lebensdauer**. Die Adresse ist die **Identität** des Behälters: Zu jedem Zeitpunkt identifiziert sie ihn eindeutig. Es gibt keine zwei unterschiedlichen Behälter mit der gleichen Adresse.

[6] Der englische C-Standard [ISO99] spricht von „object". Da dieser Begriff sehr generisch ist und leicht zu Verwechslungen führt, verwenden wir den Begriff Behälter.

Adressen von Behältern sind auch Werte. Es ist daher nur konsequent (und praktisch) Behälter zuzulassen, deren Werte Adressen anderer Behälter sind. Solche Behälter heißen **Zeiger**. Wir diskutieren sie ausführlich in Abschnitt 3.9.

Variablen sind Bezeichner, die in einer Variablenvereinbarung vorkommen. Diese besteht aus der Variable selbst und einem Typ. Beispielsweise vereinbart **`int`** `x;` eine Variable `x` des Typs **`int`**. Bei der Ausführung der Variablenvereinbarung passieren zwei Dinge: Zuerst wird ein neuer Behälter angelegt, dessen Größe anhand des Typs der Variablenvereinbarung ermittelt wird[7]. Ist zum Beispiel ein **`int`** 4 Bytes groß, so muss der Behälter, den `x` referenziert, eine Größe von mindestens 4 Bytes haben. Dann wird die Variable an die Adresse des Behälters gebunden. Man sagt auch, die Variable **referenziert** den Behälter. Diese Assoziation bleibt bis zur Freigabe des Behälters, am Ende der Lebenszeit der Variable bestehen.

[7] In C hat jeder Typ *T*, außer dem leeren Typ **`void`**, die Größe **`sizeof`**`(`*T*`)`.

Bemerkung 3.1 (Sprechweise)**.** In der Praxis verwendet man den Begriff „Variable" meist für beides: Den Bezeichner und auch den Behälter. So sagt man beispielsweise, wenn man über eine konkrete Ausführung spricht, „die Variable `x` hat den Wert 5" und meint damit, dass der Behälter, der an die Variable `x` gebunden ist, den Inhalt 5 hat. ⌟

Bemerkung 3.2 (Imperative Variablen im Vergleich zu Variablen in der Mathematik)**.** In funktionalen Sprachen, wie auch in der Mathematik, kommt Variablen eine andere Bedeutung zu als in imperativen Programmiersprachen.

Die Essenz ist, dass in beiden Paradigmen, dem funktionalen und dem imperativen, Variablen bei ihrer Vereinbarung unveränderlich gebunden werden. Bei funktionalen Programmen an einen Wert, bei imperativen Programmen jedoch an die Adresse eines Behälters, dessen Inhalt sich verändern kann.

Dementsprechend gibt es in der imperativen Programmierung zwei Arten, eine Variable auszuwerten: Zur Adresse des an sie gebunden Behälters (L-Auswertung) und zu dessen Inhalt (R-Auswertung). Beide besprechen wir im Detail in Abschnitt 3.7.2 und Kapitel 5. ⌟

```
int main() {
    int x;
    x = 1;
    x = x + 1;
    return 0;
}
```

Die Lebensdauer eines Behälters bestimmt, wann er während der Programmausführung angelegt und wieder freigegeben wird. Diesbezüglich unterscheidet man mehrere Arten:

3.3.1 Lokale Variablen

Lokale Variablen werden durch die Ausführung einer Variablenvereinbarung angelegt und werden freigegeben, nachdem die letzte Anweisung ihres innersten, umschließenden Blockes abgearbeitet wurde.

Der Bezeichner der Vereinbarung wird an die Adresse der Variable gebunden. Man kann sich also innerhalb des innersten umschließenden Blockes durch das Nennen des Bezeichners auf die Variable beziehen.

Beispiel 3.1. Betrachten wir folgendes Beispiel: Die erste Anweisung bewirkt das Anlegen eines neuen Behälters, der einen **`int`** aufnehmen kann. Der Bezeichner `x` wird dann an die Adresse dieses Behälters gebunden. Die zweite Anweisung schreibt dann den Wert `1` in den Behälter (dessen Adresse an `x` gebunden ist). Die dritte Anweisung liest den Inhalt des Behälters, addiert `1` darauf und legt das Ergebnis wieder darin ab. ⌟

3.3.2 Globale Variablen

Globale Variablen werden außerhalb von Funktionen vereinbart[8]. Ihre Größe wird wie die einer lokalen Variable bestimmt. Genauso wird der vereinbarte Bezeichner an ihre Adresse gebunden. Der Unterschied zur lokalen Variable ist, dass die Lebensdauer der globalen Variable sich vom Programmstart zum Programmende erstreckt, die Variable also immer verfügbar ist.

Globale Variablen sind schlechter Programmierstil und, wo möglich, zu vermeiden. Der Grund ist, dass über globale Variablen meist Abhängigkeiten zwischen verschiedenen Funktionen entstehen, die schwer nachvollziehbar sind. Das Verständnis und das korrekte Arbeiten eines Unterprogramms kann dann nicht mehr individuell für jedes Unterprogramm sichergestellt werden.

Beispiel 3.2 (Globale Variable)**.** Betrachten wir das folgende Unterprogramm:

```
int data[1024];

void sort() {
    /* sortiere die Daten in data */
}
```

[8] Man kann globale Variablen auch innerhalb von Funktionen vereinbaren, wenn man die Qualifizierer **`static`** oder **`extern`** verwendet. Wir gehen hier aber nicht weiter darauf ein.

Diese Sortierfunktion funktioniert nur, wenn man die Daten vorher in die Reihung `data` kopiert hat. Das macht die Funktion unflexibel, da die Eingabe nur an einer Stelle stehen darf. Des Weiteren ist die Funktion schwerer verständlich, da aus ihren Parametern nicht hervorgeht, was sortiert werden soll. ⌟

3.3.3 Dynamisch angeforderte Behälter

Oftmals bemisst sich die Anzahl der anzulegenden Behälter oder aber auch deren Größe nach der Eingabe des Programms und ist statisch[9] nicht bekannt. Solche Variablen müssen **dynamisch** angelegt werden. Die Funktion `malloc` nimmt eine Anzahl von Bytes entgegen, legt einen neuen Behälter dieser Größe an und liefert dessen Adresse zurück. Der Behälter kann mit einem Aufruf an `free`, dem man seine Adresse übergeben muss, wieder freigegeben werden.

[9] statisch heißt: ohne das Programm auszuführen

Nicht freigegebene Behälter sind ein Programmierfehler und führen zu sogenannten **Speicherlecks** (engl. memory leaks). Vor allem für länger laufende Programme sind diese ein ernsthaftes Problem, da nach und nach der verfügbare Speicher ausgeht.

Beispiel 3.3 (Dynamisch angerforderte Behälter)**.** Betrachten wir das folgende Unterprogramm:

```
int *unit_vector(int dimension, int n_dimensions) {
    int *res = malloc(sizeof(res[0]) * n_dimensions);
    for (int i = 0; i < n_dimensions; i++)
        res[i] = 0;
    res[dimension] = 1;
    return res;
}
```

Die Anweisung in der zweiten Zeile fordert zwei neue Behälter an: Der erste kann eine **Adresse** aufnehmen; seine Adresse wird an die Variable `res` gebunden. Der zweite ensteht durch den Aufruf an `malloc`. Das Argument des Aufrufs von `malloc` bestimmt die Größe des Behälters. `sizeof(res[0])` ist die Größe eines `int`s (siehe Abschnitt 3.9) und davon sollen `n_dimensions` viele bereitgestellt werden. Der Aufruf an `malloc` fordert also einen Behälter an, der so groß ist, dass `n_dimensions` viele `int`s darin Platz finden. `malloc` liefert die Adresse des Behälters zurück. Diese wird dann in dem Behälter, der an `res` gebunden ist, abgelegt.

Die darauf folgende Schleife schreibt in jeden dieser `int`s eine 0 und die Anweisung danach schreibt an der `dimension`-ten Position eine 1 in die Reihung. Schließlich liefert das Unterprogramm die Adresse des angeforderten Behälters an seinen Aufrufer zurück.

Es ist zu beachten, dass die Lebenszeit des dynamisch (mit `malloc`) angeforderten Behälters nicht bei der Rückkehr des Unterprogramms erlischt, wohl aber die des Behälters, der an `res` gebunden ist und die Adresse der Reihung enthält. ⌟

3.3.4 Das C-Speichermodell

Wir haben bereits gesehen, dass Behälter eine Adresse haben. Im Vergleich zum Hauptspeicher eines Rechners, wie wir ihn in Kapitel 2 kennen gelernt haben, weist der Speicher eines C-Programms weniger Struktur auf. Insbesondere ist nicht definiert, dass die Adresse eines Behälters auch eine Hauptspeicher-Adresse ist.[10]

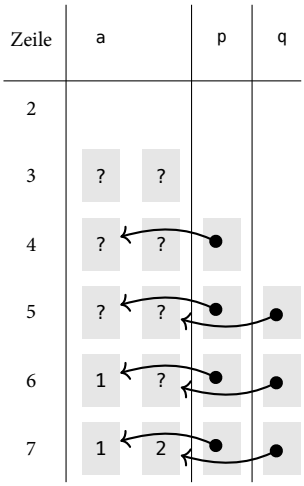
Allerdings erlaubt C Behälter, die mehr als einen Wert umfassen, was insbesondere für zusammengesetzte Daten wichtig ist. Insbesondere erlaubt es C, die einzelnen Komponenten eines solchen Datums zu adressieren. So kann man beispielsweise aus der Adresse eines Behälters und einem Versatz (engl. offset) die Adresse eines Bytes innerhalb eines Behälters ermitteln. Somit kann es mehrere unterschiedliche Adressen geben, die auf denselben Behälter verweisen, aber unterschiedliche Stellen darin adressieren. Formal besteht eine Adresse aus einem Paar aus der Behälteradresse und einem Versatz, der zwischen 0 und der Größe des Behälters liegt.

Die Adressen, die in denselben Behälter verweisen sind totalgeordnet. Dies ermöglicht dann Zeigerarithmetik, auf die wir in Abschnitt 3.9 eingehen.

[10] Zunächst hört sich das unintuitiv und restriktiv an, sollte doch C eine hardware-nahe Programmierung ermöglichen. Zum einen ist es aber sinnvoll, nicht genau zu spezifizieren, was eine Adresse ist, da viele Prozessoren, auf denen C-Programme laufen sollen, komplexere Arten der Adressierung haben, wie wir sie kennengelernt haben. Zum anderen würde eine solche Festlegung den Übersetzer stark einschränken, da er dann lokale Variablen nicht mehr so frei dem Hauptspeicher und den Registern zuordnen könnte.

Beispiel 3.4. Betrachten wir folgendes Programm:

```
1  void foo() {
2      int a[2];
3      int *p = a;
4      int *q = a + 1;
5      *p = 1;
6      *q = 2;
7  }
```



Zeile 1 (des Rumpfes) legt ein Behälter an, in dem zwei **int**s Platz finden. Zeile 2 legt einen neuen Behälter an, der eine Adresse aufnehmen kann, bindet diesen Behälter an p, und legt die Adresse des ersten **int**s von a darin ab. Zeile 3 legt einen neuen Behälter an, der Adressen aufnehmen kann, bindet diesen Behälter an q, und legt die Adresse des zweiten **int**s von a darin ab. Die beiden letzten Zeilen speichern die Werte 1 und 2 in den ersten bzw. zweiten **int** von a.

In Ausführungsprotokollen schreiben wir Adressen nicht explizit. Zeiger notieren wir so, dass wir einen Pfeil auf den referenzierten Behälter zeichnen. ⌟

3.4 Übersetzungseinheiten

Ein C-Programm besteht normalerweise aus mehreren Übersetzungseinheiten. Eine Übersetzungseinheit ist eine Datei mit der Endung `.c`. Der C-Übersetzer übersetzt die Unterprogramme einer Übersetzungseinheit nach Maschinencode (engl. object code), den er dann in einer Binärdatei mit der Endung `.o` ablegt. In einem Unix-System übersetzt das Kommando[11]

```
$ cc -o x.o -c x.c
```

die Übersetzungseinheit `x.c` in die Binärdatei `x.o`. Mehrere übersetzte Übersetzungseinheiten können mit dem C-Compiler dann zu einer ausführbaren Datei zusammengebunden werden:

```
$ cc -o prg x.o y.o z.o
```

wobei `prg` der Name des ausführbaren Programms ist und frei gewählt werden kann. Danach findet sich im aktuellen Verzeichnis eine Datei `prg`, die mit

```
$ ./prg
```

gestartet werden kann.

[11] `$` bezeichnet die Eingabeaufforderung der Unix shell.

3.4.1 main

Damit das Bauen der ausführbaren Datei gelingt, muss es in genau einer Übersetzungseinheit ein Unterprogramm mit dem Namen `main` geben. Die Programmausführung beginnt mit dem Aufruf dieses Unterprogramms. Man nennt das Unterprogramm `main` auch oft **Hauptprogramm**. Unix-Programme (und somit auch C-Programme) können beim Aufruf mit Argumenten versehen werden. Diese werden dem Unterprogramm `main` in Form der Parameter `argc` und `argv` übergeben. `argc` enthält die Anzahl der Argumente beim Aufruf, wobei der Programmname auch als Parameter zählt. Die Zeichenketten der anderen Parameter sind über die Reihung `argv` zugängig. Eine Zeichenkette ist in C meist eine null-terminierte Reihung von Buchstaben. Eine Art sich auf eine Zeichenkette zu beziehen, ist die Adresse ihres ersten Buchstabens. `argv` ist eine Reihung von Zeigern auf erste Buchstaben, also eine Reihung von Zeichenketten (siehe Abbildung 3.3). Ein Zeiger ist eine Variable, deren Inhalt eine Adresse ist.

Betrachten wir ein Hauptprogramm, das unser Unterprogramm `fac` (siehe voriges Kapitel) aufruft und das Argument an `fac` von der Kommandozeile bezieht.

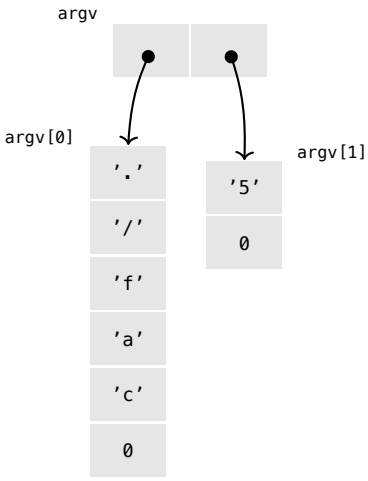


Abbildung 3.3: Darstellung der Variable `argv` nach Aufruf des Programms mit `./fac 5`

```c
#include <stdio.h>
#include <stdlib.h>
/* Hier steht das Unterprogramm fac */
int main(int argc, char *argv[]) {
    if (argc <= 1) {
        fprintf(stderr, "syntax: %s <value>\n",
                argv[0]);
        return 1;
    }

    unsigned n = atoi(argv[1]);
    unsigned r = fac(n);
    printf("%u\n", r);
    return 0;
```

Abbildung 3.4: Hauptprogramm für das Fakultäts-Unterprogramm

Nach dem Bauen mit

```
$ cc -o fac.o -c fac.c
$ cc -o fac fac.o
```

schlägt der Aufruf des Programms von der Kommandozeile fehl:

```
$ ./fac
syntax: ./fac <value>
```

Die ausgegebene Meldung ermahnt uns, die Zahl anzugeben, deren Fakultät wir berechnen wollen. Der folgende Aufruf mit Argument produziert dann die gewünschte Ausgabe:

```
$ ./fac 5
120
```

Das Hauptprogramm überprüft zunächst, ob ein Parameter übergeben wurde. Hierzu muss `argc` größer 1 sein, da der Programmname auch als Parameter zählt. Falls nein, wird eine entsprechende Warnung ausgegeben und das Hauptprogramm wird mit dem Wert 1 beendet[12]. Ansonsten wird das erste Argument (eine Zeichenkette) in eine Ganzzahl konvertiert und dessen Fakultät berechnet. Diese wird danach mittels `printf` ausgegeben und das Hauptprogramm wird mit dem Wert 0 beendet.

[12] In Unix kann jedes Programm beim Beenden einen sog. „exit code" zurückliefern. In einem C Programm ist der Rückgabewert von `main` der exit code. Die Konvention sagt, dass ein exit code von 0 bedeutet, dass das Programm fehlerfrei abgelaufen ist.

3.4.2 Unterprogramme aus anderen Übersetzungseinheiten aufrufen

Nehmen wir an, wir wollen das Hauptprogramm `main` und das Unterprogramm `fac` in unterschiedliche Übersetzungseinheiten aufteilen. Es ist oft sinnvoll, zusammengehörende Unterprogramme in eigenen Übersetzungseinheiten zu bündeln. Das Hauptprogramm steht oft alleine, da es inhaltlich nicht direkt zu den anderen Unterprogrammen gehört. Das Unterprogramm `fac` könnte man in einem anderen Projekt nochmals wiederverwenden; unser Hauptprogramm eher nicht. Daher ist es sinnvoll beide Unterprogramme in zwei Übersetzungseinheiten zu trennen. Alle Übersetzungseinheiten werden getrennt voneinander übersetzt.

Löscht man das Unterprogramm `fac` aus der Übersetzungseinheit in Abbildung 3.4, so lehnt der Übersetzer das Übersetzen ab, da er nichts

fac.c

```
#include "fac.h"

unsigned int fac(unsigned int n) {
    unsigned int res = 1;
    while (n > 1) {
        res = res * n;
        n   = n - 1;
    }
    return res;
}
```

fac.h

```
#ifndef FAC_H
#define FAC_H
unsigned fac(unsigned n);
#endif /* FAC_H */
```

main.c

```
#include <stdio.h>
#include <stdlib.h>

#include "fac.h"

int main(int argc, char *argv[]) {
    if (argc <= 1) {
        fprintf(stderr, "syntax: %s <value>\n",
                argv[0]);
        return 1;
    }

    unsigned n = atoi(argv[1]);
    unsigned r = fac(n);
    printf("%u\n", r);
    return 0;
}
```

Abbildung 3.5: Fakultäts-Unterprogramm und Hauptprogramm in zwei getrennten Übersetzungseinheiten. Die Headerdatei `fac.h` beinhaltet den Prototyp des Unterprogramms `fac`. Die Präprozessordirektiven (**`#ifdef`**, etc.) verhindern das erneute Inkludieren des Dateiinhalts. Dies ist nötig, um endlos rekursive Include-Sequenzen zu brechen.

über den Typ des Unterprogramms `fac` weiß. Für eine korrekte Übersetzung ist die Kenntnis des Typs von `fac` unerläßlich.[13] Der Typ von `fac` kann nun in der Übersetzungseinheit `main.c` dadurch bekannt gemacht werden, in dem man den **Prototyp** von `fac` angibt. Der Prototyp besteht aus dem Namen des Unterprogramms, dem Rückgabetyp und den Typen der Parameter.

```
unsigned int fac(unsigned int);
```

In C spricht hier auch von einer **Funktionsdeklaration** (im Gegensatz zu einer **Funktionsdefinition**), da man nur die Signatur der Funktion bekannt gibt. In der Praxis dupliziert man natürlich nicht die Prototypen jedes Unterprogramms in jede Übersetzungseinheit, in der sie verwendet werden. Einerseits wäre das zuviel Schreibarbeit, andererseits fehleranfällig, da man, wenn man das Unterprogramm ändert (beispielsweise einen Parameter hinzufügt oder den Typ eines Parameters ändert), alle Übersetzungseinheiten in denen der Prototyp steht, anpassen muss. Daher erstellt man eine sogenannte **Headerdatei**, die durch den **Präprozessor** eingebunden wird.

Der Präprozessor ist ein eigenes Programm, das vom C-Compiler aufgerufen wird, bevor der eigentliche Übersetzungsvorgang beginnt. Er erzeugt aus einem Text[14] einen neuen Text, in dem er Präprozessor-Direktiven expandiert. Alle Präprozessordirektiven beginnen mit einer Raute (#). Die Direktive **`#include "x.h"`** beispielsweise stoppt das Präprozessieren der Datei in der sie steht, präprozessiert die Datei `x.h` und fährt dann mit der Ausgangsdatei fort. Das hat zur Folge, dass in unserem Beispiel in Abbildung 3.5 durch das Präprozessieren der Inhalt von `fac.h` in die Dateien `fac.c` und `main.c` quasi „eingeblendet“ wird.

[13] Anhand der Typen der Parameter (die im Typ des Unterprogramms enthalten sind), kann der Compiler ermitteln, welche Parameter in welchen Registern zu übergeben sind (→ Abschnitt 2.9.2). Damit der vom Compiler an der Aufrufstelle generierte Code mit dem Code des Unterprogramms korrekt zusammenspielt, muss in allen Übersetzungseinheiten in denen ein Unterprogramm aufgerufen oder vereinbart wird, der gleiche Prototyp des Unterprogramms vereinbart sein.

[14] Der Text muss nicht notwendigerweise ein C-Programm sein. Der C-Präprozessor ist auch in anderen Sprachen beliebt.

3.4.3 Makefiles

Damit man nicht alle Übersetzungseinheiten (in der Praxis können es hunderte oder gar tausende werden) von Hand per Kommandozeile bauen muss[15], gibt es das Unix-Programm `make`. Es liest eine Datei namens `Makefile` ein, das beschreibt, wie man ein Programmierprojekt baut. Diese Beschreibung enthält die **Abhängigkeiten** der beteiligten Dateien untereinander und eine Beschreibung, wie man die Dateien auseinander erzeugt:

```
fac: fac.o main.o
  cc -o $@ $^

main.o: main.c fac.h
fac.o: fac.c fac.h

%.o: %.c
  cc -o $@ -c $<
```

Die ersten beiden Zeilen des `Makefile`s sagen, dass man, um die Datei `fac` zu bauen, die Dateien `fac.o` und `main.o` benötigt und dass das Kommando `cc -o fac fac.o main.o` dann die Datei `fac` produziert. Die letzten beiden Zeilen sagen, dass man, um eine Datei, die auf `.o` endet, zu erstellen, eine Datei benötigt, deren Dateinamen vor dem Punkt gleich lautet, aber auf `.c` endet. Das Kommando `cc -o x.o -c x.c` produziert dann aus einer Datei `x.c` die Datei `x.o`. Die verwendeten Platzhalter haben die folgende Bedeutung:

`$@`	Das „Ziel“: Text links des Doppelpunkts
`$<`	Die erste „Voraussetzung“: Erstes Wort rechts des Doppelpunkts
`$^`	Alle Voraussetzungen: Alle Worte rechts des Doppelpunkts

Der Einsatz von `make` ist nicht auf C beschränkt. Es ist ein allgemeines Werkzeug in dem Bauprozesse und Abhängigkeiten beschrieben werden kann. Seine Verwendung in C-Projekten ist allerdings weit verbreitet.

[15] Um das ausführbare Programm zu binden, muss man nicht immer alle `.o`-Dateien bauen. Lediglich wenn die Übersetzungseinheit geändert wurde, muss man mit einem erneuten Starten des Compilers die `.o` Datei neu bauen. Ist ein Projekt sehr groß, kann man sich leicht vorstellen, dass man leicht die Übersicht verliert, welche `.o` Dateien neu gebaut werden müssen. `make` nimmt einem dies ab, indem es das Datum der letzten Dateiänderung der `.o` und der `.c` Datei vergleicht und die Datei nur dann neu baut, wenn die Quelldatei neuer ist.

3.5 Basistypen

Jede Variable in C hat einen Typ. Dieser bestimmt, welche Werte in dem Behälter, für den die Variable steht, abgelegt werden können. Man unterscheidet zwischen Basistypen und **zusammengesetzten Typen** (engl. compound type). Die Basistypen in C umfassen verschiedene Arten von Zahldarstellungen und dazugehörige Wertebereiche. Man unterscheidet zwischen Ganzzahl-Typen (engl. integer types) und Gleitkomma-Typen (engl. floating point types). Die Ganzzahl-Typen **`char`**, **`short`**, **`int`**, **`long`** kommen in zwei Varianten **`unsigned`** und **`signed`**, die dem Typ vorangestellt werden. Lässt man diese Angabe weg, so ist sie implizit **`signed`**. Ein **`signed int`** entspricht einer vorzeichenbehafteten Ganzzahl, ein **`unsigned int`** einer vorzeichenlosen. Bei **`signed`**-Typen macht der C-Standard [ISO99] wenig Vorschriften über die zu verwendende Arithmetik. Insbesondere kann man nicht davon ausgehen, dass **`signed`**-Werte im Zweierkomplement dargestellt sind und die entsprechende Arithmetik zum Einsatz kommt. Bei **`unsigned`**-Zahlen kommt Moduloarithmetik zum Einsatz. Des Weiteren macht der Standard keine Aussage über die Größe der Ganzzahl-Typen. Er garantiert lediglich:

$$1 = \texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

wobei **`sizeof`**(*t*) ein Operator ist, der die Größe eines Typs liefert. Diese schwache Spezifikation ist Absicht, da sie es erlaubt, C auf vielen unterschiedlichen Rechnern korrekt zu implementieren. Sie ist aber auch Quelle vieler Software-Fehler, da viele unwissende Programmierer zu starke Annahmen über die Semantik von C treffen. Für normale Berechnungen verwendet man immer **`int`**, da ein **`int`** meist einem Maschinenwort entspricht.

Die Gleitkomma-Typen finden in numerischen Programmen Verwendung. Sie stellen einen Ausschnitt der rationalen Zahlen dar. Da man nur endlich viel Bits zur Darstellung eines Wertes hat, kommt es beim Rechnen mit Gleitkomma-Werten im Allgemeinen zu Rundungsfehlern, die sich unbemerkt fortpflanzen und schwer zu lokalisieren sind. Insbesondere erwecken Gleitkomma-Zahlen beim Programmierer leicht den Eindruck, man könne beliebig genau mit reellen Zahlen in einem Rechner rechnen; viele Programmierer ignorieren die Gesetzte der Gleitkomma-Arithmetik von Gleitkomma-Zahlen völlig und verwenden sie, als wären sie reelle Zahlen. In C gibt es drei Gleitkomma-Typen: **`float`**, **`double`**, **`long double`** die sich ihrer Größe und somit ihrer der Genauigkeit unterscheiden.

3.6 Ablaufsteuerung

C besitzt mehrere Konstrukte zur Ablaufsteuerung. Diese bestimmtem den Ablauf des Programms, das heißt, sie entscheiden, welche Anweisung als nächstes ausgeführt wird.

3.6.1 Fallunterscheidung

Die Fallunterscheidung **if** (*b*) *A* **else** *B* wertet die Bedingung *b* aus. Ist ihr Wert ungleich 0, wird die Anweisung *A* ausgeführt, ansonsten die Anweisung *B*.

```
void print_max(int a, int b) {
    if (a < b)
        printf("%d\n", b);
    else
        printf("%d\n", a);
}
```

Ein weiteres Konstrukt zur Fallunterscheidung ist **switch**. **switch** wertet einen Ausdruck vom Typ **int** aus. Nun kann man für verschiedene Ergebnisse der Ausdrucksauswertung Marken angeben, an die dann gesprungen wird:

```
enum { jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

unsigned days_per_month(unsigned month, bool is_leap_year) {
    assert (month < 12);
    switch (month) {
    case jan:
    case mar:
    case may:
    case jul:
    case aug:
    case oct:
    case dec:
        return 31;
    case feb:
        return 28 + is_leap_year;
    default:
        return 30;
    }
}
```

Passt der Wert zu keiner der angegebenen Marken, so wird die Marke **default** angesprungen. Es ist zu beachten, dass nach dem Sprung zur Marke **alle** Anweisungen bis zum Ende des **switch** Blocks abgearbeitet werden. Ein **switch** Block kann vorzeitig nur durch **break** oder **return** verlassen werden.

3.6.2 Schleifen

C verfügt über drei Arten von Schleifen. Die offene Schleife **do** *A* **while** (*b*) führt die Anweisung *A* aus und prüft danach anhand der Bedingung *b*

ob die Schleife nochmals durchlaufen werden soll. Sie heißt offene Schleife, weil A auf jeden Fall einmal ausgeführt wird:

```
int sum(void) {
    int sum = 0;
    int n;
    do {
        printf("Enter a number (0 = finish): ");
        scanf("%d", &n);
        sum += n;
    } while (n != 0);
    printf("The sum is %d\n", sum);
}
```

Die geschlossene Schleife **`while`** (b) A führt A wiederholt aus, solange die Bedingung b erfüllt ist. Sie heißt geschlossen, weil vor dem Eintritt in die Schleife geprüft wird, ob b erfüllt ist. Eventuell wird A also gar nicht ausgeführt.

```
int sum(int *a, unsigned n) {
    int res = 0;
    unsigned i = 0;
    while (i < n) {
        res += a[i];
        i = i + 1;
    }
    return res;
}
```

Die for-Schleife ist eine Abkürzung der while-Schleife um Zählvariablen kompakt vereinbaren zu können:

```
int sum(int *a, unsigned n) {
    int res = 0;
    for (unsigned i = 0; i < n; i++)
        res += a[i];
    return res;
}
```

Mit der Anweisung **`break`** kann die innerste Schleife, die sie umgibt, verlassen werden:

```
void sum(void) {
    int sum = 0;
    for (;;) {  // is equal to while (true)
        int n;
        printf("Enter a number (0 = finish): ");
        scanf("%d", &n);
        if (n == 0)
            break;
        sum += n;
    }
    printf("The sum is %d\n", sum);
}
```

Die Anweisung **`continue`** springt unmittelbar zur Inkrement-Anweisung der **`for`**-Schleife.

```
void cycle(unsigned mod) {
    for (unsigned i = 0; ; i++) {
        if (i == mod) {
            i = 0;
            continue;
        }
        printf("%d\n", i);
    }
}
```

Die Anweisung **`return`** kehrt unmittelbar zum Aufrufer zurück. Hat das Unterprogramm einen Rückgabewert (Rückgabetyp ungleich **`void`**), so erwartet **`return`** einen Parameter:

```
int max(int a, int b) {
    if (a < b)
        return b;
    else
        return a;
}
```

3.7 Ausdrücke

Berechnungen werden in C durch Ausdrücke dargestellt. In C gibt es eine Reihe von binären und unären arithmetischen Operatoren. Diese sind teilweise überladen, sprich, das gleiche Symbol wird für verschiedene Typen der Operanden verwendet. So ist die Addition zweier Gleitkommazahlen eine andere Operation als die Addition zweier Ganzzahlen.[16] Der Compiler verwendet die Typen der Operanden um den entsprechenden Operator zu identifizieren und die Überladung somit aufzulösen.

Für alle Typen $i \in \{\texttt{int}, \texttt{long}\}$ und $f \in i \cup \{\texttt{float}, \texttt{double}\}$ hat C folgende Arithmetik-Operatoren:

Symbol	Signatur	Beschreibung
`~`	$i \to i$	Bitweises Negieren
`!`	$i \to \texttt{int}$	Logisches Nicht
`+ -`	$f \to f$	Unäres Plus, Minus
`+ - * /`	$f \times f \to f$	Plus, Minus, Mal, Geteilt
`%`	$i \times i \to i$	Rest bei Division
`& | ^ << >>`	$i \times i \to i$	Und, Oder, Xor, Shifts
`== != <= < >= >`	$f \times f \to \texttt{int}$	Vergleiche

[16] Beim Übersetzen in Maschinencode muss der Übersetzer für jeden Operator einen Maschinenbefehl auswählen. Der Maschinenbefehl für die Addition zweier Gleitkommazahlen ist ein anderer als der für die Addition der Ganzzahlen. Um statisch, also zur Übersetzungszeit des Programms, zu entscheiden, welche Operation ausgewählt werden muss, benötigt der Compiler die Typen der Operanden des Ausdrucks. Bei Sprachen, die Typen nicht statisch ermitteln (z.B. JavaScript), wird die Entscheidung, welche Operation ausgewählt werden muss, zur Laufzeit des Programms getroffen, was zu langsamer laufenden Programmen führt.

Abbildung 3.6: Arithmetik-Operatoren in C

3.7.1 Automatische Typanpassung

Um die Tipparbeit für den Programmierer zu verringern, werden die Operanden vor der Anwendung des Operators automatisch angepasst. Betrachten wir folgendes Beispiel:

```
int x;
double d;
...
d = d + x;
```

Der Additionsoperator ist nicht auf einem **`double`** und einem **`int`** definiert (siehe Abbildung 3.6). Gemäß folgender Ordnung wird aber ein **`int`** zunächst in einen **`double`** umgewandelt[17]. Die Anpassung findet immer an den kleinsten gemeinsamen Typ aller Operanden statt, wobei a „kleiner“ $b \iff a \to^* b$. Es wird jedoch prinzipiell nicht „kleiner“ als **`int`** gerechnet. Dies rührt daher, dass ein **`int`** meist ein Maschinenwort ist.

[17] Man beachte, dass hierdurch Code entsteht, da die Bitfolge, die den **`int`** darstellt, in eine Bitfolge umgewandelt werden muss, die, als **`double`** interpretiert, den gleichen Zahlwert ergibt. Für manche Typen (z.B. 64-Bit long nach 64-Bit double) ist diese Konversion aufgrund unzureichender Genauigkeit nicht exakt.

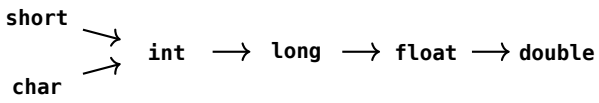


Abbildung 3.7: Automatische Typanpassung der Operanden bei Arithmetik-Operatoren. Alle Operanden werden an den kleinsten gemeinsamen Typ angepasst.

3.7.2 L- und R-Auswertung von Ausdrücken

Ausdrücke werden in C (und allen abgeleiteten imperativen Sprachen) auf zwei Arten ausgewertet, abhängig davon, wo der Ausdruck steht. Betrachten wir die Zuweisung

```
x = x + 1;
```

In C sind Zuweisungen auch Ausdrücke, sprich = ist ein binärer Operator. Der Operator hat die Nebenwirkung (siehe Abbildung 3.9), dass er den Wert, der sich durch die Auswertung des Ausdrucks auf der rechten Seite ergibt, in den Behälter schreibt, dessen Adresse sich durch die Auswertung des Ausdrucks auf der linken Seite ergibt.

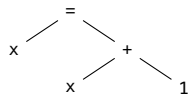


Abbildung 3.8: Ausdrucksbaum zu `x = x + 1`

Der linke Operand von = muss also zu einer Adresse, der rechte zu einem Wert auswerten. Nun taucht auf beiden Seiten der Teilausdruck x auf, der nur aus der Nennung eines Variablennamens besteht. Auf der rechten Seite wird der (Teil-)Ausdruck x dadurch ausgewertet, dass der Inhalt des Behälters für den x steht, gelesen wird. Auswerten zum Behälterinhalt nennt man **R-Auswertung**. Im Folgenden wird dann 1 auf diesen Wert aufaddiert, wodurch der Wert des Ausdrucks auf der rechten Seite zustande kommt. Auf der linken Seite wird der Ausdruck x zur **Adresse** des Behälters ausgewertet. Wir sprechen hier von **L-Auswertung**, da sie auf der **L**inken Seite der Zuweisung stattfindet. Das Auftreten eines Ausdrucks auf der linken oder rechten Seite des Zuweisungsoperators bewirkt also eine unterschiedliche Art der Auswertung.

Jeder Ausdruck kann R-ausgewertet werden, aber nicht jeder Ausdruck kann L-ausgewertet werden. Nur Ausdrücke, die sich zur Adresse eines Behälters auswerten lassen, können L-auswertbar sein. Der Ausdruck `1 + 2` lässt sich beispielsweise nicht L-auswerten. Der Ausdruck x, wobei x eine vereinbarte Variable ist, kann L-ausgewertet werden.

Die Eigenschaft „L-auswertbar“ ist statisch überprüfbar. Der Übersetzer bricht den Übersetzungsvorgang mit einer Fehlermeldung ab, wenn ein nicht L-auswertbarer Ausdruck an einer nicht zulässigen Stelle (beispielsweise auf der linken Seite der Zuweisung) steht.

Ist ein Ausdruck *e* L-auswertbar, so gilt die Regel, dass eine R-Auswertung von *e* den Inhalt des Behälters ergibt, dessen Adresse sich aus der L-Auswertung von *e* ergibt:

$$\text{R-Ausw}(e) = \text{Inhalt}(\text{L-Ausw}(e)) \tag{3.1}$$

3.7.3 Nebenwirkungen

Eine besondere Eigenheit von C ist, dass die Auswertung eines Ausdrucks Nebenwirkungen (engl. side effects) haben kann. Das heißt, dass während der Auswertung des Ausdrucks der Inhalt von Behältern geändert werden, oder Kommunikation mit dem Betriebssystem[18] stattfinden, oder gar das Programm divergieren (in eine Endlosschleife geraten) kann. Abbildung 3.9 zeigt alle C-Operatoren, die Nebenwirkungen haben.
Die Zuweisung ist ein Ausdruck mit Nebenwirkungen:

[18] beispielsweise Ausgabe von Text auf der Konsole, Verschicken von Netzwerkpaketen, Lesen und Schreiben von Dateien, usw.

```
x = x + 1
```

Die Nebenwirkung ist, dass der Wert, der sich durch die Auswertung der rechten Seite ergibt, in den Behälter, dessen Adresse sich aus dem linken Ausdruck ergibt, geschrieben wird. Der Wert des Ausdrucks = ist der geschriebene Wert.

Ausdruck	Name	Wert des Ausdrucks	Neuer Wert von e / Nebenwirkungen
$++e$	Präinkrement	Neuer Wert von e	Schreiben des Wertes von $e+1$ in e
$--e$	Prädekrement	Neuer Wert von e	Schreiben des Wertes von $e-1$ in e
$e++$	Postinkrement	Alter Wert von e	Schreiben des Wertes von $e+1$ in e
$e--$	Postdekrement	Alter Wert von e	Schreiben des Wertes von $e-1$ in e
$e = e'$	Zuweisung	Neuer Wert von e	Schreiben des Wertes von e' in e
$e \oplus= e'$	Zuweisung	Neuer Wert von e	Schreiben des Wertes $e \oplus e'$ in e
$f(...)$	Unterprogrammaufruf	Rückgabewert des UP	Nebenwirkungen des UP

Abbildung 3.9: Operatoren mit Nebenwirkungen. $\oplus$ steht für einen beliebigen binären Operator (siehe Abbildung 3.6). Selbstverständlich muss der Ausdruck e L-auswertbar sein.

Besitzt ein Ausdruck mehrere Nebenwirkungen, so könnte eine unterschiedliche Reihenfolge der Auswertung der Teilausdrücke zu unterschiedlichen Ergebnissen führen. Betrachten wir folgendes Beispiel:

```
x = 5;
y = (x = 2) * (++x);
```

Es gibt nun mehrere Möglichkeiten, die Semantik dieses Programms zu definieren:

1. Man gibt C eine nichtdeterministische Semantik, was bedeuten würde, dass sowohl `2` als auch `3` korrekte Werte für den Behälterinhalt von `x` sind.

2. Man definiert eine Ordnung, in der die Teilausdrücke auszuwerten sind (dies ist in der Sprache Java der Fall).

3. Man gibt solchen Programmen überhaupt keine Semantik (dies ist in C der Fall). Die genaue Regel, wann ein Ausdruck mit Nebenwirkungen eine Bedeutung hat, ist kompliziert[19]. Als Faustregel kann man sich merken: Wird ein Behälter während der Auswertung eines Ausdrucks zum zweiten mal beschrieben, so ist das Verhalten nicht mehr definiert. Eine Ausnahme bilden Nebenwirkungen in gerufenen Unterprogrammen:

   ```
   int a(void) { printf("A"); return 1; }
   int b(void) { printf("B"); return 2; }
   int c(void) { return a() + b(); }
   ```

   Ein Aufruf von `c` liefert den Wert `3`. Jedoch sind, sowohl `AB` als auch `BA` gültige Ausgaben des Programms. Da die wenigsten Programmierer diese Regel kennen und Programme wie das obige daher schwer verständlich sind, ist es am besten, Nebenwirkungen auf ein Minimum zu reduzieren: Höchstens eine pro Ausdruck.

[19] siehe Abschnitt J.2, Seite 491 und Abschnitt 6.5 im C-Standard

3.7.4 Faule Auswertung

C hat drei Operatoren, deren Operanden nicht strikt sondern faul ausgewertet werden. Ein Operator wird **strikt** ausgewertet, wenn zuerst **alle** Operanden ausgewertet werden, bevor der Operator ausgewertet wird. Wird er faul ausgewertet, so wird ein Operand nur dann auswertet, wenn er zum Ergebnis beiträgt.[20]

[20] In der Gegenwart von Nebenwirkungen ist es erheblich, ob ein Operator faule oder strikte Auswertung bewirkt: Wird ein Operand aufgrund fauler Auswertung nicht ausgewertet, so treten seine Nebenwirkungen nicht ein.

Operator	Signatur	Verhalten
e ? t : f	**int** $\times T \times T \to T$	Wenn e zu 0 ausgewertet wird, dann werte f aus, ansonsten t
e && e'	**int** $\times$ **int** $\to$ **int**	(e == 0) ? 0 : (e' != 0)
e || e'	**int** $\times$ **int** $\to$ **int**	(e != 0) ? 1 : (e' != 0)

Abbildung 3.10: C-Operatoren mit fauler Auswertung

3.8 Aufrufen von Unterprogrammen

Der Aufruf eines Unterprogramms ist auch ein Ausdruck (siehe oben). Betrachten wir nochmal unser einführendes Beispiel, insbesondere den Aufruf von `fac` in `main`:

```
unsigned fac(unsigned n)
{
    unsigned r = 1;
    /* ... */
    return r;
}

int main(int argc, char** argv)
{
    unsigned r, n;
    /* ... */
    r = fac(n);
    /* ... */
}
```

Wie genau geschieht die Übergabe der Parameter an `fac`? Beim Aufruf eines Unterprogramms wird für jeden Parameter ein Behälter entsprechender Größe erstellt.[21] So gesehen, sind die Parameter zusätzliche lokale Variablen des (gerufenen) Unterprogramms. Ihr Sichtbarkeitsbereich erstreckt sich über das gesamte Unterprogramm. Ihre Lebensdauer endet mit dem Verlassen des Unterprogramms. Bevor die Ausführung bei der ersten Anweisung des Unterprogramms fortgesetzt wird, werden alle (durch Kommata getrennte) **Argumente**[22] R-ausgewertet und die Werte in die entsprechenden, gerade erstellten, Behälter geschrieben. Die Behälter der Parameter werden mit dem Verlassen des Unterprogramms wieder zerstört. Dieses Vorgehen nennt man **Wertaufruf** (engl. call by value), da die Werte der Argumente übergeben werden. Andere Programmiersprachen (z.B. Pascal oder C#) bieten zusätzlich den **Referenzaufruf** (engl. call by reference), bei dem man die Adresse eines Behälters übergeben kann. Das entsprechende Argument muss dann L-auswertbar sein. In C kann man den Referenzaufruf dadurch simulieren, dass man einen Zeiger übergibt (siehe nächster Abschnitt).

Ein Unterprogramm wird durch Ausführen der **`return`**-Anweisung beendet. Hat das Unterprogramm den Rückgabe-Typ **`void`**, so kann auf **`return`** verzichtet werden und das Unterprogramm kehrt nach der Ausführung der letzten Anweisung in seinem Rumpf zu seinem Aufrufer zurück. Hat das Unterprogramm einen anderen Rückgabe-Typ, so muss das Unterprogramm mittels **`return`** e beendet werden, wobei e ein Ausdruck ist, dessen Typ gleich dem Rückgabetyp des Unterprogramms ist.

[21] Der Prototyp des Unterprogramms ordnet jedem Parameter einen Typen zu. Da der Prototyp beim Aufruf bekannt sein muss, kann die Größe des Behälters ermittelt werden.
[22] so nennt man die Ausdrücke, deren Werte dem Unterprogramm übergeben werden.

3.9 Zeiger

Bislang haben wir nur Basistypen kennen gelernt. Im Behälter einer Basistyp-Variable kann zu einem Zeitpunkt exakt ein Wert des Typs enthalten sein. Möchte man größere Datenmengen verarbeiten, beispielsweise eine Liste von n Zahlen, benötigt man zusammengesetzte Datentypen (engl. compound types). Der Wert eines zusammengesetzten Datentyps besteht aus mehreren (Unter-)Werten anderer (potentiell zusammengesetzter) Typen. C besitzt zwei Arten[23] zusammengesetzter Datentypen: Die Reihung (engl. array) und den Verbund (engl. struct, record), den wir im nächsten Absatz besprechen.

Eine Reihung der Länge n vom Typ T kann n Werte des Typs T aufnehmen.

```
int numbers[100000];
/* ... */
int m = max(numbers);
```

vereinbart eine Reihung von 100000 Elementen des Typs **`int`** und legt **einen** Behälter an, der 100000 Werte des Typs **`int`** aufnehmen kann.

Eine Besonderheit von C-Reihungen ist, dass eine R-Auswertung einer Reihung `a` des Typs T`[`n`]` nicht einen Wert dieses Typs liefert, sondern die **Anfangsadresse**, einen Zeiger auf das erste Element von `a` des Typs $T*$[24].

Daher vereinbart ein Unterprogramm, dem eine Reihung von Elementen des Typs T übergeben werden soll, den entsprechenden Parameter als **Zeiger** auf T. Da in der Reihung nur der Inhalt steht und nicht deren Länge[25], muss diese auch noch mit übergeben werden, wenn das Unterprogramm nicht von einer konkreten Länge abhängig sein soll:

```
int max(int* numbers, int len)
{
    int m = 0;
    for (int i = 0; i < len; i++)
        m = numbers[i] > m ? numbers[i] : m;
    return m;
}
```

Auf die einzelnen Elemente einer Reihung kann man mittels Adressarithmetik relativ zur Anfangsadresse zugreifen. Hierzu verwendet man den Indizierungsoperator e`[`i`]`, wie in obigem Beispiel illustriert. e`[`i`]` ist nichts weiter als „syntaktischer Zucker“ für `*(`e`+`i`)`. Ist e des Typs $T*$, so kann man auf e einen Ausdruck i vom Typ **`int`** addieren und dadurch wieder einen Zeiger des Typs $T*$ konstruieren. Somit kann man in einem Behälter, der Platz für mehrere Ts hat, einzelnen Werte adressieren. Man beachte, dass über Adressarithmetik unterschiedliche Zeiger in den gleichen Behälter entstehen können. Leider können durch Adressarithmetik auch leicht ungültige Zeiger konstruiert werden: Beispielsweise sind in obigem Code die Adressen `numbers + 100042` und `numbers - 1` ungültig, da der jeweilige Versatz (100042 und -1) jenseits der durch die Größe (100000) des Behälters gegebenen Grenzen [0; 100000[liegt. Die Bildung

[23] Streng genommen gibt es drei: Den Vereinigungstyp **`union`** besprechen wir hier der Übersicht halber nicht.

[24] Die einzige Ausnahme ist das Auftreten einer Variable eines Reihungstyps in **`sizeof`**. So liefert, unter der Annahme, dass **`sizeof`**`(int) == 4`, **`sizeof`**`(numbers)` in obigem Beispiel den Wert 400000 und nicht **`sizeof`**`(int*)`.

[25] Ein häufiger Anfängerfehler ist, zu glauben, dass man mit **`sizeof`**`(numbers)` die Länge der Reihung ermitteln kann, wenn `numbers` ein **Zeiger** auf deren erstes Element ist. **`sizeof`** wird jedoch zur Übersetzungszeit ausgewertet und liefert nur die Größe des Typs des Ausdrucks; in diesem Fall die Größe eines Zeigers.

solcher „out-of-bounds“ Adressen führt zu undefiniertem Verhalten (→ Abschnitt 3.14).[26]

Als **Zeiger** bezeichnet man eine Variable des Typs $T*$. Der Wert eines Zeigers ist eine Adresse. Sprich, der Inhalt des Behälters eines Zeigers ist die Adresse eines anderen Behälters. Nicht zu verwechseln ist die Adresse des Zeigers mit dem Wert des Zeigers, der ja auch eine Adresse ist.

Operator	Name	Auswertungsregeln	L-auswertbar	Operand ist vom Typ	Ausdruck ist vom Typ
$*$	Indirektion	L-Ausw($*e$) ≔ R-Ausw(e)	ja	$T*$	T
&	Adresse-von	R-Ausw(&e) ≔ L-Ausw(e)	nein	T	$T*$

Abbildung 3.11: Übersicht über die Eigenschaften der Zeiger-Operatoren

Zwei Operatoren sind im Umgang mit Zeigern relevant: Der Adresse-von-Operator & und der Indirektionsoperator $*$. Kurz gesprochen liefert die R-Auswertung &e die Adresse des L-auswertbaren Ausdrucks e. $*e$ hingegen wandelt e in einen L-auswertbaren Ausdruck um. Hierzu muss e zu einer Adresse R-auswerten. Somit ist mit $*e$ der Inhalt des Behälters mit der Adresse e gemeint ist. Abbildung 3.11 fasst die Bedeutung der beiden Operatoren zusammen.

Das Programm in Abbildung 3.12 zeigt die Verwendung der Operatoren $*$ und &. Betrachten wir die Effekte der einzelnen Anweisung im Detail:

1. In den Behälter von x wird der Wert 1 geschrieben.

2. Der Behälter von y kann Adressen von **int**-Behältern aufnehmen. Der Ausdruck x ist L-auswertbar: Er wird L-ausgewertet, was die Adresse des Behälters von x liefert. Diese wird durch die R-Auswertung von & zu einem Wert. Dieser wird in den Behälter von y geschrieben. Nota bene: Stünde das & nicht vor x, so würde der Übersetzer einen Typfehler melden, denn auf der rechten Seite einer Zuweisung werden Ausdrücke R-ausgewertet, und die R-Auswertung von x liest den Wert aus dem Behälter x und liefert **nicht** dessen Adresse.

3. Die R-Auswertung des Ausdrucks 2 gibt den Wert 2. Spannender ist die linke Seite der Zuweisung: Der Ausdruck $*$y ist nach Definition L-auswertbar; andernfalls dürfte er nicht auf der linken Seite einer Zuweisung stehen. Sein Wert ist das Ergebnis der R-Auswertung seines Operanden. Das Ergebnis ist der Inhalt des Behälters von y: die Adresse von x. In diesen Behälter, wird dann der Wert 2 geschrieben.

4. Die L-Auswertung von z liefert die Adresse des Behälters von z. Die R-Auswertung von y liefert den Inhalt des Behälters von y, die Adresse von x. Also wird die Adresse von x in den Behälter von z geschrieben.

5. Da z die Adresse von x beinhaltet, wird der Inhalt des Behälters von x nun auf 3 gesetzt.

6. Interessant ist das Auftreten des $*$-Operators auf der rechten Seite: Die R-Auswertung von y liefert den Inhalt des Behälters von y,

Variablendeklarationen:

```
int w;
int x;
int *y;
int *z;
```

Ausführungsprotokoll:

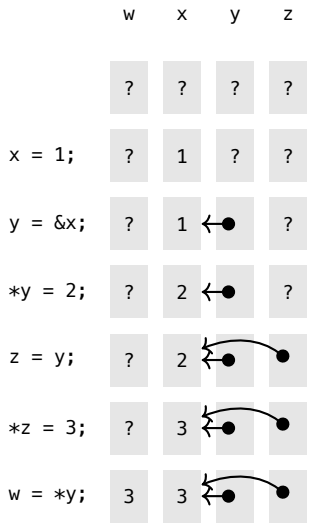


Abbildung 3.12: Ausführungsprotokoll eines C-Programms mit Zeigern. Jede Zeile entspricht der Ausführung der Anweisung links. Rechts ist der Programmzustand in Form der Variableninhalte dargestellt.

[26] Eine Ausnahme bildet die Adresse p+s wobei s die Größe des Behälters ist. Diese Adresse lässt sich ohne in undefiniertes Verhalten zu geraten, bilden. Jedoch sind Zugriffe mit dieser Adresse undefiniert. Hintergrund dieser Ausnahme ist, dass man häufig Adressarithmetik verwendet, um über Reihungen zu iterieren und es hat sich eingebürgert, diese Adresse in der Abbruchbedingung zu verwenden.

also die Adresse von x. Der Ausdruck ∗y ist L-auswertbar. Die R-Auswertung eines L-auswertbaren Ausdrucks liest den Inhalt des Behälters, dessen Adresse die L-Auswertung liefert. Hier also der Inhalt des Behälters von x, die 3. In w steht also dann der Wert 3.

3.9.1 Arrays

In einem Behälter einer Basistyp-Variable kann zu einem Zeitpunkt exakt ein Wert des Typs enthalten sein. Möchte man größere Datenmengen verarbeiten, beispielsweise eine Liste von n Zahlen, benötigt man zusammengesetzte Datentypen (engl. compound types). Der Wert eines zusammengesetzten Datentyps besteht aus mehreren (Unter-)Werten anderer (potentiell zusammengesetzter) Typen. C besitzt zwei Arten[27] zusammengesetzter Datentypen: Die Reihung (engl. array) und den Verbund (engl. struct, record), den wir im nächsten Absatz besprechen.

Eine Reihung der Länge n vom Typ T kann n Werte des Typs T aufnehmen.

```
int numbers[100000];
/* ... */
int m = max(numbers);
```

vereinbart eine Reihung von 100000 Elementen des Typs **int** und legt **einen** Behälter an, der 100000 Werte des Typs **int** aufnehmen kann.

Eine Besonderheit von C-Reihungen ist, dass eine R-Auswertung einer Reihung a des Typs T[n] nicht einen Wert dieses Typs liefert, sondern die **Anfangsadresse**, einen Zeiger auf das erste Element von a des Typs T∗[28].

Daher vereinbart ein Unterprogramm, dem eine Reihung von Elementen des Typs T übergeben werden soll, den entsprechenden Parameter als **Zeiger** auf T. Da in der Reihung nur der Inhalt steht und nicht deren Länge[29], muss diese auch noch mit übergeben werden, wenn das Unterprogramm nicht von einer konkreten Länge abhängig sein soll:

```
int max(int* numbers, int len)
{
    int m = 0;
    for (int i = 0; i < len; i++)
        m = numbers[i] > m ? numbers[i] : m;
    return m;
}
```

Auf die einzelnen Elemente einer Reihung kann man mittels Adressarithmetik relativ zur Anfangsadresse zugreifen. Hierzu verwendet man den Indizierungsoperator e[i], wie in obigem Beispiel illustriert. e[i] ist nichts weiter als „syntaktischer Zucker“ für ∗(e+i). Ist e des Typs T∗, so kann man auf e einen Ausdruck i vom Typ **int** addieren und dadurch wieder einen Zeiger des Typs T∗ konstruieren. Somit kann man in einem Behälter, der Platz für mehrere Ts hat, einzelnen Werte adressieren. Man beachte, dass über Adressarithmetik unterschiedliche Zeiger in den gleichen Behälter entstehen können. Leider können durch Adressarithmetik auch leicht ungültige Zeiger konstruiert werden: Beispielsweise sind in obigem Code die Adressen `numbers + 100000` und `numbers - 1` ungül-

[27] Streng genommen gibt es drei: Den Vereinigungstyp **union** besprechen wir hier der Übersicht halber nicht.

[28] Die einzige Ausnahme ist das Auftreten einer Variable eines Reihungstyps in **sizeof**. So liefert, unter der Annahme, dass **sizeof**(**int**) == 4, **sizeof**(numbers) in obigem Beispiel den Wert 400000 und nicht **sizeof**(**int**∗).

[29] Ein häufiger Anfängerfehler ist, zu glauben, dass man mit **sizeof**(numbers) die Länge der Reihung ermitteln kann, wenn numbers ein **Zeiger** auf deren erstes Element ist. **sizeof** wird jedoch zur Übersetzungszeit ausgewertet und liefert nur die Größe des Typs des Ausdrucks; in diesem Fall die Größe eines Zeigers.

tig, da der jeweilige Versatz (100000 und -1) jenseits der durch die Größe (100000) des Behälters gegebenen Grenzen [0; 100000[liegt. Die Bildung solcher „out-of-bounds“ Adressen führt zu undefiniertem Verhalten (→ Abschnitt 3.14).

3.9.2 const

C erlaubt es, manche Typen durch sog. **Qualifizierer (engl. qualifier)** genauer zu spezifizieren. Ein vor allem mit Zeigern häufig verwendeter Qualifizierer ist **`const`**. Der Behälter einer Variable eines **`const`**-Typs kann nicht überschrieben werden. Zum Beispiel:

```
int const x = 2;
x = 4; /* Verboten! */
```

Bei Zeigern gibt es mehrere Varianten, was genau **`const`** sein kann:

```
int w = 0;
int x = 1;
int const* p = &x;        /* Zeiger auf einen const int */
w  = *p;                  /* ok */
*p = w;                   /* nicht ok */
p  = &w;                  /* ok, da der Zeiger nicht const ist */
int * const q = &x;       /* const Zeiger auf einen int */
*q = 1;                   /* ok, da Zeiger const, aber nicht sein Ziel */
q  = &w;                  /* nicht ok, da Zeiger const */
int const *const r = &x; /* const Zeiger auf const int */
*r = 1;                   /* nicht ok */
r  = &w;                  /* nicht ok */
```

Große Daten übergibt man meist durch einen Zeiger. Häufig lesen Unterprogramme die Daten nur. Durch die Verwendung von **`const`** im Prototypen kann man dies dokumentieren. Der Übersetzer untersagt dann ein Schreiben in den Behälter, der **`const`** übergeben wurde.

Nun können wir unser Beispiel abschließen und die Implementierung des Unterprogramms `max` angeben:

```
int max(int const* table, unsigned len)
{
    assert(len > 0 && "cannot take the maximum of an empty array");
    int max = table[0];
    for (unsigned i = 1; i < len; i++) {
        int v = table[i];
        max = v > max ? v : max;
    }
    return max;
}
```

3.10 Beispiele

Jetzt können wir die Beispiele aus dem MIPS Kapitel in C implementieren.

Beispiel 3.5 (Zahlenkonversion)**.**

```
#include <stdio.h>

void print_hex(unsigned word)
{
    unsigned shift = 8 * sizeof(word);
    do {
        shift -= 4;
        unsigned digit = (word >> shift) & 0x0f;
        unsigned ascii = digit + (digit < 10 ? '0' : 'a' - 10);
        putchar(ascii);
    } while (shift != 0);
}
```

⌟

Beispiel 3.6 (Das Sieb des Eratosthenes)**.**

```
#include <string.h>

char* eratosthenes(char* table, unsigned n) {
    memset(table, 0, n);
    for (unsigned q = 2; q * q < n; q++) {
        if (table[q] == 0) {
            for (unsigned j = q * q; j < n; j += q)
                table[j] = 1;
        }
    }
    return table;
}
```

⌟

Beispiel 3.7 (Sortieren durch Einfügen)**.**

```
#include <assert.h>

static unsigned find(int* arr, unsigned n, int value)
{
    unsigned i = 0;
    while (i < n && arr[i] < value)
        i++;
    return i;
}

static void insert(int* arr, unsigned n, unsigned from, unsigned to)
{
    assert(to <= from);
```

```
    int val = arr[from];
    for (unsigned i = to; i <= from; i++) {
        int tmp = arr[i];
        arr[i]  = val;
        val     = tmp;
    }
}

void insertsort(int* arr, unsigned n)
{
    for (unsigned i = 1; i < n; i++) {
        unsigned pos = find(arr, n, arr[i]);
        insert(arr, n, i, pos);
    }
}
```

⌟

3.11 Dynamische Speicherverwaltung

Die Lebenszeit einer lokalen Variablen ist an die Ausführung des Blocks gebunden, der sie umgibt. Betrachten wir folgendes Beispiel:

```
int *numbers(unsigned n) {
    int a[n];
    for (unsigned i = 0; i < n; i++)
        a[i] = i;
    return a;
}
```

Hier wird offenbar versucht, eine Reihung zurückzugeben, die die ersten *n* Zahlen enthält. Allerdings ist die Lebenszeit von `a` auf die Ausführungsdauer des Rumpfes von `numbers` beschränkt, da dies der innerste Block ist, der ihre Vereinbarung enthält. Bevor das Unterprogramm also zum Aufrufer zurückgekehrt ist, wird der Behälter von `a` freigegeben und der entsprechende Zeiger auf das erste Element der Reihung **baumelt** (engl. dangling pointer): Er zeigt auf keinen existierenden Behälter. Das Programm ist also inkorrekt. Es gibt zwei Möglichkeiten, dieses Problem zu lösen:

1. Man übergibt `numbers` die Adresse einer Reihung, in der die Zahlen abgelegt werden sollen. Somit ist `numbers` nicht mit dem Erstellen des Behälters für die Reihung befasst. Das ist meist, die bessere, da flexiblere Lösung. Die Reihung könnte dann als lokale Variable im Aufrufer vereinbart sein.

2. Das Unterprogramm `numbers` kann den Behälter für die Reihung selbst anfordern. Da eine lokale Variable nicht in Frage kommt, kommt nur ein Behälter auf der Halde (engl. heap) in Frage. Diese werden mit `malloc` angefordert.

Möchte man Behälter, die länger leben, als der sie umschließende Block ausgeführt wird, muss man sie durch die Funktion

```
void* malloc(size_t n)
```

anfordern. Der Parameter `n` gibt die Größe des Behälters in Bytes an. Man verwendet häufig den **`sizeof`**-Operator[30], um die Größe des Behälters in Abhängigkeit des Typs des gewünschten Inhalts zu berechnen. Folgende Anweisung legt beispielsweise eine Reihung von `n` Elementen des Typs `T` an[31]:

```
T *a = malloc(n * sizeof(a[0])); // sizeof(a[0]) == sizeof(T)
```

Der Behälter lebt so lange, bis der von `malloc` zurückgegebene Zeiger durch die Funktion

```
void free(void* b)
```

wieder freigegeben wird.[32] Im Gegensatz zu den Behältern lokaler Variablen, wird ein mit `malloc` angeforderter Behälter nicht zerstört, wenn der Block, in dem der Behälter angefordert wurde, verlassen wird.

[30] **`sizeof`**`(x)` ist ein Operator und kein Unterprogramm. Der Operand `x` kann ein Typ aber auch ein Ausdruck sein. Ist es ein Ausdruck, berechnet der **Übersetzer** den Typ des Ausdrucks und bestimmt dessen Größe. Man beachte, dass **`sizeof`** nicht zur Laufzeit ausgewertet wird und man damit **nicht** die Größe eines Behälters, sondern eines Typs ermittelt.

[31] Fordert man mit `malloc` eine Reihung an, auf die man mit einem Zeiger `a` des Typs *T* verweisen will, so schreibt man besser **`sizeof`**`(a[0])` statt **`sizeof`**`(`*T*`)`. Ändert man irgendwann den Typ von `a`, so muss er in **`sizeof`** nicht mitgeändert werden.

[32] „double frees“, das Freigeben von bereits freigegebenen Behältern sind beliebte Programmierfehler, die nicht immer einfach zu entdecken sind.

Beispiel 3.8 (Eratosthenes)**.** Betrachten wir nun ein Hauptprogramm, das die Größe der Tabelle für das Sieb das Eratosthenes aus dem Kommandozeilen-Parameter ermittelt und Speicher für die Tabelle anfordert und freigibt.

```
int main(int argc, char** argv) {
    if (argc < 2) {
        printf("syntax: %s <n>\n", argv[0]);
        return 1;
    }
    unsigned n = atoi(argv[1]);
    char *table = malloc(n * sizeof(table[0]));
    eratosthenes(table, n);
    print_table(table, n);
    free(table)
    return 0;
}
```

⌟

3.11.1 realloc

Oft verwendet man Reihungen, um Listen zu implementieren. Das ist insbesondere sinnvoll, wenn man schnellen Zugriff auf das i-te Element benötigt: Bei einer Reihung kann man die Adresse des i-ten Elements durch Zeigerarithmetik bestimmen.

Die Größe einer Reihung wird bei ihrer Erstellung (bei ihrer Vereinbarung im Falle einer lokalen Variable oder beim Aufruf vom `malloc`) festgelegt und kann nicht nachträglich verändert werden. Übersteigt die Anzahl der anzufügenden Elemente die Größe der Reihung (vielleicht weil man beim Anlegen der Reihung die Anzahl der anzufügenden Elemente noch nicht kannte), so muss man eine neue, größere Reihung anlegen, die Elemente entsprechend umkopieren, und die alte Reihung freigeben. Dies erreicht man mit der Funktion `realloc`.

```
int* read_numbers_from_file(FILE *f) {
    int *numbers = NULL;
    unsigned capacity = 0;
    unsigned n = 0;
    int x;
    while (fscanf(f, "%d", &x) == 1) {
        if (n == capacity) {
            capacity = capacity == 0 ? 1 : 2 * capacity;
            numbers = realloc(numbers, sizeof(numbers[0])*capacity);
        }
        numbers[n++] = x;
    }
}
```

Es ist sinnvoll, die Größe der Reihung bei jedem `realloc` zu verdoppeln. Der Verschnitt bei n eingefügten Elementen ist höchstens $n - 1$. Die Anzahl der Kopien, die `realloc` durchführen muss, wenn eine neue Reihung angelegt wird, ist maximal $2n$.

3.12 Verbunde

Der zweite zusammengesetzte Datentyp in C ist der **Verbund**. Ein Verbund besteht aus n **Feldern** möglicherweise unterschiedlicher Typen $T_1, \ldots, T_n$. Der Behälter eines Verbunds besteht aus den n Behältern seiner Felder. Betrachten wir den Verbund „Person“ aus Abschnitt 2.6. Eine Person besteht aus und wird beschrieben durch einem Geburtsdatum, einem Vor- und einem Nachnamen.

Ein Datum ist gegeben durch einen Tag, einen Monat und ein Jahr:

```
struct date {
    unsigned day, month, year;
};
```

In C können wir sowohl Komposition („ist Teil von“) und Aggregation („verweist auf“) definieren:

Komposition

```
struct person {
    struct date date_of_birth;
    char name[16];
    char surname[16];
};
```

Aggregation

```
struct person {
    struct date date_of_birth;
    char const *name;
    char const *surname;
};
```

In beiden Vereinbarungen wurde das Feld `date_of_birth` durch Komposition hinzugefügt. Man könnte das Feld auch per Aggregation durch einen Zeiger auf einen Verbund implementieren. Dagegen spricht, dass `struct date` ein kleiner Verbund ist, so dass ein Zeiger kaum Speicherplatz spart und die Wiederverwendung von `struct date` Objekten somit keinen großen Vorteil bringt.
Auf die einzelnen Felder eines Verbund-Behälters kann mittels `.`-Operator zugegriffen werden:

```
struct person p;
...
printf("%s %s\n", p.name, p.surname);
```

Im Gegensatz zu Reihungen können Verbunde in C per Wert an Unterprogramme übergeben werden. Bei kleinen Verbunden kann das sinnvoll sein. In der Praxis verwendet man allerdings meistens die Übergabe mittels Zeiger, wie in folgendem Beispiel:

```
void init_date(struct date *d, unsigned day,
               unsigned month, unsigned year)
{
    d->day = day;
    d->month = month;
    d->year = year;
}
```

Der Operator $a\texttt{->}b$ ist eine Abkürzung für $(*a)\texttt{.}b$.

3.12.1 typedef

Man kann sich für lange Typnamen, wie beispielsweise `struct person` oder komplizierte Typkonstruktionen, wie beispielsweise `int *(*)[10]` mittels `typedef` Kurznamen definieren:

```
typedef struct {
    /* siehe oben */
} person_t;

/* ptr_table_t ist ein Zeiger auf eine Reihung von
   10 Zeigern auf ints */
typedef int *(*ptr_table_t)[10];
```

Beispiel 3.9 (Initialisierung)**.** Häufig definiert man sich für jeden Verbund auch noch ein Unterprogramm, dass ihn ordentlich initialisiert. Folgendes Unterprogramm initialisiert den Verbund, bei dem `name` und `surname` aggregiert sind.

```
person_t *person_init(person_t *p, date_t const *d,
                      char const *name, char const *surname)
{
    p->date = *d;
    p->name = name;
    p->surname = surname;
    return p;
}
```

Solchen Initialisierungsroutinen, auch **Konstruktoren** genannt, übergibt man meist ein Zeiger auf einen uninitialisierten Verbund. Ein solcher Konstruktor ist dann unabhängig davon, ob der Verbund eine globale Variable ist, oder auf dem Keller oder der Halde angelegt wurde. Man kann sie in jedem dieser Fälle verwenden:

```
person_t global_p;
void person_test() {
    person_t local_p;
    person_t *heap_p;

    heap_p = malloc(sizeof(*heap_p));

    person_init(&global_p, /* ... */);
    person_init(&local_p, /* ... */);
    person_init(heap_p, /* ... */);

    free(heap_p);
}
```

Verwendet man für die Felder `name` und `surname` Komposition statt Aggregation, so muss man die Zeichenketten in den Verbund kopieren:

```
person_t *person_init(person_t *p, date_t const *d,
                      char const *name, char const *surname)
{
```

```
    p->date = *d;
    snprintf(p->name,    sizeof(p->name),    "%s", name);
    snprintf(p->surname, sizeof(p->surname), "%s", surname);
    return p;
}
```

Hierbei ist es wichtig, nur soviele Buchstaben zu kopieren, wie auch in das Feld passen, was hier durch `snprintf` mit der Größe `sizeof(p->name)` implementiert ist. Kopiert man die Zeichenkette ohne die Größe des Zielbehälters zu berücksichtigen, tritt undefiniertes Verhalten (→ Abschnitt 3.14) auf, wenn mehr Bytes kopiert werden, als der Zielbehälter fassen kann. Weiterhin ist zu beachten, dass `p->name` den Typ `char[16]` und nicht `char *` ist, und somit `sizeof(p->name) == 16` ist. ⌟

Beispiel 3.10 (Polynome)**.** Folgender Verbund stellt ein Polynom dar:

```
typedef struct {
    unsigned n;
    int *coeff;
} polynom_t;
```

wobei `n` den Grad angibt. Die Koeffizienten sind per Aggregation implementiert. Dies ist notwendig, da die Länge des Polynoms nicht statisch bekannt ist. Insofern muss die Reihung der Koeffizienten separat alloziert werden (→ Aufgabe 3.8). ⌟

3.12.2 Unvollständige Verbunde

Um große Software-Systeme besser zu modularisieren, verwendet man eine Technik namens **Kapselung (engl. encapsulation)**. Unter Kapselung versteht man das Verbergen von Implementierungs-Details eines Moduls. Verbirgt man Details einer Implementierung vor anderen Code-Teilen, so kann man diese Implementierung ändern, ohne dadurch die anderen Code-Teile anpassen zu müssen.

C unterstützt Kapselung durch unvollständige Verbunde. Hierdurch kann man das Wissen über die Zusammensetzung eines Verbunds vor anderen Code-Teilen verbergen. Wir besprechen Kapselung im Java-Teil dieses Buches noch genauer.

Betrachten wir Beispiel 3.10. Nehmen wir an, die Art, wie ein Polynom dargestellt werden soll, soll vor anderen Code-Teilen, die diesen Code verwenden, verborgen werden. Dies erreicht man mit der Vereinbarung eines unvollständigen Verbunds. Dieser wird meist in einer Headerdatei vereinbart (siehe `poly.h` in Abbildung 3.13). Werte des Datentyps sind dann nur durch einen Satz von Unterprogrammen, eine sogenannte Schnittstelle (engl. API = Application Programming Interface) von aussen ansprechbar. Die API eines solchen Datentyps agiert dann mit Zeigern auf den unvollständigen Typ, da Aufgrund der Kapselung sein Inneres nicht bekannt ist und somit auch keine Behältergröße ermittelbar ist.

poly.h

```
#ifndef POLY_H
#define POLY_H

/* unvollständiger Verbund */
typedef struct poly_t poly_t;

poly_t *poly_alloc(unsigned degree);
void poly_free(poly_t *p);
void poly_set_coeff(poly_t *p, unsigned deg,
                    int coeff);
int poly_eval(poly_t const *p, int x);
unsigned poly_degree(poly_t const *p);

#endif /* POLY_H */
```

poly.c

```
#include <stdlib.h>
#include <assert.h>

#include "poly.h"

struct poly_t {
    unsigned degree;
    int *coeffs;
};
```

Fortsetzung: siehe Aufgabe 3.8.

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "poly.h"

int main(int argc, char *argv[])
{
    if (argc < 3) {
        fprintf(stderr, "syntax: %s x coeffs...",
                argv[0]);
        return 1;
    }

    poly_t *p = poly_alloc(argc - 3);
    for (int i = 2; i < argc; i++) {
        int coeff = atoi(argv[i]);
        poly_set_coeff(p, i - 2, coeff);
    }
    int x = atoi(argv[1]);
    int y = poly_eval(p, x);
    poly_free(p);
    printf("%d\n", y);
    return 0;
}
```

Abbildung 3.13: Implementierung des Polynoms mit Kapselung. Die Gestalt des Verbunds `poly_t` ist nur in der Übersetzungseinheit `poly.c` sichtbar.

3.13 Ein- und Ausgabe

Die Header-Datei `stdio.h` der C-Standardbibliothek stellt die elementaren Unterprogramme zur Ein- und Ausgabe zur Verfügung. Eine Datei wird hier durch einen Zeiger auf ein `FILE` dargestellt. Der Verbund `FILE` ist opak (nicht einsehbar), wir kennen also seine innere Struktur nicht und können lediglich Zeiger darauf anlegen.

`stdio.h` definiert drei globale Variablen

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

die Standard-Eingabe, -Ausgabe und -Fehlerausgabe genannt werden.

Dateien können mit der Funktion

```
FILE* fopen(char const *filename, char const *mode);
```

geöffnet und mit

```
void fclose(FILE*);
```

wieder geschlossen werden. `fopen` nimmt einen Dateinamen und einen Modus, der genauer angibt, wie die Datei geöffnet werden soll (`"r"` zum Lesen und `"w"` zum Schreiben). Es liefert einen Zeiger zu einem `FILE` wenn das Öffnen erfolgreich war, oder `NULL`, falls nicht. Zum Lesen und Schreiben bedient man sich meist der Funktionen

```
int fscanf(FILE* f, char const* format, ...);
int fprintf(FILE* f, char const* format, ...);
```

Die häufig verwendeten Funktionen `printf` und `scanf` sind Kurzformen dafür wobei `f` auf `stdin` bzw. `stdout` gesetzt wird.

3.13.1 Formatstrings

Sowohl `fprintf` als auch `fscanf` bedienen sich sogenannter Formatstrings um das Format der Ein- und Ausgabe festzulegen. Das gewählte Formatelement entscheidet dann auch, wie das entsprechende Argument zu interpretieren ist. Wir wollen hier einige Beispiele geben und für die vollständige Dokumentation auf die entsprechenden Unix manpages[33] verweisen.

[33] `$ man fprintf`

```
    char const *weekday = "Wednesday";
    char const *month = "March";
    int day = 11;
    int hour = 10;
    int min = 30;
    printf("%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

erzeugt die Ausgabe

```
Wednesday, March 11, 10:30
```

```
    printf("pi = %.5f\n", 4 * atan(1.0));
```

erzeugt die Ausgabe

```
pi = 3.14159
```

Beispiel 3.11 (Laufender Durchschnitt)**.** Wir möchten ein Programm schreiben, dass den Durchschnitt aller Zahlen aus einer Liste von Dateien berechnet. Eine Technik, den Durchschnitt einer Liste von Zahlen zu berechnen, ist der laufende Durchschnitt (engl. running average). Kennt man den Durchschnitt s_n der ersten n Zahlen $a_1, \ldots, a_n$, so ist der Durchschnitt der Zahlen $a_1, \ldots, a_n$ gegeben durch:

$$s_{n+1} = \frac{n}{n+1} s_n + \frac{1}{n+1} a_{n+1} \tag{3.2}$$

Hierzu schreiben wir zunächst ein Unterprogramm, das den Durchschnitt aller Zahlen **einer** Datei berechnet. Wir nehmen an, dass in der Datei durch Leerzeichen getrennte, textuelle Repräsentationen von Gleitkomma-Zahlen stehen. Zum Beispiel könnte die Datei so aussehen:

```
1 2.2 3
4.2
1.4
```

Die textuelle Darstellung kann mittels `fscanf` wieder in eine Bitfolge übersetzt werden, die wir in einer Variable ablegen können. Den Durchschnitt und die Anzahl der gelesenen Zahlen Speichern wir in einem Verbund

```
typedef struct {
    double avg;
    unsigned n;
} avg_t;
```

Das Unterprogramm `running_average` nimmt die Datei und einen Zeiger einen solchen Verbund als Argumente. Es liest dann solange es Zahlen in der Datei lesen kann[34] und schreibt den Durchschnitt gemäß (3.2) fort:

[34] `fscanf` gibt die Anzahl der korrekt dekodierten Elemente des Formatstrings zurück

```
avg_t *running_average(FILE* f, avg_t* avg)
{
    double val;
    while (fscanf(f, "%lg", &val) == 1) {
        double   old_n = avg->n;
        unsigned new_n = ++avg->n;
        avg->avg = (old_n / new_n) * avg->avg + val / new_n;
    }
    return avg;
}
```

Folgendes Hauptprogramm akzeptiert eine Liste von Dateinamen als Kommandozeilen-Parameter. Es öffnet diese Dateien nacheinander und ruft das Unterprogramm `running_average` darauf auf und gibt zuletzt den berechneten Durchschnitt aus. Ist kein Kommandozeilen-Parameter gegeben, ist die Eingabedatei die Standard-Eingabe `stdin`.

```
int main(int argc, char *argv[])
{
    avg_t avg;
    avg.avg = 0.0;
    avg.n   = 0;

    if (argc == 1) {
        running_average(stdin, &avg);
    }
    else {
        for (int i = 1; i < argc; i++) {
            FILE* input;
            if ((input = fopen(argv[i], "r")) != NULL) {
                running_average(input, &avg);
                fclose(input);
            }
            else {
                fprintf(stderr, "no such file: %s\n", argv[i]);
                return 1;
            }
        }
    }
    printf("%g\n", avg.avg);
    return 0;
}
```

3.14 Undefiniertes Verhalten

Betrachten wir die Anweisung

```
z = x / y;
```

Nehmen wir an, die Variable *x* enthält einen Zeiger auf ein Zeichen und *y* eine Ganzzahl. Dieses Programm hat in diesem Fall keine Bedeutung, da C die Division eines Zeigers durch eine Zahl nicht definiert. Diese Art von Fehlern nennt man Typfehler, weil ein Operator auf Werte angewandt wird, auf dessen Typen er nicht definiert ist. Typfehler werden in typisierten Sprachen durch Typüberprüfung statisch, also während der Übersetzungszeit, erkannt. Mithilfe des Typsystems kann der Übersetzer also bedeutungslose Programme zurückweisen.

Es gibt aber bedeutungslose Programme, die der Übersetzer nicht ablehnen kann. Nehmen wir im gleichen Programm an, dass *x* und *y* beides Ganzzahlen sind, *y* aber den Wert 0 enthält. Das Programm ist **wohlgetypt** (engl. well-typed) , da C die Division auf Ganzzahlen definiert. Nichtsdestotrotz ist nicht klar, was bei einer Division durch 0 passieren soll.

Es gibt im wesentlichen zwei Lösungen zu diesem Problem: Man definiert ein Verhalten, wenn eine solche Ausnahmesituation eintritt, zum Beispiel das Ausführen von Code, der den Fehler behandelt. Diesen Weg bestreiten beispielsweise Java, C#, ML und weitere Sprachen durch das Auslösen von Ausnahmen (engl. exceptions). Dem Programm wird dann für Ausnahmesituationen ein definiertes Verhalten gegeben. Allerdings muss man dann **zur Laufzeit** bei jeder Operation, die potenziell eine Ausnahmesituation auslösen könnte, überprüfen ob diese tatsächlich eingetreten ist. Dies kann die Programmlaufzeit **signifikant negativ** beeinträchtigen. Wir nennen eine Sprache **typsicher**, wenn jedes wohlgetypte Programm nicht „steckenbleibt“. Robin Milner prägte folgende Definition von Typsicherheit:

> well-typed programs cannot go wrong.

Die andere Lösung besteht darin, im Ausnahmefall kein Verhalten vorzuschreiben. Diesen Weg bestreitet C[35] Dadurch, dass man kein Verhalten vorschreibt, muss man zur Laufzeit auch nicht prüfen, ob eine Ausnahmesituation eingetreten ist. Daher können C-Programme im Allgemeinen schneller ablaufen als Programme einer typisierten Sprache. Allerdings hat der Programmierer auch keine Aussage darüber, was das Programm tut, wenn der Ausnahmefall eintritt. Dies kann zu gravierenden Fehlern führen, die nur sehr schwer zu finden sind. Annex J.2 des C-Standards[ISO99] dokumentiert alle Situationen in denen das Verhalten undefiniert ist. Neben vielen anderen Situationen sind darunter: Überläufe bei vorzeichenbehafteten Ganzzahlen, Division durch 0, Bildung von Adressen jenseits allozierter Behälter, Zugriff auf Adressen an denen kein Behälter liegt.

[35] Somit kann C nicht als typsichere Sprache gelten. Allerdings gibt das Typsystem von Garantien, die hilfreich sind und bestimmte Laufzeitfehler ausschließen.

Betrachten wir ein einführendes Beispiel. Der Code dieses Beispiels verletzt unter Umständen die Speichersicherheit, das heißt, er versucht eine Adresse zu verwenden, die zu keinem Behälter passt.

Beispiel 3.12 (Speichersicherheit (engl. memory safety))**.** Betrachten wir folgendes Programm

```
void broken(int n) {
    int x[20];
    *(x + n) = 1;
}
```

Das Programm hat definiertes Verhalten, wenn $0 \leq n < 20$. Für alle $n \geq 20$ ist das verhalten des Programms nach der Auswertung des Ausdrucks `*(x + n)` undefiniert, da die resultierende Adresse nicht innerhalb des Behälters liegt (C99 Standard, Seite 492):

> The behavior is undefined in the following circumstances: […] Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

Dies bedeutet nicht notwendigerweise, dass das Programm abstürzt oder sich mit einem Fehler beendet. Es ist einfach nicht definiert, was dann passiert. Jede Annahme über das Verhalten des Programms nach dem undefinierten Zugriff ist **falsch**. Es kann also auch gut sein, dass das Programm weiter, oder gar zu Ende läuft und das erwartete Ergebnis produziert. Das macht es schwer, den Programmierfehler in der Funktion `broken` zu finden.

Warum lässt man sich überhaupt auf undefiniertes Verhalten ein? Versetzen wir uns kurz in die Lage des Übersetzers, der `broken` in Maschinensprache übersetzt. Wenn das Verhalten eines Zugriffs mit $n \geq 20$ kein undefiniertes Verhalten auslösen soll, muss diese Bedingung abgefangen werden. Das heißt, der Übersetzer muss entweder durch eine statische Analyse des Codes beweisen, dass für alle möglichen Eingaben des Programms $n < 20$ gilt. Dies ist ein schweres Problem und in der Praxis können viele Übersetzer diese Bedingungen nicht vollständig nachweisen. Andernfalls muss der Übersetzer Code erzeugen, der diese Bedingung zur Laufzeit des Programmes testet:

```
       sltiu $t1 $a0 20
       beqz  $t1 fail
       sll   $t1 $a0 4
       addu  $t0 $t0 $t1
       li    $t1 1
       sw    $t1 ($t0)
   fail:
```

Die ersten beiden Befehle dienen nur zur Erkennung und Behandlung der Ausnahmesituation, dass der Versatz jenseits von `x` liegt. Diese Tests können die Laufzeit eines Programms signifikant negativ beeinflussen. Man stelle sich vor, sie stehen im Rumpf mehrerer geschachtelter Schleifen.

Es ist die Philosophie von C, diese Tests nicht zu erzwingen, und die Verantwortung dem Programmierer in die Hand zu geben. Für den Fall, dass das Programm, das `broken` aufruft, korrekt ist, ist der Test überflüssig und die Laufzeit kann gespart werden. Die Tatsache, dass das

Verhalten im Fall $n \geq 20$ undefiniert ist, erlaubt es dem Übersetzer jetzt, diesen Fall schlicht zu ignorieren: Der Code, den er erzeugt, muss nur für den Fall $0 \leq n < 20$ korrekt sein, andernfalls ist das Verhalten ja undefiniert. Dies erlaubt es dem Übersetzer, Code zu erzeugen, der den Test nicht enthält:

```
        sll   $t1 $a0 4
        addu  $t0 $t0 $t1
        li    $t1 1
        sw    $t1 ($t0)
```

Ist sich der Programmierer nicht sicher, dass der Code, der `broken` aufruft, sicherstellt, dass $0 \leq n < 20$, muss er die Überprüfung selbst einbauen:

```
void broken(int n) {
    int x[20];
    if (0 <= n && n < 20)
        *(x + n) = 1;
}
```

⌟

3.14.1 Unspezifiziertes und implementierungsspezifisches Verhalten

Der C-Standard unterscheidet sehr genau zwischen undefiniertem, unspezifiziertem und implementierungsspezifischem Verhalten.

Unspezifiziertes Verhalten entspricht Nicht-Determinismus. Es gibt mehrere, genau definierte Möglichkeiten, wie ein Programm sich verhalten kann. Der Programmierer kann sich darauf verlassen, dass sich das Programm nach einer der Möglichkeiten verhalten wird, weiss aber nicht welcher. Unspezifiziertes Verhalten haben wir in Abschnitt 3.7.1 gesehen: C legt nicht fest, in welcher Reihenfolge die Teilausdrücke eines Ausdrucks ausgewertet werden. Hat der Ausdruck Nebenwirkungen kann es zu unterschiedlichem Verhalten kommen. Annex J.1 des C-Standards dokumentiert das unspezifizierte Verhalten.

Implementierungsspezifisches Verhalten meint unspezifiziertes Verhalten, bei dem der Übersetzer eine Möglichkeit fest auswählt. Diese ist dann für alle mit diesem Übersetzer übersetzten Programme gleich. Implementierungsspezifisches Verhalten ist beispielsweise die maximale Länge eines Bezeichners, die Zahldarstellung von vorzeichenbehafteten Ganzzahlen (Zweierkomplement, Einerkomplement, etc.), die Größe der Integertypen, usw. Annex J.3 des C-Standards dokumentiert das implementierungsspezifische Verhalten.

3.14.2 Sicherheitsprobleme

Nachlässigkeit im Zusammenhang mit undefiniertem Verhalten kann zu Fehlern führen, die Programme nicht nur abstürzen lassen, sondern auch schwere Sicherheitslücken darstellen. Der Klassiker ist der sogenannte Pufferüberlauf (engl. buffer overflow). Nehmen wir an, ein Angreifer hat den Binärcode, den ein Übersetzer aus einem C-Programm erzeugt hat.

Er kann sich nun ungeprüftes Schreiben in Reihungen (meist Zeichenketten) zu nutze machen, um den Inhalt des Laufzeitkellers zu manipulieren um den Programmablauf zu ändern und letztendlich Schad-Code in ein Programm zu injizieren.

Betrachten wir folgendes Beispielprogramm[36]:

```
void foo(char const *msg) {
    char nice_msg[32];
    sprintf(nice_msg, "The message is: \"%s\"", msg);
    ...
}
```

[36] `sprintf` gibt den formatierten String nicht auf der Konsole aus wie `printf`, sondern schreibt die einzelnen Zeichen, in die Reihung, die als erster Parameter mitgegeben wird. Undefiniertes Verhalten tritt ein, wenn die Reihung nicht groß genug ist.

Nehmen wir nun an, der Übersetzer hat folgenden Code für `foo` erzeugt:

```
    addiu  $sp $sp -36
    sw     $ra 32($sp)

    # Code, der die Adresse von
    # "The messa ..." nach $a1 lädt
    move   $a2 $a0
    move   $a0 $sp
    jal    sprintf
    ...

    lw     $ra 32($sp)
    addiu  $sp $sp 36
    jr     $ra
```

Nehmen wir der Einfachheit halber an, dass `foo` mit einem Argument aufgerufen wird, dass über das Netzwerk empfangen wurde oder von der Platte gelesen wurde. In der Praxis ist das oft der Fall.

Problematisch an `foo` ist nun, dass es die übergebene Zeichenkette in einen Puffer fester Länge (hier 32) schreibt ohne dass beim Schreiben geprüft wird, ob der Puffer schon voll ist. Was, wenn `"The message is: "` konkateniert mit `msg` länger als 31 Zeichen ist? Da `sprintf` nur die Adresse der Zielreihung bekommt, deren Länge es nicht kennt und somit auch nicht prüfen kann, schreibt es potenziell darüber hinaus. Dadurch wird die Konsistenz des Laufzeitkellers zerstört, was dem Angreifer erlaubt, die von `foo` gesicherte Rückkehr-Adresse zu überschreiben.

Angenommen, der Angreifer kennt den Wert des Kellerpegels beim Aufruf (sagen wir, er hat den Wert 0xffffffc0). Dann kann er eine Zeichenkette fabrizieren, die (fast) beliebigen Code in das Programm einschleust. Dieser Code ist Teil der Zeichenkette (die Bytes der binärcodierten Maschinenbefehle). Dazu muss er nur die Rückkehradresse durch eine Adresse weiter oben auf dem Keller ersetzen, an der der erste auszuführende Befehl steht, nachdem die Zeichenkette auf den Keller geschrieben wurde. Der Code darf nur keine Null-Bytes beinhalten, sonst wähnt `sprintf` das Ende der Zeichenkette und bricht das kopieren mitten im Schadcode ab.

Folgender MIPS-Code gibt auf der Konsole die Zahl 666 aus und beendet dann das Programm. Um zu vermeiden, dass seine Befehlsworte Null-Bytes enthalten, müssen manche Konstanten umständlich

	Adresse	Inhalt
Schachteln der Aufrufer	0xFFFFFFFF	???
	⋮	⋮
	$sp + 36	???
Schachtel von foo	$sp + 32	Rückkehr-Addr.
	$sp + 28	???
	⋮	⋮
	$sp + 0	???
	⋮	⋮
	0x00000000	???

Speicherauszug des Laufzeitkellers beim Aufruf von `foo`.

zusammengebaut werden. Auch der Befehl `syscall` kann nicht direkt verwendet werden, da sein Befehlswort Null-Bytes enthält. Daher erstellt der Code als erstes ein kleines Unterprogramm, das einen syscall ausführt.

```
bad_code_start:
  addiu   $t9 $ra -1000         # $t9 <- $ra - 1000
  li      $t1 1012
  addiu   $t1 $t1 -1000         # $t1 <- 12
  sw      $t1 1000($t9)         # write 0x0c (syscall) to $ra + ofs
  lui     $t1 0x03e0
  ori     $t1 $t1 0x0118
  addiu   $t1 $t1 -272          # $t1 <- 0x03e00800 (jr $ra)
  sw      $t1 1004($t9)         # write return to adress after syscall
  sw      $0  1008($t9)         # fill delay slot (nop = 0)
  addiu   $t1 $t9 1000          # address of syscall in $t1
  li      $a0 666
  li      $v0 1001
  addiu   $v0 $v0 -1000         # $v0 <- 1
  jalr    $t1                   # syscall
  addiu   $0 $t9 -1             # delay slot
  li      $v0 1010
  addiu   $v0 $v0 -1000         # $v0 <- 10
  jalr    $t1
  addiu   $0  $t9 -1            # delay slot nop
bad_code:
  bgezal  $0  bad_code_start    # jump to start and get $pc in $ra
  nop
```

Wir konstruieren nun ein Argument an `foo` so, dass das Schadprogramm oberhalb der Aufrufschachtel von `foo` auf den Keller geschrieben wird. Die Rückkehradresse wird mit der Adresse der ersten Instruktion des Schadcodes überschrieben. Nehmen wir an, dass der Kellerpegel bei `foo` den Wert `0xfffff000` hat. Wir wollen die vorletzte Instruktion anspringen (`bgezal` …). Diese liegt 20 Befehle (= 80 Bytes) über dem Kellerpegel, also ist die anzuspringende Adresse `0xfffff050`. Die folgende Zeichenkette enthält Füllmaterial um die Zeichenkette `nice_msg` zu füllen, die neue Rückkehradresse und den Schadcode:

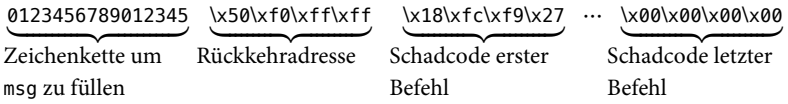


Abbildung 3.14 zeigt den Inhalt des Laufzeit-Kellers nach der Ausführung der `sprintf` Anweisung. Kehrt nun `foo` zurück, so wird die erste Instruktion des Schadcodes angesprungen und nicht zum Aufrufer zurückgekehrt.

Ruft man `foo` mit einer Zeichenkette auf, die länger als 15 Zeichen ist, so ist das Verhalten undefiniert. Das heißt, der Übersetzer kann diese Ausnahmesituation ignorieren und den Maschinencode so erzeugen, dass er nur das definierte Verhalten implementiert. Der Maschinencode hat in der Ausnahmesituation zwar ein Verhalten, nicht aber das C-Programm. Dies macht sich der Angreifer hier zu nutze.

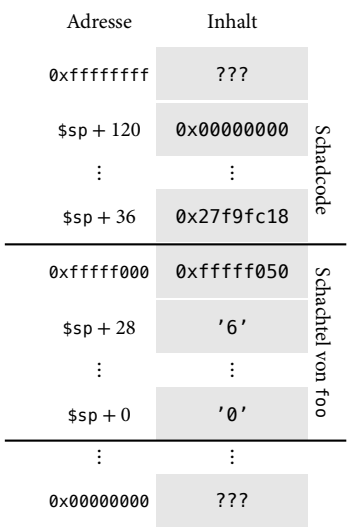


Abbildung 3.14: Speicherauszug des Laufzeitkellers beim Aufruf von `foo` mit der Zeichenkette, die den Schadcode enthält.

Solche Angriffe sind abhängig von der Art und Weise wie ein Programm übersetzt wird. In unserem Beispiel muss der Angreifer die Adresse des Kellerpegels bei Eintritt in `foo` genau kennen, um die Rückkehr-Adresse passend zu fingieren. Man muss also für solche Angriffe eine genaue Studie des Binärcodes eines Programms vornehmen.

Mittlerweile kann man C-Programme effizient gegen einfache Buffer-Overflow Attacken schützen, indem man es dem Prozessor verbietet, Code aus dem Laufzeitkeller auszuführen. Des Weiteren können moderne Übersetzer Code erzeugen, der überschriebene Rückkehradressen erkennt, was aber zu Lasten der Laufzeit geht. Allerdings gibt es gegen ersteres einen Gegenangriff, das sogenannte Return-Oriented Programming, das Befehlsschnipsel aus dem Programm selbst verwendet um den Schadcode auszuführen. Hierfür gibt es bis dato kaum wirksamen Schutz, der die Laufzeit des Programms nicht negativ beeinträchtigt.

3.14.3 Undefiniertes Verhalten und der Übersetzer

Moderne Übersetzer nutzen undefiniertes Verhalten um Programme zu optimieren. Da C nur Programmen eine Bedeutung gibt, bei denen keine Ausnahmesituationen vorkommen, muss der vom Übersetzer erzeugte Maschinencode auch nur für solche Programme korrekt sein. Das heißt im Umkehrschluss, dass der Übersetzer aus Operationen, bei denen potenziell undefiniertes Verhalten auftritt, Informationen gewinnen kann, die er zur weiteren Optimierung des Programms verwenden kann.

Beispiel 3.13 (Eliminierung von unerreichbarem Code)**.** Betrachten wir folgendes Programm:

```
int x = y / z;
if (z == 0)
    return 1;
```

Das Programm hat nur dann definiertes Verhalten, wenn $z \neq 0$. Wenn $z = 0$ ist das Verhalten des Programms nach der Ausführung der Division undefiniert. Der Übersetzer kann also für alle Anweisungen nach der Division annehmen, dass $z \neq 0$ st und dieses Wissen zur Optimierung des Programms nutzen.

Gemäß der C-Semantik ist es also korrekt, wenn der Übersetzer die if-Abfrage entfernt und das Programm wie folgt optimiert:

```
int x = y / z;
```

⌟

Diese Art von Optimierung wird von vielen Programmierern nicht verstanden und fälschlicherweise als „Compiler-Bug“ gewertet. Wie wir im folgenden sehen werden, führt dieses Unverständnis zu vielen Fehlern in weit verbreiteten Software-Systemen.

3.14.4 Beispiele aus der Praxis

Folgende Beispiele sind dem Artikel von Wang et al.[Wan+12] entnommen. Dort findet man weitere Beispiele.

Beispiel 3.14 (Shifts)**.** Betrachten wir ein Stück Code der Implementierung des ext4 Dateisystems des Linux Kernels[37]:

[37] Bug 14287, Linux kernel, 2009. https://bugzilla.kernel.org/show_bug.cgi?id=14287.

```
...
groups_per_flex = 1 << s;
flex_group_count = ... / groups_per_flex;
```

Nach Seite 493 des C99 Standards ist das Verhalten von Shifts in manchen Situationen aber undefiniert:

> The behavior is undefined in the following circumstances: […] An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7)

Nehmen wir an, dass `sizeof(int) == 4`. Ist also der Wert von `s` ≥ 32, so ist das Verhalten des Shifts undefiniert.

Der C Standard spezifiziert das Verhalten von solchen Shifts nicht, da verschiedene Prozessoren Shifts unterschiedlich implementieren. Bei den 32-Bit Versionen von MIPS und dem weit verbreiteten x86 Prozessor werden vom Shift-Betrag nur die untersten 5 Bit berücksichtigt, sprich ein Shift mit 32 entspricht einem Shift mit 0. Auf einem PowerPC Prozessor werden die unteren 6(!) Bit des Shift-Betrags berücksichtigt. Shiftet man also ein PowerPC-Register um 32 nach links, ist sein Inhalt 0. Beide Implementierungen haben ihre Berechtigung. Keine davon ist ‚falsch'.

Würde C sich auf eine der beiden oder gar auf eine dritte Bedeutung des Shifts festlegen, müsste diese auf **jedem** Prozessor genau so implementiert werden. Dies würde dann auf den Prozessoren, bei denen der Shift anders implementiert ist, zusätzlichen Code bedeuten. Es entspricht der Philosophie von C, das Verhalten hier undefiniert zu lassen um das **definierte** Verhalten möglichst effizient implementieren zu können.

Der genannte Bugreport berichtet dann auch, dass der Code auf einem PowerPC abstürzte, da `groups_per_flex` den Wert 0 hatte, was zu einer Division durch 0 führte, die auf dem PowerPC Prozessor eine Ausnahme auslöste. Ein Programmierer versuchte den Fehler dadurch zu beheben, dass die Variable nach dem Shift auf 0 getestet wird:

```
...
groups_per_flex = 1 << s;
/* There are some situations, after shift the
   value of 'groups_per_flex' can become zero
   and division with 0 will result in fixpoint
   divide exception */
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

Diese Korrektur ist falsch, denn sie verhindert nicht das Auftreten des undefinierten Verhaltens. Der C-Übersetzer, der zum Übersetzen des Linux Kernels verwendet wird, hat die eingefügte Korrektur auch umgehend „wegoptimiert". Dies ist korrekt, da das Verhalten des Programms undefiniert ist, wenn `s` größer ist, als die Anzahl von Bits in einem int.

Andernfalls ist der Inhalt von `groups_per_flex` immer ungleich 0, insofern ist die Bedingung des ifs immer falsch. Somit kann es gelöscht werden.

Die richtige Fehlerkorrektur wäre gewesen:

```
if (s < 0 || s >= CHAR_BIT * sizeof(int))
    return 1;
groups_per_flex = 1 << s;
flex_group_count = ... / groups_per_flex;
```

Beispiel 3.15 (Überlauf)**.** Folgender Code ist auch dem Linux Kernel entnommen. Der Typ `loff_t` ist ein vorzeichenbehafteter Ganzzahl-Typ.

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
...
```

Zunächst stellt der Code sicher, dass beide Variablen `offset` und `len` nicht negativ sind. Die Bedingung `offset + len < 0` versucht zu prüfen, ob bei der Addition ein Überlauf eintritt. In C ist aber das Verhalten beim Überlauf von vorzeichenbehafteten Ganzzahlen undefiniert. Der Übersetzer schließt aus dem ersten if, dass am zweiten if `offset` und `len` nicht negativ sind. Die Addition zweier nicht-negativer Zahlen führt aber entweder zu undefiniertem Verhalten (wenn ein Überlauf eintritt) oder zu einer positiven Zahl. Daher optimiert der Übersetzer das zweite if zu:

```
    ...
    if (offset + len > inode->i_sb->s_maxbytes)
        return -EFBIG;
    ...
```

Korrekt müsste der Test auf Überlauf wie folgt lauten:

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((LOFF_T_MAX - offset < len)
        || (offset + len > inode->i_sb->s_maxbytes)
        return -EFBIG;
...
```

wobei `LOFF_T_MAX` den größten Wert des Typs `loff_t` beinhaltet. Idealerweise schreibt man sich eine Funktion, um den Code lesbarer zu gestalten und potenzielle Fehler auf eine stelle zu konzentrieren:

```
int loff_overflows(loff_t offset, loff_t len)
{
    assert(offset >= 0);
    assert(len > 0);
    return LOFF_T_MAX - offset < len;
}

int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if (loff_overflows(offset, len)
        || (offset + len > inode->i_sb->s_maxbytes)
        return -EFBIG;
...
```

3.15 Aufgaben

Aufgabe 3.1 (C Datentypen und Zeichenketten. ⋆⋆)**.** Schreiben Sie ein C-Programm, das die Caesar-Verschlüsselung einer Zeichenkette mit einer Verschiebung von n berechnet und ausgibt. Bei der Caesar-Verschlüsselung wird jeder Buchstabe durch den Buchstaben ersetzt, der n Stellen weiter im Alphabet steht. Am Ende des Alphabets wird von vorne weitergezählt. Die Argumente des Programms sind n und die zu verschlüsselnde Zeichenkette. Sie dürfen dabei annehmen, dass diese nur die ASCII-Zeichen a-z enthält. Als Grundlage können Sie c-caesar-empty.c nutzen. Welche Ähnlichkeiten und Unterschiede fallen Ihnen zur MIPS Implementierung auf?

Aufgabe 3.2 (Bitoperatoren. ⋆⋆)**.** In dieser Aufgabe sollen Sie lernen, wie die ihnen bereits bekannten Bitoperationen in C realisiert sind.

C	Arithmetik
`a & b`	$a \,\&\, b$
`a ^ b`	$a \,\hat{}\, b$
`a | b`	$a \mid b$
`a << b`	$a \ll b$
`a >> b`	$a \gg b$
`~a`	$\overline{a}$

Schreiben Sie eine Funktion mit dem Prototypen

```
unsigned int f(unsigned int k, unsigned int n);
```

1. die das k-te Bit in n kippt, wenn k zwischen 0 und 31 ist.
2. die das k-te Bit in n setzt, wenn k zwischen 0 und 31 ist.
3. die das k-te Bit in n löscht, wenn k zwischen 0 und 31 ist.
4. die n um k Stellen nach links rotiert, wenn k zwischen 1 und 31 ist.
5. die n um k Stellen nach rechts rotiert, wenn k zwischen 1 und 31 ist.

⌟

Aufgabe 3.3 (Zeiger. ★★★)**.** Was gibt folgendes Programm aus?

Nachdem Sie sich eine Lösung überlegt haben, können Sie die Datei c-zeiger.c nutzen um ihre Lösung zu überprüfen.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int *dr, *ds;
  int **drr, **dss;
  int i;
  int aa[5];

  for (i = 0; i < 5; i++) *(aa + i) = -123;

  dr = &aa[3];
  ds = dr - 3;
  drr = &dr;
  dss = &ds;
  *dr = 23;
  *ds = 11;
  *(dr + 1) = 5;
  ds += 1;
  *ds = 66;
  aa[2] = 42;
  printf("%d %d %d %d %d\n", aa[0], aa[1], aa[2], aa[3], aa[4]);

  aa[0] = *(aa+2) - *(aa + 1);
  *(*drr - 1) = 7;
  *(*dss + 3) = 4;
  **dss = **drr + *(*drr - 2);
  printf("%d %d %d %d %d\n", aa[0], aa[1], aa[2], aa[3], aa[4]);

  return 0;
}
```

⌟

Aufgabe 3.4 (printf. ★★)**.** Vervollständigen Sie das folgende Programm, sodass es die Reihung von Ganzzahlen `zahlen` in verschiedenen Darstellungen ausgibt.

```c
#include <stdio.h>

int main(void) {
  int zahlen[] = {-13427, -4233, -343, -12, -5, 0,
                  3, 17, 512, 2355, 29367};
  int n = /* Anzahl der Elemente von zahlen berechnen */

  for (int i=0; i < n; ++i) {
    printf("%d ", zahlen[i]); // Diese Zeile muss jeweils angepasst werden
  }

  return 0;
}
```

Berechnen Sie zunächst mithilfe von `sizeof` die Anzahl der Elemente von `zahlen`. Geben Sie die Reihung dann in folgenden Darstellungen aus:

1. in Hexadezimaldarstellung mit vorangestelltem `0x` und bis zur Breite 8 mit Nullen aufgefüllt, also z.B. `0x00000012`. Welchen Argumenttyp verlangt der Formatspezifizierer für die Hexadezimale Ausgabe?
2. den 1000. Teil jeder Zahl auf zwei Nachkommastellen gerundet
3. die Reihung als vorzeichenbehaftete Zahlen interpretiert, und mit Vorzeichen (auch +).

Jede dieser Teilaufgaben ist mit einer entsprechend gewählten Formatzeichenkette und der Bibliotheksfunktion `printf` zu erledigen. `printf` ist eine Funktion mit variabler Argumentlänge, d.h. Aufrufe der Form `printf`($format, x_1 \ldots x_n$) sind für alle n zulässig. Als *format* wird eine Zeichenkette übergeben, die Formatspezifizierer enthalten kann. In der Ausgabe wird der n-te Formatspezifizierer durch die ensprechend formatierte Darstellung von x_n ersetzt. Ein Formatspezifizierer hat die Form

```
%[flags][min field width][precision][length]conversion specifier
```

wobei alle in `[]` gefassten Parameter optional sind. Die Beschreibung der Parameter finden Sie beispielsweise auf der Handbuchzeite zu `printf`. Diese können Sie mit dem Befehl `man 3 printf` abrufen. Die deutsche Übersetzung der Handbuchseite finden Sie mit `LANG=de_DE man 3 printf`. *Hinweis:* Versuchen Sie nicht, die Handbuchseite gänzlich zu lesen. Sie sollen lernen, den relevanten Teil zu finden und die benötigten Informationen zu extrahieren.

⌟

Aufgabe 3.5 (scanf. ★★)**.** Schreiben Sie ein Programm, das den BMI berechnet. Sie müssen dazu die Körpergröße h in Metern als Gleitkommazahl einlesen, sowie die Masse m in Kilogramm als Ganzzahl. Geben Sie dann den BMI b, der sich gemäß folgender Gleichung berechnet, aus:

$$b = \frac{m}{h^2}$$

Verwenden Sie zum Einlesen der Zahlen die Bibliotheksfunktion `scanf`, die analog zu `printf` funktioniert. `scanf` verwendet ähnliche Formatspezifizierer wie `prinf`, die Form ist

```
%[max field width][type modifier]conversion specifier
```

Beachten Sie, dass `scanf` die **Adresse des Behälters** erwartet, in den die Eingabe geschrieben werden soll. Sie können die Funktionsweise von scanf z.B. auf der Handbuchseite nachlesen, die Sie mit dem Befehl

`man 3 scanf` bzw. die deutsche Seite mit `LANG=de_DE man 3 scanf`

abrufen können. ⌟

Aufgabe 3.6 (Datei lesen und schreiben. ⋆⋆)**.** Schreiben Sie ein Programm, das eine Liste von Zahlen aus einer Datei `a.txt` liest, und die Summe der Zahlen in einer Datei `b.txt` ablegt. Sie öffnen die Datei mit der Bibliotheksfunktion `fopen`, welche ihnen einen Wert vom Typ `FILE*` liefert. Dieser Wert repräsentiert die Datei und muss an Ein- und Ausgabefunktionen, die mit der Datei arbeiten sollen, weitergereicht werden. Wie schon in den vorhergehenden Aufgaben können Sie mit `man 3 fopen` im Handbuch über `fopen` nachlesen. Benutzen Sie die Bibliotheksfunktion `fscanf` um aus einer Datei zu lesen und `fprintf`, um in eine Datei zu schreiben. Beide Funktionen arbeiten analog zu `scanf` bzw. `printf`. Denken Sie daran, die Datei mit `fclose` wieder zu schließen. ⌟

Aufgabe 3.7 (Personen. ⋆⋆⋆)**.** Schreiben Sie ein Programm, das auf dem Verbund `person_t` aus dem Skript beruht. Ihr Programm soll eine Textdatei mit Personen einlesen. In jeder Zeile der Datei steht eine Person. Eine Person ist in der Datei in folgendem Format abgelegt:

```
Name Vorname Jahr-Monat-Tag
```

1. Legen Sie die eingelesenen Personen in einer Reihung ab. Da Sie initial nicht wissen, wieviele Personen Sie lesen werden, müssen Sie die Reihung potentiell vergrößern. Verwenden Sie hierzu `realloc` und verwenden Sie `memcpy` oder `memmove` um die Reihungen gegebenenfalls zu kopieren.
2. Sortieren Sie die Datensätze nach Altern (jüngster zuerst). Verwenden Sie hierzu `qsort`. Hierzu müssen Sie sich eine Vergleichsfunktion schreiben, die zwei `date_t`s miteinander vergleichen kann. Geben Sie die sortierten Daten aus.

⌟

Aufgabe 3.8 (Polynom. ⋆⋆⋆)**.** Wir betrachten den Verbund `poly_t` aus Beispiel 3.10. Ergänzen Sie die fehlernden Unterprogramme in `poly.c` aus Abbildung 3.13:

1. Schreiben Sie einen Konstruktor für Polynome, der den Verbund und die Reihung der Koeffzienten auf der Halde anlegt.
2. Legen Sie spezielle Varianten

```
polynom_t *poly_linear(polynom_t *p, int a1, int a0);
polynom_t *poly_quadratic(polynom_t *p, int a2, int a1, int a0);
```

für lineare und quadratische Polyonome an.

3. Schreiben Sie eine Routine `polynom_free`, die ein auf der Halde angelegtes Polynom (inkl. Koeffizientenreihung) freigibt.

4. Implementieren Sie die Polynomauswertung mit dem Horner-Schema für diesen Verbund.

5. Implementieren Sie einen Konstruktor, der die Koeffizienten eines Polynoms aus einer Datei liest. Die Koeffizienten liegen in der Datei durch Leerzeichen getrennt. Der Koeffeizient für das höchste Glied steht zuerst.

⌟

Aufgabe 3.9 (Median. ★★★)**.** Schreiben Sie ein Program, das eine Liste von Zahlen aus einer Datei `a.txt` liest, und den Median der Zahlen ausgibt.

Um den Median zu berechnen, sollen Sie die Zahlen in einer verketteten Liste ablegen. Verwenden Sie dazu folgenden Verbund:

```
typedef struct el_t el_t;
struct el_t {
  et_t *next;
  int   data;
};
```

Wenn Sie alle Werte aus der Datei sortiert in die Liste eingefügt haben, können Sie den Median leicht bestimmen, indem Sie die Elemente an der entsprechenden Position zugreifen.

Schreiben Sie zunächst eine Funktion `insert`, die eine Ganzzahl in eine aufsteigend sortierte, verkettete Liste so einfügt, dass die Sortierung erhalten wird. Die Liste wird als Zeiger auf das erste Argument übergeben. `insert` soll den Zeiger auf das (möglicherweise) neue erste Element der Liste zurückliefern Verwenden Sie folgende Anweisung, um sich einen Behälter für ein neues Listenelement zu beschaffen:

```
el_t *new_el = malloc(/* Größe des Listenelements */);
```

Sie müssen lediglich die Größe des Listenelements eintragen, die Sie mithilfe von `sizeof` berechnen können. Finden Sie mithilfe der Handbuchseite zu malloc heraus, welche Headerdatei sie außer `stdio.h` noch einbinden müssen.

Schreiben Sie als nächstes eine Funktion `get`, die zu einer Position und einer verketteten Liste einen Zeiger auf das Element an dieser Position liefert.

Sie können jetzt der Programmstruktur von Aufgabe 4.2 folgen, und ihr Programm implementieren.

⌟

Aufgabe 3.10 (Generisches Sortieren durch Einfügen, reloaded. ★★★★★)**.**

1. Erweitern Sie Insertsort in C so, dass es beliebige Daten mittels einer Vergleichsfunktion sortieren kann (analog zur MIPS Aufgabe). Auf der Website der Vorlesung finden Sie einen Rumpf für ihre Implementierung. In diesem Rumpf gibt es zwei Hauptprogramme, `main_int.c` und `main_str.c`. Das erste ist vollständig und testet Ihre Routine mit vorzeichenlosen Ganzzahlen, die sie von der Eingabe liest. Lesen Sie das Hauptprogramm und verstehen Sie jede einzelne Anweisung.

2. Implementieren Sie ein weiteres Hauptprogramm, das die Sortierfunktion nutzt. Dieses soll eine Reihung von Zeichenketten sortieren. Einen Rumpf dafür (`main_str.c`) finden Sie ebenfalls in dem Paket, das Sie von der Website geladen haben. Machen Sie sich den Code, der sich dort findet klar. Wieso wird in der Schleife, die die Zeichenketten liest jedesmal Speicher angefordert? Wie ist die Reihung von Zeichenketten organisiert? Ergänzen Sie das noch unvollständige Hauptprogramm `main_str.c`.

Hinweise:

- Zum Bauen der beiden Programme können Sie das `Makefile` verwenden, indem Sie jeweils `make main_str` oder `make main_int` eingeben.
- Wenn Sie die virtuelle Maschine von der Vorlesungswebsite nutzen, können Sie den Rumpf der Aufgabe mit folgenden Kommandos

  ```
  wget http://www.cdl.uni-saarland.de/teaching/prog2/2012/misc/rumpf_aufgabe_43.zip
  unzip rumpf_aufgabe_43.zip
  ```

  von der Konsole aus runterladen und entpacken.

⌟

Aufgabe 3.11 (C Datentypen. ⋆⋆)**.**

- Speichern Sie alle Substantive und Werte aus dem folgenden Text in Variablen mit geeigneten Typen. Schreiben Sie eine `printf()`-Anweisung, die alle diese Variable als Argumente verwendet, um den Text auszugeben.

  ```
  Am 15.4.2011, erhielt der Sieger mit 34.45 Punkten ein Preisgeld von 64500 EURO.
  Trotzdem hat er in diesem Jahr einen Verlust erwirtschaftet (-8096 EURO).
  ```

- Mit welchem Datentypen müssen die folgenden Variablen deklariert werden, damit sie die selben Datentypen besitzen wie das Ergebnis auf der rechten Seite ihrer Zuweisung?

  1. `x = 256 * -32768 + 4294967296L * -2;`
  2. `y = 3 * -4L + 1.0F * .5e9L;`

⌟

4 Abstraktion, Modularisierung und Verfeinerung

Große Programme sind nur beherrschbar, wenn man sie in Komponenten aufteilt. Jede Komponente muss unabhängig von den anderen **entworfen**, **getestet** und **ausgetauscht** werden können. Nur dadurch kann man erreichen, dass mehrere Entwickler an einem Programm arbeiten und nicht jeder alles über das Programm wissen muss. Dies ist bei hinreichend großen Projekten ohnehin unmöglich. Ein weiterer Aspekt der Modularisierung ist die **Wiederverwendbarkeit**. Je unabhängiger eine Komponente vom Rest des Programms ist, desto höher ist die Wahrscheinlichkeit, dass man sie in einem anderen Kontext wieder verwenden kann.

Jede moderne Programmiersprache bietet die Möglichkeit, ein Programm in mehrere Module aufzuteilen und die Details der Implementierung eines Moduls vor den anderen Modulen zu verbergen[1]. Hierbei spricht man auch von **Kapselung (engl. encapsulation)** oder dem **Geheimnisprinzip (engl. information hiding)**. Dies ist notwendig, um Module getrennt voneinander entwerfen, testen und austauschen zu können. Die einzelnen Module müssen aber natürlich auch miteinander interagieren. Hierzu definiert jedes Modul eine **Schnittstelle (engl. interface)**, über die es „von außen" ansprechbar ist[2].

[1] In C wird dies durch das Konzept der Übersetzungseinheit erreicht.

[2] In C wird die Schnittstelle meist in Header-Dateien definiert.

Wie eine gute Aufteilung aussieht und wie man sie erreicht, kann formal schlecht beschrieben werden und unterliegt eher „weichen" Kriterien. Ein guter Software-Entwurf erfordert viel Erfahrung: Man muss mehrere schlechte Entwürfe gemacht haben, um zu verstehen, warum diese schlecht waren. Mit der Zeit wird man einen „Riecher" für gute Entwurfsentscheidungen entwickeln. Hierbei ist es oft wichtig, eine Vorahnung zu haben, wie sich bestimmte Entwurfsentscheidungen auf den weiteren Verlauf des Projekts auswirken.

Eine Technik, die uns dabei helfen kann, ist die **schrittweise Verfeinerung** [Wir71]. Dabei beginnt man mit einem Grobentwurf, den man dann schrittweise zu einer lauffähigen Implementierung verfeinert. Hierbei deutet sich oft schon an, in welche Module eine Implementierung aufgeteilt werden muss. Des Weiteren diskutieren wir grundlegende Aspekte der **Modularisierung** und der **abstrakten Datentypen** und zeigen an einem größeren Beispiel, wie man diese in C umsetzt.

4.1 Modularisierung

Unter Modularisierung verstehen wir die Aufteilung des Programms in Komponenten (Module), die nur über Schnittstellen kommunizieren. Modularisierung wird erreicht durch das Verbergung der Implementierungsdetails jeder einzelnen Komponente hinter einer Schnittstelle. Ziel ist, die Komponenten getrennt voneinander entwerfen, entwickeln, testen, austauschen und wiederverwenden zu können.

Hierbei ist es wichtig, dass andere Komponenten keinen Zugriff auf die Interna der Implementierung eines Moduls haben. Hätten sie diesen, könnten (und würden) sie sich von diesen Interna abhängig machen, was der Entkopplung der Komponenten im Wege steht. Oftmals entsprechen die Module abstrakten Datentypen, die wir im nächsten Abschnitt besprechen.

Betrachten wir die Implementierung der Polynome aus Abbildung 3.13. In der Datei `poly.h` befindet sich die **Schnittstelle** eines Moduls, das Operationen auf Polynomen anbietet. Die Operationen werden in C über Funktions-Prototypen angegeben. Die Spezifikation der Operationen ist allenfalls in Kommentaren festgehalten. Die konkrete Implementierung der Operationen ist für andere Komponenten auch nicht einsehbar. Hierbei spricht man auch von **prozeduraler Abstraktion**. Dies erlaubt es, diese Implementierung unabhängig von den anderen Komponenten austauschbar zu machen. So könnte zum Beispiel die Evaluierung des Polynoms zunächst einfach gemäß der Formel $\sum_{i=0}^{n} a_i x^i$ erfolgen und später durch das Horner-Schema ausgetauscht werden, ohne dass andere Komponenten in irgendeiner Weise davon berührt wären.

Die Unterprogramme operieren auf einem nicht näher spezifizierten Datentyp `poly_t`, der in C durch einen unvollständigen Verbund implementiert wird. Somit kann eine andere Komponente keinen Bezug auf die Implementierung eines Polynoms nehmen und sich somit auch nicht von dieser abhängig machen. Das Verbergen der Interna von Datenstrukturen nennt man auch **Datenabstraktion**. Innerhalb des Moduls (in der Übersetzungseinheit `poly.c`) befindet sich die konkrete Implementierung eines Polynoms. In dieser Implementierung werden die Koeffizienten in einer Reihung gespeichert. Die Länge der Reihung wird durch das Feld `degree` als Zahl dargestellt.

Betrachten Sie nun die nicht-modulare Implementierung in Abbildung 4.1, die die Interna ihrer Datenstruktur nach außen verrät, und sie nicht vollständig hinter einer Schnittstelle verbirgt:

Das Modul bietet keine Operation mehr zum Ermitteln des Grades des Polynoms. Der Nutzer des Moduls ist also geneigt, diesen direkt aus dem Feld `degree` des Verbunds `poly_t` zu nehmen. Möchte man irgendwann die Länge der Koeffizienten-Reihung nicht mehr durch die Angabe des Grades darstellen, sondern durch einen Ende-Zeiger (siehe Abschnitt 2.6), so muss man alle Code-Stellen, die noch das Feld `degree` verwenden, anpassen. Das Polynom-Modul kann in dieser Situation also nur schlecht unabhängig von den anderen Modulen des Programms geändert werden.

Abbildung 4.1: Nicht-modulare Implementierung der Polynome. Sie verrät Implementierungsdetails nach außen.

```
#ifndef POLY_H
#define POLY_H

typedef struct {
    unsigned degree;
    int *coeffs;
} poly_t;

int poly_eval( /* ... */ );
#endif /* POLY_H */
```

4.2 Abstrakte Datentypen

Ein abstrakter Datentyp (ADT) (engl. abstract data type) besteht aus einem Typ, einer Menge von Operationen (auf Werten dieses Typs) und einer Spezifikation[3]. „Abstrakt“ bedeutet hier, dass die Schnittstelle und die Spezifikation des ADT keine Information über die konkrete Implementierung des ADT enthält. Die Ziele sind dieselben wie bei der Modularisierung: Dadurch, dass wir die Implementierung verbergen, können sich andere Programmteile davon nicht abhängig machen und die Implementierung kann unabhängig von den anderen Programmteilen geändert und wiederverwendet werden.

[3] Spezifikationen lassen sich in den meisten Programmiersprachen nicht wirklich ausdrücken, da deren Typsysteme zu schwach sind.

Das Polynom-Modul (Beispiel 3.10) ist eine C-Implementierung des ADT Polynom. Keinerlei Interna dringen nach außen. Betrachten wir ein weiteres, einfaches Beispiel: die Liste. Abbildung 4.2 zeigt die

list.h

```c
#ifndef LIST_H
#define LIST_H

typedef struct list_t list_t;

/* Creates a list */
list_t *list_create(void);

/* Frees a list */
void list_free(list_t *l);

/* Adds an int to a list */
void list_add(list_t *l, int i);

/* Removes all ints of a certain value */
void list_remove(list_t *l, int i);

/* Checks, if an int is in the list */
void list_contains(list_t const *l, int i);

/* Determines the length of the list */
unsigned list_size(list_t const *l);

/* Calls f on every list element in order of insertion. */
void list_iterate(list_t const *l, void (f)(int v));

#endif /* LIST_H */
```

Abbildung 4.2: Schnittstelle des abstrakten Datentyps Liste (von ints) in C.

Schnittstelle des ADT Liste. Man kann eine Zahl an eine Liste anhängen, alle Einträge eines bestimmten Wertes löschen, die Länge der Liste erfahren, usw. Im Kommentar zu `list_iterate` ist festgehalten, dass die übergebene Funktion auf jedem Listenelement aufgerufen wird und

list_array.c

```c
#include "list.h"

struct list_t {
    int *arr;
    unsigned capacity;
    unsigned size;
};

void list_add(list_t *l, int i) {
    if (l->size == l->capacity) {
        l->capacity *= 2;
        l->arr = realloc(l->arr, l->capacity
                * sizeof(l->arr[0]));
    }
    l->arr[l->size++] = i;
}

unsigned list_size(list_t const *l)
{
    return l->size;
}
```

list_linked.c

```c
#include "list.h"

struct list_t {
    list_t *prev, *next;
    int value;
};

void list_add(list_t *l, int i)
{
    list_t *n = malloc(sizeof(*n));
    l->prev->next = n;
    n->prev = l->prev;
    l->prev = n;
    n->next = l;
}

unsigned list_size(list_t const *l)
{
    unsigned len = 0;
    for (list_t *i = l->next; i != l; i=i->next)
        len++;
    return len;
}
```

Abbildung 4.3: Zwei Implementierungen des abstrakten Datentyps Liste. Eine durch eine Reihung, die bei Bedarf vergrößert wird und eine durch eine verkettete Liste.

zwar in der Reihenfolge des Einfügens des Elements. Dies ist ein Teil der Spezifikation des ADTs, der nur sehr vage (durch einen Kommentar) ausgedrückt werden kann.

Abbildung 4.3 zeigt einen Auszug zweier möglicher Implementierungen des abstrakten Datentyps Liste. Die eine (list_array.c) verwendet eine Reihung, die bei Bedarf vergrößert wird, um die Zahlen abzulegen. Die andere (list_linked.c) verwendet eine verkettete Liste, bei der für jede Zahl ein Verbund alloziert wird, der entsprechend mit der Liste verzeigert wird. Man sieht, dass C die Implementierung von ADTs dahingehend unterstützt, dass man Funktions-Deklarationen von Definitionen trennen kann (prozedurale Abstraktion) und Zeiger auf unvollständige Verbunde vereinbart werden können. Dadurch können die Implementierungsdetails in den einzelnen Übersetzungseinheiten „versteckt“ werden. Benutzer der Liste können nicht unterscheiden, welche der beiden Implementierungen tatsächlich zum Einsatz kommt. Des Weiteren kann die eine Implementierung durch die andere ersetzt werden, ohne dass andere Module des Programms davon in Mitleidenschaft gezogen werden.

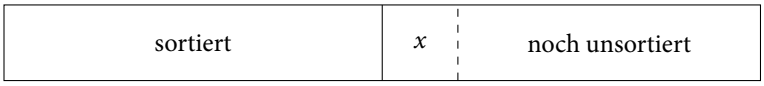
Allerdings ist es in C sehr umständlich, beide Implementierungen gleichzeitig zu verwenden und in einem Programm zu verwenden. Dies wird in objektorientierten Sprachen wie Java durch ein erweitertes Typsystem, das Subtyping bietet, elegant ermöglicht.

4.3 Schrittweise Verfeinerung

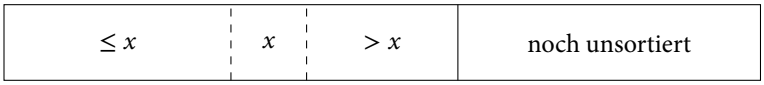
Bei der schrittweisen Verfeinerung beginnt man mit einem Grobentwurf. Dort wird die zu implementierende Aufgabe zunächst dadurch gelöst, dass man sie in kleinere Aufgaben zerteilt, sprich verfeinert. Die kleineren Aufgaben werden dann wieder in Unteraufgaben unterteilt, die dann getrennt von einander verfeinert werden. Zunächst ist der Code (unter Umständen) noch gar nicht kompilierbar und dient eher als Spielfeld für den Entwurf. Ein gängiges Vorgehen ist, pro Aufgabe ein Unterprogramm zu erstellen, das dann die Unterprogramme der Unteraufgaben ruft. Hierbei spricht man auch von **prozeduraler Abstraktion**.

Es ist zu beachten, dass nicht nur der Code, sondern oft auch die Datenstrukturen verfeinert werden. Während die Implementierung detaillierter wird, wird oft auch klarer, wie die Daten strukturiert werden müssen. Wir betrachten nun zwei Beispiele für die Verfeinerung von Code. Datenverfeinerung findet sich in einem größeren Beispiel in Abschnitt 4.4

Beispiel 4.1 (Sortieren durch Einfügen)**.** Betrachten wir nochmals das Sortieren durch Einfügen (→ Beispiel 2.8). Die Idee war, die zu sortierende Reihung in zwei Teile zu teilen. Einen „linken" Teil, der schon sortiert ist und einen „rechten", der noch zu sortieren ist.



In jedem Schritt wird nun der linke Teil um ein Element größer gemacht, in dem das erste Element aus dem rechten Teil im linken Teil passend einsortiert wird:



Dies wird solange wiederholt, bis der rechte Teil leer ist.

Wollen wir dieses Verfahren mittels schrittweiser Verfeinerung implementieren, so gehen wir wie folgt vor: Zu Beginn ist der linke Teil ein Element groß, da ein Element immer sortiert ist. Also beginnen wir mit einer Schleife, die nacheinander die Elemente $1 \ldots n-1$ in den linken Teil einfügt. Die Operationen „Suche den geeigneten Platz um das *i*te Element einzufügen" und „Bewege alle Elemente von diesem Platz eins nach rechts" lassen wir zunächst abstrakt, indem wir sie durch Unterprogrammaufrufe implementieren.

```
void insertsort(int *a, unsigned n)
{
    for (unsigned i = 1; i < n; i++) {
        int elm = a[i];
        unsigned k = search_index_to_insert(a, i, elm);
        move_right(a, k, i);
        a[k] = elm; /* insert element */
    }
}
```

Dieses Vorgehen hilft uns, uns der Implementierung schrittweise zu nähern. Des Weiteren erhöht es die Lesbarkeit der Implementierung: Unterprogrammaufrufe sind leichter zu lesen als der Code, den die Unterprogramme enthalten.

Beim schrittweisen Verfeinern müssen wir uns bei jeder Verfeinerung Gedanken machen, welches Unterprogramm welche Daten benötigt. Beispielsweise muss das Unterprogramm `search_index_to_insert` auf die Reihung zugreifen können und braucht dazu den Zeiger auf deren erstes Element, die Länge des sortierten Bereichs, sowie den Wert des Elements, das eingefügt werden soll.

Die beiden Operationen werden im folgenden weiter verfeinert.

```
unsigned search_index_to_insert(int *a, unsigned n_sorted, int elm)
{
    for (unsigned i = 0; i < n_sorted; i++)
        if (a[i] > elm)
            return i;
    return n_sorted;
}

void move_right(int *a, unsigned from, unsigned to)
{
    for (unsigned i = to; i > from; i--)
        a[i] = a[i-1];
}
```

⌟

Beispiel 4.2 (Sortieren durch Mischen (engl. merge sort))**.** Ein fundamentales Konzept im Algorithmenentwurf ist Teile und Herrsche (engl. divide and conquer). Hier bei teilen wir das zu lösende Probleme in Teilprobleme der gleichen Bauart und kombinieren die Teillösungen zu einer Lösung des Gesamtproblems. Bezüglich der schrittweisen Verfeinerung führt das dazu, dass man das Unterprogramm, das man verfeinert in seiner Verfeinerung verwendet.

Ein prominentes Beispiel hierfür ist das Sortieren durch Mischen. Hierbei sortiert man zunächst beide Hälften der Reihung getrennt und kombiniert die beiden Hälften zu einer sortierten Gesamtreihung.

```
void mergesort(int *a, unsigned from, unsigned to)
{
    assert(from <= to);
    if (to - from <= 1)
        return;

    unsigned middle = from + (to - from) / 2;
    mergesort(a, from, middle);
    mergesort(a, middle, to);
    merge(a, from, middle, to);
}
```

Man sieht, dass das Sortieren der beiden Reihungshälften durch **rekursive** Aufrufe erfolgt. Der Aufruf an `merge` fügt die beiden sortierten Hälften dann zu einer sortieren Gesamtrehihung zusammen.

```
void merge(int *a, unsigned left,
           unsigned middle, unsigned right)
{
    unsigned n = right - left;
    unsigned l = left;
    unsigned m = middle;
    unsigned i = 0;
    int merge[n];

    while (l < middle && m < right)
        merge[i++] = arr[l] <= arr[m] ? arr[l++] : arr[m++];

    while (l < middle)
        merge[i++] = arr[l++];

    while (m < right)
        merge[i++] = arr[m++];

    memcpy(a + left, merge, sizeof(a[0]) * n);
}
```

Zu beachten ist, dass man in der Praxis auch Datenstrukturen verfeinert. Hierbei läßt man die konkrete Implementierung der einzelnen Datenstrukturen **abstrakt** (siehe Abschnitt 4.2), das heißt, man legt nicht fest, wie genau die Daten tatsächlich organisiert werden. Vielmehr definiert man sich eine **Schnittstelle**, das heißt einen Satz von grundlegenden Operationen, die für alle Interaktionen mit der Datenstruktur stehen (zum Beispiel: Hänge an eine Liste an oder füge in eine Menge ein). Nach und nach können dann verschiedene Implementierungen für diese abstrakten Datentypen erstellt werden. Idealerweise werden diese von ihren Verwendern abgekapselt, so dass die Verwender nur über die jeweilige Schnittstelle mit ihnen interagieren. Dadurch bleiben die Implementierungen austauschbar.

4.4 Ein komplexeres Beispiel

Betrachten wir ein komplexeres Beispiel um zu verstehen, wie wir ein Programm durch Verfeinerung, Abstraktion und Modularisierung entwickeln. Unsere Aufgabe ist es, einen Graphen zu färben.

Ein ungerichteter Graph $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten E. Eine Kante ist eine zweielementige Menge von Knoten aus der Menge V:

$$E \subseteq \{S \in \mathscr{P}(V) \mid |S| = 2\}$$

Ist $\{v, w\} \in E$, so schreiben wir auch kurz $vw \in E$ und sagen, dass v und w benachbart sind.

Graphen spielen bei der Modellierung vieler Probleme eine wichtige Rolle. Ein klassisches Beispiel aus der Graphentheorie ist das Färben einer Landkarte: Die Länder einer Landkarte sollen so gefärbt werden, dass keine zwei benachbarten Länder die gleiche Farbe bekommen. Dieses Problem lässt sich graphentheoretisch modellieren. Wir erstellen aus der Landkarte einen Graphen G wie folgt: Die Menge der Knoten des Graphen ist die Menge der Länder. Sind zwei Länder benachbart, so existiert zwischen den Knoten der Länder eine Kante. Abbildung 4.4 zeigt den Nachbarschaftsgraphen einiger Länder Mitteleuropas.

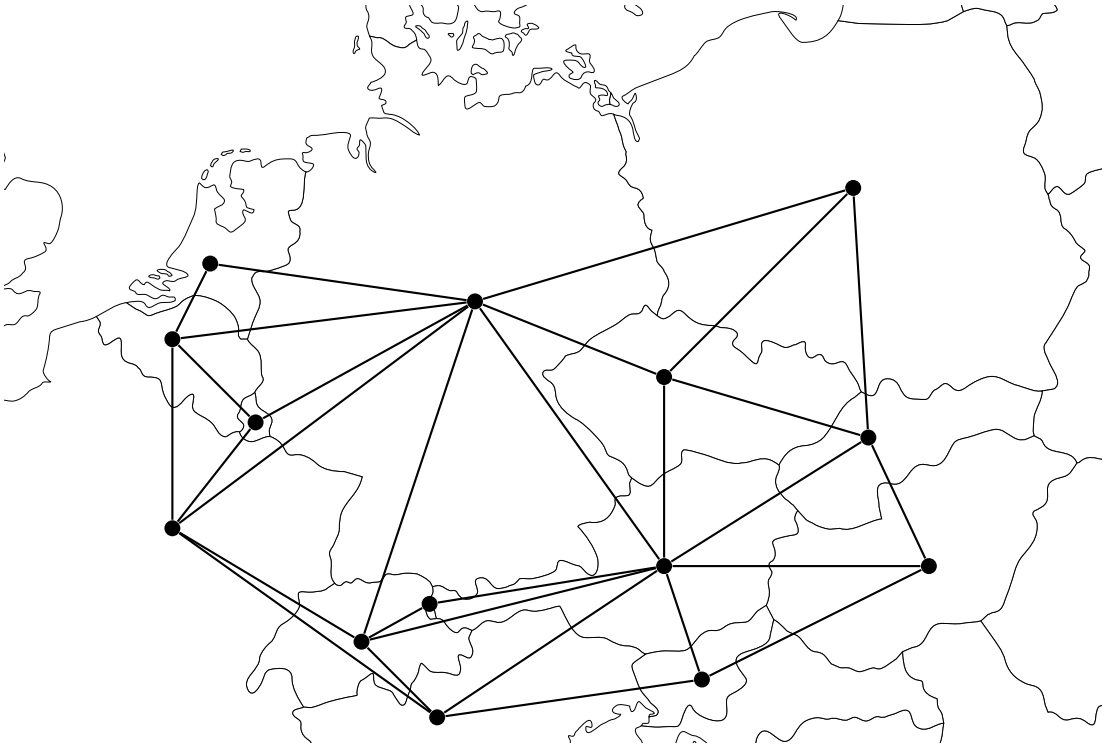


Abbildung 4.4: Karte Mitteleuropas und der Nachbarschaftsgraph der Länder (Karte basiert auf `http://www.d-maps.com/m/europa/europemin/europemin10.pdf`).

Eine k-Färbung eines Graphen ist eine Funktion

$$c : V \to \{1, \dots, k\} \text{ mit } c(v) \neq c(w) \text{ für alle } vw \in E$$

die benachbarten Knoten unterschiedliche Farben zuweist. Färbungsprobleme sind schwer, wenn wir keine Voraussetzungen an die Struktur von G stellen. Für folgende Probleme nicht bekannt, ob es einen Algorithmus gibt, der sie in polynomieller Zeit entscheidet/berechnet: Hat ein gegebener Graph eine k-Färbung? Berechne das minimale k für das ein Graph eine k-Färbung besitzt. Ermittle eine k-Färbung eines gegebenen Graphen G.

Wir wollen hier ein Verfahren entwickeln, dass für **manche** Graphen k-Färbungen berechnen kann. Das Verfahren ist eine Heuristik, also nicht vollständig: Es kann auf Graphen scheitern, für die eine k-Färbung existiert. Sei also im folgenden ein k gegeben.

Wiederhole folgende Schritte, bis der Graph leer ist: Suche einen Knoten v mit weniger als k Nachbarn. Wir sagen, ein solcher Knoten hat **insignifikanten Grad**. Existiert kein solcher Knoten, so brechen wir ab und geben auf. Andernfalls entfernen wir v und alle seine eingehenden Kanten. Wenn alle Knoten entfernt werden konnten, fügen wir sie in umgekehrter Reihenfolge wieder ein und weisen ihnen beim Einfügen eine Farbe zu. Das folgende Lemma zeigt, dass wir mit diesem Verfahren jedem Knoten beim Einfügen eine Farbe zuweisen können.

Lemma 4.1. *Gegeben eine Anzahl an Farben k, ein ungerichteter Graph $G = (V, E)$ und $v \in V$ ein Knoten mit weniger als k Nachbarn. Angenommen, es existiert eine k-Färbung c von $G - \{v\}$, dann ist auch G k-färbbar.*

Beweis. Da v nach Voraussetzung weniger als k Nachbarn hat, gibt es eine Farbe k', die von keinem seiner Nachbarn verwendet wird. Die Färbung von $G - \{v\}$ kann dann zu einer Färbung von G erweitert werden, in dem v die Farbe k' zuteilt. □

Kann der Graph also nach dem Kriterium „Eliminiere Knoten mit weniger als k Nachbarn“ vollständig geleert werden, finden wir eine k-Färbung. Abbildung 4.5 zeigt die Färbung eines einfachen Graphen mit dem gerade besprochenen Algorithmus.

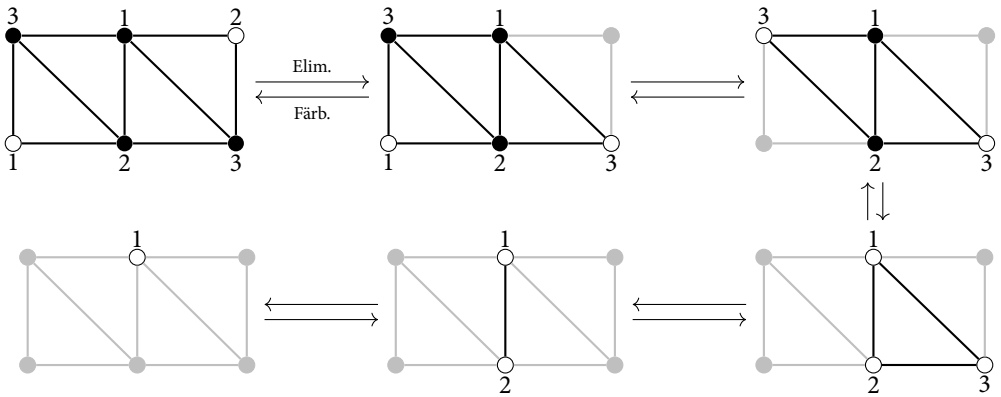


Abbildung 4.5: Ein Beispielgraph mit sechs Knoten. Insignifikante Knoten sind weiß gekennzeichnet. Ausgegraute Knoten und Kanten sind eliminiert. Die Abbildung zeigt *eine* mögliche Sequenz von Eliminerungen von insignifikanten Knoten. Die Zahlen über und unter den Knoten sind die beim Wiedereinfügen zugewiesenen Farben.

4.4.1 Grobentwurf

```
#include "graph.h"

int color(graph_t const *g, unsigned k, /* coloring */)
{
    /* get the number of nodes in the graph */
    unsigned n = graph_n_nodes(g);

    /* store the nodes in the order in which they are eliminated */
    list_t *elimination_order = list_create();

    for (unsigned i = 0; i < n; i++) {
        unsigned node = find_insignificant_node(g, k);

        if (/* none found */)
            return 0;

        list_add(node);
        eliminate_from_graph(g, node);
    }

    /* here, we have eliminated all nodes */

    /* for every node in elimination order, from back to front */
    for (unsigned i = 0; i < n; i++) {
        unsigned node = list_get(n - i);

        /* find color not occupied by any
           of the node's already inserted neighbors */
        unsigned color = get_free_color(g, k, node);
        assert(color < k && "Because of Lemma 4.1");

        /* reinsert node into the graph, and assign color */
        assign_color(node, color);
        reinsert_into_graph(g, node);
    }

    /* we succeeded to color the graph */
    return 1;
}
```

Wir lassen die konkrete Implementierung der Datenstruktur des Graphen noch offen. Wir über geben einen Zeiger auf einen anonymen Verbund. Später vereinbaren wir Unterprogramme, die auf dem Graph die gewünschten Operationen durchführen.

Wir verwalten die eliminierten Knoten in einer Liste. Früher entfernte Knoten haben den kleineren Index.

Dieses Unterprogramm muss einen Knoten finden, der weniger als k Nachbarn hat. Allerdings dürfen Knoten, die bereits eliminiert wurden, nicht berücksichtigt werden. Wir müssen uns auch überlegen, was das Unterprogramm zurückgeben soll, wenn kein solcher Knoten vorhanden ist.

Es ist noch nicht klar, wie wir das Entfernen und Wiedereinfügen der Knoten organisieren. Wenn wir einen Knoten wieder einfügen, müssen wir die Nachbarn kennen, die bereits eingefügt wurden, um eine passende Farbe ermitteln zu können. Wir können beim Entfernen des Knotens nicht einfach alle seine eingehenden Kanten mit löschen, da wir diese Information beim Einfügen wieder benötigen.

Wie genau wir die zugeteilten Farben verwalten, ist auch noch nicht klar.

Um die Darstellung zu vereinfachen, nehmen wir an, dass ein Knoten durch eine Zahl zwischen 0 und $n - 1$ dargestellt wird. Die Implementierung soll das Ergebnis dann in eine Reihung schreiben: `coloring[i] == c` heißt Knoten i hat Farbe c.

Der Graph soll durch das Färben nicht verändert werden. Die Implementierung der Färbung ist von der Implementierung des (abstrakten) Datentyps Graph unabhängig. Das nutzen wir aus und kapseln den

Graph in ein eigenes Modul und verbergen seine Interna hinter einer Schnittstelle.

4.4.2 Verfeinerung 1: Datenverfeinerung der Liste

```
#include "graph.h"

int color(graph_t const *g, unsigned k, /* coloring */)
{
    /* get the number of nodes in the graph */
    unsigned n = graph_n_nodes(g);

    /* store the nodes in the order in which they are eliminated */
    unsigned elimination_order[n];

    for (unsigned i = 0; i < n; i++) {
        unsigned node = find_insignificant_node(g, k);

        if (/* none found */)
            return 0;

        elimination_order[i++] = node;
        eliminate_from_graph(g, node);
    }

    /* here, we have eliminated all nodes */

    /* for every node in elimination order, from back to front */
    for (unsigned i = 0; i < n; i++) {
        unsigned node = elimination_order[n - i];

        /* find color not occupied by any
           of the node's already inserted neighbors */
        unsigned color = get_free_color(g, k, node);
        assert(color < k && "Because of Lemma 4.1");

        /* reinsert node into the graph, and assign color */
        assign_color(node, color);
        reinsert_into_graph(g, node);
    }

    /* we succeeded to color the graph */
    return 1;
}
```

Wir verfeinern die (abstrakte) Liste zu einer Reihung. Da wir a priori wissen, dass wir n Knoten einfügen werden, ist die Reihung die geeignete Datenstruktur. Hier geben wir also ein wenig Abstraktion auf: Wir können die Implementierung der Datenstruktur nicht mehr ohne Änderung des Verwender-Codes austauschen. Das ist hier akzeptabel, da es keine bessere Implementierung als die Reihung gibt.

4.4.3 Verfeinerung 2: Verwaltung der eliminierten Knoten

```
#include "graph.h"

int color(graph_t const *g, unsigned k, unsigned *coloring)
{
    /* get the number of nodes in the graph */
    unsigned n = graph_n_nodes(g);

    /* store the nodes in the order in which they are eliminated */
    unsigned elimination_order[n];

    /* Keep a set of nodes that have been eliminated */
    bool eliminated[n] = { false };

    for (unsigned i = 0; i < n; i++) {
        unsigned node = find_insignificant_node(g, k, eliminated);

        if (node >= n)
            return 0;

        elimination_order[i++] = node;
        eliminated[node] = true;
    }

    /* here, we have eliminated all nodes */

    /* for every node in elimination order, from back to front */
    for (unsigned i = 0; i < n; i++) {
        unsigned node = elimination_order[n - i];

        /* find color not occupied by any
           of the node's already inserted neighbors */
        unsigned color = get_free_color(g, k, coloring,
                                        node, eliminated);
        assert(color < k && "Because of Lemma 4.1");

        /* reinsert node into the graph, and assign color */
        coloring[node] = color;
        eliminated[node] = false;
    }

    /* we succeeded to color the graph */
    return 1;
}
```

Wir konkretisieren die Datenstruktur, die die Färbung speichert. Wir verwenden eine Reihung von **`unsigned`**. Da ein Knoten eine Zahl zwischen 0 und $n - 1$ ist, können wir diese verwenden, um die Reihung zu indizieren. `colors[node]` speichert dann die Farbe eines Knoten `node`.

Die Funktion, die uns den nächsten Knoten insignifikanten Grades liefert muss die Menge der bereits entfernten Knoten mitbekommen, da die bereits entfernten Knoten bei der Berechnung des Grades nicht berücksichtigt werden. Wir legen fest, dass die Funktion den ungültigen Knoten n zurückgibt, wenn kein insignifikanter Knoten gefunden werden konnte.

Wir verwalten die Menge der eliminierten Knoten in einer Reihung von Wahrheitswerten. Ein Knoten `node` ist eliminiert, genau dann wenn `eliminated[node] == true`.

Die Funktion, die eine freie Farbe auf Basis der bereits eingefügten Nachbarn ermittelt, benötigt sowohl die Färbung `coloring` als auch die Menge der entfernten Knoten `eliminated`.

4.4.4 Verfeinerung 3: Finden der freien Farbe

```
unsigned get_free_color(graph_t const *g, unsigned k,
                        unsigned *coloring, unsigned node,
                        bool const *eliminated)
{
    /* create an array of boolean variables.
       color_taken[c] == true indicates that color c has been taken.
       Initially, no color is taken */
    bool color_taken[k] = { false };

    /* for every neighbor of the node */
    graph_foreach_neighbor(g, node, neighbor) {
        if (! eliminated[neighbor]) {
            /* if the neighbor has not been eliminated
               mark its color as taken */

            unsigned color = coloring[neighbor];
            assert(color < k && "re-inserted nodes have a color");
            color_taken[color] = true;
        }
    }

    /* find the first color that has not been taken */
    for (unsigned i = 0; i < k; i++) {
        if (! color_taken[i])
            return i;
    }

    /* if there was no free color, return the illegal color k */
    return k;
}
```

Ähnlich zur Menge der eliminierten Knoten speichern wir die Menge der von den Nachbarn belegten Farben in einer Reihung von Wahrheitswerten. Ist ein Eintrag der Reihung `true`, dann ist die entsprechende Farbe von einem Nachbarn belegt.

Das Iterationskonstrukt sind ungewöhlich und nicht nach C-Syntax aus. Es soll alle Nachbarn des Knoten `node` besuchen und die Variable `neighbor` in jeder Iteration entsprechend belegen. Wir überlassen die Details es einer späteren Verfeinerung.

Hier suchen wir die erste Farbe, die nicht belegt ist. Da der Index in die Reihung `colors_taken` den Knoten angibt, beenden wir das Unterprogramm mit dem ersten Index `i`, für den `colors_taken[i] == false`.

4.4.5 Verfeinerung 4: Finden eines insignifikanten Knoten

```
unsigned find_significant_node(graph_t const *g, unsigned k,
                               bool const *eliminated);
{
    unsigned n = graph_n_nodes(g);

    /* for each node in the graph */
    for (unsigned node = 0; node < n; node++) {
        unsigned n_neighbors = 0;

        /* count the neighbors that have not been eliminated */
        graph_foreach_neighbor(g, node, neighbor) {
            n_neighbors += eliminated[neighbor] ? 0 : 1;
        }

        /* return the current node if it has less than k
           non-eliminated neighbors */
        if (n_neighbors < k)
            return node;
    }

    return n;
}
```

Wir besuchen jeden Knoten und zählen seine noch nicht eliminierten Nachbarn.

Hat ein Knoten weniger als *k* noch nicht eliminierte Nachbarn, so können wir ihn als Knoten insignifikanten Grades zurückliefern.

Konnten wir keinen Knoten insignifikanten Grades finden, so kehren wir mit dem illegalen Knoten *n* zurück.

Die Implementierung ist einfach gehalten und noch sehr ineffizient. Bei jedem Aufruf wird für jeden Knoten jeder Nachbar besucht. Die Laufzeit ist also $O(|E|)$. Ist der Graph „dicht besetzt", so beträgt die Anzahl der Nachbarn pro Knoten $O(|V|)$ und $O(|E|) = O(|V|^2)$.

Eine alternative Implementierung berechnet die Grade der Knoten einmal vor Beginn der Färbung. Beim Löschen eines Knoten wird dann der Grad der Nachbarn des Knotens um eins verringert. Dadurch muss man die Grade nicht immer neu berechnen und `find_insignificant_node` kann in $O(|V|)$ laufen[4].

Dies kann durch eine kompliziertere Datenstruktur noch weiter verbessert werden: Nach dem initialen Berechnen der Grade sortiert man eine Reihung von Knoten nach ihrem Grad. Dann merkt man sich in einer weiteren Reihung die Indices der jeweils ersten Knoten eines Grades. So kann man in $O(1)$ immer einen insignifikanten Knoten finden. Entfernt man einen Knoten, so muss man die Knoten-Reihung und die Index-Reihung entsprechend anpassen, da sich die Grade der Nachbarn ja um eins verringern.

[4] Es muss immer noch jeder Knoten einmal besucht werden, um einen mit insignifikantem Grad zu finden

4.4.6 Schnittstelle des Graphen

```
#ifndef GRAPH_H
#define GRAPH_H

struct graph_t;
typedef struct graph_t graph_t;

unsigned graph_n_nodes(graph_t const *g);

void const *graph_first_neighbor(graph_t const *g,
                                 unsigned node);
void const *graph_next_neighbor(graph_t const *g, unsigned node,
                                void const *curr);
bool graph_done_neighbor(graph_t const *g, unsigned node,
                         void const *curr);

unsigned graph_neighbor_get(void const *curr);

#define graph_foreach_neighbor(g, node, it) \
for (void const *it = graph_first_neighbor(g, node); \
     !graph_done_neighbor(g, node, it); \
     it = graph_next_neighbor(g, node, it))

#endif /* GRAPH_H */
```

Diese Funktionen implementieren das Iterieren über die Nachbarn eines Knoten. Mit `first_neighbor` können wir uns einen Zeiger beschaffen, über den wir mittels `neighbor_get` den ersten Nachbarn holen können. Diesen Zeiger nennt man auch Iterator. Dieser **`void`** Zeiger abstrahiert (so gut man es mit C eben kann) von der Repräsentation der Nachbarn eines Knoten. Mit den anderen Funktionen kann man testen, ob es noch weitere Nachbarn gibt und kann den Iterator auf den nächsten Nachbarn rücken.

Wir führen hier nur die Funktionen auf, die wir für das Färben benötigen. Sicher hat der Graph weitere Unterprogramme zum Hinzufügen und vielleicht Löschen von Knoten und Kanten. Nötig sind auch Unterprogramme, die einen leeren Graph erstellen und wieder freigeben oder einen Graphen aus einer Datei einlesen, usw.

Hervorzuheben ist, dass die Schnittstelle kaum Interna der Datenstruktur des Graphen offenbart. Wir vereinbaren den Graph hier als **abstrakten Datentyp**. Wir wissen nicht, wie genau die Knoten und Kanten des Graphen gespeichert sind[5].

Die Spracheigenschaften von C, die uns das hier ermöglichen sind anonyme Verbunde und Kapselung (durch getrennte Übersetzung). Beim Ablaufen der Nachbarn läßt uns C aber im Stich. In einer Sprache, die funktionale Programmierung unterstützt, könnte man sich einer höherstufige Funktion `iterate` definieren, die eine Funktion als Argument nimmt, die sie dann auf jedem Nachbarn eines Knoten aufruft.

In C gibt es zwar auch **Funktionszeiger**, allerdings können diese nur Unterprogramme referenzieren. Anonyme Funktionen mit automatischem Binden der freien Variablen in einer Umgebung (engl. closure) bietet C nicht.

Will man keine Details der Datenstruktur des Graphen „nach außen" offenbaren, muss man das „Protokoll" des Abschreitens (auch Iteration genannt) der Nachbarn mühselig mit entsprechenden Funktionen implementieren. Immerhin, können wir die größten „Schmerzen" bei der Benutzung hinter einem Makro verbergen.

[5] Ist der Graph durch eine Liste von Kanten gegeben? Oder gibt es für jeden Knoten eine Reihung (oder eine verkettete Liste) von Nachbarn? Oder ist der Graph gar durch seine Inzidenzmatrix gegeben? Wir können es der Schnittstelle nicht ansehen.

C++11 unterstützt Funktionsabstraktion. Hier könnte man `iterate` wie folgt nutzen:

```
...
n_neighbors = 0;
iterate([] {
    n_neighbors += ...
});
```

schreiben.

5 Die Sprache C0

In diesem Kapitel diskutieren wir die Grundlagen der Formalisierung von imperativen Programmiersprachen an der Sprachfamilie C0, die an C angelehnt ist. Wir tun dies zum einen, um kennenzulernen, wie man Programmiersprachen mathematisch beschreibt. Zum anderen hilft uns die Formalisierung, die ganzen Details (Zeiger, Adressen, L- und R-Auswertung, etc.), die wir in Kapitel 3 informell besprochen haben, genau zu verstehen, in dem wir sie mathematisch beschreiben. Um es nicht zu komplex werden zu lassen, verzichten wir bei unseren Sprachen auf viele Eigenschaften von C.

Die Sprache C0 ist einfach gehalten, um die grundlegenden Techniken der Formalisierung von Programmiersprachen zu erlernen. C0 hat keine Unterprogramme (ein Programm besteht nur aus einer Folge von Anweisungen), nur den Datentyp **`int`**, keine Blockschachtelung, keine Nebenwirkungen in Ausdrücken, usw. In zwei darauffolgenden Abschnitten werden wir formale Semantik von C0 um Zeiger und Blockschachtelung erweitern und letztlich noch eine statische Semantik inklusive eines Typsystems besprechen.

Die Erweiterung auf Zeiger zeigt formal sehr deutlich den Unterschied zwischen L- und R-Auswertung, deren Verständnis auf intuitiver Ebene (wie in Abschnitt 3.7.2) einige Übung erfordert. Die Formalisierung der Blockschachtelung legt offen, wie genau neue Behälter entstehen und freigegeben werden, wie „dangling pointer“ entstehen und was es mit dem Verdecken von lokalen Variablen auf sich hat. Zu guter Letzt zeigen wir die Vorteile einer statischen Semantik, die eine große Menge von nicht sinnvollen Programmen **statisch** ausschließen kann.

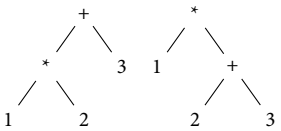
5.1 Die Syntax von Programmiersprachen

Bei der Formalisierung von Programmiersprachen unterscheidet man zwei verschiedene Arten von Syntax: Die **abstrakte Syntax** definiert die Struktur der Programme. Sie befasst sich damit, wie Konstrukte aus anderen Konstrukten zusammengesetzt werden[1]. In der abstrakten Syntax wird das Programm als **Baum**, dem sogenannten **abstrakten Syntaxbaum (engl. abstract syntax tree (AST))** dargestellt, dessen Knoten aus syntaktischen Konstrukten bestehen.

[1] Beispielsweise besteht ein Ausdruck aus Teilausdrücken und einem Operator. Oder eine While-Schleife besteht aus einem Ausdruck und einer Anweisung.

Die konkrete Syntax definiert, wie die Programme einer Sprache als Folge von Zeichen kodiert werden. In diesem Kapitel interessiert uns nur die abstrakte Syntax, da sie rein die Struktur der Programme wiedergibt und nicht mit dafür irrelevanten „syntaktischen Artefakten" wie Klammerung und Trennzeichen verunreinigt ist[2].

[2] Diese Trennzeichen sind oft nötig, da bei einer linearen Darstellung von Bäumen Mehrdeutigkeiten entstehen. So kann die Zeichenfolge $1 * 2 + 3$ zunächst folgende Syntaxbäume meinen:



Erst durch Regeln wie „Punkt vor Strich", Assoziativität, etc. kann einem Infix-notierten Ausdruck eindeutig ein AST zugeordnet werden.

Bemerkung 5.1 (Syntaxanalyse)**.** Der Vorgang, der aus der Zeichenfolge der konkreten Syntax den abstrakten Syntaxbaum erzeugt, heißt **Syntaxanalyse**. Er besteht typischerweise aus der **lexikalischen Analyse (engl. lexing)**, dem Zusammenfassen der Buchstaben zu sogenannten **Symbolen (engl. token)**, und dem **Zerteilen (engl. parsing)**, das den Symbolstrom in einen AST übersetzt. ⌟

Die abstrakte Syntax definieren wir durch eine **Syntaxdefinition**, die die folgende Form hat:

Kategorie		Form	Kommentar
$Expr \ni e$	$::=$	$e + e$	Addition
	$\mid$	$e * e$	Multiplikation
	$\mid$	c	Konstante
$Val \ni c$	$::=$	$0 \mid 1 \mid \cdots$	

Die Spalte "Kategorie" definiert eine bestimmte Art von syntaktischer Kategorie (z.B. Ausdruck, Anweisung, etc.). Pro Kategorie kann es mehrere Zeilen geben. Jede Zeile definiert ein Konstrukt der jeweiligen Kategorie. In der Spalte "Form" definiert man eine Notation für das Konstrukt. Hierbei verwendet man Variablen, die für Konstrukte stehen. Der Name der Variable muss in der Kategorie-Spalte definiert sein. Andere Zeichen kennzeichnen das syntaktische Konstrukt und werden verwendet, um es im AST zu kennzeichnen (vgl. den Ausdrucksbaum am Rand).

Bemerkung 5.2. Auf Papier notiert man die abstrakte Syntax oft in Anlehnung an die konkrete Syntax, da diese kompakter aufzuschreiben und zu erfassen ist, als ein abstrakter Syntaxbaum. Daher sehen die textuell notierten abstrakten Syntaxbäume ihrer konkreten Syntax in C sehr ähnlich. ⌟

5.2 Die abstrakte Syntax von C0

C0 besteht aus Anweisungen und Ausdrücken, die wir beide getrennt voneinander diskutieren wollen. Ein C0-Programm besteht aus einer Anweisung $s \in \mathit{Stmt}$. Diese kann aus mehreren Anweisungen zusammengesetzt sein. Um die Programme in abstrakter Syntax lesbarer zu machen, behalten wir auch noch einige der Trennzeichen der konkreten Syntax bei, wie zum Beispiel das Semikolon. Definition 5.1 zeigt die **abstrakte Syntax** der Anweisungen von C0[3].

Definition 5.1 (Die Anweisungssprache von C0)**.**

$\mathit{Stmt} \ni s$	::=	l `=` e`;`	Zuweisung
	|	$s_1 \_ s_2$	Hintereinander-Ausführung
	|	**`if`** `(`e`)` s_1 **`else`** s_2	Fallunterscheidung
	|	**`while`** `(`e`)` s	Schleife
	|	`abort();`	Abbruch des Programms
	|	`;`	Die leere Anweisung ⌟

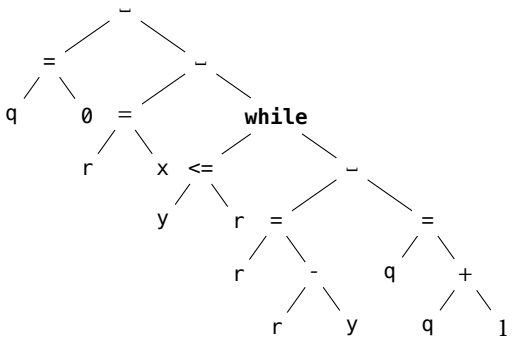
In den Anweisungen kommen an verschiedenen Stellen Ausdrücke vor, beispielsweise in der Abbruchbedingung der **`while`** Schleife, oder auf der rechten Seite der Zuweisung. Wir wollen Ausdrücke, die auf der linken Seite der Zuweisung auftreten können syntaktisch von denen, die auf der rechten Seite auftreten können, trennen.

Definition 5.2 (Die Ausdruckssprache von C0)**.**

$\mathit{LExpr} \ni l$	::=	x	Variable
$\mathit{Expr} \ni e$	::=	l	L-Wert
	|	c	Konstante
	|	$e_1\, o\, e_2$	Binärer Ausdruck
$\mathit{Op} \ni o$	::=	$r \mid m$	Operator
$\mathit{AOp} \ni r$	::=	+ | − | ⋯	Arithmetischer Operator
$\mathit{COp} \ni m$	::=	== | != | < | ⋯	Vergleichsoperator ⌟

Beispiel 5.1. Aufgrund der Art, wie wir die abstrakte Syntax von C0 definiert haben, ist das folgende Programm (gegeben in konkreter Syntax) gleichzeitig auch die lineare Darstellung seines abstrakten Syntaxbaumes.

```
q = 0;
r = x;
while (y <= r) {
    r = r - y;
    q = q + 1;
}
```



⌟

[3] Wie in Bemerkung 5.2 erwähnt, lehnen wir die Notation der Konstrukte der abstrakten Syntax an die konkrete Syntax an, um die (linearisierten) abstrakten Syntaxbäume leichter lesen zu können. So tragen die Strichpunkte und die runden Klammern in den Anweisungen von Definition 5.1 keinerlei Bedeutung und sind nur dazu da, mehr Familiarität mit C zu erzeugen.

Man könnte den if-Fall auch so definieren: $\texttt{If}(e, s_1, s_2)$.

5.3 Die Semantik von C0

5.3.1 Der Zustand

Informell haben wir bereits in Abschnitt 3.3 über die Struktur des Zustands eines C-Programms gesprochen. Hier definieren wir diesen jetzt formal. Wir gehen davon aus, dass eine Menge von Bezeichnern *Var*, und eine Menge von Adressen *Addr* gegeben ist. Wir nehmen keine innere Struktur dieser Mengen an; Die einzige Operation die wir auf Bezeichnern und Adressen zulassen, ist der Vergleich auf Gleichheit. Wir definieren die Menge der Werte zu $Val \coloneqq Addr \cup \texttt{int}$.

In Abschnitt 3.3 haben wir gesagt, dass eine Variable ein Bezeichner ist, der an die Adresse eines Behälters gebunden ist. Ein Bestandteil des Zustands ist also eine Abbildung von Variablen zu Adressen:

$$Var \rightharpoonup Addr$$

Diese Abbildung nennen wir **Variablenbelegung**. Diese Abbildung ist partiell, da eine Variable nicht zu jedem Zeitpunkt an einen Behälter gebunden sein muss.

In C0 sollen Behälter zunächst nur Adressen und Werte des Typs `int` aufnehmen können. Das heißt, dass wir, im Gegensatz zu C[4], keine Adressarithemtik unterstützen. In jedem Behälter befindet sich somit nur ein Wert. Daher modellieren wir die Behälter als Abbildung von Adressen zu Werten:

[4] siehe Abschnitt 3.3 und Abschnitt 3.9

$$Addr \rightharpoonup Val \cup \{?\}$$

Diese Abbildung heißt **Speicherbelegung** und definiert den Inhalt der Behälter. Der Wert ? ist der undefinierte Wert, der sich in dem Behälter befindet, bevor ein Wert in ihm abgelegt wurde. Ist eine Adresse nicht im Urbildbereich der Speicherbelegung, dann existiert für diese Adresse kein Behälter.

Der Zustand Σ eines C0-Programms besteht also aus einem Paar von Abbildungen:

$$\Sigma \coloneqq (Var \rightharpoonup Addr) \times (Addr \rightharpoonup Val \cup \{?\})$$

In der formalen Behandlung nennen wir die Komponenten des Paares meist ρ und μ. Damit die Darstellung nicht durch zu viele Klammern unübersichtlich wird, notieren wir das Paar (ρ, μ) im Folgenden als $\rho; \mu$. Kommt es uns auf die Komponenten im einzelnen nicht an, so kürzen wir $\rho; \mu$ einfach durch σ ab.

Definition 5.3 (Kurzschreibweise)**.** In vielen der Beispiele, die wir im Folgenden diskutieren werden, kommt es auf die konkrete Adresse nicht an. Wir verwenden daher die Kurzschreibweise $\sigma\, x \coloneqq (\mu \circ \rho)\, x$ oder verwenden, wenn wir den Zustand explizit angeben

$$\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$$

anstatt das Paar $\rho; \mu$ aus Variablen- und Speicherbelegung

$$\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}; \{a_1 \mapsto v_1, \ldots, a_n \mapsto v_n\}$$

konkret aufzuschreiben. Hierbei sind die a_i paarweise verschiedene Adressen. ⌟

5.3.2 Ausdrücke

Intuitiv haben wir eine Vorstellung davon, wie Ausdrücke ausgewertet werden. So wissen wir, dass der Ausdruck `x + 1` ausgewertet wird, indem man den Wert des Behälters liest, auf den `x` verweist und auf diesen Wert 1 aufaddiert. Der Wert eines Ausdrucks bestimmt sich also anhand des Ausdrucks selbst (gegeben als abstrakte Syntax) und eines Zustands, der die Variablen- und Speicherbelegung wiedergibt.

Analog zu unserer Diskussion in Abschnitt 3.7, gibt es zwei unterschiedliche Arten, Ausdrücke auszuwerten. Die L-Auswertung $[\![\cdot]\!]_L$, die **manche** Ausdrücke zu Adressen auswertet, und die R-Auswertung $[\![\cdot]\!]$. In C0 sind zunächst nur Bezeichner selbst L-auswertbar, da in C0 Adressen kein Resultat der R-Auswertung sein können, sprich, C0 keine Zeiger hat.

Definition 5.4 (Ausdrucksauswertung von C0)**.**

$$[\![\cdot]\!]\cdot : \mathit{Expr} \to \Sigma \rightharpoonup \mathit{Val} \qquad [\![\cdot]\!]_L\cdot : \mathit{LExpr} \to \Sigma \rightharpoonup \mathit{Addr}$$

$$\begin{aligned} [\![c]\!]\sigma &= c \\ [\![e_1\ o\ e_2]\!]\sigma &= [\![e_1]\!]\sigma\ o\ [\![e_2]\!]\sigma \\ [\![l]\!]\rho;\mu &= \mu\left([\![l]\!]_L\rho;\mu\right) \qquad [\![\mathtt{x}]\!]_L\rho;\mu = \rho\,\mathtt{x} \end{aligned}$$

⌟

Die beiden untersten Definitionen in Definition 5.4 sind die interessantesten. Pauschal wird die R-Auswertung eines L-Ausdrucks dazu definiert, dass man den L-Ausdruck zunächst L-auswertet, also seine Adresse bestimmt und dann aus dem Speicher an dieser Adresse lädt. Analog besteht die L-Auswertung eines Bezeichners daraus, nicht den Inhalt des entsprechenden Behälters zu laden, sondern dessen Adresse zu liefern. Insgesamt führt das dazu, dass die R-Auswertung eines Bezeichners zuächst die Adresse des assoziierten Behälters aus ρ ermittelt und dann den Inhalt des Behälters mittels μ liest.

Beide Funktionen zur Ausdrucksauswertung sind partiell, dass heißt, dass sie auf bestimmten Kombinationen von Ausdrücken und Zuständen nicht definiert sind. Ein einfaches Beispiel hierfür ist die Division durch 0. Die Funktion $[\![e/0]\!]$ hat für keinen Ausdruck e unter keinen Zustand ein Bild. Genauso hat $[\![\mathtt{x}]\!]_L$ auf einem Zustand kein Bild, in dem `x` nicht an eine Adresse gebunden ist.

5.3.3 Anweisungen

Wir definieren die Ausführung eines C0-Programms durch eine Folge von sogenannten **Konfigurationen**. Jeder Schritt der Ausführung (vorausgesetzt, das Programm kann noch einen Schritt machen) überführt eine Konfiguration in eine neue Konfiguration.

Definition 5.5 (Konfiguration)**.** Eine Konfiguration k ist ein Element der Menge $K = \{\lightning\} \cup \Sigma \cup (\mathit{Stmt} \times \Sigma)$ ⌟

Eine Konfiguration k ist also entweder:

1. $\lightning$ um anzudeuten, dass das Programm „abgebrochen“ ist.
2. Ein Zustand $\sigma \in \Sigma$ um auszudrücken, dass die Programmausführung im Zustand σ beendet wurde.
3. Ein Paar $\langle s \mid \sigma \rangle$ aus einem noch abzuarbeitenden Restprogramm und dem aktuellen Zustand.

Die Semantik einer Programmiersprache definiert nun, welche Konfigurationen ineinander übergehen können. Dadurch wird beschrieben, welche Anweisungen noch abzuarbeiten sind und wie sich der Zustand (der Inhalt der Variablen) ändert. Mathematisch definiert man die Semantik als zweistellige Relation $\to$ zwischen Konfigurationen. Definition 5.6 definiert eine sogenannte **operationale Semantik** für unsere kleine Sprache. Sie besteht aus den Axiomen [Empty], …, [While] und den Schlussregeln [Exec], [Crash] und [Subst]. Das Axiom [Abort] ist im späteren Verlauf des Buchs von Interesse.

Definition 5.6 (Operationale Semantik der Anweisungen von C0)**.**

[Empty] $\quad \langle \texttt{;} \mid \sigma \rangle \to \sigma$

[Assign] $\quad \langle l \texttt{ = } e\texttt{;} \mid (\rho, \mu) \rangle \to (\rho, \mu[a \mapsto v])$ falls $[\![e]\!]\sigma = v \in \mathit{Val}$ und $[\![l]\!]_L\sigma = a \in \mathit{Addr}$

[IfTrue] $\quad \langle \texttt{if (}e\texttt{) }s_1\texttt{ else }s_2 \mid \sigma \rangle \to \langle s_1 \mid \sigma \rangle$ falls $[\![e]\!]\,\sigma \neq 0$

[IfFalse] $\quad \langle \texttt{if (}e\texttt{) }s_1\texttt{ else }s_2 \mid \sigma \rangle \to \langle s_2 \mid \sigma \rangle$ falls $[\![e]\!]\,\sigma = 0$

[While] $\quad \langle \texttt{while (}e\texttt{) }s \mid \sigma \rangle \to \langle \texttt{if (}e\texttt{) \{}s\texttt{ while (}e\texttt{) }s\texttt{\} else ;} \mid \sigma \rangle$

[Abort] $\quad \langle \texttt{abort();} \mid \sigma \rangle \to \lightning$

$$\text{[Exec]}\ \frac{\langle s_1 \mid \sigma \rangle \to \sigma'}{\langle s_1\, s_2 \mid \sigma \rangle \to \langle s_2 \mid \sigma' \rangle} \qquad \text{[Crash]}\ \frac{\langle s_1 \mid \sigma \rangle \to \lightning}{\langle s_1\, s_2 \mid \sigma \rangle \to \lightning}$$

$$\text{[Subst]}\ \frac{\langle s_1 \mid \sigma \rangle \to \langle s_1' \mid \sigma' \rangle}{\langle s_1\, s_2 \mid \sigma \rangle \to \langle s_1'\, s_2 \mid \sigma' \rangle}$$

⌟

Definition 5.7 (Ausführung)**.** Eine Ausführung $s\,\sigma$ eines C0-Programms s unter einem Zustand σ ist eine Folge $\langle s \mid \sigma \rangle = k_1, k_2, \ldots$ von Konfigurationen, für die gilt, dass $k_i \to k_{i+1}$ für alle $i \geq 1$. ⌟

Definition 5.8 (Beendigung)**.** Wir schreiben $s\,\sigma \Downarrow k_n$, wenn es eine Ausführung $\langle s \mid \sigma \rangle = k_1, \ldots, k_n$ gibt und kein k' mit $k_n \to k'$ existiert. ⌟

Die Beendigung eines Programms kann nun, je nach der Gestalt von k_n (siehe Definition 5.5), verschiedene Formen annehmen:

Terminierung: $s\,\sigma \Downarrow \sigma'$ (und $\sigma' \in \Sigma$).

Die Ausführung $s\,\sigma$ endet in einem Zustand σ'. Wir sagen, dass s unter σ in einem Zustand σ' **terminiert**.

Abbruch: $P\,\sigma \Downarrow \lightning$.

Die Ausführung $s\,\sigma$ endet in der Konfiguration $\lightning$. Wir sagen, dass das Programm unter σ abgebrochen ist.

Steckenbleiben: $s\,\sigma \Downarrow \langle s' \mid \sigma' \rangle$.

Endet eine Ausführung $s\,\sigma$ in der Konfiguration $\langle s' \mid \sigma' \rangle$ und gibt es keine Konfiguration k'' mit $\langle s' \mid \sigma' \rangle \to k''$, so sagen wir, dass $s\,\sigma$ **stecken bleibt (engl. is stuck)**.

Beendet sich das Programm nicht, so sagen wir, dass es **divergiert** und schreiben $s\,\sigma \Uparrow$.

5.3.4 Steckenbleiben

Steckenbleiben resultiert daraus, dass die Funktion $[\![\cdot]\!]$ nicht total ist und die Regeln [Assign, IfTrue, IfFalse] nur eine Folge-Konfiguration oder einen Endzustand definieren, wenn $[\![e]\!]\,\sigma$ zu einem Wert auswertet. Dass es neben dem Abbruch, der Terminierung und der Divergenz noch einen weiteren Ausgang einer Programmausführung gibt, ist unschön. Man kann die Semantik einer Sprache so definieren, dass Steckenbleiben nicht vorkommt. Hierzu ist es notwendig, der Sprache auch auf den Ausdrücken und Zuständen, auf denen sie nicht definiert ist, eine Semantik zu geben, sie sozusagen zu „totalisieren". In Java und ML werden beispielsweise Ausnahmen ausgelöst, wenn der Divisor einer Division Null ist.

Eine weitere Möglichkeit, das Steckenbleiben zu verhindern ist, über eine statische Analyse (beispielsweise ein Typsystem) Programme, die zum Steckbleiben führen, zu identifizieren. So schreibt der Java-Standard dem Java-Übersetzer beispielsweise vor, Programme zurück zu weisen, für die eine vorgegebene Code-Analyse nicht nachweisen kann, dass jede lokale Variable beschrieben wurde, bevor von ihr gelesen wird. Sprachen, deren wohlgetypte Programme kein Steckenbleiben aufweisen können, nennt man **typsicher** (engl. type safe). C und C0 sind nicht typsicher. Wir gehen darauf noch genauer in Abschnitt 5.6 ein.

Interessant ist nun, ob und in welcher Weise wir das Steckenbleiben beobachten, und von der Terminierung, dem Abbruch und der Divergenz unterschieden können. Auf dem Papier, sprich durch Anwenden der Regeln aus Definition 5.6, können wir das Steckenbleiben leicht identifizieren. In der Praxis führen wir Programme aber nicht mit einer formalen Semantik, sondern mit einem Rechner aus. Wir verwenden hierzu einen Übersetzer, der aus einem Programm P ein Maschinen-Programm $\hat{P}$ macht, das wir dann auf unserem Rechner ausführen können. Von diesem Maschinenprogramm fordert der C-Standard, dass es (1) in einem äquivalenten Zustand terminiert, wenn P terminiert (2) abbricht, wenn P abbricht (3) divergiert, wenn P divergiert. Da C vom Gedanken der Effizienz beseelt ist, macht es Steckenbleiben zu undefiniertem Verhalten (siehe Abschnitt 3.14). Das bedeutet, dass an das Maschinen-Programm $\hat{P}$ keinerlei Forderungen gestellt werden, in dem Moment, in dem P stecken bleibt. Dies wiederum heißt, dass wir einer Ausführung von $\hat{P}$ nicht ansehen können, ob die dazugehörige Ausführung von P stecken blieb.

Beispiel 5.2. Betrachten wir folgendes C0-Programm und eine korrekte Implementierung in MIPS-Assembler:

```
if (y == 0)
    x = 42;
while (x != 0)
    if (x == 42)
        abort();
```

```
      bnez $a1 L2
      li   $t0 42
      move $a0 $t0
      b    L2
L1:   bne  $a0 $t0 L2
      jal  abort
L2:   bnez $a0 L1
```

Nur für $y = 0$ wird die Variable x überhaupt beschrieben. Das heißt, dass für alle Werte von y ungleich 0 das C0-Programm mit der Auswertung der Schleifenabbruchbedingung steckenbleibt.

Abhängig vom Inhalt der Register `$a0` (x) und `$a1` (y) zum Programmstart, kann das Maschinen-Programm, das der Übersetzer aus dem C0-Programm erzeugt hat, aber terminieren, abbrechen oder aber divergieren. ⌟

5.3.5 Einige Beispiele

Betrachten wir nun einige Beispielprogramme, die wir mit der Semantik in Definition 5.6 ausführen wollen.

Beispiel 5.3 (Vertauschen)**.** Das folgende Programm vertauscht den Inhalt der Variablen x und y.

```
t = x; x = y; y = t;
```

Führen wir das Programm auf dem Zustand

$$\begin{aligned} \rho &:= \{\mathtt{x} \mapsto \triangle, \mathtt{y} \mapsto \pentagon, \mathtt{t} \mapsto \diamondsuit\} \\ \mu &:= \{\triangle \mapsto 1, \pentagon \mapsto 2, \diamondsuit \mapsto ?\} \end{aligned}$$

aus. Die Symbole $\triangle$, $\pentagon$ und $\diamondsuit$ kennzeichnen konkrete Adressen. Wir schreiben in diesem Beispiel ρ und μ explizit aus, um die Effekte auf den Zustand zu verdeutlichen. In den folgenden Beispielen verwenden wir dann die Kurzschreibweise aus Definition 5.3.

Den ersten Schritt des Programms leiten wir mit den Regeln [Exec] und [Assign] ab:

$$\text{Exec}\frac{\text{Assign}\dfrac{[\![\mathtt{t}]\!]_L\,\rho;\mu = \diamondsuit \qquad [\![\mathtt{x}]\!]\,\rho;\mu = 1}{\langle \mathtt{t = x;} \mid \sigma\rangle \to \rho;\{\triangle \mapsto 1, \pentagon \mapsto 2, \diamondsuit \mapsto 1\} =: \sigma'}}{\langle \mathtt{t = x; x = y; y = t} \mid \sigma\rangle \to \langle \mathtt{x = y; y = t} \mid \sigma'\rangle}$$

Die nächsten Schritte sind sehr ähnlich. Wir verzichten nun darauf, die Ableitungen explizit zu notieren und schreiben die Regeln, die die jeweilige Folgekonfiguration bestimmen, rechts neben die Folgekonfiguration:

$$\begin{aligned} & \langle \mathtt{x = y; y = t;} \mid \rho, \{\triangle \mapsto 1, \pentagon \mapsto 2, \diamondsuit \mapsto 1\}\rangle && \text{[Assign], [Exec]} \\ \to\ & \langle \mathtt{y = t;} \mid \rho, \{\triangle \mapsto 2, \pentagon \mapsto 2, \diamondsuit \mapsto 1\}\rangle && \text{[Assign], [Exec]} \\ \to\ & \{\mathtt{x} \mapsto \triangle, \mathtt{y} \mapsto \pentagon, \mathtt{t} \mapsto \diamondsuit\}, \{\triangle \mapsto 2, \pentagon \mapsto 1, \diamondsuit \mapsto 1\} && \text{[Assign]} \end{aligned}$$ ⌟

Beispiel 5.4 (Minimum)**.** Das folgende Programm legt das Minimum von x und y in der Variable r ab:

```
if (x < y) r = x; else r = y;
```

Probieren wir das Programm auf dem Zustand $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{r} \mapsto ?\}$ aus:

$$
\begin{array}{lll}
 & \langle \texttt{if (x < y) r = x; else r = y;} \mid \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}\rangle & \\
\to & \langle \texttt{r = x;} \mid \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}\rangle & \text{[IfTrue]} \\
\to & \{\mathtt{r} \mapsto 1, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\} & \text{[Assign]}
\end{array}
$$

⌟

Beispiel 5.5 (Division)**.** Das folgende Programm berechnet den Quotienten q und den Rest r der Division einer nicht negativen Zahl x durch eine positive Zahl y.

```
q = 0;
r = x;
while (y <= r) {
    r = r - y;
    q = q + 1;
}
```

Probieren wir das Programm auf dem Zustand $\{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?\}$ aus. Um die Lesbarkeit zu verbessern, kürzen wir den Schleifenrumpf durch S ab.

$$
\begin{array}{lll}
 & \langle \texttt{q = 0; r = x; while (y <= r) } S & \mid\{\mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \\
\to & \langle \texttt{r = x; while (y <= r) } S & \mid\{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto ?, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign], [Exec]} \\
\to & \langle \texttt{while (y <= r) } S & \mid\{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 5, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign], [Exec]} \\
\to & \langle \texttt{if (y <= r) \{} S \texttt{ while (y <= r) } S \texttt{ else ;\}} & \mid\{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 5, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[While]} \\
\to & \langle \texttt{r = r - y; q = q + 1; while (y <= r) } S & \mid\{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 5, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[IfTrue]} \\
\to & \langle \texttt{q = q + 1; while (y <= r) } S & \mid\{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign], [Exec]} \\
\to & \langle \texttt{while (y <= r) } S & \mid\{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign], [Exec]} \\
\to & \langle \texttt{if (y <= r) \{} S \texttt{ while (y <= r) } S \texttt{\} else ;} & \mid\{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[While]} \\
\to & \langle \texttt{r = r - y; q = q + 1; while (y <= r) } S & \mid\{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[IfTrue]} \\
\to & \langle \texttt{q = q + 1; while (y <= r) } S & \mid\{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign], [Exec]} \\
\to & \langle \texttt{while (y <= r) } S & \mid\{\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign], [Exec]} \\
\to & \langle \texttt{if (y <= r) \{} S \texttt{ while (y <= r) } S \texttt{\} else ;} & \mid\{\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[While]} \\
\to & \langle \texttt{;} & \mid\{\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[IfFalse]} \\
\to & \{\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\} & & \text{[Empty]}
\end{array}
$$

⌟

5.4 Zeiger (C0p)

C0p ist die Erweiterung von C0 um Zeiger. Hierzu erweitern wir die Ausdruckssprache um die Operatoren $*$ und &.

$LExpr \ni l$	::=	$\cdots$	wie Definition 5.2
	|	$*e$	Indirektion
$Expr \ni e$	::=	$\cdots$	wie Definition 5.2
	|	$\&l$	Adresse von

Die Ausdrucksauswertung muss nun um die beiden Fälle erweitert werden. Zunächst ist klar, dass der Adresse-von Operator nur auf einen Ausdruck angewendet werden kann, der zu einer Adresse auswertbar ist, sprich einen L-auswertbaren Ausdruck. Entsprechend ist der Ausdruck $*$e L-auswertbar, falls e eine Adresse liefert.

Definition 5.9 (Erweiterung der Ausdrucksauswertung in C0p)**.**

$$\begin{aligned} [\![\&l]\!]\sigma &= [\![l]\!]_L\sigma \\ [\![*e]\!]_L\sigma &= [\![e]\!]\sigma \quad \text{für } [\![e]\!]\sigma \in Addr \end{aligned}$$ ⌟

Interessant ist die „Dualität“ der beiden Operatoren. Der Operator & macht aus der R-Auswertung eine L-Auswertung und $*$ das Umgekehrte.[5] Die Semantik der Anweisungen bleibt wie in C0.

[5] Es gibt im Zusammenhang mit Zeigern aber auch weitere Fälle in denen die Ausdrucksauswertung undefiniert ist: So kann es beispielsweise sein, dass die L-Auswertung von $*e$ undefiniert ist, da die R-Auswertung von e keine Adresse sondern eine Zahl liefert. Dies tritt zum Beispiel im Ausdruck $*3$ auf, der gemäß der Syntax von C0p zulässig ist, aber keine Bedeutung (Semantik) hat.

Beispiel 5.6. Gegeben sei der Zustand

$$\sigma = \rho;\mu \quad \text{mit} \quad \rho := \{\mathsf{x} \mapsto \triangle, \mathsf{y} \mapsto \pentagon\} \quad \mu := \{\triangle \mapsto \pentagon, \pentagon \mapsto 3\}$$

Man sieht, dass x ein Zeiger ist, der auf den gleichen Behälter zeigt, auf den auch y verweist. Die Variable y verweist auf einen Behälter, der den Wert 3 enthält. Betrachten wir die R-Auswertung des Ausdrucks `*x + y - 1` anhand von Definition 5.4:

$$\begin{aligned} [\![\mathtt{*x + y - 1}]\!]\sigma &= [\![\mathtt{*x + y}]\!]\sigma - [\![1]\!]\sigma \\ &= [\![\mathtt{*x + y}]\!]\sigma - 1 \\ &= [\![\mathtt{*x}]\!]\sigma + [\![\mathtt{y}]\!]\sigma - 1 \end{aligned}$$

Betrachten wir nun die R-Auswertungen der beiden ersten Teilausdrücke im Detail:

$$\begin{aligned} [\![\mathtt{y}]\!]\sigma &= \mu\ [\![\mathtt{y}]\!]_L\sigma \\ &= \mu(\rho\,\mathsf{y}) \\ &= \mu\pentagon \\ &= 3 \end{aligned}$$

$$\begin{aligned} [\![\mathtt{*x}]\!]\sigma &= \mu\ [\![\mathtt{*x}]\!]_L\sigma \\ &= \mu\ [\![\mathtt{x}]\!]\sigma \\ &= \mu(\mu\ [\![\mathtt{x}]\!]_L\sigma) \\ &= \mu(\mu(\rho\,\mathsf{x})) \\ &= \mu(\mu\triangle) \\ &= \mu\pentagon \\ &= 3 \end{aligned}$$

Somit wertet der Ausdruck unter σ zum Wert 5 aus. ⌟

5.5 Blockschachtelung (C0b)

Im Moment muss man einem C0-Programm noch einen Zustand mitgeben, der Behälter für alle lokalen Variablen bereitstellt. Das Programm ist nicht in der Lage zur Laufzeit neue Behälter anzufordern und wieder freizugeben. In diesem Abschnitt werden wir C0 um Variablenvereinbarungen und Blockschachtelung erweitern, so dass man das Anfordern und Freigeben von lokalen Variablen umsetzen kann.

Hierzu fügen wir zunächst zwei neue Anweisungen in die abstrakte Syntax von C0 ein.

$\mathit{Stmt} \ni s$	::=	$\cdots$	wie in Definition 5.1
	|	$\{k_1\ x_1; \ldots k_n\ x_n; s\}$	Block mit Variablenvereinbarungen
	|	$\blacksquare$	Blockende

wobei die k_i Typen (ungleich **`void`**) sind. Noch spielen Typen hier keine Rolle, was wir im nächsten Abschnitt aber ändern werden. Wir wollen auch fordern, dass die Variablen x_i alle paarweise unterschiedlich sind.

In der Semantik müssen wir nun ein wenig mehr arbeiten. Die Bedeutung eines Blockes soll sein, dass für alle in ihm vereinbarten lokalen Variablen neue Behälter angelegt werden und diese nach Verlassen des Blocks freigegeben werden. Ferner soll die Variablenbelegung nach Verlassen des Blocks zu der Variablenbelegung unmittelbar vor Betreten des Blocks entsprechen.

Beispiel 5.7. Wir erwarten in folgendem Programm, dass y nach Programmende den Wert 3 hat und nicht 2.

```
{ int x; x = 3; { int x; x = 2; } y = x; }
```

⌟

Hierzu erweitern wir den Zustand um einen **Keller (engl. stack)** von Variablenbelegungen. Beim Eintritt in den Block wird eine neue Variablenbelegung auf den Keller gelegt, die die im Block vereinbarten Variablen enthält. Beim Austritt aus dem Block wird diese Belegung wieder vom Keller genommen. Damit wir in der Semantik auf das Ende eines Blockes entsprechend reagieren können, fügt die Regel [Block] die Anweisung $\blacksquare$ in das „Restprogramm" ein. Zusätzlich stellt die Regel [Block] **neue** Behälter für die Variablen, die in dem Block vereinbart wurden, bereit. Die Neuheit der Behälter wird dadurch modelliert, dass die Bezeichner an Adressen gebunden werden, die in der aktuellen Speicherbelegung nicht in Gebrauch sind.

$$\text{[Block]}\quad \langle \{k_1 x_1; \ldots k_n x_n; s\} \mid \rho_1, \ldots, \rho_m; \mu\rangle \to \langle s \ \blacksquare \mid \rho_1, \ldots, \rho_m, \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}; \mu[a_1 \mapsto ?, \ldots, a_n \mapsto ?]\rangle \quad \text{mit } a_i \notin \operatorname{dom}\mu \text{ für } 1 \le i \le n$$

$$\text{[Leave]}\quad \langle \blacksquare \mid \rho_1, \ldots, \rho_m, \rho_{m+1}, \mu\rangle \to \rho_1, \ldots, \rho_m; \mu' \quad \text{mit } \mu' \coloneqq \mu \setminus \operatorname{ran}\rho_{m+1}$$

Die letzte Regel drückt aus, dass $\blacksquare$ die Variablenbelegung vom Keller nimmt und alle Behälter aus dem Speicher freigibt, die innerhalb des

Blockes angelegt wurden. Das sind genau die Adressen, die im Bildbereich von ρ_{m+1} liegen.

Da wir nun einen Keller von Variablenbelegungen haben, müssen wir die Ausdrucksauswertung geringfügig anpassen:

$$[\![x]\!]_L\, \rho_1, \ldots, \rho_m; \mu = \begin{cases} \rho_m\, x & \text{falls } x \in \rho_m \\ [\![x]\!]_L\, \rho_1, \ldots, \rho_{m-1}; \mu & \text{andernfalls} \end{cases}$$

Beispiel 5.8. Betrachten wir nun die Ableitung des Programms aus obigem Beispiel auf dem Startzustand

$\rho \coloneqq \{\mathsf{y} \mapsto \triangle\} \quad \mu \coloneqq \{\triangle \mapsto ?\}$

	$\langle$`{int x; x = 3; { int x; x = 2; } y = x;}`$\,\vert\{\mathsf{y} \mapsto \triangle\}; \{\triangle \mapsto ?\}\rangle$	
$\to$	$\langle$`x = 3; { int x; x = 2; } y = x; ■` $\vert\{\mathsf{y} \mapsto \triangle\}, \{\mathsf{x} \mapsto \pentagon\}; \{\triangle \mapsto ?, \pentagon \mapsto ?\}\}\rangle$	[Block]
$\to$	$\langle$`{ int x; x = 2; } y = x; ■` $\vert\{\mathsf{y} \mapsto \triangle\}, \{\mathsf{x} \mapsto \pentagon\}; \{\triangle \mapsto ?, \pentagon \mapsto 3\}\}\rangle$	[Assign], [Exec]
$\to$	$\langle$`x = 2; ■; y = x; ■` $\vert\{\mathsf{y} \mapsto \triangle\}, \{\mathsf{x} \mapsto \pentagon\}, \{\mathsf{x} \mapsto \diamondsuit\};$ $\{\triangle \mapsto ?, \pentagon \mapsto 3, \diamondsuit \mapsto ?\}\rangle$	[Block], [Subst]
$\to$	$\langle$`■; y = x; ■` $\vert\{\mathsf{y} \mapsto \triangle\}, \{\mathsf{x} \mapsto \pentagon\}, \{\mathsf{x} \mapsto \diamondsuit\};$ $\{\triangle \mapsto ?, \pentagon \mapsto 3, \diamondsuit \mapsto 2\}\rangle$	[Assign], [Exec]
$\to$	$\langle$`y = x; ■` $\vert\{\mathsf{y} \mapsto \triangle\}, \{\mathsf{x} \mapsto \pentagon\}; \{\triangle \mapsto ?, \pentagon \mapsto 3\}\rangle$	[Leave], [Exec]
$\to$	$\langle$`■` $\vert\{\mathsf{y} \mapsto \triangle\}, \{\mathsf{x} \mapsto \pentagon\}; \{\triangle \mapsto 3, \pentagon \mapsto 3\}\rangle$	[Assign], [Exec]
$\to$	$\{\mathsf{y} \mapsto \triangle\}; \{\triangle \mapsto 3\}$	[Leave] ⌟

5.6 Ein Typsystem (C0pbt)

Die Menge aller C0-Programme, die durch die abstrakte Syntax in Definition 5.1 und Definition 5.2 definiert ist, enthält Programme, die steckenbleiben. So ist folgende Anweisung zwar syntaktisch zulässig, führt aber bei der Programmausführung sofort in eine Konfiguration, von der aus die Programmausführung nicht fortgesetzt werden kann.

```
int x;
x = 666;
*x = 42;
```

Einige steckenbleibende Programme, so wie obiges, können wir **statisch** bereits erkennen und aussondern. Dies gelingt uns durch eine **Statische Semantik**. Diese gibt den Konstrukten einer Sprache eine „Bedeutung", die sich an statischen Eigenschaften orientiert. Statische Eigenschaften sind solche, die unabhängig von der Eingabe an das Programm auftreten, wie zum Beispiel Typen von Variablen.

Aus der Information, dass x den Typ **int** hat, können wir schließen, dass x immer einen **int** als Wert hat und nie eine Adresse. Daher führt das Anwenden des Indirektions-Operators auf x immer zum Steckenbleiben. Im Folgenden stellen wir Typregeln auf, die solche „offensichtlich falschen" Programme ausschließen. Programme, die die Typregeln befolgen, nennen wir **wohlgetypt**.

Allerdings gibt es wohlgetypte C0-Programme, die trotzdem steckenbleiben können, wie zum Beispiel

```
int x;
x = x + 1;
```

Von der Variable x wird gelesen, obwohl sie vorher nicht beschrieben wurde. Dies führt dazu, dass der uninitialisierte Wert ? gelesen wird, auf dem die Ausdrucksauswertung nicht definiert ist.

```
int x;
int *p;
{ int y; p = &y; }
x = *p;
```

Der Zeiger p zeigt nach der Ausführung des Blocks auf einen Behälter, den es nicht mehr gibt.

```
int x = 666;
x = x / 0;
```

Hier wird schlicht durch 0 geteilt.

Das heißt, dass es in C0 und C Ausnahmesituationen gibt, die durch die statische Semantik nicht erkannt werden können. Sprachen, in denen wohlgetype Programme niemals steckenbleiben heißen **statisch typsicher** [6].

Herkömmlicherweise definiert man die statische Semantik eines Programms durch eine Relation, die eine **Typumgebung** mit der abstrakten Syntax in Beziehung setzt. Eine Typumgebung ordnet, basierend auf den Variablenvereinbarungen, Variablen und anderen Bezeichnern statische Informationen zu, beispielsweise einen Typ. Diese Relation ist induktiv über der abstrakten Syntax definiert, das heißt, dass sich die statische Semantik von Programmteilen zur statischen Semantik des gesamten Programms zusammensetzen lässt. Wir wollen hier der Übersichtlichkeit halber die statische Semantik für Ausdrücke und Anweisungen getrennt formulieren.

[6] Statisch typsichere Sprachen (wie z.B. Java oder ML) stellen durch ihre Semantik sicher, dass Ausnahmesituationen, die statisch im Allgemeinen nicht erkannt werden können, ein fest definiertes Verhalten bekommen. So löst Java beispielsweise eine Ausnahme aus, wenn die **null**-Referenz dereferenziert wird, oder durch 0 geteilt wird.

Zunächst führen wir verschiedene Arten von Typen ein, um (im wesentlichen) Zeiger von **`int`**s unterscheiden zu können.

Definition 5.10 (Die Datentypen von C0)**.** Die Benennung der Typen ist dem C-Standard entnommen.

$ITy \ni i$	$::=$	**`char`** | **`int`**	Integer-Typ
$PTy \ni p$	$::=$	$t*$	Zeiger-Typ
$STy \ni k$	$::=$	$p \mid i$	Skalarer Typ
$Ty \ni t$	$::=$	k | **`void`**	Typ

⌟

C unterstützt implizite Typanpassung, was bedeutet, dass an manchen Stellen die Typen in andere umgewandelt werden können. Beispielsweise kann ein **`int`**$*$ automatisch in einen **`void`**$*$ umgewandelt werden. Die Relation $\leftrightarrow$ modelliert, welche Typen an welche angepasst werden können.

Definition 5.11 (Automatische Typanpassung)**.**

$$\frac{}{i_1 \leftrightarrow i_2} \qquad \frac{}{t* \leftrightarrow t*} \qquad \frac{}{t* \leftrightarrow \texttt{void}*} \qquad \frac{}{\texttt{void}* \leftrightarrow t*}$$

⌟

5.6.1 Ausdrücke

Die statische Semantik der Ausdrücke ist durch die Relation

$$ExprS \subseteq (Var \rightharpoonup Ty) \times Expr \times Ty$$

definiert, die wir im folgenden induktiv über der abstrakten Syntax definieren werden. Zur besseren Lesbarkeit verwenden wir die folgende Notation, die sich eingebürgert hat:

$$\Gamma \vdash e : t \quad :\Longleftrightarrow \quad (\Gamma, e, t) \in ExprS$$

$\Gamma \vdash e : t$ sagt also, dass der Ausdruck e unter der Typumgebung Γ den Typ t hat.

Definition 5.12 (Statische Semantik der C0-Ausdruckssprache)**.**

$$\text{Var}\frac{\Gamma\, x = k}{\Gamma \vdash x : k} \qquad \text{Const}\frac{-N \le c < N \quad N = 2^{8 \cdot \textbf{sizeof(int)}-1}}{\Gamma \vdash c : \textbf{int}}$$

$$\text{BinArith}\frac{\Gamma \vdash e_1 : i_1 \quad \Gamma \vdash e_2 : i_2}{\Gamma \vdash e_1\ r\ e_2 : \textbf{int}} \qquad \text{BinCmp}\frac{\Gamma \vdash e_1 : k_1 \quad \Gamma \vdash e_2 : k_2 \quad k_1 \leftrightarrow k_2}{\Gamma \vdash e_1\ m\ e_2 : \textbf{int}}$$

$$\text{CmpPtrL}\frac{\Gamma \vdash e : t*}{\Gamma \vdash e\ m\ 0 : \textbf{int}} \qquad \text{CmpPtrR}\frac{\Gamma \vdash e : t*}{\Gamma \vdash 0\ m\ e : \textbf{int}}$$

$$\text{Indir}\frac{\Gamma \vdash e : k*}{\Gamma \vdash *e : k} \qquad \text{Addr}\frac{\Gamma \vdash l : t}{\Gamma \vdash \&l : t*}$$

⌟

Binäre arithmetische Operatoren sind immer vom Typ `int`, auch wenn der Typ kleiner ist (in C0 gibt es nur noch `char`, das kleiner ist). In „echtem“ C gibt es die sogenannte Integer-Promotion, die festlegt, dass mindestens in `int` gerechnet wird, da `int` oft der Maschinenwortbreite der Zielarchitektur entspricht. Vergleichsoperatoren können auch Zeiger als Operanden haben. Arithmetische Operationen in C0 jedoch nicht. Aufgabe 5.15 beschäftigt sich damit, das Typsystem um Zeigerarithmetik zu erweitern.

Die Konstante 0 hat beim Vergleich von Zeigern eine Sonderrolle (wie auch später bei der Zuweisung). Sie dient sowohl als `int`-Konstante mit dem Wert 0 als auch als Zeigerkonstante, die den Nullzeiger darstellt, und kann daher in Vergleichen mit Zeigern auftreten.

Der Operator `&` kann die Adresse eines Behälters bestimmen. Hierzu muss der entsprechende Ausdruck **L-auswertbar** sein.

5.6.2 Anweisungen

Analog zu den Ausdrücken führen wir wieder eine Relation ein, die Anweisungen mit Typumgebungen in Beziehung setzt. Anweisungen haben keinen Typ, daher ist die Relation nur zweistellig und nicht dreistellig wie bei den Ausdrücken.

$$StmtS \subseteq (Var \rightharpoonup Ty) \times Stmt$$

Zur besseren Lesbarkeit verwenden wir die folgende Notation:

$$\Gamma \vdash s \quad :\Longleftrightarrow \quad (\Gamma, s) \in StmtS$$

Definition 5.13 (Statische Semantik der C0-Anweisungen)**.**

$$\text{Assign}\ \frac{\Gamma \vdash e : k_1 \quad \Gamma \vdash l : k_2 \quad k_1 \leftrightarrow k_2}{\Gamma \vdash l \texttt{ = } e\texttt{;}} \qquad \text{Abort}\ \frac{}{\Gamma \vdash \texttt{abort();}}$$

$$\text{While}\ \frac{\Gamma \vdash e : k \quad \Gamma \vdash s}{\Gamma \vdash \texttt{while (}e\texttt{)}\ s} \qquad \text{If}\ \frac{\Gamma \vdash e : k \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if (e)}\ s_1\ \texttt{else}\ s_2}$$

$$\text{Seq}\ \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1 \_ s_2} \qquad \text{Block}\ \frac{\Gamma[x_1 \mapsto k_1, \ldots, x_n \mapsto k_n] \vdash s}{\Gamma \vdash \{k_1\ x_1; \ldots k_n\ x_n; s\}}$$

⌟

Die Regeln [While] und [If] stellen sicher, dass die Bedingungen auch einen skalaren Typ haben, also nicht **`void`** sein können. Die Regeln [Assign] stellt sicher, dass der Typ des Ausdrucks an den Typ linken Seite anpassbar ist. Dadurch wird zum Beispiel ein Programm wie das folgende ausgeschlossen:

```
{ int *p; p = 5; }
```

Die Regel [Block] blendet die Variablenvereinbarungen des Blocks für seinen Rumpf in die Typumgebung ein.

5.6.3 Beispiele

Beispiel 5.9. Betrachten wir zunächst die Ableitung des Typs eines Ausdrucks unter der Typumgebung $\Gamma \coloneqq \{\mathtt{x} \mapsto \mathtt{char*}, \mathtt{y} \mapsto \mathtt{int}\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\,\mathtt{x} = \mathtt{char*}}{\Gamma \vdash \mathtt{x} : \mathtt{char*}}\,\text{Var}}{\Gamma \vdash \mathtt{*x} : \mathtt{char}}\,\text{Indir} \quad \dfrac{\Gamma\,\mathtt{y} = \mathtt{int}}{\Gamma \vdash \mathtt{y} : \mathtt{int}}\,\text{Var}}{\Gamma \vdash \mathtt{*x + y} : \mathtt{int}}\,\text{BinArith} \quad \dfrac{}{\Gamma \vdash \mathtt{1} : \mathtt{int}}\,\text{Const}}{\Gamma \vdash \mathtt{*x + y - 1} : \mathtt{int}}\,\text{BinArith}$$

⌟

Beispiel 5.10. In diesem Beispiel betrachten wir den gleichen Ausdruck wie im vorigen Beispiel, allerdings mit einer anderen Typumgebung:

$$\Gamma \coloneqq \{\mathtt{x} \mapsto \mathtt{char}, \mathtt{y} \mapsto \mathtt{int}\}$$

Hier sieht man nun schön, dass man für die Regel [Var] über [Indir] keine passende Prämisse erzeugen kann, da die Typumgebung für x einen Zeiger bereitstellen müsste.

$$\dfrac{\dfrac{\dfrac{\dfrac{\lightning}{\Gamma \vdash \mathtt{x} : \mathtt{char*}}\,\text{Var}}{\Gamma \vdash \mathtt{*x} : \mathtt{char}}\,\text{Indir} \quad \dfrac{\Gamma\,\mathtt{y} = \mathtt{int}}{\Gamma \vdash \mathtt{y} : \mathtt{int}}\,\text{Var}}{\Gamma \vdash \mathtt{*x + y} : \mathtt{int}}\,\text{BinArith} \quad \dfrac{}{\Gamma \vdash \mathtt{1} : \mathtt{int}}\,\text{Const}}{\Gamma \vdash \mathtt{*x + y - 1} : \mathtt{int}}\,\text{BinArith}$$

⌟

Beispiel 5.11. Betrachten wir eine Ableitung in der statischen Semantik für ein komplexeres Beispiel, in dem lokale Variablen in Blöcken definiert werden. Wir lassen hier die Beweisbäume der Prämissen von [Assign] der Übersichtlichkeit halber weg.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{x = 3;}}\,\text{Assign} \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma[\mathtt{x} \mapsto \mathtt{int*}] \vdash \mathtt{x = \&y;}}\,\text{Assign}}{\Gamma \vdash \mathtt{\{\ int*\ x;\ x = \&y;\ \}}}\,\text{Block} \quad \dfrac{\vdots}{\Gamma \vdash \mathtt{y = x;}}\,\text{Assign}}{\Gamma \vdash \mathtt{\{\ int*\ x;\ x = \&y;\ \}\ y = x;}}\,\text{Seq}}{\Gamma \coloneqq \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}\} \vdash \mathtt{x = 3;\ \{\ int*\ x;\ x = \&y;\ \}\ y = x;}}\,\text{Seq}}{\varnothing \vdash \mathtt{\{int\ x;\ int\ y;\ x = 3;\ \{\ int*\ x;\ x = \&y;\ \}\ y = x;\ \}}}\,\text{Block}$$

⌟

5.7 Aufgaben

Aufgabe 5.1 (Logische Formeln. ⋆)**.** Definieren Sie die abstrakte Syntax von aussagenlogischen Formeln. Diese bestehen aus den binären Operatoren $\wedge, \vee, \Rightarrow, \Leftrightarrow$, dem unären Operator $\neg$ und Bezeichnern. ⌟

Aufgabe 5.2 (Arithmetische Ausdrücke. ⋆⋆⋆⋆⋆)**.** Versuchen Sie eine **konkrete** Syntax für arithmetische Ausdrücke zu konstruieren. In der konkreten Syntax darf es für jede Folge von Zeichen nur einen Ableitungsbaum geben. Hinweis: Die konkrete Syntax muss Regeln wie „Punkt vor Strich“ und Klammerung behandeln. ⌟

Aufgabe 5.3 (Do-while Schleife. ⋆⋆)**.** Erweitern Sie die abstrakte Syntax und die Anweisungs-Semantik von C0 um die do-while Schleife (siehe auch Abschnitt 3.6.2). ⌟

Aufgabe 5.4 (For Schleife. ⋆⋆⋆)**.** Erweitern Sie die abstrakte Syntax und die Anweisungs-Semantik um eine eingeschränkte for-Schleife. Diese soll die (abstrakte) Syntax **`for`** `(`x `=` e`;` e`;` x`++)` s haben. ⌟

Aufgabe 5.5 (C0 AST. ⋆⋆)**.** Zeichnen Sie den abstrakten Syntaxbaum des folgenden Programms:

```
r = 1;
while (r <= n) {
    r = r * n;
    n = n - 1;
}
```

⌟

Aufgabe 5.6 ($\sigma = (\rho, \mu)$. ⋆⋆)**.** Betrachten Sie folgenden Zustand in Kurzschreibweise gemäß Definition 5.3:

$$\sigma = \{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2\}$$

Geben Sie ρ und μ an. Verwenden Sie $\triangle, \pentagon \ldots$ um Adressen darzustellen. ⌟

Aufgabe 5.7 (Ausführungsprotokoll. ⋆⋆)**.** Führen Sie folgendes Programm mit der Semantik von C0

```
r = 1;
while (r <= n) {
    r = r * n;
    n = n - 1;
}
```

auf dem Zustand $\sigma = \{\mathsf{n} \mapsto 3, \mathsf{r} \mapsto ?\}$ aus. ⌟

Aufgabe 5.8 (Ausführung und Zeiger. ⋆⋆)**.** Führen Sie folgendes Programm mit der Semantik von C0p auf dem Zustand

$$\{\mathsf{x} \mapsto \triangle, \mathsf{y} \mapsto \pentagon, \mathsf{z} \mapsto \diamondsuit, \mathsf{w} \mapsto \triangledown\}, \{\triangle \mapsto ?, \pentagon \mapsto ?, \diamondsuit \mapsto ?, \triangledown \mapsto ?\}$$

aus:

```
x = 42;
y = &x;
z = &y;
w = **z;
```

Beobachten Sie vor allem, wie die Behälter „verzeigert“ sind. ⌟

Aufgabe 5.9 (Auswertung von Ausdrücken. ★★★)**.** Betrachten Sie die Ausdrucksauswertung in C0 auf einem Ausdruck e. Was ist der Unterschied zwischen folgenden Situationen: $[\![e]\!]\ \sigma$ ist nicht definiert und $[\![e]\!]\ \sigma = ?$. Was passiert bei der Ausführung der Anweisung `x = e` wenn $[\![e]\!]\ \sigma = ??$ ⌟

Aufgabe 5.10 (Fehlende Deklaration. ★★)**.** Was passiert, wenn man das Programm

```
x = 1;
```

auf dem leeren Zustand ausführt? ⌟

Aufgabe 5.11 (C0 Ausdrücke. ★★)**.** Betrachten Sie die Semantik der Ausdrücke. Geben Sie drei Zustände an, unter denen die Ausdrucksauswertung undefiniert ist. ⌟

Aufgabe 5.12 (Lokale Variablen. ★★)**.** Kann es in C0b jemals zwei lokale Variablen geben, die denselben Behälter referenzieren? ⌟

Aufgabe 5.13 (Baumelnde Zeiger. ★★★)**.** Ein Zeiger heißt „baumelnd“ (engl. dangling pointer), wenn er eine Adresse enthält, zu der es keinen Behälter gibt. Konstruieren Sie ein C0pb Programm, in dessen Ablauf ein baumelnder Zeiger vorkommt. Vergewissern Sie sich dessen, in dem Sie das Programm mit der formalen Semantik ausführen. ⌟

Aufgabe 5.14 (Unterprogramme. ★★★★★)**.** Fügen Sie der formalen Semantik von C0 Unterprogramme und Unterprogrammaufrufe hinzu. Hierzu gehen Sie wie folgt vor:

1. Definieren Sie eine neue Kategorie *Func* für Unterprogramme in der abstrakten Syntax.
2. Ein Programm besteht nicht mehr nur aus einer Anweisung, sondern aus Unterprogramm-Definitionen.
3. Definieren Sie eine neue Komponente $\phi : \mathit{Var} \rightharpoonup \mathit{Func}$ im Zustand. Diese bildet Bezeichner auf den „Code“ der Unterprogramme ab.
4. Bei der Definition des Unterprogrammaufrufs können sie dann in ϕ den Code des Unterprogramms „nachschlagen“ und eine entsprechende Konfiguration erzeugen.

⌟

Aufgabe 5.15 (Pointer Arithmetik. ★★)**.** Fügen Sie dem Typsystem von C0 Zeigerarithmetik hinzu. Hierbei kann zu einem Zeiger ein int hinzuaddiert oder von ihm abgezogen werden. ⌟

Aufgabe 5.16 (Unterprogramme. ★★★★★)**.** Fügen Sie der formalen Semantik von C0 Unterprogramme und Unterprogrammaufrufe hinzu. Hierzu gehen Sie wie folgt vor:

1. Definieren Sie eine neue Kategorie *Func* für Unterprogramme in der abstrakten Syntax.
2. Ein Programm besteht nicht mehr nur aus einer Anweisung, sondern aus Unterprogramm-Definitionen.
3. Definieren Sie eine neue Komponente $\phi : \mathit{Var} \rightharpoonup \mathit{Func}$ im Zustand. Diese bildet Bezeichner auf den „Code“ der Unterprogramme ab.
4. Bei der Definition des Unterprogrammaufrufs können sie dann in ϕ den Code des Unterprogramms „nachschlagen“ und eine entsprechende Konfiguration erzeugen.

⌟

Aufgabe 5.17 (Die Statische Semantik von C0 ist deterministisch. ★★★)**.** Zeigen Sie, dass die statische Semantik von C0 deterministisch ist. Hierzu zeigen Sie durch strukturelle Induktion über der abstrakten Syntax von C0 die Eigenschaft

$$\Gamma \vdash e : t \land \Gamma \vdash e : t' \implies t = t' \quad \text{für alle } t, t' \in \mathit{Ty}, e \in \mathit{Expr}$$

⌟

5.8 Zusammenfassung und Literaturhinweise

Dieses Kapitel ist nur eine kleine Einführung in die formale Semantik von Programmiersprachen. Zur Auswertung von Anweisungen haben wir eine sogenannte *small step operational semantics* verwendet. Diese hat gegenüber anderen Techniken, wie zum Beispiel der *big step* Semantik, den Vorteil, dass man Divergenz, Steckenbleiben und Abbruch unterscheiden kann. Dafür ist sie ein wenig umständlicher. Insbesondere wäre die Formalisierung von C0b in einer big step Semantik knapper und weniger überladen ausgefallen.

Eine gute Einführung in die formale Semantik von Programmiersprachen, die auch die Unterschiede der einzelnen Semantiken gegenüberstellt, sind die Lehrbücher von Nielson und Nielson [NN92] und Winskel [Win93]. Weiterführende Themen werden zum Beispiel im Buch von Harper [Har13] behandelt. Historisch gesehen, ist die Arbeit von Strachey [Str00] wegweisend (wir zitieren hier einen 2000 erschienen Nachdruck des Originals von 1967), in der zum ersten mal versucht wurde, imperative Programmiersprachen formal zu fassen.

Die Formalisierung von C0, die wir in diesem Kapitel diskutiert haben, behandelt viele Eigenschaften von C gar nicht, zum Beispiel Unterprogramme, Zeigerarithmetik oder aber auch Nebenwirkungen in Ausdrücken. Es ist nicht kompliziert, der Semantik Unterprogramme hinzuzufügen. Allerdings erfordert das Hinzufügen von Zeigerarithmetik, die man benötigt, um Verbunde und Reihungen zu modellieren, ein feiner formuliertes Speichermodell, was wir, um die Darstellung kurz und überschaubar zu halten, hier nicht diskutieren. Interessante Details finden sich, unter anderem, in der Formalisierung des C-Speichermodells des verifizierten Übersetzers CompCert [LB08] und den Dissertationen von Norrish [Nor98] und Krebbers [Kre15]. Beide enthalten eine weitestgehend vollständige Formalisierung von C.

6 Korrekte Software

6.1 Funktionale Korrektheit

Was bedeutet es, wenn wir sagen, dass ein Programm P korrekt ist? Zunächst gibt es verschiedene Arten von Korrekheitskriterien: Ein wichtiges Kriterium ist die **funktionale Korrektheit**, die fordert, dass das Programm „die richtigen Ergebnisse produziert". Weitere Korrektheitskriterien sind zum Beispiel die Einhaltung von Resourcenbeschränkungen, wie Zeit oder Speicher, oder aber auch die Abwesenheit von Sicherheitslücken. In diesem Abschnitt diskutieren wir die **funktionale Korrektheit**. Wie spezifizieren wir aber, was die richtigen Ergebnisse sind? Hierzu benötigt man eine **Spezifikation**.

Definition 6.1 (Spezifikation)**.** Eine Spezifikation $S \subseteq \Sigma \times \Sigma$ ist eine Menge von Zustandspaaren. ⌟

Eine Spezifikation beschreibt also eine Menge von Ein- und Ausgabepaaren und legt somit fest, welche Ausgabe(n) zu welcher Eingabe gehören.

Beispiel 6.1. Folgende Spezifikation fordert, dass die Variable $\mathtt{y}$ im Endzustand den Wert $\mathtt{x} + 1$ hat.

$$S_{\text{add}} = \{(\sigma, \sigma') \mid \sigma'\,\mathtt{y} = \sigma\,\mathtt{x} + 1\}$$ ⌟

Beispiel 6.2 (Maximum)**.** Betrachten wir die Spezifikation $s_{\max}$ zur Berechnung des Maximums $\mathtt{m}$ zweier Variablen $\mathtt{x}$ und $\mathtt{y}$. Sie ist gegeben durch die folgende Menge von Zustandspaaren:

$$\begin{aligned} S_{\max} = \{&(\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}, \{\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{m} \mapsto 1\}), \\ &(\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}, \{\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{m} \mapsto 2\}), \\ &\vdots \\ &(\{\mathtt{x} \mapsto 42, \mathtt{y} \mapsto 37\}, \{\{\mathtt{x} \mapsto 42, \mathtt{y} \mapsto 37, \mathtt{m} \mapsto 42\})\} \\ &\vdots \end{aligned}$$

oder kompakter

$$S_{\max} = \left\{(\sigma, \sigma') \mid \sigma'\,\mathtt{m} \geq \sigma\,\mathtt{x} \wedge \sigma'\,\mathtt{m} \geq \sigma\,\mathtt{y} \wedge (\sigma'\,\mathtt{m} = \sigma\,\mathtt{x} \vee \sigma'\,\mathtt{m} = \sigma\,\mathtt{y})\right\}$$ ⌟

Eine Menge von Zuständen kompakt beschreiben zu können, verwenden wir Zusicherungen:

Definition 6.2 (Zusicherung (engl. assertion))**.** Eine Zusicherung z ist ein Prädikat über einen Zustand. Wir modellieren eine Zusicherung hier als Ausdruck der Sprache C0 angereichert mit den herkömmlichen logischen Operatoren $\vee, \wedge, \ldots$ Wir sagen, ein Zustand σ **erfüllt** z, genau dann wenn $[\![z]\!]\,\sigma \neq 0$. Wir schreiben dann $\sigma \vDash z$. Die Menge $\{\sigma \mid \sigma \vDash z\}$ aller Zustände, die z erfüllen, kürzen wir, unter leichtem Missbrauch der Notation, mit $[\![z]\!]$ ab. ⌟

Bemerkung 6.1. Die Zusicherung **`true`** wird von allen Zuständen erfüllt. Die Zusicherung **`false`** wird von keinem Zustand erfüllt. ⌟

Eine Spezifikation spricht immer über ein Paar von Zuständen. Es bietet sich nun an, eine Spezifikation durch ein Paar von Zusicherungen, der Vorbedingung V und der Nachbedingung N darzustellen. Man kann einen Zustand der Nachbedingung meist aber nicht unabhängig von einem Zustand der Vorbedingung ausdrücken. Daher definiert man sich eine Menge von Variablen K, die in den Zuständen der Vor- und Nachbedingung den gleichen Wert haben müssen.

Definition 6.3 (Kompatible Zustände)**.** Ein Paar von Zuständen (σ, σ') ist **kompatibel** bezüglich einer Menge von Variablen K, wenn für jede Variable $x \in K$ gilt, dass $\sigma\, x = \sigma'\, x$. ⌟

Definition 6.4 (Vor- und Nachbedingung)**.** Ein Paar von Zusicherungen (V, N) und eine Menge von Variablen K definieren eine Spezifikation

$$S_{V,N,K} = \{(\sigma, \sigma') \mid \sigma \vDash V \text{ und } \sigma' \vDash N \text{ und } \sigma, \sigma' \text{ kompatibel bzgl. } K\}$$

Die Zusicherung V heißt **Vorbedingung (engl. precondition)** und N heißt **Nachbedingung (engl. postcondition)**. ⌟

Beispiel 6.3 (Zusicherungen, Maximum)**.** Die Vorbedingung und Nachbedingung

$$V = \texttt{true} \quad N = \texttt{m} \geq \texttt{x} \wedge \texttt{m} \geq \texttt{y} \wedge (\texttt{m} = \texttt{x} \vee \texttt{m} = \texttt{y}) \quad K = \{\texttt{x}, \texttt{y}\}$$

bilden die Spezifikation für ein Programm, dass das Maximum der Variablen `x` und `y` in der Variable `m` speichert. ⌟

Um nicht immer die Kompatibilität der Zustände fordern zu müssen, schränken wir die Menge der Programme ein: Wir verbieten, dass die Variablen mit Großbuchstaben `X`, `Y`, … überschrieben werden dürfen. Dann sind per Definition alle Vor- und Nachbedingungen auf diesen Variablen kompatibel. Mit dieser Einschränkung können wir nun die Menge der kompatiblem Variablen einfach weglassen.

Definition 6.5 (Totale Korrektheit)**.** Ein Programm P ist (total) korrekt bezüglich einer Spezifikation (V, N), wenn für jeden Zustand $\sigma \vDash V$ ein Zustand $\sigma' \vDash N$ existiert, so dass P, angewandt auf σ, mit σ' terminiert:

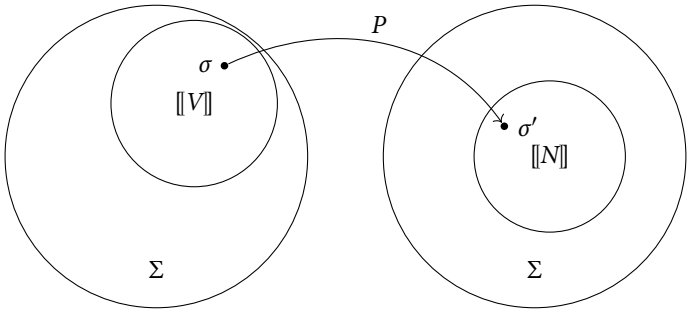
$$[\![V]\!] \subseteq \{\sigma \mid P\,\sigma \Downarrow \sigma' \wedge \sigma' \vDash N\}$$

Wir notieren die Tatsache, dass ein Programm total korrekt ist, durch ein sogenanntes **Hoare-Tripel**[1]: $[V]\ P\ [N]$ ⌟

[1] Tony Hoare definierte die nach ihm benannten Tripel ursprünglich nur für partielle Korrektheit [Hoa69]. Hierfür hat sich die Notation $\{V\}\,P\,\{N\}$ eingebürgert.

Definition 6.6 (Partielle Korrektheit)**.** *P* ist partiell korrekt bezüglich (V, N), wenn es für jeden Zustand $\sigma \vDash V$ entweder divergiert oder in einem Zustand $\sigma' \vDash N$ terminiert: $[\![V]\!] \subseteq \{\sigma \mid P\,\sigma \Downarrow \sigma' \Rightarrow \sigma' \vDash N\}$. ⌟

Wir verwenden folgende grafische Anschauung für ein Hoare-Tripel $[V]\ P\ [N]$.



Wir zeichnen jeweils die Menge aller Zustände Σ vor der Programmausführung und nach der Programmausführung und zeichnen einen Pfeil von σ zu σ', wenn $P\,\sigma \Downarrow \sigma'$. Der mit $[\![V]\!]$ beschriftete Kreis stellt die Menge der Zustände dar, die die Zusicherung *V* erfüllen. Analog stellt der mit $[\![N]\!]$ beschriftete Kreis die Menge aller Zustände dar, die die Zusicherung *N* erfüllen. Die Grafik verdeutlicht, dass jeder Zustand der die Zusicherung *V* erfüllt durch *P* (wenn es terminiert) in einen Zustand überführt wird, der die Zusicherung *N* erfüllt.

6.1.1 Verstärken und Abschwächen

Eine Aussage *B* ist **stärker** als eine Aussage *C*, wenn $B \Rightarrow C$. Entsprechend ist *C* **schwächer** als *B*. In unserem Kontext beschreibt eine schwächere Aussage **mehr** Zustände als eine stärkere. So ist beispielsweise

$$\begin{aligned} x > 10 \quad &\text{stärker als} \quad x > 3 \\ x = 10 \wedge y = 5 \quad &\text{stärker als} \quad y = 5 \end{aligned}$$

Die Menge $[\![B]\!]$ aller Zustände, die *B* erfüllen ist also kleiner als die Menge $[\![C]\!]$ aller Zustände, die *C* erfüllen.

In einem Hoare-Tripel $[V]\ P\ [N]$ kann man **Vorbedingungen verstärken** und **Nachbedingungen abschwächen**:

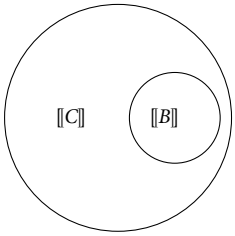


Abbildung 6.1: Die Zusicherung *B* ist stärker als die Zusicherung *C*.

Satz 6.1 (Abschwächen und Verstärken)**.** *Angenommen* $V' \Rightarrow V$, $N \Rightarrow N'$ *und* $[V]\ P\ [N]$. *Dann gilt* $[V']\ P\ [N']$.

Beweis. Sei σ ein Zustand, der V' erfüllt: $\sigma \vDash V'$. Da $V' \Rightarrow V$ erfüllt σ auch *V*. Wegen $[V]\ P\ [N]$ liefert die Ausführung von *P* einen Zustand σ', der *N* erfüllt. Da $N \Rightarrow N'$ gilt $\sigma' \vDash N'$. □

Beispiel 6.4. Betrachten wir die Spezifikation aus Beispiel 6.3 und nehmen an, dass wir ein Programm *P* haben, das $[V]\ \ P\ \ [N]$ erfüllt. Nach Satz 6.1 können wir *V* ohne weiteres auf `x > 0` verstärken und $[V \wedge$ `x > 0`$]\ P\ [N]$ gilt. ⌟

6.2 Fehler

Betrachten wir ein Programm P und eine Spezifikation (V, N). Ist P nicht partiell korrekt, so nennen wir es **fehlerhaft**. Dann existiert nach Definition 6.6 ein Zustand $\sigma \vDash V$, so dass P angewendet auf σ zu einem der folgenden Ergebnisse führt:

1. $P\,\sigma$ bleibt stecken.
2. $P\,\sigma$ hat abgebrochen.
3. $P\,\sigma \Downarrow \sigma'$ und $\sigma' \nvDash N$.

Definition 6.7 (Fehlschlag (engl. failure))**.** Ein Zustand σ heißt Fehlschlag von P bezüglich der Spezifikation (V, N), wenn er eine der drei oben genannten Bedingungen erfüllt. ⌟

Bemerkung 6.2. Bleibt P stecken, ist dessen Semantik undefiniert und das Maschinenprogramm, das der Compiler für P erzeugt, kann (a) divergieren, (b) abbrechen, oder in einem Zustand enden, der die Nachbedingung (c) **erfüllt** oder (d) nicht erfüllt. Das bedeutet, dass wir im Falle von Steckenbleiben einen Fehlschlag durch die Ausführung des Programms unter Umständen **nicht** beobachten können, nämlich genau in den Fällen (a) und (c). Will man, wie beim Testen, Fehlschläge beobachten, ist es daher in Praxis wichtig, Steckenbleiben zu verhindern, beispielsweise durch sogenannte Sanitizer, die undefiniertes Verhalten zur Laufzeit erkennen und in Abbrüche verwandeln. ⌟

Bemerkung 6.3. Ist ein Programm partiell aber nicht total korrekt, so gibt es eine gültige Eingabe σ auf der das Programm divergiert. Existiert für ein Programm also kein Fehlschlag, so können wir daraus nur die partielle Korrektheit des Programms schließen. ⌟

Ein Fehlschlag ist der Beleg für einen Fehler von P bezüglich (V, N). Aus einem Fehlschlag kann man aber im Allgemeinen nicht direkt erkennen, an welchen Stellen das Programm **fehlerhaft** ist. Ein Fehlschlag ist also ein **Symptom** und keine Diagnose.

Beispiel 6.5. Betrachten wir die folgenden beiden Unterprogramme:

```
unsigned min(unsigned x, unsigned y) {
    if (x < y)
        return y;
    else
        return x;
}

unsigned sum_first_n(unsigned *arr, unsigned sz, unsigned n) {
    unsigned sum = 0;
    n = min(n, sz);
    for (unsigned i = 0; i < n; i++)
        sum += arr[i];
    return sum;
}
```

Die Spezifikation ist, wie in der Praxis oft üblich, in Form von Prosa gegeben und lautet:

> Das Unterprogramm `sum_first_n` soll die ersten `n`, jedoch höchstens `sz`, Elemente der Reihung `arr` aufsummieren und zurückliefern.

Betrachten wir die folgenden Aufrufe an `sum_first_n`:

```
unsigned arr[] = { 1, 2, 3 };
unsigned r1 = sum_first_n(arr, 3, 2); // (1)
unsigned r2 = sum_first_n(arr, 3, 3); // (2)
unsigned r3 = sum_first_n(arr, 3, 4); // (3)
```

(1) Der Aufruf ist ein Fehlschlag, denn nach unserer Spezifikation muss das Ergebnis `r1 = 3` sein, es ist aber `r1 = 6`.

(2) ist kein Fehlschlag.

(3) führt dazu, dass das Programm stecken bleibt: Die letzte Iteration der Schleife im Unterprogramm `sum_first_n` will auf das Element `arr + 3` zugreifen, das aber nicht existiert. Das Verhalten dieses Zugriffs ist in C undefiniert und das Programm bleibt stecken.

    In Bemerkung 6.2 haben wir diskutiert, dass wir in diesem Fall keine Annahmen über den weiteren Ablauf der **Implementierung** von `sum_first_n` treffen können. Es könnte also sein, dass auch dieser Aufruf ohne Abbruch zurückkehrt und ein Ergebnis liefert. Dieses könnte sogar richtig sein, obwohl das C-Programm stecken blieb!

Die **Ursache (engl. cause)** für den Fehlschlag in (1) und das Steckenbleiben in (3) ist ein **Fehler (engl. defect, error, bug)** im Unterprogramm `min`. Dieses berechnet nicht das Minimum, sondern das Maximum. ⌟

An diesem Beispiel sieht man deutlich, dass ein Fehlschlag wenig Auskunft über die Fehlerursache gibt. Insbesondere ist es schwer durch ein Fehlverhalten den Fehler zu lokalisieren: Der erste Aufruf bringt das Programm nicht zum Absturz; es wird einfach ein falscher Wert zurück gegeben. Der dritte Aufruf führt zum Steckenbleiben, was wir nur beobachten können, wenn wir einen Sanitizer verwenden, der die Semantik „totalisiert".

Die Fehlersuche wird erheblich vereinfacht, wenn man dafür sorgt, dass das Programm so schnell wie möglich abbricht, wenn ein Fehler aufgetreten ist. Es rechnet dann nicht mit falschen Werten weiter und entfernt sich dadurch von der Ursache des Fehlschlags. Wir wollen also Fehler so früh wie möglich **beobachten**. Dies erreichen wir dadurch, dass wir nicht mit fehlerhaften Zuständen weiter rechnen, sondern einen Abbruch herbeiführen. Hierzu machen wir die Spezifikation, oder Teile davon, durch Zusicherungen (engl. assertions) im Code explizit. Viele Programmiersprachen, wie auch C und Java, bieten Konstrukte (sogenannte assertions), um Zusicherungen zur Laufzeit des Programms zu prüfen. Erfüllt ein Zustand die Zusicherung nicht, bricht das Programm ab.

Definition 6.8 (Zusicherung (engl. assertion))**.** Wir definieren `assert(`*e*`);` als Abkürzung für `if (e) ; else abort();`. ⌟

Diese Konstrukte sind speziell gekennzeichnet und können meist mit einem Übersetzer-Schalter deaktiviert werden, um die Laufzeit der Prüfung zu sparen, wenn man das Programm für „hinreichend richtig" hält.

Beispiel 6.6 (Minimum mit Assertions)**.**

```
#include <assert.h>
unsigned min(unsigned x, unsigned y) {
    unsigned m = x < y ? x : y;
    assert(m <= x && m <= y && (m == x || m == y));
    return m;
}
```

⌟

Die Bedingung, die das Argument von `assert` darstellt, wird ausgewertet. Ist sie 0, so ruft `assert` die Funktion `abort` auf, die die Programmausführung abbricht. So erreicht man ein Prüfen der Nachbedingung an der Stelle, an der das Programm steht, das die Nachbedingung erfüllen soll.

Genauso verfährt man mit Code, der von einer Vorbedingung abhängt, die nicht `true` ist, wie in folgendem Beispiel.

Beispiel 6.7 (Minimum einer Reihung)**.** Wollen wir das minimale Element einer Reihung bestimmen, so muss die Reihung mindestens ein Element beinhalten, weil nicht definiert ist, was das Minimum der leeren Reihung ist. Es ist also schlicht nicht zulässig, folgendes Unterprogramm mit einer leeren Reihung aufzurufen. Wir stellen die Vorbedingung durch die Zusicherung `assert(n > 0);` sicher. Danach können wir das erste Element der Reihung laden um die Variable `r` zu initialisieren, denn Aufrufe mit `n == 0` werden an der Zusicherung scheitern.

```
unsigned min_arr(unsigned* arr, unsigned n) {
    assert(n > 0);
    unsigned res = arr[0];
    for (unsigned i = 1; i < n; i++)
        res = min(res, arr[i]);
    return res;
}
```

⌟

Bemerkung 6.4 (Defensives Programmieren)**.** Defensiv Programmieren bedeutet, dass man beim Programmieren nicht nur gültige Eingaben bedenkt. Man berücksichtigt alle, also auch falsche Eingaben. Man bemüht sich also besonders darum, dass die Vorbedingungen aller Unterprogramme eingehalten werden. Oft wird dies dahingehend missverstanden, dass das Programm möglichst auch im Fehlerfall weiter laufen soll. Dies versuchen manche Programmierer dadurch zu erreichen, dass sie Pseudo-Ergebnisse „erfinden", die dann im Fehlerfall zurückgeliefert werden.

Folgendes Beispiel zeigt die Problematik. Das Unterprogramm `min_idx` soll den Index des kleinsten Elements einer Reihung zurück geben. Wie oben ist dieser Index im Fall der leeren Reihung nicht definiert, der Programmierer hat aber hier den „Pseudo-Wert“ `-1` zurückgegeben:

```
int min_arr(int* arr, unsigned n) {
    if (n == 0)
        return -1;
    int min = arr[0];
    int idx = 0;
    for (unsigned i = 1; i < n; i++) {
        if (arr[i] < min) {
            min = arr[i];
            idx = i;
        }
    }
    return idx;
}
```

Durch die Rückgabe von `-1` ist niemandem geholfen. Das Unterprogramm, das `min_arr` aufruft, kann diesen Wert nicht verwenden, um die Reihung zu indizieren. Es muss also seinerseits prüfen, ob `-1` zurückkam und diesen Fall gesondert behandeln:

```
    ...
    int res = min_arr(a, n);
    if (res == -1) {
        printf(”something strange happened...\n”):
    }
```

In diesem Fall war das rufende Unterprogramm wohl selbst nicht dafür gewappnet, dass `a` leer sein kann. Es hat also offensichtlich seine Vorbedingung auch nicht sichergestellt. Hier besteht die Gefahr, dass falsche Werte (`-1`) durch das Programm gereicht werden, das dann irgendwann mit einer unerwarteten Ausgabe terminiert. Das hat aber nur zur Konsequenz, dass das Symptom des Fehlers (die falsche Ausgabe) im Programmablauf weit von der Fehlerursache entfernt wird, was es unnötig schwer macht, den Fehler zu finden.

Richtiges defensives Programmieren platziert eine Zusicherung an den Beginn von `min_arr` die **prüft und dokumentiert**, dass die Vorbedingung von `min_arr` `n > 0` erfüllt ist:

```
int min_arr(int* arr, unsigned n) {
    assert(n > 0);
    ...
```

⌟

Bemerkung 6.5 (Undefiniertes Verhalten)**.** Zusicherungen eignen sich auch gut, um Fehler zu finden, die in undefiniertem Verhalten resultieren. Nach Kapitel 5 gibt es keine Aussage darüber, wie sich das Maschinen-Programm $\hat{P}$ eines C0-Programm P verhält, wenn P stecken bleibt. Macht man die Bedingungen unter denen P stecken bleibt ex-

plizit, so bricht P (und somit auch $\hat{P}$) ab, wenn an der entsprechenden Programmstelle undefiniertes Verhalten auftreten würde:

```
int arr[n];
/* ... */
unsigned idx = compute_idx(...);
assert(idx < n); // idx >= 0, because it is unsigned
printf("The n-th element is: %d\n", arr[idx]);
```

⌟

6.3 Testen

Testen ist eine Technik zur Fehlersuche in Programmen. Beim Testen konstruiert man Eingaben an ein Programm (oder an einen Programmteil, z.B. ein Unterprogramm), führt es darauf aus und beurteilt, ob das Ergebnis korrekt ist. Ist das Ergebnis inkorrekt – man sagt, der Test ist fehlgeschlagen – so ist das Programm fehlerhaft. Schlägt keiner der vorhandenen Tests fehl, kann man keine Aussage über die Korrektheit des Programms treffen. Edsger Dijkstra prägte folgende Aussage, die den Kern des Testens beschreibt:

> Testing shows the presence, not the absence of bugs.

Definition 6.9 (Test (engl. test case))**.** Gegeben ein Programm P und eine Spezifikation (V, N). Ein Zustand σ, der die Vorbedingung erfüllt (sprich $\sigma \vDash V$) ist ein Test von P. Ein Test ist **bestanden**, wenn $P\,\sigma \Downarrow \sigma'$ und $\sigma' \vDash N$. Ist $P\,\sigma$ ein Fehlschlag, so ist der Test σ nicht bestanden. Eine Menge von Tests nennen wir **Testsuite**. ⌟

Beispiel 6.8 (Maximum)**.** Betrachten wir die Spezifikation zur Berechnung des Maximums aus Beispiel 6.3 und das folgende C0-Programm P:

```
if (x > y)
    m = x;
else
    m = x;
```

Das C0-Programm besteht den Test $\sigma_{✓} = \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 1\}$. Der Test $\sigma_{×} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$ wird hingegen nicht bestanden. ⌟

Häufig verwenden wir ein Programm P_σ um den Test σ herzustellen. Dieses Programm ruft dann das zu testende Programm auf und stellt sicher, dass der Endzustand die Nachbedingung erfüllt:

```
void test1() {
    unsigned x = 2;
    unsigned y = 1;
    unsigned m = max(2, 1);
    assert(m == x);
}
```

Testen wir einzelne Module oder Unterprogramme, so sprechen wir von **Modultests (engl. unit test)**. Beim Modultesten zerlegt man das Programm in einzelne Module, meist ein Unterprogramm, oder mehrere zu einem Verbund gehörende Unterprogramme und testet dieses unabhängig von Rest des Programms.

Modultests sind hilfreich um Fehler in den einzelnen Modulen zu isolieren. Hat man keine Modultests, können Fehler in Modul A leicht zu Fehlschlägen führen, die man „weit entfernt“ von A in einem anderen Modul beobachtet. Solche Fehler sind dann schwer zu finden. Des Weiteren verwendet man Modultests um **Regressionen** zu verhindern: Bucht man Änderungen an einem Modul in eine Versionsverwaltung ein, so muss sichergestellt werden, dass man nicht weniger Tests besteht als zuvor. So erschwert man das Hinzufügen von Fehlern.

Beim Testen wird häufig zwischen zwei Ausprägungen unterschieden:

Funktionsbasiertes Testen (black-box testing). Die Tests werden nur anhand der Spezifikation entworfen. Der zu testende Code hat keinen Einfluss auf die Gestalt der Tests.

Dies hat den Vorteil, dass man eventuell nicht implementierte Fälle der Spezifikation entdecken kann. Der Nachteil ist allerdings, dass die Tests bestimmte Code-Pfade eventuell nicht ausführen, die dann ungetestet bleiben.

Strukturbasiertes Testen (auch white-box testing). Man nutzt die Kenntnis des Quellcodes um Tests zu entwerfen, die soviel Code des zu testenden Programms (SUT = subject under test) wie möglich auszuführen. Man versucht, anhand bestimmter Metriken (siehe Abschnitt „Abdeckung"), so viele Pfade des Moduls wie möglich **abzudecken**. Besteht ein Programm eine Testsuite, die es gut abdeckt, ist das noch nicht zwingend ein Indiz dafür, dass das Programm wenig Fehler hat. Bestimmte Fälle der Spezifikation könnten schlicht nicht implementiert sein.

In der Praxis wird beim Modultest beides eingesetzt. Tester (Personen, die Code testen, aber nicht entwickeln) führen meist black-box Tests durch um sich nicht vom zu testenden Code beeinflussen zu lassen und eventuell Fälle zu vergessen. Des Weiteren ist es oft üblich, dass Tests vor, oder zumindest während des Implementierens erstellt werden. Entwickler neigen eher zu white-box Tests, um so viel von ihrem Code wie möglich auszuführen. Allerdings müssen sie sich auch der Spezifikation bedienen, um die Nachbedingung der Tests überprüfen zu können.

Neben Modultests unterscheidet man noch zwischen **Integrationstests (engl. integration test)**, die das Zusammenspiel mehrerer Module testen und **Systemtests (engl. system test)**, die das gesamte Programm testen. Die Begriffe Modul- und Systemtest bezeichnen keine Test-Technik an sich, sondern lediglich den Umfang des Tests.

6.3.1 Das Orakelproblem

In der Praxis kommt es häufig vor, dass keine Spezifikation explizit vorliegt. Sie existiert in diesem Fall nur in den Köpfen der Programmierer und liegt weder formal noch in Prosa vor. Teile der Spezifikation können jedoch durch Zusicherungen (siehe Definition 6.2) im Programm dokumentiert sein.

Man zieht sich daher auf die Spezifikation (**`true`**, **`true`**) zurück. Diese Spezifikation beschreibt alle Programmläufe, die nicht abbrechen oder divergieren (was in der Regel die Mindestanforderung an ein brauchbares Programm ist). Sind Teile der Spezifikation im Programm durch Zusicherungen dokumentiert, so schlagen Tests, die diese auslösen, fehl. Wir beobachten dann einen Fehlschlag, dessen Ursache wir beheben können. Man beachte, dass die Vorbedingung **`true`** auch Zustände beschreibt, die die Vorbedingung (der nicht explizit vorliegenden Spezifikation) nicht erfüllen. Bei einer Ausführung des Programms auf einem solchen Zustand beobachten wir möglicherweise unerwünschtes

Verhalten (Abbruch, Fehlschlag, Stecken bleiben), das aber nicht notwendigerweise auf einen Fehler im Programm hindeutet. Gute Software sollte jedoch bei einer ungültigen Eingabe abbrechen.

6.3.2 Funktionsbasiertes (black-box) Testen

Beim funktionsbasierten Testen betrachtet man das zu testende Programm (engl. subject under test) nicht. Man erstellt die Tests einzig und allein anhand der Spezifikation. Betrachten wir folgendes Beispiel [PY07].

Beispiel 6.9. Gegeben ist die Spezifikation eines Unterprogramms, das die Nullstellen von quadratischen Polynomen berechnet.

> Das Unterprogramm
>
> ```
> unsigned roots(double a, double b, double c, double *r);
> ```
>
> soll die drei Koeffizienten eines quadratischen Polynoms entgegennehmen und die reellen Wurzeln bestimmen. Die Anzahl der Wurzeln soll zurückgegeben werden. Hat das Polynom mehr als zwei reelle Wurzeln, so soll der Rückgabewert `UINT_MAX` sein. In diesem Fall wird die Reihung `r` nicht beschrieben.
>
> Die Wurzeln selbst sollen in die Reihung `r` geschrieben werden. Diese muss mindestens zwei **`double`** Werte aufnehmen können.

Wie wir wissen, kann es nun mehrere Fälle geben, die das Unterprogramm behandeln muss.

1. Das Polynom kann das Null-Polynom sein. Dann muss `roots` den Wert `UINT_MAX` zurückgeben.
2. Das Polynom kann linear sein $a = 0$. Dann hat es eine ($b \neq 0$) oder keine ($b = 0$) Nullstelle.
3. Das Polynom kann quadratisch sein $a \neq 0$, dann kann es zwei, eine oder keine Nullstelle haben. Jeder Fall muss einzeln geprüft werden.

⌟

Ein typisches Vorgehen beim black-box Testen ist die Unterteilung der Eingabe in **Äquivalenzklassen**. Für jede Äquivalenzklasse (in unserem Beispiel: Polynom ist linear, Polynom ist Null-Polynom, etc.) wird dann je ein Test erzeugt, der für die Klasse repräsentativ ist. Die Intuition dahinter ist, dass jeder Fall auch im Code eigens geprüft und behandelt werden muss. Man versucht durch eine Partitionierung des Eingaberaums eine geeignete Abdeckung des Codes zu erreichen und zu überprüfen, ob tatsächlich alle Fälle der Spezifikation implementiert wurden.

In unserem Beispiel sind folgende Tests geeignet, aber nicht ausreichend. Weitere können Sie in Aufgabe 6.2 entwickeln.

```
void test_null(void) {
    unsigned res = roots(0, 0, 0, NULL);
    assert(res == UINT_MAX);
```

```
}

void test_linear(void) {
    double r[2];
    unsigned res = roots(0, 2, 2, r);
    assert(res == 1 && r[0] == -1);
}

void test_linear_none(void) {
    double r[2];
    unsigned res = roots(0, 0, 2, r);
    assert(res == 0);
}
```

6.3.3 Abdeckung

Beim strukturbasierten (white-box) Testen spielt die **Abdeckung (engl. coverage)** des Programmes durch Tests eine wesentliche Rolle. Diese bezeichnet den Teil des Programms, der durch die Tests ausgeführt wird. Gegeben sei ein Programm P. Um die einzelnen Anweisungen des Programms eindeutig identifizieren zu können, definieren wir eine injektive Funktion $lab : \{1, \ldots, n\} \to Stmt$, die die Anweisungen in unserem Programm geeignet durchnummeriert.

Definition 6.10 (Anweisungsabdeckung (engl. statement coverage))**.** Ein Test σ deckt eine Anweisung k ab, wenn in der Ausführung $P\,\sigma$ eine Konfiguration $\langle P_i \mid \sigma_i \rangle$ existiert, mit $lab\, k = P_i$. ⌟

Definition 6.11 (Zweigabdeckung (engl. branch coverage))**.** Ein Test σ deckt den Zweig (k_1, k_2) ab, wenn in der Ausführung $P\,\sigma$ zwei Konfigurationen $\langle lab\, k_1 \mid \sigma_1 \rangle$ und $\langle lab\, k_2 \mid \sigma_2 \rangle$ aufeinanderfolgen. ⌟

Bemerkung 6.6. Zweigabdeckung unterscheidet sich auf unserer Sprache nicht von der Anweisungsabdeckung: Für jeden Zweig findet sich auch eine Anweisung, deren Abdeckung die des Zweiges impliziert. Hat man Fallunterschediungen ohne Alternativ-Fall, wie zum Beispiel

```
r = 0;
if (x < 0)
    r = -x;
/* nächste Anweisung */
```

so muss man zwischen Anweisungs- und Zweigabdeckung unterscheiden: In diesem Beispiel deckt der Testfall $\sigma = \{x \mapsto -1\}$ alle Anweisungen aber nicht alle Zweige ab. ⌟

Definition 6.12 (Pfadabdeckung (engl. path coverage))**.** Ein Test σ deckt einen Pfad $p : k_1, \ldots, k_m$ ab, wenn in der Ausführung $P\,\sigma$ die Folge von Konfigurationen $\langle lab\, k_1 \mid \sigma_1 \rangle \to \cdots \to \langle lab\, k_m \mid \sigma_m \rangle$ vorkommt. ⌟

Bemerkung 6.7. Da die Menge der unterschiedlichen Pfade durch ein Programm riesig, wenn nicht gar unendlich sein kann (im Falle von Schleifen), hat man in der Praxis keine Testsuites, die große Pfad-Abdeckung erreichen. ⌟

Beispiel 6.10. Betrachten wir eine fehlerhafte Implementierung der Spezifikation aus Beispiel 6.9.

```
unsigned roots_buggy(double a, double b, double c, double *r) {
    if (a != 0) {                          // 1
        double p = b / a;                  // 2
        double q = c / a;                  // 3
        double d = p * p / 4 - q;          // 4

        if (d > 0) {                       // 5
            r[0] = -p / 2 + sqrt(d);       // 6
            r[1] = -p / 2 - sqrt(d);       // 7
            return 2;                      // 8
        }

        r[0] = -p / 2;                     // 9
        return 1;                          // 10
    }

    if (b != 0) {                          // 11
        r[0] = -b / c;                     // 12
        return 1;                          // 13
    }

    return 0;                              // 14
}
```

Die Tests aus Beispiel 6.9 decken folgende Anweisungen ab: 1, 11, 12, 13, 14. Die abgedeckten Pfade sind

`test_null`	1, 11, 14
`test_linear`	1, 11, 12, 13, 14
`test_linear_none`	1, 11, 14

Der erste Test wird nicht bestanden und findet den Fehler, dass das Polynom 0 nicht korrekt behandelt wird. Der zweite Test wird bestanden und findet den Fehler in Anweisung 12 nicht, die korrekt `r[0] = -c / b;` heißen müsste. Hier zeigt sich, dass die Eingabe $\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 2\}$ schlecht ist, da man die Vertauschung von Dividend und Divisor nicht bemerkt. Ein besserer Test wäre hier:

```
void test_linear(void) {
    double r[2];
    unsigned res = roots(0, 4, 2, r);
    assert(res == 1 && r[0] == -0.5);
}
```

Der letzte Test wird bestanden, da die Funktion im Falle eines konstanten Polynoms $\neq 0$ korrekt ist. Keiner der Tests entdeckt, dass ein Fall für quadratische Gleichungen nicht implementiert ist: Ist die Diskriminante kleiner 0, so liefert die Funktion fälschlicherweise eine Nullstelle mit dem Wert $-p/2$.

Die Testsuite hat keine gute Anweisungsabdeckung. Die Anweisungsabdeckung beträgt

$$\frac{|\{1, 11, 12, 13, 14\}|}{|\{1, \ldots, 14\}|} = \frac{5}{14} \approx 35{,}7\%$$

Die Pfadabdeckung liegt bei 50%. Es ist selten, dass die Pfadabdeckung höher als die Anweisungsabdeckung ist. Dies liegt hier daran, dass das Unterprogramm keine Schleife und keine Fallunterscheidungen enthält, die nacheinander ausgeführt werden können (jeder Pfad endet hier in einem **`return`** bevor die nächste Fallunterscheidung kommt). ⌟

Bemerkung 6.8. Testsuites, die nicht alle Anweisungen abdecken, sind unzureichend. ⌟

Wie gut man mit einer Testsuite Fehler finden kann, bemisst sich nicht nur allein anhand der Abdeckung. Fehlt Code, der eine bestimmte verlangte Funktionalität implementiert, so hilft eine hohe Abdeckung nicht weiter. Hohe Abdeckung ist also notwendig, aber nicht hinreichend.

6.3.4 Zufallgesteuertes Testen (fuzzing)

Eine weitere Technik um Tests zu erzeugen, ist das Fuzzing. Hierbei schreibt man ein Programm, das (mehr oder weniger) zufällig Eingaben erzeugt. Fuzzing wird erst dann möglich, wenn man ein Verfahren hat, mit dem man leicht Eingaben erzeugen kann, die die Vorbedingung erfüllen. Je nach zu testendem Programm kann das sehr schwierig sein.

Fuzzing ist ein black-box Verfahren, da man die Implementierung des zu testenden Programms ignoriert. Je nachdem, wie kompliziert die Spezifikation ist, kann es schwierig sein, ein geeignetes Orakel für einen Fuzz-Test zu finden, da man für die zufällig bestimmten Eingaben die korrekte Ausgabe kennen muss. Oft wird Fuzz-Testen eingesetzt, um schwächere Spezifikationen zu testen, zum Beispiel, „das Programm bricht nicht ab“, oder „bleibt nicht stecken“.

Eine andere weit verbreitete Technik ist das Fuzzen einer Implementierung A unter Verwendung einer einfacheren (meist ineffizienteren) Implementierung B als Orakel.

6.4 Schwächste Vorbedingungen

Wir haben gesehen, dass man durch Testen die Fehlerhaftigkeit eines Programms nachweisen kann, jedoch nicht seine Korrektheit. In diesem Abschnitt wollen wir untersuchen, wie man ein Hoare-Tripel $[V]\ P\ [N]$ formal beweisen kann. Man spricht dabei auch von der formalen Verifikation des Programms P bezüglich der Spezifikation (V, N).

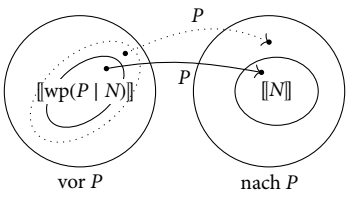
Hierzu übersetzen wir das Verifikationsproblem in ein Logik-Problem: Nehmen wir an, wir könnten aus einem Programm P und einer Nachbedingung N eine Formel $\text{wp}(P \mid N)$ mit der folgenden Eigenschaft konstruieren:

$$\sigma \vDash \text{wp}(P \mid N) \iff \langle P \mid \sigma \rangle \Downarrow \sigma' \wedge \sigma' \vDash N \tag{6.1}$$

Die erfüllenden Zustände von $\text{wp}(P \mid N)$ sind genau die Zustände, unter denen P in einem Zustand terminiert, der N erfüllt. Wenn nun jeder Zustand, der die Vorbedingung V erfüllt, auch $\text{wp}(P \mid N)$ erfüllt, ist das Programm korrekt. Es gilt:

$$V \Rightarrow \text{wp}(P \mid N) \quad \text{genau dann wenn} \quad [V]\ P\ [N] \tag{6.2}$$

Die Zusicherung $\text{wp}(P \mid N)$ heißt **schwächste Vorbedingung (engl. weakest precondition)** von P unter N. Sie heißt *schwächste* Vorbedingung, weil sie die **größte** Menge von Zuständen beschreibt, unter der P in N terminiert. Jede schwächere Bedingung enthielte einen Zustand, unter dem P nicht in N terminiert.



Im nächsten Abschnitt werden wir sehen, dass wir die Funktion

$$\text{wp} : \mathit{Stmt} \to \mathit{Expr} \to \mathit{Expr}$$

für jede Anweisung der Sprache C0, außer der Schleife, direkt angeben können. Die Argumente von wp sind **Syntax**: wp nimmt eine Anweisung und eine Zusicherung (in Form eines Ausdrucks[2]) und liefert eine Zusicherung. Wir nennen sie daher auch **syntaktische** schwächste Vorbedingung. Somit können wir ein schleifenfreies Programm P direkt in die Verifikationsbedingung

$$V \Rightarrow \text{wp}(P \mid N)$$

übersetzen.

[2] wir verwenden im Folgenden der besseren Lesbarkeit halber die Symbole $\wedge, \vee, \ldots$ anstatt &&, ||, ...

Für Schleifen kann man im Allgemeinen keine schwächste Vorbedingung in Form einer Zusicherung angeben. Allerdings kann man die **Zustandsmenge** der schwächsten Vorbedingung charakterisieren. Wir werden dann zeigen, dass spezielle Zusicherungen, sogenannte **Schleifeninvarianten**, stärker sind als die schwächste Vorbedingung einer Schleife, sie also implizieren. Daraus werden wir ein Verifikationsverfahren für Programme der Sprache C0 konstruieren.

6.4.1 Die schwächste Vorbedingung einfacher Anweisungen

Definition 6.13 definiert die syntaktische schwächste Vorbedingung für alle Anweisungen der Sprache C0, außer der Schleife. Im Folgenden wollen wir die Definitionen aus Definition 6.13 informell besprechen und an einfachen Beispielen ausprobieren. Im nächsten Abschnitt diskutieren wir die Korrektheit dieser Definitionen.

Definition 6.13 (Syntaktische schwächste Vorbedingung)**.**

Anweisung s	$\mathrm{wp}(s \mid N)$
`;`	N
`abort();`	**`false`**
`x = e;`	$\mathrm{def}\, e \wedge N[e/x]$
$s_1 \_ s_2$	$\mathrm{wp}(s_1 \mid \mathrm{wp}(s_2 \mid N))$
`if` …	$\mathrm{def}\, e \wedge \big[(e \wedge \mathrm{wp}(s_1 \mid N)) \vee (\neg e \wedge \mathrm{wp}(s_2 \mid N))\big]$

Die Funktion $\mathrm{def}\, e$ liefert eine Formel, die die Menge aller Zustände beschreibt, auf denen e definiert ist:

$$\sigma \vDash \mathrm{def}\, e \quad \text{genau dann wenn} \quad \exists v.\ [\![e]\!]\, \sigma = v$$ ⌟

Die leere Anweisung. Die schwächste Vorbedingung der leeren Anweisung ist die Nachbedingung, da die leere Anweisung den Zustand nicht verändert.

Der Abbruch. Die schwächste Vorbedingung von `abort();` ist **`false`**, da für keinen Zustand die Ausführung von `abort();` in irgendeinem Zustand endet, sondern in der Konfiguration $\lightning$. Die Zusicherung **`false`** wird exakt von der leeren (Zustands-) Menge erfüllt.

Die Zuweisung. Die Definition sagt, dass man die schwächste Vorbedingung aus der Nachbedingung erhält, indem man die Variable darin durch den Term e ersetzt (ausgedrückt durch $n[e/\mathtt{x}]$) und mit einer Bedingung $\mathrm{def}\, e$ konjugiert, die alle Zustände beschreibt, unter denen e auswertbar ist. Intuitiv kann man sich diese Definition so klar machen:

Die Zuweisung setzt x auf $[\![e]\!]$. Vor der Ausführung der Zuweisung hat muss also die Bedingung gelten, die aus der Nachbedingung hervorgeht, indem man x durch $[\![e]\!]$ ersetzt.

Die Hintereinanderausführung. Die schwächste Vorbedingung der Anweisung s_2 ist exakt die Zustandsmenge für die S_2 in einem Zustand der Nachbedingung terminiert. Also ist die schwächste Vorbedingung der Hintereinanderausführung der Anweisungen s_1 und s_2 also die Menge aller Zustände für die s_1 in der schwächsten Vorbedingung von s_2 terminiert.

Die Fallunterscheidung. Die schwächste Vorbedingung der Fallunterscheidung setzt sich aus den schwächsten Vorbedingungen der

Konsequenz s_1 und der Alternative s_2 zusammen. Die Zustände der schwächsten Vorbedingung von s_1 (s_2) sind Teil der schwächsten Vorbedingung der Fallunterscheidung, wenn auf ihnen die Bedingung e ($\neg e$) erfüllt ist.

Betrachten wir nun einige Beispiele indem wir die syntaktische Definition der schwächsten Vorbedingung auf einige Programme anwenden. Mithilfe von (6.2) können wir so schon einfache Verifikationsprobleme lösen.

Beispiel 6.11 (Zuweisung)**.**

1. $\mathrm{wp}(\texttt{x = 2} \mid \texttt{x > 0}) = (\texttt{2 > 0}) \equiv \textbf{true}$
2. $\mathrm{wp}(\texttt{x = 2 * y} \mid \texttt{y = x}) = (\texttt{2 * y = y}) \equiv \texttt{y = 0}$
3. $\mathrm{wp}(\texttt{x = y + 1} \mid \texttt{x = 5}) = (\texttt{y + 1 = 5}) \equiv \texttt{y = 4}$
4. $\mathrm{wp}(\texttt{x = y / z} \mid \texttt{x = 5}) = (\texttt{z} \neq \texttt{0}) \wedge (\texttt{5 = y / z})$

In den ersten drei Beispielen ist jeweils $\mathrm{def}\, e = \textbf{true}$, da alle rechten Seiten auf allen Werten definiert sind. Die Konjunktion mit **true** wurde in den Beispielen der Übersichtlichkeit halber weggelassen. Im vierten Beispiel ist $\mathrm{def}\, \texttt{y / z} = \texttt{z} \neq \texttt{0}$. ⌟

Beispiel 6.12 (Vertauschen)**.** Mit den Regeln für die Zuweisung und den Block können wir die schwächste Vorbedingung des Programms

```
{
    t = x;
    x = y;
    y = t;
}
```

bezüglich der Nachbedingung $\texttt{X} = \texttt{y} \wedge \texttt{Y} = \texttt{x}$ ausrechnen.

$$
\begin{aligned}
& \mathrm{wp}(\texttt{t = x; x = y; y = t;} \mid \texttt{X = y} \wedge \texttt{Y = x}) \\
= \;& \mathrm{wp}(\texttt{t = x;} \mid \mathrm{wp}(\texttt{x = y;} \mid \mathrm{wp}(\texttt{y = t;} \mid \texttt{X = y} \wedge \texttt{Y = x}))) \\
= \;& \mathrm{wp}(\texttt{t = x;} \mid \mathrm{wp}(\texttt{x = y;} \mid \texttt{X = t} \wedge \texttt{Y = x})) \\
= \;& \mathrm{wp}(\texttt{t = x;} \mid \texttt{X = t} \wedge \texttt{Y = y}) \\
= \;& \texttt{X = x} \wedge \texttt{Y = y}
\end{aligned}
$$

Nach (6.2) gilt dann das Hoare-Tripel

$$[\texttt{X = x} \wedge \texttt{Y = y}]\ \texttt{t = x; x = y; y = t;}\ [\texttt{X = y} \wedge \texttt{Y = x}]$$

und wir haben **bewiesen**, dass obiges Programm den Inhalt der beiden Variablen x und y vertauscht. ⌟

Beispiel 6.13 (Minimum)**.** Die Spezifikation für ein Programm, welches das Minimum der Variablen x und y berechnet und in der Variable r ablegt, ist:

$$V = \textbf{true} \qquad N = \texttt{r} \leq \texttt{x} \wedge \texttt{r} \leq \texttt{y} \wedge (\texttt{r = x} \vee \texttt{r = y})$$

Wir wollen beweisen, dass das folgende Programm P bezüglich dieser Spezifikation korrekt ist.

```
{
    if (x < y)
        r = x;
    else
        r = y;
}
```

Hierzu berechnen wir die schwächste Vorbedingung des Programms bezüglich der Nachbedingung der Spezifikation und prüfen, gemäß (6.2), ob diese schwächer ist, als die Vorbedingung der Spezifikation.

$$\begin{aligned} \mathrm{wp}(P \mid N) &= (\mathtt{x} < \mathtt{y} \wedge \mathrm{wp}(\mathtt{r = x;} \mid N)) \vee (\mathtt{x} \geq \mathtt{y} \wedge \mathrm{wp}(\mathtt{r = y;} \mid N)) \\ &\equiv (\mathtt{x} < \mathtt{y} \wedge \mathtt{x} \leq \mathtt{x} \wedge \mathtt{x} \leq \mathtt{y} \wedge (\mathtt{x} = \mathtt{x} \vee \mathtt{x} = \mathtt{y})) \\ &\vee (\mathtt{x} \geq \mathtt{y} \wedge \mathtt{y} \leq \mathtt{x} \wedge \mathtt{y} \leq \mathtt{y} \wedge (\mathtt{y} = \mathtt{x} \vee \mathtt{y} = \mathtt{y})) \\ &\equiv (\mathtt{x} \leq \mathtt{y}) \vee (\mathtt{x} \geq \mathtt{y}) \equiv \mathtt{true} \end{aligned}$$

Damit und mit (6.2) ist das Hoare-Tripel $[V]\ P\ [N]$ bewiesen und das Programm korrekt. ⌟

6.4.2 Korrektheit

Wir wollen nun untersuchen, warum die Definition der schwächsten Vorbedingung, wie wir sie im letzten Abschnitt besprochen haben, korrekt ist. Hierzu müssen wir für jeden Fall in Definition 6.13 zeigen, dass sie (6.1) erfüllt. Hierzu definieren wir die **semantische** schwächste Vorbedingung:

Definition 6.14 (semantische schwächste Vorbedingung)**.**

$$\mathrm{Wp} : \mathit{Stmt} \to \mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma) \quad \mathrm{Wp}(P \mid N) \coloneqq \{\sigma \mid \langle P \mid \sigma \rangle \Downarrow \sigma' \wedge \sigma' \in N\}$$ ⌟

Die Funktion Wp bildet keine Zusicherungen auf Zusicherungen ab, sondern Mengen auf Mengen. Sie liefert die **Menge** der Zustände, unter denen P in einem Zustand der Nachbedingung terminiert. Sie heißt *semantische* schwächste Vorbedingung, weil sie durch die Semantik C0 der Sprache definiert ist: In der Definition von Wp taucht die Ableitungsrelation von C0 direkt auf. Folgender Satz gibt uns die Korrektheit von wp:

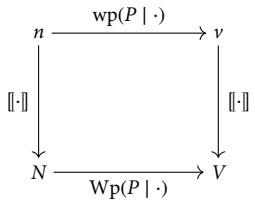


Abbildung 6.2: Diagramm zu Satz 6.2.

Satz 6.2. *Für jede der Anweisungen s in Definition 6.13 und jede Zusicherung n gilt:*

$$\mathrm{Wp}(s \mid [\![n]\!]) = [\![\mathrm{wp}(s \mid n)]\!]$$

Beweis. Wir beweisen den Satz durch Induktion über die Syntax der Sprache C0. Die leere Anweisung und der Programmabbruch folgen sofort durch Einsetzen der Regeln [Empty] und [Abort] in Definition 6.14.

Die Fallunterscheidung. Die Induktionshypothese ist:

$$\mathrm{Wp}(s_i \mid [\![n]\!]) = [\![\mathrm{wp}(s_i \mid n)]\!] \text{ für alle Zusicherungen } n \text{ und } 1 \leq i \leq 2$$

Aus [IfTrue] und [IfFalse] erhält man

$$\begin{aligned}
\mathrm{Wp}(\texttt{if}\ \cdots \mid [\![n]\!]) &= \mathrm{Def}\, e \cap \left[([\![e]\!] \cap \mathrm{Wp}(s_1 \mid [\![n]\!])) \cup ([\![\neg e]\!] \cap \mathrm{Wp}(s_2 \mid [\![n]\!])) \right] \\
&= [\![\mathrm{def}\, e]\!] \cap \left[([\![e]\!] \cap [\![\mathrm{wp}(s_1 \mid n)]\!]) \cup ([\![\neg e]\!] \cap [\![\mathrm{wp}(s_2 \mid n)]\!]) \right] && \text{Induktionshyopthese} \\
&= [\![\mathrm{def}\, e \wedge (e \wedge \mathrm{wp}(s_1 \mid n)) \vee (\neg e \wedge \mathrm{wp}(s_2 \mid n))]\!] && \text{Mengenlehre} \\
&= [\![\mathrm{wp}(\texttt{if}\ \cdots \mid n)]\!] && \text{Definition 6.13}
\end{aligned}$$

Die Hintereinanderausführung. Die Induktionshypothese ist:

$$\mathrm{Wp}(s_i \mid [\![n]\!]) = [\![\mathrm{wp}(s_i \mid n)]\!] \text{ für alle Zusicherungen } n \text{ und } 1 \leq i \leq 2$$

Wir zeigen zunächst, dass

$$\mathrm{Wp}(s_1 \_ s_2 \mid [\![n]\!]) = \mathrm{Wp}(s_1 \mid \mathrm{Wp}(s_2 \mid [\![n]\!])) \tag{6.3}$$

$\subseteq$: Sei σ ein Zustand in $\mathrm{Wp}(s_1 \_ s_2 \mid [\![n]\!])$. Nach Definition 6.14 gibt es dann einen Zustand $\sigma'' \vDash n$ in dem $s_1 \_ s_2$, angewendet auf σ, terminiert. In dieser Ausführung gibt es, gemäß Definition 5.1, eine Anwendung der Regel [Exec], die eine Konfiguration $\langle s_2 \mid \sigma' \rangle$ produziert, in der s_1 vollständig abgearbeitet wurde. Da s_2 (nach Voraussetzung) auf σ' terminiert, ist $\sigma' \in \mathrm{Wp}(s_2 \mid [\![n]\!])$. Da s_1 auf σ in σ' terminiert, gilt (6.3).

$\supseteq$: Sei $\sigma \in \mathrm{Wp}(s_1 \mid \mathrm{Wp}(s_2 \mid [\![n]\!]))$. Dann terminiert s_1 auf σ in einem Zustand $\sigma' \in \mathrm{Wp}(s_2 \mid [\![n]\!])$. Auf σ' terminiert nun wiederum s_2 in einem Zustand $\sigma'' \in [\![n]\!]$. Somit ist $\sigma \in \mathrm{Wp}(s_1 \_ s_2 \mid [\![n]\!])$.

Zuletzt nutzen wir die Induktionshypothese, um die rechte Seite von (6.3) entsprechend umzuschreiben.

Die Zuweisung. Die Zuweisung ist ein wenig komplizierter, da wir hier das Ersetzen der Variablen x im Ausdruck e in der syntaktischen Version mit dem Auswerten von e und dem Setzen von x in der semantischen Version abgleichen müssen.

$$\begin{aligned}
& [\![\mathrm{wp}(\mathrm{x} := e; \mid n)]\!] = \mathrm{Wp}(\mathrm{x} := e; \mid [\![n]\!]) \\
\iff\ & [\![\mathrm{def}(e) \wedge n[e/\mathrm{x}]]\!] = [\![\mathrm{def}(e)]\!] \cap \{\sigma \mid \sigma[\mathrm{x} \mapsto [\![e]\!]\sigma] \in [\![n]\!]\} \\
\impliedby\ & [\![n[e/\mathrm{x}]]\!] = \{\sigma \mid \sigma[\mathrm{x} \mapsto [\![e]\!]\sigma] \in [\![n]\!]\} \\
\impliedby\ & \forall \sigma.\ [\![n[e/\mathrm{x}]]\!]\sigma = [\![n]\!]\sigma[\mathrm{x} \mapsto [\![e]\!]\sigma]
\end{aligned}$$

Die letzte Aussage zeigen wir durch strukturelle Induktion über dem Ausdruck n:

1. Sei n eine Konstante $n = c$. Dann ist für alle σ:

   $$[\![c]\!]\sigma = c = [\![c]\!]\sigma[\mathrm{x} \mapsto [\![e]\!]\sigma]$$

2. Der Fall, in dem n eine Variable ungleich x ist, ist analog zu $n = c$.
3. Sei $n = \mathrm{x}$. Dann ist für alle σ:

   $$[\![\mathrm{x}[e/\mathrm{x}]]\!]\sigma = [\![e]\!]\sigma = [\![x]\!]\ \sigma[\mathrm{x} \mapsto [\![e]\!]\sigma]$$

4. Sei $n = e_1 \; o \; e_2$: Dann ist für alle σ:

$$
\begin{aligned}
[\![(e_1 \; o \; e_2)[e/\mathrm{x}]]\!]\,\sigma &= [\![e_1[e/x]]\!]\,\sigma \; o \; [\![e_2[e/x]]\!]\,\sigma && \text{Definition 5.2} \\
&= [\![e_1]\!]\,\sigma' \; o \; [\![e_2]\!]\,\sigma' && \text{IH} \\
&\quad \text{mit } \sigma' = \sigma[x \mapsto [\![e]\!]\,\sigma] \\
&= [\![e_1 \; o \; e_2]\!]\,\sigma' && \text{Definition 5.2} \quad \square
\end{aligned}
$$

6.5 Schleifeninvarianten und Terminierung

Nach (6.1) umfasst die schwächste Vorbedingung einer Anweisung nur Zustände, unter denen die Anweisung auch terminiert. Sei nun $H_k(N)$ die Menge von Zuständen, unter denen

```
while (b) s
```

in k oder weniger Iterationen in einem Zustand terminiert, der N erfüllt. Wir schreiben im Folgenden kurz H_k statt $H_k(N)$, da es sich hier immer um die gleiche gegebene Nachbedingung N handelt.

Nach den Regeln [While] und [IfFalse] ergibt sich für $k = 0$:

$$H_0 \coloneqq \overline{B} \cap N \qquad \text{mit } B \coloneqq [\![b]\!] \text{ und } \overline{B} = \Sigma \setminus B = [\![\neg b]\!] \tag{6.4}$$

H_0 ist also die Menge aller Zustände, für die die Schleife nicht betreten und die Nachbedingung erfüllt wird.

Angenommen, wir haben die Menge H_k der Zustände für die die Schleife in höchstens k Iterationen in einem Zustand terminiert der N erfüllt. Dann können wir, mit der Regel [While] aus Definition 5.6 daraus die Menge H_{k+1} konstruieren, die alle Zustände enthält, unter denen die Schleife in höchstens $k+1$ Iterationen in einem Zustand terminiert, der N erfüllt:

$$H_{k+1} \coloneqq \mathrm{Wp}(\texttt{if (}b\texttt{) }s \mid H_k) = (B \cap \mathrm{Wp}(s \mid H_k)) \cup (\overline{B} \cap H_k) \tag{6.5}$$

$$= (B \cap \mathrm{Wp}(s \mid H_k)) \cup H_0 \tag{6.6}$$

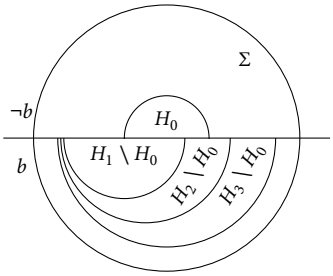


Abbildung 6.3: Skizze der Mengen H_i. Dass $H_0 \subseteq H_1 \subseteq \cdots$ gilt, geht aus Lemma 6.3 hervor.

Das folgende Lemma wird später noch benötigt und rechtfertigt die Skizze in Abbildung 6.3.

Lemma 6.3. *Für alle $i \geq 0$ gilt $H_i \subseteq H_{i+1}$.*

Beweis. Durch Induktion über i. Der Basisfall $i = 0$ folgt aus

$$H_0 = \overline{B} \cap N \subseteq (\overline{B} \cap N) \cup (B \cap \mathrm{Wp}(s \mid H_0)) = H_1$$

Der Induktionsschritt folgt aus der Induktionshypothese $H_i \subseteq H_{i+1}$ und der Monotonität (→ Aufgabe 6.11) von Wp. □

Die **semantische** schwächste Vorbedingung besteht aus **allen** Zuständen, unter denen die Schleife in einem Zustand terminiert, der N erfüllt. Es muss also nur ein k geben, so dass $\sigma \in H_k$:

$$\mathrm{Wp}(\texttt{while (}b\texttt{) }s \mid N) = \{\sigma \mid \exists k \geq 0.\, \sigma \in H_k\} = H_0 \cup H_1 \cup \cdots \tag{6.7}$$

Interessant an dieser Konstruktion ist nun, dass es im Allgemeinen kein k mit $H_k = H_{k+1}$ gibt, wie folgendes Beispiel zeigt:

Beispiel 6.14. Betrachten wir das Programm

```
while (x > 0)
    x = x - 1;
```

und die Nachbedingung `x = 0`. Nach (6.4) und (6.5) sind die Mengen H_i gegeben durch

$$H_i = \{\sigma \mid 0 \leq \sigma\,\mathtt{x} \leq i\}$$

Für alle i ist H_{i+1} echt größer als H_i. ⌟

Das bedeutet, dass wir die Menge der schwächsten Vorbedingung im Allgemeinen nicht durch ein endliches Anwenden von Wp(**`if`** `(`b`)` $s \mid \cdot$) konstruieren können. Wir können die Menge nur (wie zum Beispiel in (6.7)) **beschreiben**.

Dies hat zur Konsequenz, dass wir eine geeignete **syntaktische** schwächste Vorbedingung (im Allgemeinen) nicht nach dem gleichen Prinzip konstruieren können. Wir können zwar, analog zu H_k, eine Funktion h_k

$$h_0 \coloneqq \neg b \wedge N \qquad h_{k+1} \coloneqq \mathrm{wp}(\mathtt{if}\ (b)\ s \mid h_k)$$

definieren, die für ein gegebenes k eine **Zusicherung** dafür liefert, dass die Schleife in höchstens k Schritten terminiert. Gibt es allerdings, **für alle Zustände,** kein k, so dass **`while`** `(`b`)` s terminiert, so entspräche eine, analog zu (6.7) konstruierte Zusicherung wp, einer unendlich großen Formel, die wir nicht materialisieren können. Auch wenn es ein solches k gäbe und wir es ermitteln könnten, so ist die Größe der Zusicherung linear von k abhängig, was sie für große k ebenfalls unbrauchbar macht.

Beispiel 6.15 (Fortsetzung Beispiel 6.14)**.** Berechnet man die Funktionswerte h_k für obiges Beispiel, so bekommt man für jedes k eine Disjunktion über k Terme.

$$h_k = (\mathtt{x = 0}) \vee (\mathtt{x = 1}) \vee \cdots \vee (\mathtt{x = k})$$

Eine analog zu (6.7) konstruierte schwächste Vorbedingung, wäre dann eine Disjunktion über alle h_k: eine unendlich große Formel. Man sieht aber leicht ein, dass

$$\mathrm{wp}(\mathtt{while\ (x > 0)\ x = x - 1;} \mid \mathtt{x = 0}) = \mathtt{x} \geq \mathtt{0}$$ ⌟

Das Fazit ist, dass wir die syntaktische schwächste Vorbedingung einer Schleife nicht mechanisch (wie in Definition 6.13) berechnen können. Dies führt uns zu speziellen Zusicherungen, den sogenannten Schleifeninvarianten, die wir zur Verifikation von Programmen mit Schleifen einsetzen können.

6.5.1 Schleifeninvarianten

Um trotzdem noch Aussagen über Programme mit Schleifen treffen zu können, verwendet man Schleifeninvarianten. Eine Schleifeninvariante ist eine Zusicherung, die vor und nach jeder Ausführung des Schleifenrumpfes gilt. Im Allgemeinen ist sie stärker als die schwächste Vorbedingung der Schleife.

Definition 6.15 (Schleifeninvariante)**.** Eine Zustandsmenge I (analog eine Zusicherung i) ist eine Schleifeninvariante der Schleife **`while`** `(`b`)` s, wenn

$$B \cap I \subseteq \mathrm{Wp}(s \mid I) \qquad \text{analog } b \wedge i \Rightarrow \mathrm{wp}(s \mid i)$$ ⌟

Eine Schleifeninvariante kann allerdings Zustände umfassen, unter denen die Schleife nicht terminiert. Hat man eine Schleifeninvariante I, so weiß man nur, dass **wenn** die Schleife unter einem Zustand $\sigma \in I$ terminiert, der Zustand nach der Beendigung der Schleife in I liegt.

Da wir aber an **totaler** Korrektheit interessiert sind, brauchen wir noch eine Spezifikation, die ausdrückt, dass die Schleife tatsächlich terminiert. Um das zu gewährleisten, muss man eine **Terminierungsfunktion (engl. ranking function)** in Form eines Ausdrucks $t \in \mathit{Expr}$ angeben. Kann man zeigen, dass ihr Wert in jedem Schleifendurchlauf kleiner wird und, solange die Schleife ausgeführt wird, nicht negativ ist, so kann man daraus schließen, dass die Schleife nicht divergiert. Gemeinsam mit der Invariante i muss die Terminierungsfunktion t die folgende Spezifikation erfüllen:

Definition 6.16 (Terminierungsfunktion, Schleifeninvariante)**.** Ein Ausdruck t heißt Terminierungsfunktion und ein Ausdruck i heißt Schleifeninvariante einer Schleife **`while`** (b) s, wenn gilt:

$$[b \wedge i \wedge 0 \leq t \leq k]\ s\ [i \wedge 0 \leq t < k]$$

wobei k eine Variable ist, die in s nicht vorkommt. ⌟

Im Gegensatz zu den Zusicherungen, die wir für schleifenfreie Anweisungen automatisch aus der Nachbedingung konstruieren können, muss eine Schleifeninvariante anderweitig ermittelt werden, gegebenenfalls vom Programmierer zur Verfügung gestellt werden.

Wir zeigen nun, dass die Menge der Zustände, die die Schleifeninvariante erfüllen und auf denen die Terminierungsfunktion kleiner gleich k ist, eine Teilmenge von $H_k(\overline{B} \cap I)$ ist. Das bedeutet, dass die Schleife auf diesen Zuständen in einem Zustand terminiert, der $i \wedge \neg b$ erfüllt.

Lemma 6.4. *Für alle $k \geq 0$:* $I \cap \underbrace{\{\sigma \mid 0 \leq [\![t]\!]\sigma \leq k\}}_{T_k} \subseteq H_k(\overline{B} \cap I)$.

Beweis. Wir zerlegen die Menge $I \cap T_k$ in zwei disjunkte Mengen $(B \cap I \cap T_k) \cup (\overline{B} \cap I \cap T_k)$ und zeigen für jede Teilmenge, dass sie Teil von $H_k(\overline{B} \cap I)$ ist. Es gilt für alle $k \geq 0$:

$$\overline{B} \cap I \cap T_k \subseteq \overline{B} \cap I = H_0(\overline{B} \cap I)$$

und somit gilt nach Lemma 6.3 dann $\overline{B} \cap I \cap T_k \subseteq H_i(\overline{B} \cap I)$. Wir zeigen nun durch Induktion über k das Lemma für $B \cap I \cap T_k$:

$$\begin{array}{lll}
k = 0: & B \cap I \cap T_0 \subseteq \mathrm{Wp}(s \mid I \cap \{\sigma \mid 0 \leq [\![t]\!]\sigma < 0\}) & \text{Definition 6.16} \\
& = \mathrm{Wp}(s \mid \varnothing) = \varnothing \subseteq H_0(\overline{B} \cap I) & \text{Aufgabe 6.10} \\
\\
k-1 \to k: & B \cap I \cap T_k \subseteq B \cap \mathrm{Wp}(s \mid I \cap T_{k-1}) & \text{Definition 6.16} \\
& \subseteq \mathrm{Wp}(\textbf{if}\ (b)\ s \mid I \cap T_{k-1}) & \text{Definition 6.13 und Satz 6.2} \\
& \subseteq \mathrm{Wp}(\textbf{if}\ (b)\ s \mid H_{k-1}(\overline{B} \cap I)) & \text{IH und Aufgabe 6.11} \\
& \subseteq H_k(\overline{B} \cap I) & (6.5)\quad \square
\end{array}$$

Lemma 6.4 erlaubt es uns nun folgenden Satz über Hoare-Tripel von Schleifen zu zeigen:

Satz 6.5 (Hoare-Tripel der Schleife)**.** *Gegeben eine Schleife*

while *(b) s*

Sei i eine Schleifeninvariante und t eine Terminierungsfunktion für die der Schleifenrumpf der Spezifikation in Definition 6.16 genügt. Dann gilt

$[i \wedge t \geq 0]$ **while** *(b) s* $[i \wedge \neg b]$

Beweis. Sei σ ein Zustand, der die Vorbedingung erfüllt und somit auch i. Es existiert also ein $k \geq 0$ mit $[\![t]\!]\sigma = k$. Nach Lemma 6.4 ist dieser Zustand in $H_k(I \cap \overline{B})$ enthalten. Nach (6.5) terminiert somit die Schleife in höchstens k Iterationen in einem Zustand, der die Nachbedingung $i \wedge \neg b$ erfüllt. □

Beispiel 6.16 (Division)**.** Betrachten wir das folgende Programm

```
r = x;
q = 0;
while (y <= r) {
    r = r - y;
    q = q + 1;
}
```

Dieser Algorithmus soll den Quotient und Rest der Division von x durch y berechnen. Dies drücken wir durch folgende Spezifikation aus:

$V \coloneqq \mathtt{x} \geq \mathtt{0} \wedge \mathtt{y} > \mathtt{0} \qquad N \coloneqq \mathtt{x} = \mathtt{q} * \mathtt{y} + \mathtt{r} \wedge \mathtt{0} \leq \mathtt{r} < \mathtt{y}$

Wir wählen die Schleifeninvariante und Terminierungsfunktion:

$i \coloneqq \mathtt{x} = \mathtt{q} * \mathtt{y} + \mathtt{r} \wedge \mathtt{r} \geq \mathtt{0} \wedge \mathtt{y} > \mathtt{0} \qquad t \coloneqq \mathtt{r}$

Damit wir Satz 6.5 für die Schleife verwenden können, müssen wir zunächst zeigen, dass i tatsächlich eine Schleifeninvariante und t eine Terminierungsfunktion ist. Hierzu müssen wir das Hoare-Tripel aus Definition 6.16 zeigen. Dies machen wir, indem wir die schwächste Vorbedingung des Schleifenrumpfes bezüglich $i \wedge 0 \leq t < k$ berechnen und überprüfen, dass diese von $i \wedge 0 \leq t \leq k$ impliziert wird:

$$
\begin{aligned}
&\mathrm{wp}(\mathtt{r = r - y; q = q + 1;} \mid \mathtt{x = q * y + r} \wedge \mathtt{r \geq 0} \wedge \mathtt{y > 0} \wedge \mathtt{0 \leq r < k}) \\
=\ &\mathrm{wp}(\mathtt{r = r - y;} \mid \mathtt{x = (q + 1) * y + r} \wedge \mathtt{r \geq 0} \wedge \mathtt{y > 0} \wedge \mathtt{0 \leq r < k}) \\
\equiv\ &\mathtt{x = (q + 1) * y + (r - y)} \wedge \mathtt{y > 0} \wedge \mathtt{0 \leq r - y < k} \\
=\ &\mathtt{x = q * y + r} \wedge \mathtt{y \leq r} \wedge \mathtt{y > 0} \wedge \mathtt{r < k + y}
\end{aligned}
$$

was der Fall ist. Manuell überprüft man am besten dadurch, dass man sich davon überzeugt, dass jede Klausel von $b \wedge i \wedge 0 \leq t \leq k$ einzeln impliziert wird. Somit ist die Vorbedingung vom Satz 6.5 erfüllt, wir können den Satz anwenden, und es gilt das Hoare-Tripel

$[i \wedge t \geq 0]$ **while** `(y <= r) { ... }` $[i \wedge \neg b]$

Nun zeigt sich, dass wir die Schleifeninvariante gut gewählt haben, da man $i \wedge \neg b$ zur Nachbedingung abschwächen kann. Der Korrektheitsbeweis des Programms kann nun dadurch vervollständigt werden, dass

wir zeigen, dass die schwächste Vorbedingung der ersten beiden Anweisungen bezüglich der Nachbedingung $i \wedge t \geq 0$ von der Vorbedingung impliziert wird.

$$
\begin{aligned}
& \text{wp}(\texttt{r = x; q = 0;} \mid \texttt{x = q * y + r} \wedge \texttt{y > 0} \wedge \texttt{r} \geq \texttt{0}) \\
= \; & \text{wp}(\texttt{r = x;} \mid \texttt{x = 0 * y + r} \wedge \texttt{y > 0} \wedge \texttt{r} \geq \texttt{0}) \\
= \; & \texttt{x = 0 * y + x} \wedge \texttt{y > 0} \wedge \texttt{x} \geq \texttt{0} \\
\equiv \; & \texttt{y > 0} \wedge \texttt{x} \geq \texttt{0} \\
\equiv \; & V \quad \lrcorner
\end{aligned}
$$

6.6 Automatisierung der Verifikation

Wir haben nun alle Werkzeuge beisammen, um die Verifikation von Programmen zu automatisieren: Die Gültigkeit des Hoare-Tripel $[V]\ s\ [N]$ einer schleifenfreien Anweisung können wir, dank (6.2), in die Aussage

$$V \Rightarrow \mathrm{wp}(S \mid N)$$

übersetzen. Für Schleifen sind wir, nach Satz 6.5, auf Schleifeninvarianten und Terminierungsfunktionen angewiesen. Diese müssen entweder von Programmierer annotiert werden oder durch andere automatische Verfahren, auf die wir hier nicht eingehen, ermittelt werden.

Wir kombinieren diese Ergebnisse zu einem Verfahren, dass aus einem (annotierten) Programm s und einer Spezifikation (V, N) eine logische Aussage konstruiert, deren Allgemeingültigkeit das Hoare-Tripel $[V]\ s\ [N]$ impliziert.

Hierzu definieren wir uns zwei Funktionen, die die entsprechenden Komponenten der Aussage aus dem Programm ermitteln. Die erste Funktion pc (= precondition) ist wp ähnlich, liefert aber für die (annotierte) Schleife die Vorbedingung des Hoare-Tripels aus Satz 6.5. Die zweite Funktion vc (= verification condition) generiert die zusätzlichen Verifikationsbedingungen (i ist eine Schleifeninvariante, t eine Terminierungsfunktion), die benötigt werden, um Satz 6.5 anwenden zu können.

Definition 6.17 (Regeln zur Erzeugung von Verifikationsbedingungen)**.**

$$\begin{aligned}
&\mathrm{pc}(s_1 _ s_2 \mid N) &&= \mathrm{pc}(s_1 \mid \mathrm{pc}(s_2 \mid N)) \\
&\mathrm{pc}(\texttt{if}\ldots \mid N) &&= \mathrm{def}\, e \wedge \big[(e \wedge \mathrm{pc}(s_1 \mid N)) \vee (\neg e \wedge \mathrm{pc}(s_2 \mid N))\big] \\
&\mathrm{pc}(\texttt{while (}e\texttt{) _Inv(}i\texttt{) _Term(}t\texttt{) }s \mid N) &&= t \geq 0 \wedge i \\
&\mathrm{pc}(s \mid N) &&= \mathrm{wp}(s \mid N) \quad \text{für alle anderen Anweisungen} \\
\\
&\mathrm{vc}(s_1 _ s_2 \mid N) &&= \mathrm{vc}(s_1 \mid \mathrm{pc}(s_2 \mid N)) \wedge \mathrm{vc}(s_2 \mid N) \\
&\mathrm{vc}(\texttt{if}\ldots \mid N) &&= \mathrm{vc}(s_1 \mid N) \wedge \mathrm{vc}(s_2 \mid N) \\
&\mathrm{vc}(\texttt{while (}e\texttt{) _Inv(}i\texttt{) _Term(}t\texttt{) }s \mid N) &&= \mathrm{vc}(s \mid i \wedge 0 \leq t \leq k) \\
&&&\wedge\ (i \wedge \neg e \Rightarrow N) \\
&&&\wedge\ (e \wedge i \wedge 0 \leq t \leq k+1 \\
&&&\qquad \Rightarrow \mathrm{pc}(s \mid i \wedge 0 \leq t \leq k)) \\
&\mathrm{vc}(s \mid N) &&= \texttt{true} \quad \text{für alle anderen Anweisungen}
\end{aligned}$$ ⌟

Satz 6.6. *Ist* $\mathrm{vc}(s \mid N)$ *allgemeingültig, so gilt* $[\![\mathrm{pc}(s \mid N)]\!] \subseteq \mathrm{Wp}(s \mid N)$

Beweis. Wie Satz 6.2 beweisen wir diesen Satz durch Induktion über die abstrakte Syntax der Anweisung s. Wir skizzieren den Beweis hier nur, in dem wir die drei Fälle der Zuweisung, Hintereinanderausführung, und der while-Schleife betrachten:

Die Zuweisung $s = x = e;$.

Da $\mathrm{vc}(x = e; \mid N) = \texttt{true}$ und $\mathrm{pc}(x = e; \mid N) = \mathrm{wp}(x = e; \mid N)$ gilt nach Satz 6.2 der Satz in diesem Fall.

Die Hintereinanderausführung $s = s_1 \_ s_2$.

Nehmen wir an, dass $\mathrm{vc}(s_1 \_ s_2 \mid N)$ allgemeingültig ist. Mit Definition 6.17 gilt, dass $\mathrm{vc}(s_1 \mid \mathrm{pc}(s_2 \mid N))$ und $\mathrm{vc}(s_2 \mid N)$ allgemeingültig sind. Nach der Induktionshypothese gilt dann:

$$[\mathrm{pc}(s_2 \mid N)]\ s_2\ [N] \qquad \text{und} \qquad [\mathrm{pc}(s_1 \mid \mathrm{pc}(s_2 \mid N))]\ s_1\ [\mathrm{pc}(s_2 \mid N)]$$

Kombiniert man beide Aussagen unter Ausnutzung der Aussage, dass $[A]\ s_1 \_ s_2\ [C]$ aus $[A]\ s_1\ [B]$ und $[B]\ s_2\ [C]$ folgt, erhält man:

$$[\mathrm{pc}(s_1 \_ s_2 \mid N)]\ s_1\ [N]$$

Die Schleife $s = \texttt{while}\ (e)\ \texttt{\_Inv}(i)\ \texttt{\_Term}(t)\ s'$.

Der Übersicht halber kürzen wir die Vor- und Nachbedingung von Definition 6.16 ab:

$$A \coloneqq e \wedge i \wedge 0 \leq t \leq k + 1$$

$$B \coloneqq i \wedge 0 \leq t \leq k$$

Nehmen wir an, dass $\mathrm{vc}(s \mid N)$ allgemeingültig ist. Daraus folgt, dass jede der einzelnen Klauseln von $\mathrm{vc}(s \mid N)$

$$\mathrm{vc}(s' \mid B) \tag{6.8}$$

$$A \Rightarrow \mathrm{pc}(s' \mid B) \tag{6.9}$$

$$\neg e \wedge i \Rightarrow N \tag{6.10}$$

(siehe auch Definition 6.17) auch allgemeingültig ist. Mit (6.8) können wir die Induktionshypothese anwenden und schließen, dass

$$[\![\mathrm{pc}(s' \mid B)]\!] \subseteq \mathrm{Wp}(s' \mid B)$$

Aus (6.9) folgt, dass

$$[\![A]\!] \subseteq [\![\mathrm{pc}(s' \mid B)]\!] \subseteq \mathrm{Wp}(s' \mid B) \quad \text{sprich} \quad [A]\ s'\ [B]$$

Somit ist die Prämisse von Satz 6.5 erfüllt und es gilt:

$$[t \geq 0 \wedge i]\ s\ [\neg e \wedge i]$$

was mit (6.10) und Satz 6.1 zur Behauptung abgeschwächt werden kann. □

Mit Satz 6.6 bekommen wir ein Werkzeug an die Hand, mit dem wir für eine Spezifikation (V, N) und ein annotiertes Programm s, zwei Formeln $\mathrm{pc}(s \mid N)$ und $\mathrm{vc}(s \mid N)$ ableiten können, so dass gilt:

Korollar 6.7. *Wenn* $\mathrm{vc}(s \mid N) \wedge (V \Rightarrow \mathrm{pc}(s \mid N))$ *allgemeingültig ist, dann gilt* $[V]\ s\ [N]$.

Unser Ansatz ist nicht auf Vor- und Nachbedingungen beschränkt, sondern kann auch dazu verwendet werden, Zusicherungen im Code zu beweisen, die wir mit `_Assert(`e`)` kennzeichnen. Darüberhinaus ist es sinnvoll, eine Funktion `_Assume(`e`)` einzuführen, mit der man Vorbedingungen ausdrücken kann. Neben der Spezifikation der Vorbedingung kann dies zum Beispiel sinnvoll sein, wenn man bei Eingaben an das Programm bestimmte Annahmen spezifizieren möchte (beispielsweise kann ein Temparaturwert nie kleiner als −273,15 Grad Celsius sein).

Definition 6.18 (Assume, Assert)**.**

$$\begin{aligned} \mathrm{pc}(\texttt{\_Assert}(e) \mid N) &= e \wedge N \text{ (siehe Definition 6.8)} \\ \mathrm{pc}(\texttt{\_Assume}(e) \mid N) &= e \Rightarrow N \end{aligned}$$

⌟

Wenn wir `_Assume` und `_Assert` von nun an verwenden, um die Vor- und Nachbedingung im Programmtext zu spezifizieren, können wir für V und V jeweils **`true`** einsetzen.

Durch die Verwendung von `_Assert` und `_Assume` zur Dokumentation der Vor- und Nachbedingung können wir V und N im Hauptprogramm auf **`true`** setzen.

Beispiel 6.17. Betrachten wir nochmal den Divisionsalgorithmus aus Beispiel 6.16, annotieren ihn entsprechend und leiten die Verifikationsbedingungen systematisch ab.

```
_Assume(x >= 0 && y > 0);
r = x;
q = 0;
while (y <= r)
    _Inv(x == y * q + r && r >= 0 && y > 0)
    _Term(r)
{
    r = r - y;
    q = q + 1;
}
_Assert(x == y * q + r && 0 <= r && r < y);
```

Wir kürzen den Rumpf der Schleife der Übersichtlichkeit halber durch s', und die Schleifeninvariante durch i ab:

$$\begin{aligned} \mathrm{vc}(s' \mid i \wedge 0 \leq \mathtt{r} < k) &= \texttt{true} \\ \mathrm{pc}(s' \mid i \wedge 0 \leq \mathtt{r} < k) &= \mathtt{x = q * y + r} \wedge \mathtt{0} \leq \mathtt{r - y} \leq \mathtt{k} \wedge \mathtt{y > 0} \\ \mathrm{vc}(\texttt{while ...} \mid N) &\equiv \underbrace{\texttt{true}}_{\mathrm{vc}(s' \mid i \wedge 0 \leq r < k)} \\ &\wedge \ (i \wedge \mathtt{y > r} \Rightarrow N) \\ &\wedge \ (\mathtt{y} \leq \mathtt{r} \wedge i \wedge \mathtt{0} \leq \mathtt{r} \wedge \mathtt{r} \leq \mathtt{k + 1} \Rightarrow \\ &\quad \underbrace{\mathtt{x = q * y + r} \wedge \mathtt{0} \leq \mathtt{r - y} \leq \mathtt{k} \wedge \mathtt{y > 0}}_{\mathrm{pc}(s' \mid i \wedge 0 \leq r < k)}) \\ \mathrm{pc}(\texttt{while ...} \mid N) &= \mathtt{x = q * y + r} \wedge \mathtt{r} \geq \mathtt{0} \wedge \mathtt{y > 0} \end{aligned}$$

Somit haben wir:

$$
\begin{aligned}
& \mathrm{vc}(\texttt{\_Assume}(V)\texttt{; r = x; q = 0; while ...; \_Assert}(N) \mid \texttt{true}) \\
\equiv\ & \mathrm{vc}(\texttt{while ...} \mid N) \\
& \mathrm{pc}(\texttt{\_Assume}(V)\texttt{; r = x; q = 0; while ...; \_Assert}(N) \mid \texttt{true}) \\
\equiv\ & \mathrm{pc}(\texttt{\_Assume}(V)\texttt{; r = x; q = 0;} \mid \texttt{x = y * q + r} \wedge \texttt{r} \geq \texttt{0} \wedge \texttt{y > 0}) \\
\equiv\ & \mathrm{pc}(\texttt{\_Assume}(V)\texttt{;} \mid \texttt{x} \geq \texttt{0} \wedge \texttt{y > 0}) \\
\equiv\ & \texttt{x} \geq \texttt{0} \wedge \texttt{y > 0} \Rightarrow \texttt{x} \geq \texttt{0} \wedge \texttt{y > 0}
\end{aligned}
$$

Die Prämisse für Korollar 6.7 setzt sich also wie folgt zusammen:

$$
\underbrace{((i \wedge y > r \Rightarrow N) \wedge (y \leq r \wedge i \wedge 0 \leq r \leq k+1 \Rightarrow i \wedge 0 \leq r - y \leq k))}_{\mathrm{vc}} \wedge
$$

$$
\Big[\underbrace{(x \geq 0 \wedge y > 0)}_{V} \Rightarrow \underbrace{(x \geq 0 \wedge y > 0)}_{\mathrm{pc}}\Big]
$$

Diese Formel ist allgemeingültig, womit dann nach Korollar 6.7 $[V]\ s\ [N]$ gilt. ⌟

Durch die beiden Funktionen pc und vc können wir also syntaxgesteuert Formeln generieren, die mithilfe von Korollar 6.7 zu einer Formel zusammengesetzt werden können, deren Allgemeingültigkeit die Korrektheit des Programms impliziert.

6.7 Zusammenfassung und Literaturhinweise

Der Begriff der schwächsten Vorbedingung geht zurück auf Edsger Dijkstra [Dij75]. Der Kalkül der schwächsten Vorbedingungen ist eng verwandt mit der Floyd-Hoare-Logik, die wir in Ansätzen diskutiert haben. Das Buch „The Science of Programming“ von Gries [Gri81] ist ein Klassiker und bietet einen formaleren Zugang zum Programmieren, der stark auf Dijkstras WP-Kalkül beruht.

In diesem Kapitel haben wir stillschweigend Aussagenlogik um die Theorie der Arithmetik auf den ganzen Zahlen erweitert, um Zusicherungen mit Termen wie `x > 0` zuzulassen. Beschränkt man sich nicht auf C0 und betrachtet weiterführende Spracheigenschaften wie Reihungen und Zeiger, so braucht man auch mächtigere Theorien und Logiken um die gewünschten Spezifikationen überhaupt erst formulieren zu können. Das Buch von Bradley und Manna [BM07] gibt hierzu eine moderne Einführung und einen Überblick.

Die Darstellung von Abschnitt 6.6 ist dem Skript von Mike Gordon [Gor15] angelehnt, das auch eine gute Einführung in den WP-Kalkül und die Hoare-Logik bietet.

Aufgaben

Aufgabe 6.1 (Kaputt. ★★★)**.** Schreiben Sie Zusicherungen für die unterstrichene Lücke, so dass die Funktion `supersum` immer terminiert.

```
int supersum(int a, int b, int c) {
  assert(c >= 0);
  ________________
  int x = 0;
  for (int i = a; x < b; i = i + c) {
    x = x + i;
  }
  return x;
}
```

Aufgabe 6.2 (Funktionale Tests. ★★)**.** Die angegebene Testsuite für Beispiel 6.9 deckt nicht alle Fälle ab. Schreiben Sie weitere Tests für die noch nicht behandelten Fälle.

Aufgabe 6.3 (Abdeckung. ★★★)**.** Betrachten Sie folgendes Programm:

```
int substr(char * dest, int begin, int end, char * data) {
  int delta = 0;
  if (end < 0) {
    delta = -end;
  }

  if (delta == 0) {
    if (begin <= end) {
      for (int i = begin; i < end; ++i) {
        dest[i - begin] = data[i];
      }
    }
    dest[end - begin] = '\0';
  } else {
    int i = begin;
    if (data[i + delta] != '\0') {
      for (; data[i + delta] != '\0'; ++i) {
        dest[i - begin] = data[i];
      }
    }
    dest[i - begin] = '\0';
  }
}
```

1. Beschreiben Sie die Funktion des Programms in ein bis zwei Sätzen.
2. Schreiben Sie eine Testsuite, die alle Anweisungen abdeckt.
3. Erweitern Sie die Testsuite ggf., so dass alle Zweige abdeckt werden.

Aufgabe 6.4 (Ausführungsprotokoll. ⋆⋆)**.** Führen Sie folgende Konfigurationen gemäß der Operationalen Semantik von C0 aus. Rechtfertigen Sie jeden Schritt in der Semantik entsprechend durch Nebenrechnungen oder einen Inferenzbaum.

1. $\langle$`{a = 2; b = 2 * a;}` $\mid \{\}\rangle$
2. $\langle$`if (x < 0) x = -x; else {}` $\mid \{\mathtt{x} \mapsto 1\}\rangle$
3. $\langle$`if (x < 0) x = -x; else {}` $\mid \{\mathtt{x} \mapsto -1\}\rangle$
4. $\langle$`{a = 1; while (n > 1) { a = a * n; n = n - 1; }}` $\mid [\mathtt{n} \mapsto 2]\rangle$

⌟

Aufgabe 6.5 (Nach Spezifikation. ⋆⋆)**.** ⌟

1. Schreiben Sie ein C0 Programm, dass folgende Spezifikation implementiert.

$$S = \{(\sigma, \sigma') \mid \begin{array}{l} \sigma x > 1 \implies \sigma' y = \sigma b \wedge \\ \sigma x < 0 \implies \sigma' y = \sigma a \wedge \\ \sigma x \geq 0 \wedge \sigma x \leq 1 \implies \sigma' y = \sigma b \cdot \sigma x + (1 - \sigma x) \cdot \sigma a\} \end{array}$$

2. Beweisen Sie, dass Ihr Programm korrekt ist. Machen Sie hierzu eine Fallunterscheidung über die Eingabe ($\sigma x > 1$, etc.) und führen Sie das Programm dann für jeden Fall mit der Semantik von C0 aus.

Aufgabe 6.6 (⋆)**.** Welche Zustandsmengen werden durch die Zusicherungen *true* und *false* beschrieben? ⌟

Aufgabe 6.7 (Programmsuche. ⋆⋆⋆)**.** Geben Sie das kürzeste Programm an, das folgendes Hoare-Tripel erfüllt

$$[\mathit{true}]\ P\ [(\mathtt{m} = \mathtt{a} \vee \mathtt{m} = \mathtt{b}) \wedge \mathtt{m} \geq \mathtt{a} \wedge \mathtt{m} \geq \mathtt{b}]$$

Korrigieren Sie die Spezifikation durch den Einsatz von Hilfsvariablen, so dass sie das intendierte Programm beschreibt. ⌟

Aufgabe 6.8 (Abschwächen und Verstärken. ⋆⋆)**.** Veranschaulichen Sie den Beweis von Satz 6.1 grafisch. ⌟

Aufgabe 6.9 (WP Konjunktion. ⋆⋆)**.** Beweisen Sie $\mathrm{Wp}(S \mid A \cap B) = \mathrm{Wp}(S \mid A) \cap \mathrm{Wp}(S \mid B)$ ⌟

Aufgabe 6.10 (Empty. ⋆⋆)**.** Beweisen Sie $\mathrm{Wp}(S \mid \varnothing) = \varnothing$ für alle $S \neq$ `abort();`. ⌟

Aufgabe 6.11 (Monotonität von WP. ⋆⋆)**.** Beweisen Sie $A \subseteq B \implies \mathrm{Wp}(S \mid A) \subseteq \mathrm{Wp}(S \mid B)$. ⌟

Aufgabe 6.12 (Schwächste Vorbedingung. ⋆⋆)**.** Berechnen Sie

1. $\mathrm{wp}($`x = x - 1;` $\mid \mathtt{x} = 5)$
2. $\mathrm{wp}($`x = 7;` $\mid \mathtt{x} = 5)$
3. $\mathrm{wp}($`x = 7;` $\mid \mathtt{x} = 7)$
4. $\mathrm{wp}($`a = a - b;` $\mid \mathtt{a} > \mathtt{b})$

⌟

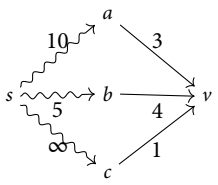
7 Dynamische Programmierung

Ein gängiges Verfahren der Algorithmenkonstruktion ist das **Teile und Herrsche** Prinzip. Hierbei wird ein Problem in mehrere Teilprobleme unterteilt, die dann einzeln gelöst werden. Nach der Bearbeitung der Teilprobleme wird deren Lösung dann zur Lösung des Gesamtproblems verwendet. Ein prominentes Beispiel ist das Sortieren durch Mischen (engl. Mergesort). Hierbei wird die zu sortierende Reihung in zwei Hälften geteilt, diese dann getrennt voneinander sortiert (über Rekursion) und dann durch Mischen die beiden sortierten Teilreihungen zu einer sortierten Gesamtreihung zusammen gesetzt (siehe auch Beispiel 4.2).

In manchen Problemen tritt darüberhinaus die Situation auf, dass die Lösung mancher Teilprobleme mehrmals verwendet wird. Man spricht hier von **überlappenden Teilproblemen**. Anstatt diese mehrfach zu berechnen, wie es die rekursive Definition solcher Algorithmen nahelegt, tabelliert die Zwischenergebnisse und benutzt sie wieder. Diese Implementierungstechnik wird **Memoisierung** genannt.

Als dynamische Programmierung bezeichnet man die Anwendung dieses Ansatzes auf bestimmte Optimierungsprobleme. Manche Optimierungsprobleme haben die Eigenschaft, dass sich eine optimale Lösung des Gesamtproblems aus optimalen Lösungen der Teilproblemen zusammensetzen läßt. Man sagt dann, das Problem hat **optimale Teilstruktur**. Ein Beispiel ist die Ermittlung des **kürzesten** Pfades von einem Ursprungsknoten s zu einem Knoten v eines Graphen. Kennt man die kürzesten Pfade zu allen Vorgängerknoten eines Knotens v, dann kann man durch eine lokale Betrachtung der kürzesten Pfade zu den Vorgängern den kürzesten Pfad zu v ermitteln.

Wir betrachten in diesem Kapitel zunächst einige Beispiele, in denen wir Rekursion durch Memoisierung ersetzen. Danach betrachten wir einige Optimierungsprobleme, zu deren Lösung wir dynamische Programmierung einsetzen.



$sp(s, v) = \min\{10+3, 5+4, \infty+1\} = 9$

Abbildung 7.1: Den kürzesten Pfad von s nach v können wir anhand der kürzesten Pfade zu v's Vorgänger ermitteln.

7.1 Die Fibonacci Zahlen

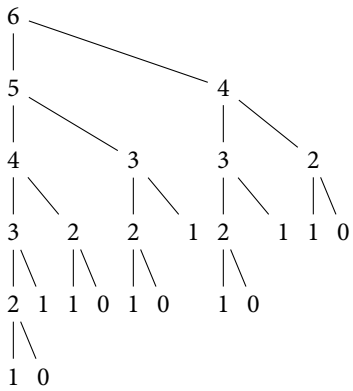
Betrachten wir die Berechnung der **Fibonacci-Zahlen**. Diese sind (rekursiv) definiert durch:

$$F_0 := 0 \qquad F_1 := 1 \qquad F_n := F_{n-1} + F_{n-2} \text{ für } n > 1$$

Eine erste Implementierung setzt diese Definition direkt in Code um:

```
unsigned int fib(unsigned int n) {
    if (n <= 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Betrachten wir den Aufruf `fib(6)`. Er erzeugt folgenden Rekursionsbaum:



Die Anzahl der Aufrufe, die `fib(n)` erzeugt ist F_n. `fib(45)` hat beispielsweise $F_{45} = 1.836.311.903$ Aufrufe. Dauert ein Aufruf beispielsweise 10 Taktzyklen eines Prozessors mit 1 GHz, so dauert die Berechnung von `fib(45)` ca. 18 Sekunden. Das geht jedoch erheblich schneller. Man sieht, dass viele Aufrufe mehrfach vorkommen. Beispielsweise wird `fib(4)` zweimal berechnet. Man kann die Berechnung erheblich schneller durchführen, indem man sich mehrfach berechnete Teilergebnisse merkt. Folgende Implementierung setzt dies um:

```
unsigned int fib_compute(unsigned int n, unsigned int* table)
{
    if (n <= 1)
        return n;
    if (table[n] == 0)
        table[n] = fib_compute(n - 1, table)
                 + fib_compute(n - 2, table);
    return table[n];
}

unsigned int fib(unsigned int n)
```

```
{
    /* Early exit. Für n <= 1 brauchen wir keine Tabelle. */
    if (n <= 1)
        return n;
    /* Erzeuge mit 0 initialisierten Behälter für n unsigneds */
    unsigned int table[n] = { 0 };
    /* Berechne die Fibonacci Zahl */
    unsigned int res = fib_compute(n, table);
    return res;
}
```

In diesem Beispiel ist aber die Verwaltung einer Tabelle für alle Zwischenergebnisse nicht notwendig, wenn man statt der rekursiven Implementierung eine Implementierung mit Schleife wählt:

```
unsigned int fib(unsigned int n)
{
    unsigned int fm1 = 1;
    unsigned int fm2 = 0;
    unsigned int f   = n;

    for (; n >= 2; --n) {
        f   = fm1 + fm2;
        fm2 = fm1;
        fm1 = f;
    }
    return f;
}
```

Hierbei berechnet man die Zahlen nun „von unten“ und nicht „von oben“. Zur Berechnung von F_n muss man nur noch die Zahlen F_{n-1} und F_{n-2} wissen.

Bemerkung 7.1. Die Fibonacci-Zahlen haben auch eine geschlossene Form:

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor \qquad \text{mit } \phi \coloneqq \frac{1+\sqrt{5}}{2}$$

ϕ ist der sogenannte **goldene Schnitt**. ⌟

7.2 Berechnung der Binomial-Koeffizienten

Der Binomialkoeffizient zweier natürlicher Zahlen $k \leq n$ ist definiert durch

$$\binom{n}{k} := \frac{n!}{k!(n-k)!} \qquad \text{mit } n! := n \cdot (n-1) \cdots 2 \text{ und } 0! := 1$$

Um den Binomialkoeffizienten nach obiger Formel zu berechnen brauchen wir

$$(n-2) + (k-2) + (n-k-2) = 2n - 6$$

Multiplikationen. Allerdings fallen die Terme $k!$ und $(n-k)!$ als Teilergebnisse an:

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-m+1)}{m!} \qquad \text{mit } m := \min(k, n-k) \leq n/2$$

Und somit brauchen wir nur $n - (n-m+1) + m - 1 = 2m - 2$ Multiplikationen. Da $m \leq n/2$, also im schlimmsten Fall $n - 2$. Allerdings werden sowohl Zähler als auch Nenner bei dieser Art den Binomialkoeffizienten zu berechnen sehr groß: Nehmen wir an, wir verwenden vorzeichenlose 32 Bit Zahlen für n und k und die anfallenden Zwischenergebnisse. Das größte n mit $n! < 2^{32}$ ist $n = 12$. Aufgrund der Größe der anfallenden Zwischenergebnisse ist der größte Binomialkoeffizient, den wir ohne Überlauf berechnen können $\binom{17}{8} = 24310$. Allerdings ist der größte darstellbare Binomialkoeffizient wesentlich größer.

Daher verwendet man das Pascal'sche Dreieck, um die Binomialkoeffizienten ohne Überlauf in den Zwischenergebnissen zu berechnen. Das Pascal'sche Dreieck

	$k=0$	1	2	3	4	5	6	7	$\cdots$
$n=0$	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

(7.1)

ist definiert über die Gleichungen:

$$\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{7.2}$$

Diese rekursive Definition kann direkt in ein Programm gegossen werden:

```
unsigned int binomial(unsigned int n, unsigned int k)
{
    if (k > n)
        return 0;
    if (k == 0 || k == n)
        return 1;
    return binomial(n - 1, k - 1) + binomial(n - 1, k);
}
```

Allerdings stellt man auch hier fest, dass manche Teilergebnisse mehrfach berechnet werden, denn

$$\binom{n}{k+1} = \binom{n-1}{k} + \binom{n-1}{k+1}$$

Der erste Summand wird auch benötigt, um $\binom{n}{k}$ zu berechnen (siehe oben). Ohne Tabellierung produziert obiges Programm wieder exponentiell viele Aufrufe. Daher tabelliert man die Zwischenergebnisse wieder.

Das Pascal'sche Dreieck ((7.1)) ist spiegelsymmetrisch zur „Seitenhalbierenden", da

$$\binom{n}{k} = \binom{n}{n-k}$$

Das heißt, dass wir nur Binomialkoeffizienten mit $k \leq \lfloor n/2 \rfloor$ tabellieren müssen. Aus (7.2) folgt auch, dass wir, um $\binom{n}{k}$ zu berechnen, niemals Einträge $> k$ berechnen müssen. Es reicht also aus, ein k Zahlen breites Band des Pascal'schen Dreiecks zu berechnen.

Beispiel 7.1. Zur Berechnung von $\binom{7}{3}$ berechnen wir folgende Tabelle:

	$k=0$	1	2	3	4	5	6	7	⋯
$n=0$	1	0	0	0					
1	1	1	0	0					
2	1	2	1	0					
3	1	3	3	1					
4	1	4	6	4	.				
5	1	5	10	10	.	.			
6	1	6	15	20	.	.	.		
7	1	7	21	35	.	.	.	.	

(7.3)

⌟

Folgender Code setzt diese Überlegung um:

```
unsigned int binomial(unsigned int N, unsigned int K) {
    if (K > N)
        return 0;
    if (K > N - K)
        K = N - K;
    if (K == 0)
        return 1;
```

```
    unsigned table[N + 1][K + 1];
    for (unsigned n = 0; n <= N; n++) {
        table[n][0] = 1;
        unsigned const end = n < K ? n : K;
        for (unsigned k = 1; k <= end; k++)
            table[n][k] = table[n - 1][k] + table[n - 1][k - 1];
        for (unsigned i = end + 1; i <= K; i++)
            table[n][i] = 0;
    }
    return table[N][K];
}
```

Der Speicherverbrauch kann noch optimiert werden, da man von dem Ausschnitt des Pascal'schen Dreieicks immer nur die letzten beiden Zeilen benötigt (→ Aufgabe 7.9).

7.3 Minimale Editierdistanz

Betrachten wir ein weiteres Problem, das sich mit Tabellierung effizient lösen lässt. Gegeben zwei Worte, a und b. Finde die minimale Anzahl von Editieroperationen (Zeichen ersetzen, löschen, anfügen), um das Wort a in das Wort b zu ändern. Diese Anzahl wird die **minimale Editierdistanz** oder auch **Levenshtein-Distanz** zwischen a und b genannt. Machen wir ein Beispiel:

$a = \texttt{Tore} \qquad b = \texttt{Tag}$

`Tore` kann in `Tag` überführt werden, indem man vier Buchstaben löscht und drei hinzufügt. Das wären dann sieben Editieroperationen. Das Mininum hingegen ist drei: `T` beibehalten (kostet nichts), `o` durch `a` ersetzen, `r` durch `g` ersetzen, `e` löschen. Die minimale Editierdistanz findet zum Beispiel bei Rechtschreibekorrektur-Systemen oder unscharfer Suche Anwendung.

Man definiert die minimale Editierdistanz bis zu den Stellen i, j in den Wörtern a, b wie folgt:

Definition 7.1 (minimale Editierdistanz)**.**

$$
\begin{aligned}
edist_{a,b} &: \{0,\dots,|a|\} \times \{0,\dots,|b|\} \to \mathbb{N} \\
edist_{a,b}(0,j) &\mapsto j \\
edist_{a,b}(i,0) &\mapsto i \\
edist_{a,b}(i,j) &\mapsto \min \begin{cases}
edist_{a,b}(i-1,j)+1 & \text{anfügen} \\
edist_{a,b}(i,j-1)+1 & \text{löschen} \\
edist_{a,b}(i-1,j-1) & \text{wenn } a_i = b_j \quad \text{beibehalten} \\
edist_{a,b}(i-1,j-1)+1 & \text{wenn } a_i \neq b_j \quad \text{ersetzen}
\end{cases}
\end{aligned}
$$

⌟

An der (rekursiven) Definition von *edist* sieht man, dass manche Teilergebnisse öfter verwendet werden. Daher wird Berechnung der minimalen Editierdistanz durch Tabellierung durchgeführt.

Beispiel 7.2. Seien a, b wie oben. Die Berechnung von *edist* geschieht am besten über eine Tabelle. Um aus dem leeren Wort ☒ das Wort `Tag` zu erhalten, müssen die Buchstaben `T`, `a` und `g` angefügt werden. Es wird also drei mal der erste Fall der minimalen Editierdistanz angewendet:

	ε	T	a	g
ε	0	1	2	3
T				
o				
r				
e				

Das leere Wort erhalten wir, indem wir alle Buchstaben löschen, also den zweiten Fall der Editierdistanz anwenden:

	ε	T	a	g
ε	0	1	2	3
T	1			
o	2			
r	3			
e	4			

Für die verbleibenden Matrixeinträge muss der dritte Fall betrachtet werden, i ist der Zeilenindex, j der Index der Spalte:

löschen

	ε	T	a	g
ε	0	1	2	3
T	1	2		
o	2			
r	3			
e	4			

anfügen

	ε	T	a	g
ε	0	1	2	3
T	1	2		
o	2			
r	3			
e	4			

beibehalten

	ε	T	a	g
ε	0	1	2	3
T	1	0		
o	2			
r	3			
e	4			

Wir entscheiden uns laut Rekursionsformel der Edit-Distanz für das Minimum und füllen den Rest der Matrix auf die gleiche Weise:

	ε	T	a	g
ε	0	1	2	3
T	1	0	1	2
o	2	1	1	2
r	3	2	2	2
e	4	3	3	3

⌟

Der Eintrag in Zeile i und Spalte j (in rot) zeigt nun die minimale Editierdistanz zwischen den Teilzeichenketten $a[1 : i]$ und $b[1 : j]$ an. Jeder Eintrag der Tabelle wurde durch die Inspektion der Nachbarschaft oben, links und links oben gemäß Definition 7.1 gebildet. Die 0-te Zeile und Spalte sind immer gleich und entsprechen den ersten beiden Zeilen in Definition 7.1.

Bei der Bestimmung der minimalen Editierdistanz handelt es sich um ein **Optimierungsproblem**. Die Eigenschaft des Problems, die den Ansatz mit dynamischer Programmierung möglich macht, ist die, dass sich die optimale Lösung für $edist(i, j)$ aus den optimalen Lösungen der Teilprobleme $edist(i, j-1)$, $edist(i-1, j)$ und $edist(i, j)$ bestimmen lässt. Dies nennt man auch **optimale Teilstruktur**.

7.4 Der Floyd-Warshall-Algorithmus

Betrachten wir einen gerichteten Graphen $G = (V, E)$ und eine Funktion $w : V \times V \to \mathbb{N} \cup \{\infty\}$. Die Funktion w ordnet jeder Kante ein Gewicht zu. Darüberhinaus soll gelten:

$$\forall v \in V : w(v, v) = 0 \qquad \forall (x, y) \notin E : w(x, y) = \infty \tag{7.4}$$

Wir sind nun daran interessiert, für **jedes** Knotenpaar $(v, w) \in V^2$ die Länge des kürzesten Pfades zwischen v und w zu berechnen. Seien nun der Einfachheit halber die Knoten des Graphen positive natürliche Zahlen, sprich $V \subset \mathbb{N}^+$. Wir nennen die Menge der Knoten $v_2, \ldots, v_{n-1}$ die **inneren Knoten** eines Pfades $p : v_1, \ldots, v_n$.

Der Algorithmus nutzt folgende Beobachtung aus: Sei P_k die Menge alle Pfade von v_1 nach v_n deren innere Knoten alle aus der Menge $\{1, \ldots, k\}$ stammen. Sei $p \in P_k$ ein kürzester Pfad aus der Menge P_k. Sprich, es könnte noch einen kürzeren Pfad in G geben, aber keinen, dessen Knoten alle aus $\{1, \ldots, k\}$ stammen. Betrachten wir den Knoten k genauer:

- k ist kein innerer Knoten des Pfades p. Also ist ein kürzester Pfad der Menge P_{k-1} auch ein kürzester Pfad der Menge P_k.
- k ist ein innerer Knoten des Pfades p. Wir zerlegen den Pfad p in einen Pfad $p_1 : v_1, \ldots k$ und $p_2 : k, \ldots, v_n$. Beide Pfade, p_1 und p_2, sind kürzeste Pfade in P_k (Lemma 7.1, unten). Da aber k weder in p_1 noch in p_2 innerer Knoten ist, gilt nach obiger Überlegung, dass p_1 und p_2 kürzeste Pfade in P_{k-1} sind.

Zusammengefasst heißt das: Hat man alle kürzesten Pfade der Menge P_{k-1}, so kann man diese zu kürzesten Pfaden der Menge P_k erweitern. Letztendlich ist $k = |V|$ und man hat alle kürzesten Pfade zwischen allen Knotenpaaren in G gefunden. Hier tritt wieder die Eigenschaft der **optimalen Teilstruktur** auf: Man kann die optimale Lösung des Problems aus optimalen Lösungen der Teilprobleme konstruieren.

Es bleibt folgendes Lemma zu zeigen:

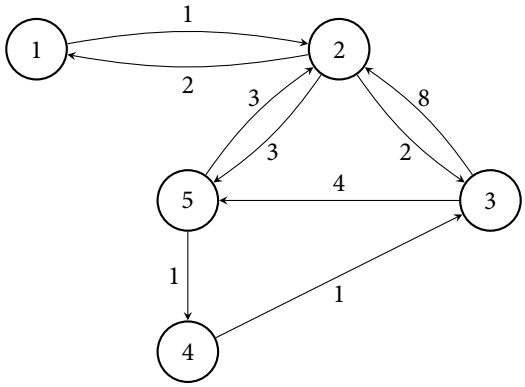
Lemma 7.1. *Gegeben ein Pfad $p : v_1, \ldots, v_l, \ldots v_n$. Ist p ein kürzester Pfad, so sind $v_1, \ldots v_l$ und $v_l, \ldots, v_n$ auch kürzeste Pfade.*

Berechnet werden alle kürzesten Pfade nun wieder mittels Tabellierung. Hierzu erstellt man eine Tabelle $T^{n \times n}$, in der es für jeden Knoten eine Zeile und eine Spalte gibt. In jedem Schritt berechnet der Algorithmus von Floyd-Warshall nun eine neue Version der Tabelle (Matrix) $T^{(k)}$, indem ein weiterer Knoten k hinzu genommen wird:

$$\begin{aligned} T_{ij}^{(0)} &= w(i, j) \\ T_{ij}^{(k)} &= \min\left(T_{ij}^{(k-1)}, T_{ik}^{(k-1)} + T_{kj}^{(k-1)}\right) \end{aligned} \tag{7.5}$$

Die beiden Alternativen des Minimums in (7.5) spiegeln die Situationen wider, die oben diskutiert wurden: k ist innerer Knoten des Pfades in P_k (dann ist der letzte Operand des Minimums kleiner gleich dem ersten). k ist kein innerer Knoten, dann ist der erste Operand kleiner.

Beispiel 7.3. Sei nun folgender Graph gegeben:



Ziel ist es alle kürzesten Pfade mithilfe des Floyd-Warshall-Algorithmus zu bestimmen. Hierzu initialisieren wir zuerst die Startmatrix $T^{(0)}$ mit Teil 1 in (7.5):

$T^{(0)}$	1	2	3	4	5
1	0	1	∞	∞	∞
2	2	0	2	∞	3
3	∞	8	0	∞	4
4	∞	∞	1	0	∞
5	∞	3	∞	1	0

Nun können wir Schritt für Schritt Teil 2 der (7.5) anwenden. Hier wird deutlich, dass im Gegensatz zur Rekursion bei zum Beispiel der minimalen Editierdistanz, der aktuelle Tabelleneintrag nicht von anderen Einträgen dieser Tabelle abhängt, sondern von vorherigen Tabellen abhängt. Es wird also in jedem Schritt des Algorithmus eine neue Matrix auf Basis der bisherigen Matrix erzeugt.

Schauen wir uns nun an wie sich der Eintrag (2,3) für die erste Iteration ergibt. Eingesetzt in (7.5) erhalten wir: $T^{(1)}_{2,3} = min\{T^{(0)}_{2,3}, T^{(0)}_{2,1} + T^{(0)}_{1,3}\}$ und damit $T^{(1)}_{2,3} = min\{2, 2 + \infty\} = 2$

$T^{(0)}$	1	2	3	4	5
1	0	1	∞	∞	∞
2	2	0	2	∞	3
3	∞	8	0	∞	4
4	∞	∞	1	0	∞
5	∞	3	∞	1	0

$T^{(1)}$	1	2	3	4	5
1	0	1	∞	∞	∞
2	2	0	2		
3	∞		0		
4	∞			0	
5	∞				0

Die Ausführung des gesamten Algorithmus ergibt schrittweise (Änderungen rot markiert):

$T^{(1)}$	1	2	3	4	5
1	0	1	∞	∞	∞
2	2	0	2	∞	3
3	∞	8	0	∞	4
4	∞	∞	1	0	∞
5	∞	3	∞	1	0

$T^{(2)}$	1	2	3	4	5
1	0	1	3	∞	4
2	2	0	2	∞	3
3	10	8	0	∞	4
4	∞	∞	1	0	∞
5	5	3	5	1	0

$T^{(3)}$	1	2	3	4	5
1	0	1	3	∞	4
2	2	0	2	∞	3
3	10	8	0	∞	4
4	11	9	1	0	5
5	5	3	5	1	0

$T^{(4)}$	1	2	3	4	5
1	0	1	3	∞	4
2	2	0	2	∞	3
3	10	8	0	∞	4
4	11	9	1	0	5
5	5	3	2	1	0

$T^{(5)}$	1	2	3	4	5
1	0	1	3	5	4
2	2	0	2	4	3
3	9	7	0	5	4
4	10	8	1	0	5
5	5	3	2	1	0

Die nun erzeugte Matrix $T^{(5)}$ gibt die Länge aller minimalen Pfade zwischen Knotenpaaren im Graphen an. ⌟

7.5 Der Algorithmus von Dijkstra

Gegeben ein gerichteter Graph $G = (V, E)$, ein Knoten $s \in V$ und eine Funktion $\ell : E \to \mathbb{R}^+$, die jeder Kante ein positives Gewicht zuordnet. Der Algorithmus von Dijkstra berechnet die kürzesten Pfade von s zu allen anderen Knoten in G. Die Länge $\ell(p)$ eines Pfades $p : s = x_1, \dots, x_n$ ist hierbei gegeben als Summe der Gewichte der Kanten des Pfades:

$$\ell(p) := \sum_{i=1}^{n-1} \ell(x_i, x_{i+1})$$

Der Algorithmus von Dijkstra ein sogenannter **single source** shortest path Algorithmus im Gegensatz zu Floyd-Warshall, der ein **all pairs** shortest path Algorithmus ist. Der Algorithmus beruht auf folgender Idee:

Angenommen, wir kennen für eine Menge $S \subseteq V$ von Knoten den kürzesten Pfad von s. Der Knoten s ist trivialerweise in S enthalten. Sei v ein Knoten aus S und $p : s, \dots, v$ ein kürzester Pfad von s nach v. Nach Lemma 7.1 sind dann alle Teilpfade auch kürzeste Pfade. Die Menge S enthält somit auch auf alle Knoten auf kürzesten Pfaden zu Knoten in S.

Wir betrachten nun die Menge der Pfade P_S von s zu Knoten, die bis auf den letzten Knoten nur Knoten aus S enthalten.

$$P_S = \{s, v_1, \dots, v_n, w \mid v_1, \dots, v_n \in S \text{ und } w \notin S\} \tag{7.6}$$

Betrachten wir nun den kürzesten Pfad $p : s, \dots, w$ der Menge P_S.

Lemma 7.2. *p ist der kürzeste Pfad von s nach w in G.*

Beweis. Beweis durch Widerspruch: Angenommen p ist nicht der kürzeste Pfad von s nach w in G. Dann gibt es einen kürzeren Pfad p' von s nach w.

Betrachten wir den ersten Knoten x von p', der nicht in S ist. p' hat also die Form:

$$p' : \underbrace{s, v_1, \dots, v_n}_{\in S}, x, \dots, w$$

Da p' kürzer ist als p, kann x nicht gleich w sein, da das der Wahl von p als kürzestem Pfad aus P_S widerspricht. Betrachten wir also den Fall $x \neq w$ und das Präfix $p'' : s, v_1, \dots, v_n, x$ von p'. Nach Voraussetzung liegt p'' in P_S. Da p ein kürzester Pfad von P_S ist, kann p'' nicht kürzer als p sein. Da $x \neq w$, ist der Pfad p' länger als p, da sein Präfix p'' schon mindestens so lang ist wie p. Das ist ein Widerspruch zur Annahme. □

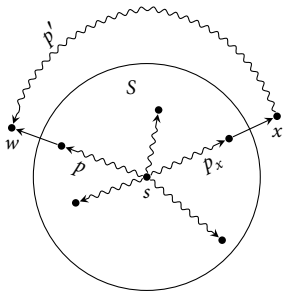


Abbildung 7.2: Die Situation in Lemma 7.2

Dies bedeutet, dass nun die Menge S schrittweise erweitert werden kann, bis sie alle Knoten des Graphen enthält und somit die kürzesten Pfade von s zu allen anderen Knoten bekannt sind. Interessant ist nun, wie wir den Pfad p effizient bestimmen. Die Menge P_S aufzuzählen um den kürzesten Pfad zu ermitteln wäre viel zu ineffizient. Wir merken uns daher in einer Abbildung `dist` die Entfernung jedes Knoten w, der

entweder in S oder eine Kante von S entfernt liegt. Entdeckt der Algorithmus einen neuen kürzesten Pfad $p : s, \ldots, w$ aus P_S, so können wir die Entfernungen der Nachbarn von w aktualisieren: Wir ermitteln für jeden Nachbarn v von w die Länge eines Pfades von s durch w zu v. War v bereits durch einen anderen Knoten w' aus S erreichbar, so aktualisieren wir die Entfernung von v, wenn der Pfad durch w kürzer ist als durch w'. War der Knoten v bislang noch nicht durch Knoten aus S erreichbar, so ist der Pfad durch w der aktuell kürzeste Pfad nach v.

```
public Map<Node, Double> shortestPath(Graph g, Node s) {
    Set<Node> nodes         = g.nodes();       // The set V
    Set<Node> done          = new HashSet<>(); // The set S
    Map<Node, Double> dist  = new HashMap<>();
    dist.put(s, 0.0);
    while (nodes.size() != done.size()) {
        // find node of dist \ done with minimum distance
        Node w = findMinimum(dist, done);
        assert !done.contains(w);
        double d = dist.get(w);
        // d == length of shortest path from s to w
        for (Edge e : g.edges(w)) {
            // look at every outgoing edge of w
            Node t = e.target();
            // t is end point of edge
            // if it is shorter to reach t via w than before
            // update length of the shortest path to t via nodes in done
            int distViaW = d + e.getLength();
            if (!dist.containsKey(t) || distViaW < dist.get(t))
                dist.put(t, distViaW);
        }
        done.add(w);
    }
    return dist;
}
```

Momentan ist die Laufzeit des Algorithmus quadratisch in der Anzahl der Knoten. Die äußere Schleife wird für jeden Knoten einmal durchlaufen. In der inneren Schleife ermittelt die Funktion `findMinumum` den Knoten dadurch, dass sie die ganze Abbildung `dist` durchsucht.

Das Finden des Minimums kann man durch die Wahl der richtigen Datenstruktur beschleunigen. Hierzu verwenden wir eine Prioritäts-Warteschlange, die auf der Heap-Datenstruktur [Cor+09] beruht. Eine Prioritäts-Warteschlange kann das Minimum der Einträge in $O(1)$ liefern und Elemente in $O(\log n)$ einfügen und löschen. In Java wird sie durch die Klasse `PriorityQueue` implementiert. Sie erwartet, dass die Elemente die Schnittstelle `Comparable` implementiert, um Elemente vergleichen zu können. Wir tragen dem hier Rechnung, indem wir die Abbildung `dist` dadurch ersetzen, dass wir die Entfernung direkt in den Knoten spichern. Die Klasse `Node` muss dann `Comparable` implementieren. Die Entfernung muss initial auf ∞ gesetzt werden.

```
public void shortestPath(Graph g, Node s) {
    Queue<Node> queue = new PrioirityQueue<Node>();
    queue.offer(s);
    while (! queue.isEmpty()) {
        // get node with minimal distance
        Node w = queue.poll();
        for (Edge e : g.edges(w)) {
            Node t = e.target();
            double currDist = t.getDist();
            double newDist = w.getDist() + e.getLength();
            if (newDist < currDist) {
                // if t is already in q, the priority queue
                // has to be updated. PriorityQueue does not implement
                // update so we have to remove and add the node
                if (currDist < Double.POSITIVE_INFINITY)
                    q.remove(t);
                t.setDist(d);
                q.add(t);
            }
        }
    }
}
```

7.5.1 Eine Heuristik

Der Algorithmus von Dijkstra geht ähnlich einer Breitensuche vor: Er verfolgt immer den aktuell kürzesten Pfad und bestimmt somit die kürzesten Pfade zu **allen** Knoten. Oft sind wir aber nur am kürzesten Pfad zu einem bestimmten Knoten t interessiert. In vielen Anwendungen, wie zum Beispiel der Routenplanung, hat man zusätzliche Informationen, wie zum Beispiel die Koordinaten des Quell- und Zielorts und eine Metrik, die man verwenden kann, um Pfade „zielgerichteter" zu verfolgen.

Diese zusätzlichen Informationen integrieren wir in die Kantengewichte, so dass „zielführendere" Kanten niedrigere Gewichte bekommen. Hierzu nehmen wir an, dass wir eine **Heuristik** $h : V \to \mathbb{R}^+$ haben, die uns für jeden Knoten die Entfernung zum Ziel t abschätzt. Wir verwenden h, um eine neue Kostenfunktion der Kanten ℓ' auf Basis der alten zu definieren:

$$\ell'(x, y) = \ell(x, y) + h(y) - h(x) \tag{7.7}$$

Der Term $\ell(x, y)$ entspricht den tatsächlichen Kosten der Kante und der Term $h(y) - h(x)$ beschreibt, wieviel man gemäß der Heuristik „gutgemacht" hat, wenn man die Kante (x, y) verfolgt. Der Algorithmus von Dijkstra fordert, dass die Kantengewichte positiv sind (sonst gilt die Argumentation in Lemma 7.2 nicht mehr). Also muss die Heuristik h die Eigenschaft

$$h(x) < \ell(x, y) + h(y) \text{ für alle } (x, y) \in E \tag{7.8}$$

erfüllen. Des Weiteren fordern wir, dass $h(t) = 0$ ist, da die Entfernung vom Zielknoten zu sich selbst 0 ist. Dies impliziert, dass die Heuristik die

tatsächliche Länge eines Pfades $p : s = x_1, \ldots, x_n = t$ von s nach t immer unterschätzt, da

$$h(x_1) < \ell(x_1, x_2) + h(x_2) < \cdots < \sum_{i=1}^{n-1} \ell(x_i, x_{i+1}) + \underbrace{h(t)}_{=0} = \ell(p)$$

Für die Länge des Pfades p bezüglich der Gewichte ℓ' gilt dann:

$$\begin{aligned} \sum_{i=1}^{n-1} \ell'(x_i, x_{i+1}) &= \sum_{i=1}^{n-1} \ell(x_i, x_{i+1}) + h(x_{i+1}) - h(x_i) \\ &= \ell(p) - h(x_1) + \underbrace{h(x_n)}_{=0} = \ell(p) - h(s) \end{aligned}$$

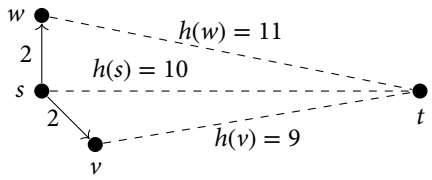
Die Länge des Pfads bezüglich der Gewichte ℓ' unterscheidet sich nur um den Subtrahend $h(s)$ von der Länge des Pfads bezüglich ℓ. Für alle Pfade, die von s ausgehen, ist dieser Subtrahend der gleiche. Daher ist ein kürzester Pfad von s bezüglich ℓ' auch ein kürzester Pfad bezüglich ℓ. Mit der Kostenfunktion aus (7.7), die eine Heuristik verwendet, die (7.8) erfüllt, liefert der Algorithmus von Dijkstra also immer noch den kürzesten Pfad.

Beispiel 7.4. Betrachten wir folgenden Graph. Gemäß (7.7) haben die Kanten folgende Gewichte:

$$\ell'(s, v) = 2 + 9 - 10 = 1$$

$$\ell'(s, w) = 2 + 11 - 10 = 3$$

Der Algorithmus von Dijkstra mit den Gewichten ℓ' würde also zuerst den Knoten v verfolgen.



Bemerkung 7.2. Mit einer Heuristik, die (7.8) erfüllt, ist der Algorithmus von Dijkstra ein Spezialfall des A*-Suchalgorithmus. Dieser ist dahingehend allgemeiner, dass er von einer Heuristik nur fordert, dass sie die Länge des kürzesten Pfades zum Ziel unterschätzt.

7.6 Aufgaben

Aufgabe 7.1 (★★★★). Wieviele rekursive Aufrufe erzeugt ein Aufruf von `fib(n)` (für die rekursive Implementierung von `fib` ohne Tabellierung). ⌟

Aufgabe 7.2 (★★★★). Aus

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

folgt mit $\det(A^n) = (\det A)^n$, dass

$$(-1)^n = F_{n+1}F_{n-1} - F_n^2$$

Nutzen Sie diesen Zusammenhang um eine Rekursionsgleichung für die Fibonacci-Zahlen abzuleiten, die F_n in $\log n$ Aufrufen berechnet. Hinweis: Für quadratische Matrizen gilt: $A^{2n} = A^n A^n$. ⌟

Aufgabe 7.3 (★). Bestimmen Sie $\binom{n}{k} < 2^{32}$ mit der Eigenschaft, dass keine weiteren n', k' existieren mit $\binom{n}{k} < \binom{n'}{k'} < 2^{32}$. ⌟

Aufgabe 7.4 (★★). Bestimmen sie die Levenshtein-Distanz zwischen `Wolle` und `Troll`. ⌟

Aufgabe 7.5 (★★★). Schreiben Sie ein C-Programm, dass die minimale Editierdistanz zweier Zeichenketten berechnet. Beide Zeichenketten sollen als Kommandozeilenparameter an das Hauptprogramm übergeben werden. Bedenken Sie, dass Sie die Länge der Zeichenketten statisch nicht kennen. Sie müssen also via `malloc` dynamisch Speicher anfordern und die Adressarithmetik selbst vornehmen. ⌟

Aufgabe 7.6 (★★★). Erweitern Sie ihr Programm aus Aufgabe 7.5 so, dass Folge von Editieroperationen ausgegeben wird, die zur minimalen Editierdistanz führen. Gibt es mehrere minimale Lösung, so reicht die Ausgabe einer Folge. ⌟

Aufgabe 7.7 (★★). Beweisen Sie Lemma 7.1. ⌟

Aufgabe 7.8 (★★★). Die rekursive Gleichung des Algorithmus von Floyd-Warshall ((7.5)) berechnet in jedem Schritt eine neue Matrix. Kann man den Algorithmus auch so implementieren, dass er das Element an Stelle i, j überschreibt, sprich man mit einer Matrix auskommt, die man modifiziert? ⌟

Aufgabe 7.9 (★★). Modifizieren Sie die Berechnung des Binomialkoeffizienten so, dass nur eine Tabelle der Größe $2 \times (K + 1)$ benötigt wird. Realisieren Sie dies, ohne Elemente der Reihung umzukopieren. ⌟

8 Objektorientierte Programmierung mit Java

Java ähnelt C syntaktisch sehr stark. Die meisten Anweisungen und Operatoren, die wir von C kennen, finden wir in Java wieder. In diesem Kapitel wollen wir kurz auf die wichtigsten Unterschiede eingehen. Die objektorientierten Sprachmittel von Java besprechen wir im nächsten Kapitel ausführlich.

8.1 Übersetzung, Hauptprogramm, Pakete

Ein Java-Programm besteht aus einer oder mehreren Dateien mit der Endung `.java`. In jeder `.java`-Datei wird eine **Klasse** vereinbart.[1] Diese Klasse muss den selben Namen wie die Datei tragen. Das heißt, in der Datei `X.java` muss eine Klasse `X` vereinbart werden:

```
public class X {
}
```

[1] Streng genommen kann man pro `.java`-Datei mehrere Klassen vereinbaren. Davon wird aber nur selten gebrauch gemacht. Jedoch darf höchstens eine der Klassen einer `.java`-Datei mit **`public`** gekennzeichnet sein.

Eine Datei `X.java` wird mit dem Java-Übersetzer `javac` in die **Klassendatei (engl. class file)** `X.class` übersetzt. Diese Datei entspricht den von C bekannten `.o` Objektdateien. In ihr sind Informationen zur Klasse und der sogenannte **Java-Bytecode** der Unterprogramme der Klasse gespeichert. Java-Bytecode ist eine abstrakte Maschinensprache, die vom Java-Laufzeitsystem zur Ausführung gebracht wird. Starten kann man ein Java-Programm durch einen Aufruf an das Java-Laufzeitsystem: `java X`

Hierbei muss eine Klassendatei `X.class` existieren, die ein Unterprogramm namens `main` aufweist. Unser erstes Java-Programm lautet also:

```
public class X {
    public static void main(String[] args) {
        System.out.println(”Hallo Welt”);
    }
}
```

Durch

```
javac X.java
```

übersetzen wir es in eine Klassendatei, die wir dann mit

```
java X
```

ausführen. Java erkennt eine Methode `main` nur als Startpunkt des Programms an, wenn sie den Rückgabetyp **`void`** und einen Parameter des Reihungstyps `String[]` hat und des Weiteren mit den Schlüsselwörtern **`public`** und **`static`** dekoriert ist. Der Parameter `args` von `main` ist eine Reihung von Zeichenketten. Dort sind die Kommandozeilenparameter des Programms, die ihm beim Starten des Programms mitgegeben wurden. Zum Beispiel gibt

```
public class ArgPrinter {
    public static void main(String[] args) {
        for (String s : args)
            System.out.println(s);
    }
}
```

alle Argumente zeilenweise aus:

```
java ArgPrinter das sind jetzt fuenf Argumente
```

erzeugt die Ausgabe

```
das
sind
jetzt
fuenf
Argumente
```

8.1.1 Mehrere Klassen

Realistische Java-Programme bestehen aus mehreren Klassen (und somit aus mehreren Dateien). Jede Klasse definiert einen eigenen Sichtbarkeitsbereich. Das heißt, die Bezeichner, die in einer Klasse vereinbart werden (wie beispielsweise die Methode `main` oben), sind zunächst nur innerhalb einer Klasse sichtbar. Durch **Qualifikation** können sie aber dennoch aus anderen Sichtbarkeitsbereichen benutzt werden:

```
// Y.java
public class Y {
    static void sageHallo() {
        System.out.println(”Hallo”);
    }
}

// X.java
public class X {
    public static void main(String[] args) {
        Y.sageHallo();
    }
}
```

Die Möglichkeit der Qualifikation kann vom Programmierer innerhalb gewisser Grenzen eingeschränkt werden. Abschnitt 8.5 geht darauf genauer ein.

8.1.2 Pakete

Die vielen Klassen eines Java-Programms können in Paketen strukturiert werden. Man stellt der Klassenvereinbarung die Anweisung

```
package paket;

public class X {
}
```

voran, um auszudrücken, dass die Klasse `X` im Paket `paket` liegt. Die Paketstruktur ist implizit mit der Verzeichnisstruktur der `.java`-Dateien verbunden. Der Java-Übersetzer erwartet die Datei `X.java` nun im Unterverzeichnis `paket`. Klassen in anderen Paketen können entweder über Qualifizierung angesprochen werden, oder durch importieren der Pakete. Will man beispielsweise auf eine Klasse `Y` im Paket `paket1`

```
// paket1/Y.java
package paket1;

public class Y {
    static void sageHallo() {
        System.out.println(”Hallo”);
    }
}
```

in einer Klasse `X` im Paket `paket2` Bezug nehmen, so schreibt man entweder:

```
// paket2/X.java
package paket2;
public class X {
    public static void main(String[] args) {
        paket1.Y.sageHallo();
    }
}
```

oder

```
// paket2/X.java
package paket2;

import paket1.Y;

public class X {
    public static void main(String[] args) {
        Y.sageHallo();
    }
}
```

8.2 Typen

In Java unterscheidet man zwischen **Basistypen** und **Referenztypen**. Die Referenztypen unterteilen sich in **Klassen** und **Reihungstypen**. Die Basistypen sind ähnlich zu C, jedoch ist ihre Größe und Arithmetik genau definiert.

Abbildung 8.1: Basistypen in Java

Typ	Beschreibung	Wertemenge	Standardwert
byte	8-bit Ganzzahl	$-128 \dots 127$	`0`
short	16-bit Ganzzahl	$-32768 \dots 32767$	`0`
int	32-bit Ganzzahl	$-2^{31} \dots 2^{31} - 1$	`0`
long	64-bit Ganzzahl	$-2^{63} \dots 2^{63} - 1$	`0L`
char	16-bit Unicode Zeichen	`'\u0000'`…`'\uFFFF'`	`'\u0000'`
float	32-bit IEEE 754 Gleitkomma		`0.0f`
double	64-bit IEEE 754 Gleitkomma		`0.0`
boolean	Wahrheitswerte	{**false**, **true**}	**false**

Auf den Basistypen findet automatische Typumwandlung (Konversion) gemäß der Ordnung in Abbildung 8.2[1] statt. Der Typ **boolean** ist nicht automatisch in einen anderen Typen konvertierbar und kein anderer Typ ist automatisch nach **boolean** konvertierbar.

[1] **Hinweis:** Diese Ordnung ergibt sich direkt aus der Teilmengenbeziehung der Wertemengen aus Abbildung 8.1. **char** ist der einzige **unsigned** Java-Datentyp und ist daher keine Teil- oder Obermenge von **short** und **byte**.

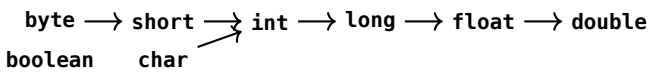


Abbildung 8.2: Automatische Typumwandlung zwischen Basistypen

Der Typ eines Ausdrucks *e* kann mittels expliziter Typumwandlung durch den Ausdruck `(T)e` in den Typ *T* (außer **boolean**) konvertiert werden. Durch explizite Typumwandlung kann Genauigkeit verloren gehen, so ist beispielsweise `(byte)1024` gleich `0`.

Beispiel 8.1 (Implizite und explizite Typumwandlung)**.**

```
short s = 1;
byte  b = 2;
int   i = 3;
i = s; // ok
b = s; // nicht ok, da short nicht
       // automatisch in byte umwandelbar
b = (byte)s; // ok, da explizite Umwandlung
```

⌟

8.3 Referenztypen

Der Behälter einer Variable eines Referenztyps enthält immer die Referenz[2] auf ein Objekt oder den Wert **`null`**. Im Gegensatz zu C gibt es in Java keine Referenzen auf bereits freigegebene Objekte deren Dereferenzierung zu undefiniertem Verhalten führen würde. Dies resultiert daraus, dass in Java keine manuelle Speicherbereinigung möglich ist (es gibt kein `free` wie in C). So lange ein Objekt vom aktuellen Programmzustand aus transitiv über Zeiger erreichbar ist, ist es lebendig. Um Speicher wieder freigeben zu können, setzen Java-Laufzeitumgebungen **automatische Speicherbereinigung (engl. Garbage Collection)** ein, um unerreichbar gewordene Objekte zu löschen.

[2] In Java verwendet man den Term „Referenz“ für „Adresse“.

Im Gegensatz zu C ist es in Java nicht möglich, Variablen einer Klasse oder einer Reihung zu vereinbaren. Implizit sind diese immer Referenzen auf dieselben. Die Objekte und Reihungen werden immer auf der Halde (engl. Heap) mittels **`new`** angelegt (siehe nächstes Kapitel).

8.3.1 Reihungen

Eine Variable eines Reihungstyps enthält immer eine **Referenz** auf einen Behälter, der die Länge der Reihung und die Elemente der Reihung enthält. Eine neue Reihung des Typs `T` kann mittels `new T[`*e*`]` erstellt werden, wobei *e* ein Ausdruck ist, der die Länge der Reihung ergibt. `new` liefert eine **Referenz** auf die neu erstellte Reihung. Reihungen können nicht vergrößert oder verkleinert werden: Bei ihrer Erstellung wird die Größe festgelegt, sie ändert sich während ihrer Lebenszeit nicht. Zum Beispiel:

```
int[] a = new int[10];
int[] b = a;
a[2] = 42;
// Gibt 42 aus
System.out.println(b[2]);
```

Hier wird also nicht die Reihung kopiert, sondern die Referenz darauf. Die Länge einer Reihung `a` kann durch `a.length` ermittelt werden:

```
// Gibt 10 aus
System.out.println(a.length);
```

Java kennt auch mehrdimensionale Reihungen, zum Beispiel:

```
int[][] arr    = new int[4][4];
int[]   subarr = arr[2];
```

Eine mehrdimensionale Reihung besteht allerdings nicht aus einem Behälter, der Platz für alle Elemente der mehrdimensionalen Reihung enthält. Er ist vielmehr eine Reihung von Reihungen. Da eine Reihung ein Referenztyp ist, ist jedes Element einer mehrdimensionalen Reihung eine Referenz auf eine weitere Reihung. In unserem Beispiel hat `arr[2]` den Typ `int[]`. Die Reihung `arr` enthält Referenzen auf Reihungen des Typs `int`. Abbildung 8.3 zeigt die Behälter nach Ausführung der obigen Zeilen.

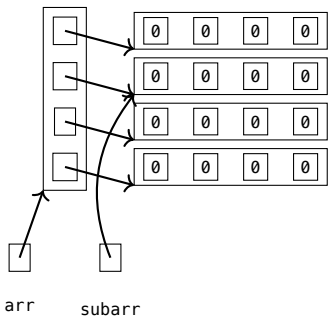


Abbildung 8.3: Mehrdimensionale Reihungen in Java

8.4 Klassen

Sehr häufig arbeiten wir in größeren Programmen mit **Verbunddatentypen** (in C `struct` genannt). Ein Verbunddatentyp ist das Kreuzprodukt mehrerer Typen. Ein Wert eines Verbunddatentyps ist also ein Tupel mehrerer Werte. Betrachten wir folgenden Verbund in C, der einen Punkt in der zweidimensionalen Ebene darstellen soll:

```
typedef struct {
  double x, y;
} vec2_t;
```

Da Änderungen und Berechnungen auf Verbunden meist mehrere Felder eines Verbundes beinhalten, lagert man diese in Unterprogramme aus:

```
void vec2_translate(vec2_t *p, double dx, double dy) {
  p->x += dx;
  p->y += dy;
}

void vec2_length(vec2_t const *p) {
  return sqrt(p->x * p->x + p->y * p->y);
}
```

Diese Unterprogramme haben eins gemeinsam: Sie bekommen als ersten Parameter einen Zeiger auf einen Verbund (Wert eines Verbundtyps). In der objektorientierten Programmierung schreibt man nun die Unterprogramme in den Verbundtyp. In Java schreibt man:

```
class Vec2 {
  double x, y;

  void translate(double dx, double dy) {
    this.x += dx;
    this.y += dy;
  }

  double length() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}
```

In der objektorientierten Programmierung hat sich eine andere Nomenklatur eingebürgert: Verbundtypen heißen **Klassen**, ein Wert einer Klasse heißt **Objekt** oder **Instanz**, Unterprogramme heißen **Methoden**, Zeiger heißen **Referenzen**. Jede Methode bekommt einen impliziten Parameter namens **`this`**: Er wird vom Compiler automatisch vereinbart und muss vom Programmierer nicht hingeschrieben werden. **`this`** ist eine **Referenz** auf ein Objekt der Klasse, in der die Methode steht. In obigem Beispiel ist der Typ von **`this`** also `Vec2`. Werden die Bezeichner, die in einer Klasse vereinbart werden, nicht von lokalen Variablen der Methode verdeckt, so kann **`this.`** auch entfallen. So könnte `dist` in `Vec2` auch so geschrieben werden:

```
class Vec2 {
  ...
  double length() {
    return Math.sqrt(x * x + y * y);
  }
}
```

8.4.1 Konstruktoren

Eine Klasse muss einen oder mehrere **Konstruktoren** anbieten, mit denen frisch erzeugte Objekte der Klasse initialisiert werden. Ein Konstruktor trägt denselben Namen wie die Klasse und hat keinen Rückgabetyp. In unserem Beispiel würde man einen Konstruktor einführen, der ein `Vec2` Objekt mit einer x- und einer y-Koordinate initialisiert.

```
class Vec2 {
  double x, y;

  public Vec2(double x, double y) {
    this.x = x;
    this.y = y;
  }
  ...
}
```

Alle Felder eines Objekts, die von einem Konstruktor nicht explizit initialisiert werden, werden (in Java) zu ihrem **Standardwert** initialisiert (siehe Abbildung 8.1).

Eine besondere Rolle spielt der **vorgegebene Konstruktor** (default constructor), der von Java automatisch vereinbart wird, sobald kein anderer Konstruktor in einer Klasse vereinbart wird. Er entspricht folgender Vereinbarung:

```
public class Vec2 {
  public Vec2() {
  }
}
```

Neue Objekte können nun mittels **`new`** und einem in der Klasse vereinbarten Konstruktoren erstellt werden:

```
public class Test {
  public static void main(String[] args) {
    Vec2 a = new Vec2(1.0, 0.0);
    System.out.println(a.length());
  }
}
```

Der Wert des Ausdrucks **`new`** `Vec2(1.0, 0.0)` ist vom Typ `Vec2` und stellt eine Referenz (Verweis) auf ein neu erzeugtes Objekt der Klasse `Vec2` dar. Die Lebensdauer des Behälters, der zur Variable `a` gehört, ist auf den umgebenden Block beschränkt. Das Objekt jedoch ist solange lebendig, solange es noch Behälter gibt, die es referenzieren. Hat ein Objekt keine

referenzierenden Behälter mehr, kann es von der **Speicherbereinigung** beseitigt werden.

Beim Methodenaufruf ist zu beachten, dass die Referenz auf das Objekt, das der Methode übergeben wird, nicht wie gewöhnlich in der Parameterliste auftaucht, sondern dem Methodennamen, abgetrennt durch einen Punkt, vorangestellt ist. Der Wert dieses besonders hervorgehobenen Parameters wird beim Aufruf in einen frischen Behälter kopiert, der im Rumpf der Methode durch `this` angesprochen werden kann.

Beispiel 8.2. Betrachten Sie den Aufruf `a.length()` im Beispiel oben. Obwohl kein Ausdruck in den runden Klammern steht, bekommt die Methode `length` ein Argument, nämlich den Wert des Ausdrucks `a` (eine Referenz auf ein Objekt). Zur Gegenüberstellung betrachten Sie nochmals das C-Unterprogramm `vec2_length`, das einen Parameter nimmt. ⌟

Bemerkung 8.1. Der Grund für die Sonderschreibweise ist, dass dieser erste Parameter eine Sonderstellung bei der Auswahl der zu rufenden Methode einnimmt, wie wir in Abschnitt 8.7 sehen werden. ⌟

8.5 Kapselung

Objektorientierte Sprachen wie Java bieten sprachliche Unterstützung zur Umsetzung von **Kapselung**. Darunter versteht man das Verbergen der Interna einer Datenstruktur. Warum es sinnvoll ist, Interna einer Datenstruktur zu verbergen, erschließt sich an unserem Beispiel. Betrachten wir hierzu eine andere Klasse, in der `Vec2` Objekte verwendet werden:

```
public class Rectangle {
  Vec2 upperLeft, lowerRight;
  ...
  double area() {
    double dx = lowerRight.x - upperLeft.x;
    double dy = lowerRight.y - upperLeft.y;
    return Math.abs(dx * dy);
  }
}
```

Die Klasse `Rectangle` modelliert anscheinend ein Rechteck und hat zwei Felder `upperLeft`, `lowerRight`, die die linke obere und rechte untere Ecke eines Rechteckes angeben. In der Methode `area` wird die Fläche des Rechteckes berechnet.

Aktuell verwendet `Vec2` kartesische Koordinaten. Die Implementierung von `area` in `Rectangle` macht davon auch explizit Gebrauch. Was ist aber, wenn der Autor der Klasse `Vec2` die interne Darstellung eines Punktes ändern will, indem er beispielsweise Polarkoordinaten einsetzt? Um diese Änderung durchführen zu können muss nun sämtlicher Code, der auf die Felder von `Vec2` zugreift mit angepasst werden. Das ist natürlich nicht zumutbar.

Um `Vec2` ändern zu können, ohne alle Verwender von `Vec2` modifizieren zu müssen, verwendet man **Kapselung**.

```
public class Vec2 {
  private double x, y;

  public double getX() { return x; }
  public double getY() { return y; }
  ...
}
```

Jeder Vereinbarung einer Klasse, eines Feldes oder einer Methode kann in Java das Schlüsselwort **`private`**, **`protected`** oder **`public`** vorangestellt werden. Hierdurch wird die Sichtbarkeit der vereinbarten Bezeichner eingeschränkt. **`private`** vor Methoden und Feldern bewirkt, dass der vereinbarte Bezeichner nur innerhalb der Klasse der Vereinbarung verwendet werden darf. Das würde den Zugriff auf die Felder `x` und `y` aus `Vec2D` aus der Klasse `Rectangle` unterbinden. **`public`** schränkt die Sichtbarkeit nicht ein. Abbildung 8.4 fasst alle Sichtbarkeitsmodifizierer von Java zusammen

Der Zugriff auf die Daten eines Objektes wird durch Zugriffsmethoden (auch genannt Getter und Setter) zur Verfügung gestellt. Dies

	Klasse	Paket	Unterklasse	Rest
public	✓	✓	✓	✓
protected	✓	✓	✓	–
fehlt	✓	✓	–	–
private	✓	–	–	–

Abbildung 8.4: Sichtbarkeitsmodifizierer in Java

hat den Vorteil, dass man nun die interne Repräsentation einer Klasse ändern kann, ohne dass der Code anderer Klassen modifiziert werden muss. Die Zugriffsmethoden stellen „nach außen“ immer die gleiche Sicht auf ein Objekt zur Verfügung. Ändert man nun `Vec2D` so, dass es Polarkoordinaten verwendet, so müssen die **Zugriffsmethoden** `getX` und `getY` entsprechend angepasst werden. Es ist ferner sinnvoll auch innerhalb der Klasse von den Zugriffsmethoden Gebrauch zu machen. In unserem Beispiel muss man dann die Implementierung der anderen Methoden, deren Implementierung auf kartesischen Koordinaten beruht, nicht anpassen.

```
public class Vec2 {
  private double r, phi;

  public double getX() { return r * Math.cos(phi); }
  ...
  public void setX(double x) {
    phi = Math.atan2(getY(), x);
    r   = x / Math.cos(phi);
  }
  ...

  public void translate(double dx, double dy) {
    setX(getX() + dx);
    setY(getY() + dy);
  }

  public double length() {
    return r;
  }
}
```

Kapselung ist auch hilfreich, um die Einhaltung von Invarianten, die auf den Feldern eines Objektes gelten, zu gewährleisten. Betrachten wir die folgende Klasse `Fraction`, die Brüche modellieren soll:

```
public class Fraction {
  private int numerator, denominator;
  ...
  public setDenominator(int d) {
    assert d > 0 : ”Denominator must be > 0”;
    this.denominator = d;
  }

  public boolean isPositive() {
```

```
    return numerator >= 0;
  }
}
```

Der Programmierer basiert seine Implementierung der Klasse `Fraction` anscheinend darauf, dass der Nenner immer größer als Null ist. Das heißt, `d` > 0 ist eine Invariante für alle `Fraction`-Objekte. Ist diese Invariante verletzt, funktioniert der Code der Klasse nicht mehr korrekt. In unserem Beispiel würde die Methode `isPositive` ein falsches Ergebnis liefern für `denominator` < 0 und `numerator` < 0. Mittels Zugriffsmethoden kann man nun vermeiden, dass das Objekt jemals in einen ungültigen Zustand gelangt, in dem man bei Verletzung der Invariante eine entsprechende Ausnahme auslöst.

Kapselung dient also der Modularisierung, sprich der Unabhängigkeit verschiedener Programmteile. Dies ermöglicht im Allgemeinen einfacheres Testen und leichtere Fehlersuche, da der zu testende bzw. zu untersuchende Bereich auf einzelne Module eingegrenzt werden kann.

8.6 Referenzen, Aliasing und unveränderliche Objekte

In Java sind Variablen, deren Typ eine Klasse ist, immer **Referenzen** (Zeiger) auf die jeweiligen Objekte. Betrachten wir folgendes Beispiel:

```
public class Foo {
    int x;
    int y;

  public static void main(String[] args) {
    Foo a = new Foo();
    a.x = 3;
    // Ab hier sind a und b Aliase
    Foo b = a;
    b.x = 5;
    System.out.println(a.x);  // gibt 5 aus
    System.out.println(b.x);  // gibt 5 aus
    // Ab hier nicht mehr
    b = new Foo();
    b.x = 3;
    System.out.println(a.x);  // gibt 5 aus
    System.out.println(b.x);  // gibt 3 aus
  }
}
```

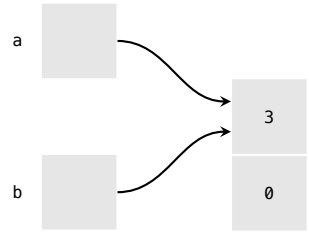


Abbildung 8.5: Visualisierung der Aliase a und b im Speicher aus dem Beispielcode

Hat ein Objekt, zu einem Zeitpunkt, zwei Referenzen *a* und *b*, die es referenzieren, so sagen wir *a* und *b* sind **Aliase**. In unserem Beispiel sind a und b zwischen den beiden Kommentaren Aliase auf das selbe Objekt.

8.6.1 Aliase

Referenzen haben den Vorteil, dass sie kompakt implementierbar sind. Sie werden vom Compiler üblicherweise durch eine Adresse im Hauptspeicher umgesetzt, an der die Felder des Objekts stehen. Beim Kopieren (wie es implizit auch beim Unterprogrammaufruf auftritt) muss also nur eine Adresse kopiert werden und nicht das gesamte Objekt. Verweisen mehrere Referenzen zum einem Zeitpunkt auf das selbe Objekt, kann das aber auch schwer zu findende Fehler hervorrufen. Problematisch daran ist, dass Aliasing der Modularität entgegensteht. Schreiben wir beispielsweise ein Unterprogramm, das ein Objekt verändert, so muss der Aufrufer des Unterprogramms darüber informiert sein, welche anderen Objekte dieses Objekt noch referenzieren und ob sie mit einem potentiell geänderten Objekt „klarkommen". Die Korrektheit des Programms hängt nun von einem korrekten Zusammenspiel mehrerer Objekte (von potentiell unterschiedlichen Klassen) ab. Dies steht der Modularisierung diametral entgegen.

8.6.2 Unveränderliche Klassen

Um Fehler, die durch Aliasing verursacht werden, zu vermeiden, kann man Klassen unveränderlich machen. Hierbei wird dafür gesorgt, dass sich die Werte der Felder nach der Konstruktion des Objekts nicht mehr

ändern. Somit ist insbesondere über Aliase keine Modifikation des Objekts mehr möglich. Das Kopieren der Referenz kommt nun dem Kopieren des Wertes gleich. Java unterstützt das unveränderlich-Machen durch das Schlüsselwort **`final`**:

```
public class Vec2 {
  private final double r, phi;

  public Vec2(double r, double phi) {
    this.r   = r;
    this.phi = phi;
  }
  ...
}
```

`final`-Feldern einer Klasse müssen im Konstruktor einmal ein Wert zugewiesen werden. Der Java-Übersetzer weist zur Übersetzungszeit jeglichen Code zurück, der ein **`final`**-Feld eines Objekts ändern könnte. Allerdings können nun auch Methoden der Klasse, wie die Methode `translate` oben, das Objekt nicht mehr ändern. Man behilft sich dadurch, dass `translate` nun ein neues Objekt, das den um `dx` und `dy` verschobenen Punkt repräsentiert, zurückgibt:

```
public class Vec2 {
  ...
  public Vec2 translate(double dx, double dy) {
    return cartesian(getX() + dx, getY() + dy);
  }

  public static Vec2 cartesian(double x, double y) {
    double phi = Math.atan2(y, x);
    double r   = x / Math.cos(phi);
    return new Vec2(r, phi);
  }
}
```

Die Methode `cartesian` hat hier die Aufgabe, die Argumente für den Konstruktoraufruf vorzubereiten, in diesem Fall die kartesischen Koordinaten in Polarkoordinaten zu transformieren. Die Methode ist mit **`static`** gekennzeichnet. Dies bedeutet, dass sie auf den impliziten Parameter **`this`** verzichtet, also unabhängig von einem Objekt der Klasse `Vec2` aufgerufen werden kann. Sie ist lediglich im Sichtbarkeitsbereich der Klasse `Vec2` vereinbart. Eine solche Methode, die dazu dient neue Objekte zu erzeugen, nennt man auch **Fabrikmethode**.

8.6.3 Wann unveränderliche Klassen?

Wann man Klassen unveränderlich macht und wann nicht, muss von Fall zu Fall abgewogen werden. Da man bei unveränderlichen Klassen Modifikation durch Replikation ersetzt, entstehen bei unveränderlichen Klassen viele Objekte, die alle alloziert und initialisiert werden müssen. Das kostet Zeit und Speicher. Eine Klasse `Image`, die ein Bild repräsentiert, ist beispielsweise kein guter Kandidat für eine unveränderliche

Klasse. Nehmen wir an, das Bild hat 1.000 × 1.000 Pixel und jeder Pixel belegt vier Byte. Dann benötigt ein Bild vier Megabyte. Will man nun einen Pixel ändern, ist es eine schlechte Idee, 999.999 Pixel zu kopieren um ein **neues** Bild zu kreieren.

Bemerkung 8.2. Viele wichtige Klassen in Java sind unveränderlich, die trickreiche Implementierungen verwenden um die Ineffizienzen, die durch das Kopieren von Objekten entstehen, zu kompensieren. Ein prominenter Vertreter ist `String`. ⌟

8.7 Vererbung

Wir haben gesehen, wie Java den Programmierer durch die Sichtbarkeitsmodifizierer beim Kapseln von Klassen unterstützt. Wir haben die Implementierung von `Vec2` zwar gekapselt, aber noch nicht von ihrer Spezifikation getrennt: Vereinbart man in einem Programm eine Variable vom Typ `Vec2`, so ist damit immer unsere Implementierung von `Vec2`, zuletzt mit Polarkoordinaten, gemeint. In größeren Programmen will man sich aber nicht auf eine konkrete Implementierung beziehen, sondern auf einen **abstrakten Datentypen**, der eine gewisse **Schnittstelle** anbietet. Der Code, der so einen abstrakten Datentypen verwendet, soll dann mit **jeder** Implementierung des abstrakten Datentypen funktionieren. Verschiedene Implementierungen sind dann austauschbar, was die Modularität des Codes erhöht.

8.7.1 Schnittstellen

Nehmen wir an, wir wollen in unserem Programm sowohl `Vec2`s mit kartesischen als auch `Vec2`s mit Polarkoordinaten verwenden. Dann müssten wir uns in einem Programm, dass diese Klassen verwendet, immer konkret auf eine der beiden Klassen beziehen:

```
public class Rectangle {
  private PolarVec2 upperLeft, lowerRight;
  ...
}
```

Man will bei der Implementierung der obigen Klassen `Rectangle` aber sich nicht darauf achten müssen, ob die Vektoren nun in Polarkoordinaten oder kartesischen Koordinaten gegeben sind. **Jede** Implementierung des ADT „Zweidimensionaler Vektor“ tut es hier. Um diese Flexibilität zu erreichten, bietet Java zwei wichtige Sprachmittel an: **Untertypen (engl. subtyping)** und **Schnittstellen (engl. interfaces)**. Durch eine Schnittstelle vereinbart man die Signatur eines abstrakten Datentyps: Alle seine Methodensignaturen[3] mit deren Parameter- und Ergebnisstypen.

[3] In Java heißen Prototypen Signaturen.

```
public interface Vec2 {
  void setX(double x);
  void setY(double x);
  double getX();
  double getY();
  void scale(double r);
  void add(Vec2 v);
  double length();
}
```

Alle vereinbarten Methoden haben die Sichtbarkeit **`public`**. Beachten Sie, dass man in Schnittstellen keine Felder vereinbaren kann. Das ist Absicht, denn man möchte ja in einer Schnittstelle gerade keine Aussage über die interne Implementierung eines ADTs treffen. Konkrete Implementierungen können nun diese (und beliebig viele andere) Schnittstellen **implementieren**:

```
public class PolarVec2 implements Vec2 {
```

```
  private double r, phi;

  public PolarVec2(double r, double phi) {
    this.r   = r;
    this.phi = phi;
  }

  public double getX() { return r * Math.cos(phi); }
  ...
  public void scale(double r) {
    setX(r * getX());
    setY(r * getY());
  }
  ...
}

public class CartesianVec2 implements Vec2 {
  private double x, y;

  public CartesianVec2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() { return x; }
  ...
  public void scale(double r) {
    setX(r * getX());
    setY(r * getY());
  }
  ...
}
```

PolarVec2 und CartesianVec2 sind nun Untertypen des Typs Vec2. Hierfür schreiben wir PolarVec2 $\leq$ Vec2 und CartesianVec2 $\leq$ Vec2[4]. Ist A ein Untertyp von B (man kann auch sagen: ist A Unterklasse von B), so können Referenzen auf Objekte der Klasse A auch in Behältern abgelegt werden, deren Variablen den Typ B haben. In unserem Beispiel erlaubt uns das, die Klasse Rectangle gegen die Schnittstelle Vec2 zu programmieren:

[4] $\leq$ ist kein Java-Operator; wir verwenden die Schreibweise A $\leq$ B nur als Abkürzung der Aussage A ist Untertyp von B.

```
public class Rectangle {
  private Vec2 upperLeft, lowerRight;

  public Rectangle(Vec2 upperLeft, Vec2 lowerRight) {
    this.upperLeft  = upperLeft;
    this.lowerRight = lowerRight;
  }
  ...
}
```

Wir können jedoch keine konkreten `Vec2`-Objekte anlegen, da in der Schnittstelle `Vec2` keine konkrete Implementierung ihrer Methoden angegeben ist (das war ja gerade beabsichtigt). Wir können Objekte beliebiger **Unterklassen** von `Vec2` zur Initialisierung eines `Rectangle` Objekts verwenden:

```
public class RectTest {
  public static void main(String[] args) {
    Vec2      v = new PolarVec2(1, Math.PI); // geht, da PolarVec2
                                             // Unterklasse von Vec2
    Vec2      w = new CartesianVec2(1, 0);   // dito
    Rectangle r = new Rectangle(v, w);
   ...
  }
}
```

Die Untertypsbeziehung ist eine Halbordnung. Sie ist

reflexiv. Für alle Typen T : $T \leq T$

transitiv. Für alle Typen U, V, W : $U \leq V \wedge V \leq W \implies U \leq W$

antisymmetrisch. Für alle Typen V, W : $V \leq W \wedge W \leq V \implies V = W$

8.7.2 Überschreiben von Methoden

Betrachten wir die beiden Klassen `CartesianVec2` und `PolarVec2` genauer, so sehen wir, dass einiger Code darin gleich ist. In unserem abgespeckten Beispiel ist das die Methode `scale`. In einer vollständigen Implementierung wären das weitaus mehr Methoden. `scale` verwendet lediglich die Setter und Getter von `Vec2` und verwendet keine weiteren Implementierungsdetails von `PolarVec2` oder `CartesianVec2`. Die aktuelle Implementierung dupliziert den Code von `scale`. Code-Duplikation ist der Feind jedes Programmierers und ist unter allen Umständen zu vermeiden: Ist der duplizierte Code fehlerhaft, muss er an mehreren Stellen repariert werden, die unter Umständen nicht leicht zu finden sind. Java hilft uns auch hier und erlaubt es uns zwischen `Vec2` und `CartesianVec2` einerseits und `PolarVec2` andererseits noch eine Klasse „einzuziehen“, die einige Teile von `Vec2` implementiert, andere aber **abstrakt** lässt:

```
public abstract class BaseVec2 implements Vec2 {
  public void scale(double r) {
    setX(r * getX());
    setY(r * getY());
  }
  public double length() {
    double x = getX();
    double y = getY();
    return Math.sqrt(x * x + y * y);
  }
}
```

```
public class PolarVec2 extends BaseVec2 {
  ...
}

public class CartesianVec2 extends BaseVec2 {
  ...
}
```

Wie von `Vec2` können von `BaseVec2` keine Objekte erzeugt werden, da einige Methoden noch nicht implementiert sind. Java zwingt uns dazu, den „unfertigen" Zustand von `BaseVec2` kenntlich zu machen, indem wir in der Klassenvereinbarung das Schlüsselwort **`abstract`** aufführen. Von abstrakten Klassen können prinzipiell keine Objekte erzeugt werden. `PolarVec2` und `CartesianVec2` **erben (engl. inherit)** dann von `BaseVec2` und **überschreiben** die restlichen Methoden aus `Vec2`.[5] Beim Vererben können aber auch bereits implementierte Methoden **überschrieben** werden. Das heißt, dass eine Unterklasse die von der Oberklasse vorgegebene Implementierung einer Methode verwirft und eine neue angibt. So kann zum Beispiel `PolarVec2` die Länge eines Vektors effizienter bestimmen, als das in `BaseVec2` geschieht:

```
public PolarVec2 extends BaseVec2 {
  private double r, phi;

  public double length() { return r; }
}
```

Somit entsteht eine ganze Vererbungshierarchie (Abbildung 8.6), die in der Praxis leicht mehrere Dutzend Klassen umfassen kann. Meist sind Vererbungshierarchien flach und breit anstatt tief.

[5] Es ist anzumerken, dass Schnittstellen nur eine spezielle Art von Klassen sind. Sämtliches Vokabular von Klassen ist auch auf Schnittstellen übertragbar. Die Tatsache, dass Java ein **`implements`** bei Schnittstellen und ein **`extends`** bei Klassen fordert, hat keine konzeptionellen, sondern nur ästhetische Gründe.

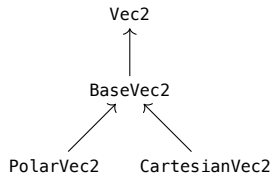


Abbildung 8.6: Vererbungshierarchie von `Vec2`

8.7.3 Der Methodenaufruf

Der Typ einer **Variable** (sofern der Typ eine Klasse ist) ist nun nur noch eine **obere Schranke** an den konkreten Typ des Objekts, das von ihr referenziert wird. Die Variablenvereinbarung

```
T a;
```

sagt letztlich, dass der Behälter zu `a` eine Referenz auf ein Objekt enthält, das **mindestens** den Typen `T` hat. Aber auch Objekte jeder Unterklasse von `T` sind erlaubt. Diese Eigenschaft des Java-Typsystems ermöglicht es ja gerade Implementierungen (Unterklassen) auszutauschen. Dadurch ergeben sich aber Mehrdeutigkeiten bei der Auswahl der zu rufenden Methode. Betrachten wir das Beispiel oben: Die Methode `length` wird von `BaseVec2` und von `PolarVec2` implementiert. Welche Methode tatsächlich gerufen wird, entscheidet der **konkrete Typ** des Objekts. Der konkrete Typ des Objekts ist der, dessen Konstruktor verwendet wurde, um das Objekt zu initialisieren.

```
Vec2 v;
double l;
v = new PolarVec2(1, 0.5);
// Ruft length aus PolarVec2
```

```
l = v.length();

v = new CartesianVec2(1, 0);
// Ruft length aus CartesianVec2
l = v.length();
```

Der konkrete Typ des Arguments `v` hat also einen Einfluss auf die zu rufende Methode. Das ist ein weiterer Grund, warum das **`this`**-Argument spezielle Syntax hat.

Es ist wichtig, den Unterschied zwischen dem **statischen Typen** und dem konkreten Typen zu verstehen. Der statische Typ ist der Typ, den der Übersetzer während der Übersetzungszeit für die Variable ermittelt. Er ergibt sich aus der Vereinbarung der Variable und ist, wie bereits erwähnt, eine obere Schranke an den konkreten Typen. Es gilt also immer:

$$\text{konkreter Typ} \leq \text{statischer Typ}$$

In obigem Beispiel ist der statische Typ der Variable `v` beide Male `Vec2`. Die Auswahl der zu rufenden Methode geschieht aber anhand des konkreten Typen.

Bemerkung 8.3. Der konkrete Typ eines Objekts ist nicht an jeder Programmstelle vom Übersetzer ermittelbar:

```
if (heute ist Dienstag)
  v = new PolarVec2(1, 0.5);
else
  v = new CartesianVec2(1, 0);
double l = v.length();
```

Es erfordert also zusätzlichen Aufwand, den konkreten Typen während der Programmausführung zu ermitteln. ⌟

Bemerkung 8.4. Durch den **`instanceof`**-Operator kann man **zur Laufzeit** testen, ob der konkrete Typ eines Objekts eine Unterklasse einer gegebenen Klasse ist. Dies wird häufig dazu eingesetzt, statische Typinformation zurückzugewinnen.

```
if (v instanceof PolarVec2) {
  PolarVec2 w = (PolarVec2)v;
}
```

Die Typumwandlung `(PolarVec2)v` prüft zur Laufzeit, ob der konkrete Typ des Objekts, das vom Behälter von `v` referenziert wird, ein Untertyp von `PolarVec2` ist. Falls das nicht der Fall ist, wird zur Laufzeit eine Ausnahme ausgelöst. ⌟

8.7.4 Die Klasse `Object`

Jede Java-Klasse erbt (in)direkt von der Klasse `Object`. Ist bei der Klassenvereinbarung keine Oberklasse angegeben, fügt Java ein implizites **`extends`** `Object` hinzu. In der Klasse `Object` sind einige Methoden vereinbart, die wichtig für das Zusammenspiel mit der Java-Standardbibliothek sind. Sie müssen unter Umständen geeignet überschrieben werden.

8.7.5 equals

In Java kann man zwischen dem gleichen und demselben Objekt unterschieden:

```
Vec2 v = new PolarVec2(1, 0.5);
Vec2 w = new PolarVec2(1, 0.5);
```

Die Objekte v und w[6] gleichen sich zwar, es sind aber nicht dieselben. Es sind zwei unterschiedliche Objekte, die aber die selben Werte in ihren Feldern haben. Der Gleichheitsoperator == vergleicht in Java prinzipiell die Referenzen, das heißt

[6] wir verzichten hier auf die ausführliche Sprechweise: „die Objekte, die von den Behältern …“.

```
System.out.println(v == w);
```

gibt `false` aus. Nun kommt es aber häufig vor, dass man mehr an der Gleichheit interessiert ist, als an der Identität. Gerade bei unveränderlichen Klassen kommt es häufig vor, dass mehrere **gleiche** Objekte gleichzeitig existieren. Daher verwendet Java an vielen Stellen, unter anderem im Collections Framework (siehe Kapitel 9) Gleichheit statt Identität. Hierzu muss eine Klasse aber erst definieren, wann sich zwei ihrer Objekte gleichen. **`boolean`** `equals(Object)` der Klasse `Object` überschreibt:

```
public class BaseVec2 {
  ...
  public boolean equals(Object o) {
    // Breche ab, wenn o nicht mindestens den Typ Vec2 hat.
    // Dies gilt beim Test mit instanceof auch, wenn o == null.
    if (!(o instanceof Vec2))
      return false;
    // o ist also mindestens ein Vec2 und nicht null
    Vec2 v = (Vec2)o;
    // Beide Vektoren sind gleich, wenn die Koordinaten gleich sind
    return getX() == v.getX() && getY() == v.getY();
  }
}
```

Bemerkung 8.5. In obigem Beispiel ist es essentiell, den Parametertyp von `equals` zu beachten. Oft passiert der Fehler, als Parametertyp des anderen Objekts, mit dem man sich vergleicht, die aktuelle Klasse anzugeben. In unserem Beispiel wäre das **`boolean`** `equals(BaseVec2)`. Java liefert in diesem Fall keinen Fehler, da dadurch die Methode `equals` **überladen** und nicht **überschrieben** (siehe Abschnitt 8.8) wird. ⌟

Bemerkung 8.6. Die Standard-Implementierung von `equals` in Objekt ist

```
public boolean equals(Object o) {
  return this == o;
}
```

⌟

8.7.6 `hashCode`

Die Methode `hashCode` dient dazu, für ein Objekt eine möglichst eindeutige Ganzzahl zu berechnen. Dies ist nötig, um ein Objekt in einer **Streutabelle (engl. hash table)** ablegen zu können. Überschreiben muss man `hashCode` nur, wenn man auch `equals` überschrieben hat. Denn jede Streutabellenimplementierung fordert, dass zwei gleiche Objekte auch denselben `hashCode` haben. Sprich, wenn für zwei Objekte p, q gilt, dass p`.equals(`q`) == true`, dann muss `hashCode` so implementiert sein, dass p`.hashCode() == `q`.hashCode()`.

Java kann die Einhaltung dieser Bedingung statisch nicht erzwingen. Es liegt also beim Programmierer, sie einzuhalten. Auf weitere Details bezüglich `hashCode` werden wir noch in Kapitel 9 eingehen.

8.7.7 `toString`

Wird dazu verwendet, um eine „menschenlesbare“, textuelle Repräsentation eines Objekts zu erstellen. Java ruft `toString` an einigen Stellen auf, um diese textuelle Darstellung zu erhalten:

- Viele Ausgabe-Routinen, beispielsweise `PrintStream.println()` akzeptieren eine Referenz vom Typ `Object`, um dann mittels `toString` die entsprechende textuelle Darstellung auszugeben.
- Der Operator + ist in Java vielfach überladen. Ist einer seiner Operanden vom Typ `String` und sein anderer Operand mindestens ein `Object`, so wird vom diesem Operanden durch das Rufen von `toString()` eine textuelle Darstellung des Objekts erzeugt und die beiden Texte werden aneinandergehängt.

In unserem Beispiel bietet sich folgende Implementierung an:

```
public class BaseVec2 {
  public String toString() {
    return "[" + getX() + ", " + getY() + "]";
  }
}
```

8.8 Überladen von Methoden

In Java ist es möglich, in einer Klasse mehrere Methoden mit selbem Namen zu vereinbaren. Um diese Methoden noch auseinander halten zu können, werden in Java Methoden nicht alleine durch ihren Namen, sondern über ihre **Signatur** identifiziert. Betrachtet man nun den Aufruf einer Methode `m`:

$$a.\texttt{m}(e_1, \ldots, e_n) \qquad a : U \quad e_i : U_i,\ 1 \leq i \leq n$$

so stellt sich die Frage, welche der überladenen Methoden `m` für einen Methodenaufruf überhaupt in Betracht kommen. Man beachte, dass die zu rufende Methode aufgrund des Überschreibens erst zur Laufzeit des Programms ermittelt werden kann. Die Auflösung der Überschreibung geschieht auf Basis des **konkreten (dynamischen) Typen** von a und nicht dessen statischen Typen.

Zunächst spielt die Anzahl der Argumente eine Rolle. Für obigen Aufruf kommen nur Vereinbarungen in Betracht, deren Signatur n-stellig ist. Dann wird überprüft, ob die statischen Typen der Argumente an die der Methodenparameter anpassbar sind. Die Menge dieser Signaturen nennt man die **passenden** Signaturen. Unter diesen wird nun die spezifischste Signatur gewählt. Da die Auflösung der Überladung zur Übersetzungszeit passiert, sind allein die statischen Typen relevant.

Definition 8.1 (Spezifischere Signatur)**.** Eine Signatur $(T_1, \ldots, T_n)$ ist **spezifischer** als eine Signatur $(U_1, \ldots, U_n)$, wenn $T_i \sqsubseteq U_i$ für alle $1 \leq i \leq n$ gilt. Wobei $T \sqsubseteq U$, wenn T anpassbar an U nach Abbildung 8.2 oder $T \leq U$. ⌟

Beispiel 8.3.

1. (`A`, **`int`**, **`int`**) ist spezifischer als (`A`, **`int`**, **`float`**).
2. (`A`) ist spezifischer als (`B`) wenn $A \leq B$ (beachte: $\leq$ impliziert $\sqsubseteq$). ⌟

Die zu rufende Methode wird nun aus der Menge der Methoden ermittelt, deren Signatur weniger (oder gleich) spezifisch ist als die Signatur des Aufrufs. Java fordert, dass diese Menge **eine** maximal spezifische Signatur enthält, deren Methode dann gewählt wird.

Definition 8.2 (Maximal spezifische Signatur)**.** Sei $\mathcal{S}$ eine Menge von Signaturen der Stelligkeit n. $s \in \mathcal{S}$ ist eine **maximal spezifische Signatur** von $\mathcal{S}$, wenn kein $s' \in \mathcal{S} \setminus \{s\}$ spezifischer ist als s. ⌟

Definition 8.2 schließt nicht aus, dass es pro Signaturenmenge mehrere maximal spezifische Signaturen gibt. In diesem Fall ist der Methodenaufruf **mehrdeutig** und Java weist das Programm mit einer Fehlermeldung zurück.

Beispiel 8.4. Betrachten wir den Aufruf der Methode `foo` in der Methode `call`.

```
class A {
  void foo(int x, int y, float z) {}
```

```
  void foo(boolean x, float y) {}
  void foo(int x, float y) {}
  void foo(long x, int y) {}
  void call() { this.foo(2, 2); }
}
```

Die Typen der Argumente sind (**int**, **int**). `foo` hat vier Vereinbarungen mit den Signaturen

(**int**, **int**, **float**), (**boolean**, **float**), (**int**, **float**), (**long**, **int**)

Die erste scheidet aus, da die Stelligkeit nicht passt. Die zweite scheidet auch aus, da die Typen der Argumente nicht an die der Parameter anpassbar sind (**int** nicht an **boolean** anpassbar). Die letzten beiden sind passend, jedoch sind sie beide **maximal spezifische** Signaturen. Java kann keine Vereinbarung ermitteln und meldet einen Fehler. Würde man eine Methode

```
  void foo(int x, int y) {}
```

hinzufügen, so wäre deren Signatur spezifischer als (**int**, **float**) und (**long**, **int**) und die Menge hätte eine eindeutige maximal spezifische Signatur. ⌟

Bemerkung 8.7. Zur Einfachheit der Betrachtung haben wir hier variadische Argumente (**...**), Boxing/Unboxing und Typvariablen ignoriert. Details findet man in Abschnitt 15.12.2 im Java Sprachbericht [Gos+05]. Die Relation $\sqsubseteq$ heißt dort **Method Invocation Conversion** (siehe Abschnitt 5.3 des Sprachberichts). Zur besseren Darstellung haben wir hier auf diese Details verzichtet. ⌟

8.8.1 Überladen und Überschreiben

Ist eine Methode `m` in einer Klasse `A` überladen, so kann beim Erben von `A` jede der überladenen Methoden getrennt überschrieben werden. Es wird also streng genommen nicht `m` überschrieben, sondern `m`$(T_1, \ldots, T_n)$, sprich das Überschreiben einer Methode bezieht sich immer auf eine konkrete überladene Methode. Es ist zu beachten, dass die überladene Methode statisch zur Übersetzungszeit ermittelt wird.

Beispiel 8.5. Betrachten wir folgende Klassen:

```
public class A {
  public void m(double X)  { System.out.println("A.m(double)"); }
  public void m(boolean x) { System.out.println("A.m(boolean)"); }
}

public class B extends A {
  public void m(int x)     { System.out.println("B.m(int)"); }
  public void m(boolean x) { System.out.println("B.m(boolean)"); }
}
```

Um deutlich zu machen, welche Methode welche überschreibt, betrachten wir folgende Tabelle:

Klasse	Methoden		
A	`A.m(double)`	`A.m(boolean)`	
B		`B.m(boolean)`	`B.m(int)`

Methoden einer Zeile, die gleichen Namen haben, überladen einander. Eine Methode in einer Spalte überschreibt die Methode in der gleichen Spalte in einer darüber liegenden Zeile. Hier überschreibt also `B.m(boolean)` die Methode `A.m(boolean)`. Betrachten wir folgende Programmfragmente, die `m` in verschiedenen Konstellationen aufrufen:

1. ```
   A a = new A();
   a.m(5);
   ```

   gibt `A.m(double)` aus. Trivial, da der konkrete Typ gleich dem statischen ist.

2. ```
   A a = new B();
   a.m(5);
   ```

   gibt `A.m(double)` aus. Der statische Typ von `a` ist `A`, der konkrete Typ bei diesem Programmlauf `B`. Jedoch überschreibt `B` die Methode `A.m(double)` nicht, sondern überlädt die Methoden `m(double)` und `m(boolean)` noch einmal mit dem Parametertyp `int`. Da die überladende Methode anhand des statischen Typs ermittelt wird, hier also `A.m(double)`, ist es irrelevant, dass im konkreten Typ `B` noch eine Methode `m(int)` existiert.

3. ```
   B b = new B();
   b.m(5);
   ```

   gibt `B.m(int)` aus. Der statische Typ ist nun `B` und `m` ist in `B` dreifach überladen.

⌟
   ```

8.9 Objektorientierung richtig nutzen

Objektorientierte Programmiersprachen helfen, ein wichtiges Problem bei der Entwicklung großer Software-Systeme zu lösen: Einfache Erweiterbarkeit. Einfach heißt hier, dass die Änderungen, die zu einer Erweiterung des Systems notwendig sind, lokal begrenzt sind und nicht eine große Zahl unterschiedlicher Programmstellen betreffen. Dies ist wichtig, da Änderungen an vielen unterschiedlichen Stellen sehr fehleranfällig sind: Der, der die Änderungen durchführt, muss den gesamtem Code, der geändert wird, verstehen, um sicherzustellen, dass die Änderungen korrekt und vollständig sind. Objektorientierte Sprachen erleichtern dies, da sie Mittel in der Programmiersprache bereitstellen, die, wenn sie richtig genutzt werden, Änderungen lokal begrenzen. Folgende Eigenschaften objektorientierter Sprachen sind hierzu wesentlich:

Die Trennung von Schnittstelle und Implementierung hilft bei der Implementierung von abstrakten Datentypen. Dies erleichtert die Erweiterbarkeit, da sich Code nicht auf eine konkrete Implementierung beziehen muss, sondern auf die Schnittstelle beziehen kann. Subtyping stellt sicher, dass alle Erweiterungen die von der Schnittstelle geforderte Funktionalität auch bereitstellen.

Die Konzentration von Daten und zugehörigem Code in einer Klasse verhindert, dass Implementierungsdetails über weite Code-Teile verstreut werden und für Teile des Codes zugänglich werden, der sie eigentlich nicht benötigt.

Wir wollen dies nun an einem Beispiel diskutieren, das wir in C und Java implementieren. Unser Programm soll die abstrakte Syntax einfacher arithmetischer Ausdrücke darstellen. Diese bestehen nur aus Variablen und der Addition von Ausdrücken:

$$\begin{array}{lcll} Exp \ni e & ::= & x & \text{Variable} \\ & | & e_1 + e_2 & \text{Addition} \end{array}$$

Auf diesen Ausdrücken wollen wir zusätzlich folgende Operationen ausführen: Ausgeben (oder in eine Zeichenkette verwandeln) und Auswerten. Abbildung 8.7 zeigt die Implementierung dieser Datenstruktur und der beiden Operationen in C und Java.

Betrachten wir zunächst die Implementierung in C. Die Tatsache, dass ein Ausdruck zwei **Varianten** hat (Variable und Addition), lässt sich in C nur umständlich ausdrücken: Der Verbund `exp_t` enthält eine **union**, die die Daten der beiden Varianten enthält. Das Feld `op` speichert, ob es sich bei dem Ausdruck um eine Addition oder das Auftreten einer Variable handelt. Festgelegt um welche Variante es sich handelt, wird im jeweiligen Konstruktor (siehe `exp_init_var` und `exp_init_add`).

Die Implementierung der Ausdrucksauswertung (in `exp_eval`) und des Ausgebens (in `exp_print`) nutzt dieses Feld, um für die jeweilige Variante den entsprechenden Code auszuführen. Diese beiden Funktionen beinhalten also den jeweiligen Code aller Varianten. Die Implementierung einer Variante ist in der C Version also auf viele Stellen verteilt: Das

enum, das alle Varianten auflistet, ein Feld in der **union** für die Daten und ein Zweig in jeder Funktion, die die Operationen des Datentyps implementiert.

Der C-Code kann nicht auf Vererbung zurückgreifen, weswegen in dem Verbund `exp_t` alle Felder aller Unterklassen versammelt sind. Das Fehlen von Klassen sorgt dafür, dass die Implementierungen der Methoden als Zweige einer Fallunterschiedung auftauchen (siehe `exp_eval` und `exp_print`). Diese Funktionen haben Zugriff auf alle Felder, wodurch es keine Kapselung der einzelnen Klassen `Add` und `Const` gibt.

Die Implementierung in Java nutzt die Vorteile der Objektorientierung. Zunächst wird eine Schnittstelle definiert, die alle Operationen vereinbart, die man auf Ausdrücken ausführen können soll. Jede Klasse, die diese Schnittstelle implementiert, entspricht einer Variante. Alle Aspekte dieser Variante (Implementierung der Operationen, Initialisierung, Daten) sind in der Klasse versammelt. Subtyping gewährleistet, dass man eine solche Klasse an allen Stellen verwenden kann, wo ein Objekt des Typs `Exp` erwartet wird. Dem Wert des Feldes `op` in der C-Implementierung entspricht in der Java-Implementierung der konkrete Typ des Objekts. Die Auflösung der Methodenüberschreibung zur Laufzeit entspricht den Fallunterscheidungen in den C-Funktionen `exp_eval` und `exp_print`. Sie wird, ohne Zutun des Programmierers, automatisch vom Übersetzer eingefügt. Des Weiteren können neue Varianten leicht hinzugefügt werden, wie Abbildung 8.8 zeigt. Hier wird ein neues Konstrukt „Konstante“ der abstrakten Syntax hinzugefügt. Hierzu muss, im Gegensatz zur C-Implementierung, keine Zeile des bereits existierenden Codes angepasst werden. Die Erweiterung um die Klasse `Const` ist lokal und betrifft den restlichen Code nicht. Die Pfeile in Abbildung 8.8 zeigen, wie die Java-Konstrukte in der C-Version implementiert werden.

Dass nun die jeweilige Methode `eval` und `toString` aus der Klasse `Const` aufgerufen wird, wenn der konkrete Typ eines Objekts `Const` ist, wird vom Übersetzer automatisch sichergestellt. Das heißt, dass die lästige und fehleranfällige Implementierung der Fallunterscheidungen in `exp_eval` und `exp_print` vom Übersetzer eingefügt wird. Wie dies genau funktioniert, diskutieren wir im nächsten Abschnitt.

C Implementierung

```c
struct env_t;
typedef struct env_t env_t;
int env_get(env_t const *e, char const* name);

typedef enum {
    ADD, VAR,
} operator_t;

typedef struct exp_t {
    operator_t op;
    union {
        struct exp_t *opnds[2];
        char const* var_name;
    };
} exp_t;

exp_t* exp_init_add(exp_t* t,
                    exp_t* l, exp_t* r) {
    t->op = ADD;
    t->opnds[0] = l;
    t->opnds[1] = r;
    return t;
}

exp_t* exp_init_var(exp_t* t, char const* n) {
    t->op = VAR;
    t->var_name = n;
    return t;
}
int exp_eval(exp_t const* t, env_t const* e) {
    switch(t->op) {
    case ADD:  return exp_eval(t->opnds[0], e)
                    + exp_eval(t->opnds[1], e);
    case VAR:  return env_get(e, t->var_name);
    }
}

void exp_print(exp_t const* t, FILE* f) {
    switch(t->op) {
    case ADD:  exp_print(t->opnds[0], f);
               fprintf(f, " + ");
               exp_print(t->opnds[1], f);
               break;
    case VAR:  fputs(t->var_name, f);
               break;
    }
}
```

Java Implementierung

```java
public interface Env {
    int get(String varName);
}

public interface Exp {
    public int eval(Env e);
    public String toString();
}

public class Add implements Exp {
    private Exp left, right;

    public Add(Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }

    public int eval(Env e) {
        return left.eval(e) + right.eval(e);
    }

    public String toString() {
        return "" + left + " + " + right;
    }
}

public class Var implements Exp {
    private String name;

    public Var(String name) {
        this.name = name;
    }

    public int eval(Env e) {
        return e.get(this.name);
    }

    public toString() {
        return this.name;
    }
}
```

Abbildung 8.7: Implementierung der abstrakten Syntax einer einfachen Ausdruckssprache in C und in Java.

C Implementierung

```
struct env_t;
typedef struct env_t env_t;
int env_get(env_t const *e, char const* name);

typedef enum {
    ADD, VAR,
    CNST
} operator_t;

typedef struct exp_t {
    operator_t op;
    union {
        struct exp_t *opnds[2];
        char const* var_name;
        int cnst_value;
    };
} exp_t;

exp_t* exp_init_add(exp_t* t,
                    exp_t* l, exp_t* r) {
    t->op = ADD;
    t->opnds[0] = l;
    t->opnds[1] = r;
    return t;
}

exp_t* exp_init_var(exp_t* t, char const* n) {
    t->op = VAR;
    t->var_name = n;
    return t;
}

exp_t* exp_init_cnst(exp_t* t, int value) {
    t->op = CNST;
    t->cnst_value = value;
    return t;
}
```

Fortsetzung

```
int exp_eval(exp_t const* t, env_t const* e) {
    switch(t->op) {
    case ADD:  return exp_eval(t->opnds[0], e)
                    + exp_eval(t->opnds[1], e);
    case VAR:  return env_get(e, t->var_name);
    case CNST: return t->cnst_value;
    }
}

void exp_print(exp_t const* t, FILE* f) {
    switch(t->op) {
    case ADD:  exp_print(t->opnds[0], f);
               fprintf(f, " + ");
               exp_print(t->opnds[1], f);
               break;
    case VAR:  fputs(t->var_name, f);
               break;
    case CNST: printf("%d", t->cnst_value);
               break;
    }
}
```

Java Implementierung

```
class public Const implements Exp {
    private int value;

    public Const(int value) {
        this.value = value;
    }

    public int eval(Env e) {
        return this.value;
    }

    public String toString() {
        return "" + this.value;
    }
}
```

Abbildung 8.8: Erweiterung der Datenstruktur aus Abbildung 8.7 um eine weitere Variante.

8.10 Objektorientierte Programmierung mit C

Im letzten Abschnitt haben wir gesehen, welche Unterschiede zwischen einer objektorientierten Sprache und einer rein imperativen Sprache wie C bestehen. In diesem Abschnitt wollen wir skizzieren, wie Objektorientierung in C umgesetzt werden kann.

Betrachten wir wieder das Beispiel aus Abbildung 8.7. Der Einfachheit gehen wir davon aus, dass jede Klasse nur von einer Klasse oder Schnittstelle erben kann. Die Unterstützung mehrerer Schnittstellen verkompliziert die Implementierung leicht, trägt aber nichts Fundamentales zum Verständnis bei.

Anstatt Funktionen zu erzeugen, in denen jede Klasse als Zweig abgehandelt wird (wie in Abbildung 8.7), erzeugt man für jede Methode eine Funktion. Für jede Klasse gibt es eine Tabelle von Funktionen (genau genommen von Funktionszeigern). Diese Tabelle heißt **Virtuelle Methodentabelle** oft auch durch VMT, vtable oder vtab abgekürzt. In dieser Tabelle stehen, für jede Methode der Klasse (und aller Oberklassen) die Adressen der Funktionen, die die jeweilige Methode implementieren. Da jede Klasse eine eigene VMT hat, identifiziert die Adresse der VMT die Klasse eindeutig. Sie entspricht daher dem „Etikett“ (engl. tag) `op` in des Verbunds `exp_t` in Abbildung 8.7. Jedes Objekt erhält bei der Initialisierung (an der ja der konkrete Typ bekannt ist), in einem vom Programmierer versteckten Feld (`vtab` in Abbildung 8.9) einen Zeiger auf die VMT der Klasse. Damit man die Adresse der VMT ohne Kenntnis des konkreten Typen ermitteln kann, legt man fest, dass der Zeiger auf die VMT in **allen** Objekten an der gleichen Stelle steht (normalerweise an Offset 0). Somit kann jedes Objekt zur Laufzeit seinen konkreten Typ ermitteln.

Die Adresse der zu rufenden, überschriebenen Methode ergibt sich nun in zwei Schritten: Zuerst muss der Eintrag in der VMT ermittelt werden. Dieser ergibt sich aus dem Methodennamen (und der Signatur, wenn man Überladung mit berücksichtigt) und kann somit **statisch**, zur Übersetzungszeit durchgeführt werden. In unserem Beispiel in Abbildung 8.9 entspricht der Eintrag dem entsprechenden Feld in den Verbunden `…vtab_t`. Die statische Semantik von Java garantiert uns, dass es zu der rufenden Methode in **allen** in Frage kommenden VMTs auch einen Eintrag für diese Methode gibt. Im zweiten Schritt wird die konkrete VMT bestimmt. Da dem Objekt bei der Konstruktion die Adresse seiner VMT in das versteckte Feld `vtab` geschrieben wurde, muss **zur Laufzeit** also nur noch diese Adresse geladen werden, um die richtige VMT zu erhalten.

Einen solchen Aufruf sieht man in `exp_add_eval` in Abbildung 8.9 in der **`return`**-Anweisung. Der Ausdruck `l->vtab->eval` ermittelt die Adresse der überschriebenen Methode `eval` durch die VMT.

```
// Dies geht aus der Vereinbarung des
// Interface Exp
typedef struct {
    int  (*eval)(void* this,
                 env_t const* e);
    void (*print)(void* this, FILE* f);
} exp_vtab_t;

typedef struct {
    exp_vtab_t* vtab;
} exp_t;

// var und exp haben die gleiche VMT,
// da Add keine neuen Methoden einführt.
typedef exp_vtab_t exp_add_vtab_t;

typedef struct {
    exp_add_vtab_t const* vtab;
    exp_t* opnds[2];
} exp_add_t;

// Const hat eine Methode mehr (get_value)
// Es ist wichtig, dass die Methoden in exakt
// der gleichen Reihenfolge eingetragen sind wie
// in exp_vtab_t, so dass die VMT von exp_t eine
// Untertabelle dieser VMT ist.
typedef struct {
    int  (*eval)(void* this,
                 env_t const* e);
    void (*print)(void* this, FILE* f);
    int  (*get_value)(void* this);
} exp_const_vtab_t;

typedef struct {
    exp_const_vtab_t const* vtab;
    int value;
} exp_const_t;
```

```
int exp_add_eval(void* this, env_t const *e) {
    exp_add_t* add = this;
    exp_t* l = add->opnds[0];
    exp_t* r = add->opnds[1];
    return l->vtab->eval(l, e)
         + r->vtab->eval(r, e);
}

// VMT von Add
static const exp_add_vtab_t exp_add_vtab = {
    exp_add_eval,
    exp_add_print
};

// Konstruktor von Add
exp_add_t* exp_init_add(exp_add_t* this,
                        exp_t* l, exp_t* r) {
    this->vtab     = &exp_add_vtab;
    this->opnds[0] = l;
    this->opnds[1] = r;
    return this;
}

int exp_const_eval(void* this, env_t const *e) {
    exp_const_t* t = this;
    return t->value;
}

// VMT von Const
static const exp_const_vtab_t exp_const_vtab = {
    exp_const_eval,
    exp_const_print,
    exp_const_get_value
};

// Konstruktor von Const
exp_const_t* exp_init_const(exp_const_t* this,
                            int value) {
    this->vtab  = &exp_const_vtab;
    this->value = value;
    return this;
}
```

Abbildung 8.9: „Objektorientierte“ Variante in C des Beispiels aus Abbildung 8.7.

8.11 Entwurfsmuster

Beim Entwurf objektorientierter Programme treten eine gewisse Menge an Problemen und Fragestellungen auf, für die man in den letzten Jahren und Jahrzehnten Patentrezepte entwickelt hat. Diese Patentrezepte werden **Entwurfsmuster (engl. design pattern)** genannt. Diese Fragestellungen beeinflussen nicht die Korrektheit des Codes in dem Sinn, dass ein Nicht-Anwenden des Entwurfsmusters zu falschem Code führt. Das Verwenden des Entwurfsmusters führt hingegen meist übersichtlicherem Code, der weniger Abhängigkeiten zu anderen Code-Teilen aufweist und somit leichter wartbar und erweiterbar ist. Daher empfiehlt es sich, von einigen wichtigen Entwurfsmustern Kenntnis zu haben, um beim Entwickeln neuer Programme zu bemerken, wann der Einsatz eines Entwurfsmusters angebracht ist. Wir wollen hier einige der wichtigsten Entwurfsmuster vorstellen.

8.11.1 Adapter

Ein Adapter kommt zum Einsatz, wenn man eine Schnittstelle mithilfe bereits existierender Klassen implementieren will. Will man beispielsweise eine Menge (siehe Abschnitt 9.3) mit einer Reihungsliste implementieren, so benötigt man eine Klasse, die die Schnittstelle `Set` implementiert und deren Methoden mit den Methoden der Klasse `ArrayList` implementiert. Ziel ist hierbei die Wiederverwendung bereits existierenden Codes. Dies kann man (meist) über zwei Techniken realisieren: Delegation oder Vererbung.

8.11.2 Adaption durch Delegation

Bei der Delegation enthält der Adapter eine Referenz auf ein Objekt der Klasse, die er adaptiert:

```
public class ArrayListSet<T> implements Set<T> {
    private final List<T> list = new ArrayList<T>();

    public boolean add(T e) {
        if (list.containts(e))
            return false;
        list.add(e);
        return true;
    }

    public boolean remove(T e) {
        boolean wasRemoved = list.remove(e);
        assert !list.contains(e);
        return wasRemoved;
    }

    public boolean contains(T e) {
        return list.contains(e);
    }
}
```

8.11.3 Adaption durch Vererbung

Meist kann man aber auch Vererbung einsetzen, um einen Adapter zu realisieren. Hierbei wird direkt von der zu adaptierenden Klasse geerbt:

```
public class ArrayListSet<T> extends ArrayList<T> implements Set<T> {
    public boolean add(T e) {
        if (contains(e))
            return false;
        return super.add(e);
    }
}
```

In diesem Fall wird die Implementierung sehr kompakt, da viele ererbte Methoden die von `Set` geforderte Funktionalität bereits implementieren. Durch Voranstellen des Schlüsselworts **`super`** können wir die Methode der Oberklasse rufen.

Delegation ist meist flexibler, da sich der Adapter nicht auf den konkreten Typ der zu adaptierenden Klasse festlegen muss. Er kann ja beispielsweise ein **Objekt**, das er adaptieren soll bei der Konstruktion übergeben bekommen. So könnte der Mengen-Adapter von oben beliebige Objekte, die mindestens `List` sind, an ein `set` adaptieren. Ein delegierender Adapter muss jede Methode einzeln implementieren (siehe oben). Dagegen resultiert Adaption durch Vererbung meist in kürzerem und eleganterem Code.

8.11.4 Abstrakte Fabrik

Das Ziel der abstrakten Fabrik ist es, bei der Erzeugung eines neuen Objektes, den konkreten Typen des zu erzeugenden Objektes nicht nennen zu müssen. Betrachten wir folgendes Beispiel:

```
public class Vec2Reader {

    public void read(Collection<Vec2> vectors) {
        for (/* file not empty */) {
            double x, y;
            /* read x, y from file into local variables */
            Vec2 p = new CartesianVec2(x, y);
            vectors.add(p);
        }
    }
}
```

Die Methode `read` soll hier eine Menge von zweidimensionalen Vektoren lesen, deren Koordinaten kartesisch sind. `read` muss also für die gelesenen Daten neue Objekte erzeugen. Dazu muss es sich auf einen konkreten Typ festlegen. Dies ist oft nicht gewünscht, da der Aufrufer von `read` vielleicht selbst entscheiden will, welche konkrete Implementierung von zweidimensionalen Vektoren verwendet werden soll (kartesische oder polare). Man möchte hier `read` also so schreiben, dass es zwar neue Objekte erzeugen kann, der konkrete Typ des erzeugten Objekts aber „von außen“ kontrolliert werden kann. Hierzu verwendet man eine abstrakte Fabrik:

```
interface Vec2Factory {
    /**
     * Creates a Vec2 for given cartesian coordinates.
     */
    Vec2 createCartesian(double x, double y);

    /**
     * Creates a Vec2 for given polar coordinates.
     */
    Vec2 createPolar(double r, double phi);
}

public class Vec2Reader {
    public void read(Collection<Vec2> vectors, Vec2Factory factory) {
        for (/* file not empty */) {
            double x, y;
            /* read x, y from file into local variables */
            Vec2 p = factory.createCartesian(x, y);
            vectors.add(p);
        }
    }
}
```

Diese abstrakte Fabrik kann `Vec2` Objekte für kartesische und polare Koordinaten erzeugen. Das heißt aber nicht, dass `createPolar` Objekte vom Typ `PolarVec2` erzeugt. Lediglich die gegebenen Koordinaten sind polar. Eine `Vec2Factory` implementierende Klasse legt den konkreten Typ des erzeugten Objekts fest. `read` bekommt nun ein Fabrikobjekt übergeben, das es nutzt, um neue Objekte zu erzeugen. Unterschiedliche Aufrufe an `read` können natürlich auch unterschiedliche Fabriken bekommen. Dadurch entkoppelt man die Implementierungen der Klasse `Vec2` vom Einlesen vollständig. `read` kann unabhängig von allen Implementierungen von `Vec2` geschrieben werden. Dies ermöglicht einen flexiblen Einsatz von `read`, da der Aufrufer nun selbst entscheiden kann, welche Objekte der Reader erzeugt, solange er eine passende Fabrik zur Verfügung stellt.

```
List<Vec2> vectors = new ArrayList<Vec2>();
reader.read(vectors, new Vec2Factory() {
    public Vec2 createCartesian(double x, double y) {
        return PolarVec2.newFromCartesian(x, y);
    }

    public Vec2 createPolar(double r, double phi) {
        return new PolarVec2(r, phi);
    }
});
```

8.11.5 Kompositum

Ein Kompositum K' ist eine Unterklasse einer Klasse K, deren Objekte jeweils eine Multimenge[7] von Objekten von K enthalten. Das Kompositum K' ist also ein K, enthält aber gleichzeitig mehrere Ks. Komposita kommen immer zum Einsatz, wenn eine (Multi-)Menge von Objekten der Klasse K behandelt werden kann/soll wie ein Objekt von K. In gewisser Weise ist ein Kompositum ein Adapter einer Multimenge von Ks an K. Komposita enthalten oft wenig „eigene" Funktionalität, sondern implementieren die geforderten Methoden dadurch, dass sie die entsprechenden Methoden der enthaltenen Objekte aufrufen und deren Ergebnisse geeignet aggregieren.

[7] Die Struktur der Multimenge hängt vom konkreten Fall ab. So kann sie einfach eine Liste oder aber vielleicht auch Baum sein.

Ein klassisches Kompositum sind zum Beispiel Klassen, die die Elemente einer grafischen Benutzeroberfläche repräsentieren. Unter gewissen Aspekten (zum Beispiel die geometrische Ausdehnung auf dem Bildschirm) kann man eine Gruppe von Bedienelementen auch als ein Bedienelement sehen:

```
public interface GUIElement {
    Coordinate getUpperLeft();
    Coordinate getLowerRight();
}

public class Button implements GUIElement {
    // ...
}

public class ElementGroup implements GUIElement {
    List<GUIElement> elements;
    Coordinate upperLeft, lowerRight;

    // ...

    public void add(GUIElement e) {
        upperLeft = upperLeft.min(e.getUpperLeft());
        elements.add(e);
    }

    public Coordinate getUpperLeft() {
        return upperLeft;
    }
}
```

8.11.6 Besucher

Der Besucher ist ein komplexes Entwurfsmuster, das im wesentlichen dazu dient, eine bereits existierenden Klassenhierarchie um Funktionalität zu erweitern, ohne die Klassen verändern zu müssen. Dies ist oft sinnvoll, da diese Funktionalität oft andere Aspekte hat, die nicht direkt der entsprechenden Klasse zuzuordnen sind. Betrachten wir folgende Klassenhierarchie, die arithmetische Ausdrücke repräsentieren soll.

```
public interface Expr {
}

public class Binary implements Expr {
    private Expr left, right;
    public Expr getLeft() { return left; }
    public Expr getRight() { return right; }
}

public class Variable implements Expr {
    private String name;
    public String getName() { return name; }
}

public class Constant implements Expr {
    private int value;
    public int getValue() { return value; }
}
```

Nehmen wir an, in einem größeren Projekt spielen diese Klassen eine große Rolle. In dem Projekt soll allerhand Funktionalität für die Ausdrücke hinzugefügt werden. So sollen Ausdrücke ausgewertet werden können, als herkömmlicher Text und als XML ausgegeben werden können. Es ist nicht auszuschließen, dass zukünftig noch mehr Funktionalität hinzukommt.

Eine Möglichkeit ist, jeder Unterklasse entsprechende Methoden hinzuzufügen, die die entsprechende Funktionalität implementieren. Ist die Funktionalität komplexer, beispielsweise bei der XML-Ausgabe, so verlieren die Klassen aber schnell ihrere „Identität“: Ursprünglich sollten sie ja nur arithmetische Ausdrücke darstellen und nichts mit XML-Darstellung, etc. zu tun haben.

Eine (schlechte) Möglichkeit, die Funktionalität ausserhalb der Klassenhierarchie anzusiedeln, ist eine Fallunterscheidungs-Kaskade:

```
public class ExprXMLWriter {
    public void write(Expr e) {
        if (e instanceof Constant) {
            Constant c = (Constant) e;
            write("<constant>" + c.getValue() + "</constant>");
        }
        else if (e instanceof Variable) {
            Variable v = (Variable) e;
            write("<variable>" + v.getName() + "</variable>");
        }
        else if // etc.
    }
}
```

Dieser Code ist aus mehreren Gründen schlecht. Zum einen vergisst man leicht Fälle. In diesem Fall warnt einen der Compiler nicht. Zum anderen ist der Code ineffizient, da solange **`instanceof`** Bedingungen

geprüft werden müssen, bis der zutreffende Fall auftritt. Zudem bestimmt die Reihenfolge der `if`s die Laufzeit. Man würde gerne Code für jede Unterklasse in einer eigenen Methode haben um den Code sauber zu trennen, in etwa so:

Daher schafft man sich eine generische Möglichkeit um beliebige Funktionalität der Klassenhierarchie hinzuzufügen zu können.

```
public class ExprXMLWriter {
    public void write(Constant c) {
        write("<constant>" + c.getValue() + "</constant>");
    }
    public void write(Variable v) {
        write("<variable>" + v.getName() + "</variable>");
    }
    // die anderen faelle ...
}
```

Nun muss man aber dafür sorgen, dass diese Methoden auch aufgerufen werden. Das ist insofern problematisch, da man an der Stelle, wo man die Methode aufrufen will, vermutlich nur den statischen Typ `Expr` hat und den konkreten Typ nicht kennt:

```
public class Main {
    public static void main(String[] args) {
        Expr e = constructExpressionFromInput();
        ExprXMLWriter w = new ExprXMLWriter();
        w.write(e);
    }
}
```

Dieser Code wird nicht funktionieren, da keine Methode `write(Expr)` in `ExprXMLWriter` vorhanden ist. Man müsste also wieder eine Fallunterscheidung bauen um den konkreten Typ des Objekts herauszufinden, eine explizite Typumwandlung durchzuführen, und dann die entsprechende Methode aufzurufen:

```
public class ExprXMLWriter {
    public void write(Constant c) {
        write("<constant>" + c.getValue() + "</constant>");
    }
    public void write(Variable v) {
        write("<variable>" + v.getName() + "</variable>");
    }
    // die anderen faelle ...

    public void write(Expr e) {
        if (e instanceof Constant)
            write((Constant) c);
        else if (e instanceof Variable)
            write((Variable) v);
        else // ...
    }
}
```

Das ist hässlich und nur wenig besser als der Code oben.

Durch eine Indirektion kann man die Fallunterscheidung eliminieren. Das Objekt e aus dem letzten Code-Abschnitt kennt ja seinen konkreten Typ. Wir erweitern die Schnittstelle `Expr` um eine einzige Methode `accept`, die nichts anderes tut, als eine andere Methode `visit` einer Schnittstelle `ExprVisitor` zurückzurufen. Hierbei übergibt sie **`this`** an `visit`.

```
public interface ExprVisitor<T> {
    T visit(Constant c);
    T visit(Variable c);
    T visit(Binary c);
}

public interface Expr {
    <T> T accept(ExprVisitor<T> v);
}

public class Constant implements Expr {
    // ...
    <T> public T accept(ExprVisitor<T> v) {
        return v.visit(this);
    }
}

public class Variable implements Expr {
    // ...
    <T> public T accept(ExprVisitor<T> v) {
        return v.visit(this);
    }
}

public class Binary implements Expr {
    // ...
    <T> public T accept(ExprVisitor<T> v) {
        return v.visit(this);
    }
}
```

Der `XMLWriter` implementiert nun die Schnittstelle `ExprVisitor` und implementiert die entsprechende Funktionalität für jeden der Fälle.

```
public class ExprXMLWriter implements ExprVisitor<Void> {
    // ...
    public Void visit(Constant c) {
        write("<constant>" + c.getValue() + "</constant>");
        return null;
    }

    public Void visit(Variable v) {
        write("<variable>" + v.getName() + "</variable>");
```

```
        return null;
    }

    public Void visit(Binary b) {
        write("<binary>");
        b.getLeft().accept(this);
        b.getRight().accept(this);
        write("</binary>");
        return null;
    }
}
```

Die Besucherklasse hat einen Typparameter. Dieser wird für die Besuchsfunktionen verwendet und erlaubt somit, dass eine Implementierung der Klasse einen Rückgabetyp wählen kann. Typparameter müssen immer mit einer Klasse belegt werden. Um keinen Rückgabewert zu simulieren, wird in diesem Beispiel die Klasse `Void` und **`null`** als Wert verwendet.

Im Hauptprogramm wird ein `XMLWriter`-Objekt dann als Argument an `accept` übergeben:

```
public class Main {
    public static void main(String[] args) {
        Expr e = constructExpressionFromInput();
        ExprXMLWriter w = new ExprXMLWriter();
        e.accept(w);
    }
}
```

8.12 Aufgaben

Aufgabe 8.1 (★★). In dieser Aufgabe entwickeln wir eine Klasse `Fraction`, die Brüche darstellt.

1. Überlegen Sie, welche Felder die Klasse hat und kapseln Sie die Felder.
2. Schreiben Sie zwei Konstrukturen: Der erste soll einen Bruch mit Zähler und Nenner initialisieren. Der zweite soll den Bruch mit einer ganzen Zahl implementieren. Verwenden Sie den ersten Konstruktur um mit ihm den zweiten zu implementieren. Sie können mit `this(...);` einen Konstruktor aus dem anderen rufen.
3. Schreiben Sie getter und setter für Zähler und Nenner.
4. Fügen Sie Methoden hinzu, die einen Bruch mit einem anderen addieren und multiplizieren.
5. Überschreiben Sie die Methode `equals` angemessen. Zwei Brüche, die nach Kürzen gleich sind, sollen gleich sein.
6. Angenommen, Sie würden eine Klasse für eine ganze Zahl einführen, wäre diese Klasse eine Unter- oder Oberklasse von `Fraction`?
7. Erstellen Sie eine unveränderliche Version der Klasse `Fraction`

⌟

Aufgabe 8.2 (Equals. ★★). Führen Sie folgenden Code aus und erklären Sie die Ausgabe mithilfe von Überladung und Überschreibung. Ändern Sie die Implementierung von `Foo` sodass sie sich wie gewünscht verhält. Lesen Sie über die Annotation `@Override` nach und führen Sie kurz aus, wie diese Annotation den Fehler hätte vermeiden können.

```
class Foo {
   private int a;
   pubic Foo(int a) { this.a = a; }
   public boolean equals(Foo b) { return a == b.a; }
   public static void main(String[] args) {
     Object a = new Foo(1);
     Object b = new Foo(1);
     System.out.println(a.equals(b));
     System.out.println(a.equals(a));
   }
}
```

⌟

Aufgabe 8.3 (Überladen und Überschreiben. ★★★). Betrachten Sie folgenden Code und begründen Sie, welche Methoden in Zeilen 17–19 aufgerufen werden. Argumentieren Sie mit Überladung und Überschreibung.

```
1  class A {
2    void f (float v) { System.out.println("A.f(float)"); }
```

```
}

class B extends A {
  void f (int v) { System.out.println("B.f(int)"); }
}

class C extends B {
  void f (int v) { System.out.println("C.f(int)"); }
}

A a = new B();
B b = new B();
B c = new C();

a.f(15);
b.f(15);
c.f(15);
```

Aufgabe 8.4 (Statische und dynamische Typen in Java. ⋆⋆⋆). Berechnen Sie für jede linke und rechte Seite einer Zuweisung den statischen Typ. Berechnen Sie außerdem für jede Variable und jedes Feld auf der linken Seite den dynamischen Typ nach dieser Zuweisung. Überlegen Sie was das Programm ausgibt und erläutern Sie wie die Ausgabe zustande kommt.

```
public class A{
    public A f;

    public void print() {
        System.out.println("A");
    }

    public static void main(String[] args) {
        A u = new A();
        B v = new B();
        C w = new C();
        A x = u;
        A y = v;
        A z = w;
        x = v;
        x.print();
        y = w;
        y.print();
        x.f = x;
        x = y;
        x.print();
        v.g = (B)x;
        x.f = y;
        w.h = (C) v.g;
```

```
        ((A) w.h).print();
        v.f.print();
    }
}
class B extends A{
    public B g;
}
class C extends B{
    public C h;

    public void print() {
        System.out.println("C");
    }
}
```

Aufgabe 8.5 (Referenzen. ⋆⋆). Implementieren Sie eine Klasse `Rectangle`, die ein Rechteck mithilfe von zwei `Vec2` Objekten darstellt. Implementieren Sie einen geeigneten Konstruktor. Schreiben sie Zugriffsmethoden für den Punkt links oben und den Punkt rechts unten.

Testen Sie ihr Programm nach dem folgenden Schema, um sicherzustellen, dass die Klasse `Rectangle` ihre internen Daten richtig kapselt:

```
Vec2 v = new Vec2(1.0, 2.0);
Vec2 w = new Vec2(3.0, 4.0);
Rectangle r = new Rectangle(v,w);
System.out.println(r);
v.setX(10.0);
System.out.println(r);
r.getTopLeft().setX(10.0);
System.out.println(r);
```

Aufgabe 8.6 (Vektoren und Methodenverkettung. ⋆⋆⋆). In dieser Aufgabe schreiben Sie eine Klasse `Vec2`, die Punkte aus $\mathbb{R}^2$ darstellt.

1. Definieren Sie die Klasse `Vec2` mit den privaten Feldern x, y vom Typ `double`. Erstellen Sie einen Konstruktor mit zwei Argumenten vom Typ `double`.
2. Fügen Sie die Methode `toString` in ihre Klasse ein, die einen String zurück gibt, der einen `Vec2` in einem für Menschen leicht lesbaren Format darstellt. Erweitern Sie dazu die folgende Implementierung um die Ausgabe des Feldes y.

   ```
   public String toString() {
     return "(" + x + ")";
   }
   ```

3. Fügen Sie Zugriffsmethoden für x und y hinzu.
4. Testen Sie ihre Implementierung, indem Sie die folgende Methode `main` in ihre Klasse einfügen und ausführen:

```
  static public void main(String[] args) {
    // hier deinen Vec2 v erstellen
    // Folgende Anweisung gibt v gemäß seiner toString Methode aus.
    System.out.println(v);
  }
```

Ihr Programm sollte die Ausgabe `(1.0, 2.0)` erzeugen.

5. Implementieren Sie die Methoden `addAssign` und `mulAssign` mit Rückgabetyp `void` und den folgenden Signaturen:

```
addAssign(double scalar)
addAssign(final Vec2 point)
mulAssign(double scalar)
mulAssign(final Vec2 point)
```

   `mulAssign` soll die beiden Vektoren komponentenweise multiplizieren, und nicht das Punktprodukt bilden. Testen Sie die Implementierung, indem Sie zwei Vektoren erstellen, addieren, mit 5 multiplizieren, und ausgeben.

6. Die Methoden aus dem vorherigen Aufgabenteil hatten den Rückgabetyp **`void`**. In diesem Aufgabenteil ändern wir die Methoden so ab, dass wir die Berechnung $v * 5 + w$ mit `v.mulAssign(5).addAssign(w)` bewerkstelligen können. Dieses Programmieridiom wird Methodenverkettung (engl. **method chaining**) genannt. Passen Sie `mulAssign` und `addAssign` so an, dass Sie den Vec2 selbst zurückgeben. Stellen Sie sicher, dass ihr geändertes Programm nach wie vor die selbe Ausgabe erzeugt.

7. Implementieren Sie die Methode `clone`, sodass `v.clone()` einen neuen `Vec2` liefert, der die gleichen Koordinaten wie v besitzt. Testen Sie `clone` in ihrer `main` Methode.

⌟

Aufgabe 8.7 (Statische Methoden. ⋆⋆)**.** In Aufgabe 8.6 haben wir die Addition und Multiplikation für `Vec2` so definiert, dass das Objekt selbst verändert wurde. Jetzt implementieren wir statische Methoden, die jeweils ein neues Objekt zurückgeben.

1. Implementieren Sie `add`, `mul`, und `mod` nach dem folgenden Schema:

```
public static Vec2 add(final Vec2 v, final Vec2 w) {
  //TODO
}
```

   Die Methoden sollen v und w jeweils unverändert lassen und ein neues `Vec2` Objekt zurückgeben. Testen Sie in ihrer `main`, dass `Vec2.mul(v,w)` die `Vec2` Objekte v und w tatsächlich nicht verändert.

2. Implementieren Sie jetzt eine weitere statische Methode `rotate`, die zu einem `Vec2` v und einem `double` a einen neuen Vektor liefert, der den um a rotierten Vektor v darstellt. Der Winkel a soll im Bogenmaß

angegeben werden. Es kann hilfreich sein, sich über die Bibliotheksklasse `java.lang.Math` zu informieren.

⌟

Aufgabe 8.8 (. ★★★)**.** Warum ist es wichtig, dass die VMTs aller Unterklassen die gleiche Anordnung haben. Sprich, wenn die Methode *m* einer Klasse *A* das Offset *o* in der VMT von *A* hat, warum muss *m* dann auch Offset *o* in den VMTs aller Unterklassen von *A* haben? Kann man das immer erreichen? ⌟

Aufgabe 8.9 (. ★★)**.** Betrachten Sie die Umsetzung der Objektorientierung in C. Fügen Sie dem Beispiel aus noch den Fall für Variablen hinzu. ⌟

9 Einfache Datenstrukturen

9.1 Listen

Der abstrakte Datentyp Liste hat die folgenden Operationen:

```
interface List<E> {
    int         size();
    E           get(int i);
    void        set(int i, E x);
    void        add(E x);
    void        remove(int i);
    boolean     contains(E x);
    Iterator<E> iterator();
}
```

`E` ist ein **Typparameter** der Schnittstelle (Klasse) `List`. Eine Liste kann jetzt mit unterschiedlichen **konkreten** Datentypen implementiert werden.

Von einer Liste erwarten wir, dass die Elemente in der Reihenfolge ihres Einfügens geordnet sind. Rufen wir also `get(i)` so soll das Element zurückgegeben werden, das als *i*-tes eingefügt wurde. Des Weiteren soll der Iterator (siehe unten) die Liste in dieser Reihenfolge abschreiten.

Wir betrachten hier zunächst die Reihungsliste, die eine Liste mit einer Reihung implementiert.

9.2 Reihungslisten

Die Idee der Reihungsliste ist es, die Listenelemente in einer Reihung abzulegen. Aus Effizienzgründen ist es sinnvoll, die Größe der Reihung nicht bei jedem Hinzufügen und Löschen von Elementen zu ändern. Daher muss man sich zusätzlich den Füllstand der Reihung merken:

```
public class ArrayList<E> implements List<E> {
    private E[] elements;
    private int size;

    @SuppressWarnings(”unchecked”)
    public ArrayList(int capacity) {
        // Java erlaubt es nicht, Reihungen von Typvariablen (new E[x]) zu
        // erzeugen. Daher der Umweg über Object[] und ein Cast in get().
```

```
        assert capacity > 0;
        this.elements = (E[]) new Object[capacity];
        this.size     = 0;
    }

    public int size() { return size; }
}
```

9.2.1 set und get

`set` setzt das *i*-te Listenelement und `get` holt das *i*-te Element aus der Liste. In beidem Fällen muss *i* kleiner als die Länge der Liste sein, was wir durch Zusicherungen dokumentieren.

```
    public E get(int i) {
        assert 0 <= i && i < size;
        return elements[i];
    }

    public void set(int i, E x) {
        assert 0 <= i && i < size;
        elements[i] = x;
    }
```

9.2.2 add

Beim Hinzufügen muss die Reihung verlängert werden, wenn `size == elements.length`. Die Länge einer Reihung ist aber nicht änderbar. Daher muss in diesem Fall eine neue Reihung erzeugt werden, und der Inhalt der alten in die neue kopiert werden. Dies geschieht durch die (statische) Methode `copyOf` der Klasse `Arrays`. Sie erstellt eine Kopie der übergebenen Reihung, jedoch mit einer frei wählbaren Länge. In unserer Implementierung verdoppeln wir die Länge der Reihung.

```
    public void add(E x) {
        int l = elements.length;
        if (size == l) {
            assert l > 0;
            elements = Arrays.copyOf(elements, 2 * l);
        }
        assert size < elements.length;
        elements[size++] = x;
    }
```

Beachten Sie die beiden Zusicherungen im Code. Die erste dokumentiert (und überprüft), dass die Länge der Reihung tatsächlich größer 0 ist. Dies ist wichtig, da `2 * l` die Reihung nicht verlängern würde, wenn `l == 0`. Somit ist `elements.length > 0` eine **Invariante** unserer Klasse. Der Konstruktor sichert zu, dass niemals ein Objekt der Klasse `ArrayList` konstruiert wird, das diese Invariante verletzt.

Die zweite Zusicherung dokumentiert, dass der folgende Code erfordert, dass `size < elements.length`. Obwohl sich das am Code darüber

ablesen lässt, ist es sinnvoll diese Zusicherung einzufügen, da sie explizit macht, worauf sich die folgenden Anweisungen verlassen. Dies ist hilfreich, wenn der Code darüber (vielleicht von jemandem, der den Code bislang nicht kennt) modifiziert wird. Die Zusicherung ist hier prägnanter, wie der Code der folgt.

Zum Abschluss wollen wir noch untersuchen, warum es sinnvoll ist, die Größe der Reihung zu verdoppeln. Jedesmal, wenn die Reihung voll ist, müssen alle Elemente umkopiert werden. Nehmen wir an, die Reihung ist zu Beginn ein Element lang und nehmen wir weiterhin an, die finale Länge der Reihung ist $N = 2^n$ für $n > 0$. Wieviele Kopien mussten durchgeführt werden? Hierzu bemerken wir, dass bei der Erweiterung der Reihung von 2^i auf 2^{i+1} Elemente, 2^i Elemente kopiert werden müssen. Also werden insgesamt

$$\sum_{i=0}^{n-1} 2^i \overset{\text{Lemma 1.2}}{=} 2^n - 1 = N - 1$$

Kopien durchgeführt, bis die Reihung die Größe N erreicht hat. Da für die Anzahl der Elemente s in der Reihungsliste $N/2 < s \leq N$ gilt, hängt die Anzahl der durchgeführten Kopien **linear** von der Anzahl der eingefügten Elemente ab.

Im schlimmsten Fall, wenn $s = N/2 + 1$ und somit $N/2 - 1$ Elemente ungenutzt bleiben, werden $N - 1 = 2s - 3$ Kopien durchgeführt.

9.2.3 remove

Beim Löschen muss nun eine eventuell entstehende Lücke geschlossen werden. Dies geschieht durch das Unterprogramm `System.arraycopy`, die hier den Teil der Reihung hinter dem Index i um eine Position nach vorne kopiert. Danach ist die letzte Stelle, die zuvor belegt war, frei geworden. Diese wird mit **`null`** überschrieben, um die Referenz auf das Objekt zu löschen. Dies ist wichtig, da Objekte von der automatischen Speicherbereinigung nur dann beseitigt werden, wenn keine Referenzen mehr auf sie existieren. Würden wir den freigewordenen Eintrag nicht überschreiben, bliebe unter Umständen ein Objekt von der Liste referenziert, dass gar nicht mehr in ihr enthalten ist.

```
public void remove(int i) {
    assert 0 <= i && i < size;
    int restLen = size - (i + 1);
    System.arraycopy(elements, i + 1, elements, i, restLen);
    elements[--size] = null;
}
```

9.2.4 contains

Um zu testen, ob ein Element in einer Liste enthalten ist, müssen wir die Liste abschreiten und testen, ob das Element an der aktuellen Position dem übergebenen gleicht. Hier ist zu beachten, das wir daran interessiert sind **gleiche** und nicht **identische** Objekte zu finden:

```
public boolean contains(E x) {
```

```
        for (int i = 0; i < size; i++) {
            if (elements[i].equals(x))
                return true;
        }
        return false;
    }
```

9.2.5 Iteration

Eine wichtige Funktionalität von Datenstrukturen ist das Abschreiten aller enthaltenen Elemente. Hierbei muss sich Zustand gemerkt werden. Bei der Implementierung der Methode `contains` oben ist das beispielsweise die Zählvariable `i`, die die aktuelle Position innerhalb der Reihung speichert. Die Art dieses Zustands hängt aber direkt von der Implementierung der Datenstruktur ab. Daher ist ein Iterator auch ein abstrakter Datentyp:

```
interface Iterator<E> {
  // Gibt aktuelles Element zurück und rückt den Iterator eine Position weiter
  E next();

  // Testet, ob der Iterator schon am Ende angekommen ist
  boolean hasNext();
}
```

Eine Liste verfügt über eine Methode, die einen neuen Iterator auf ihr erzeugt. Dieser steht zu Beginn auf dem ersten Element der Liste. Typisch ist das folgende Muster zum iterieren über eine Liste:

```
List<E> l;
...
for (Iterator<E> i = l.iterator(); i.hasNext(); ) {
  E e = i.next();
  doSomethingWith(e);
}
```

Dies kommt so häufig vor, dass Java eine spezielle Syntax für **`for`**-Schleifen anbietet, die das immergleiche und umständliche Aufrufen der Iterator-Methoden vor dem Programmierer verbirgt.

```
List<E> l;
...
for (E e : l) {
  doSomethingWith(e);
}
```

Intern entspricht dies obigem Code, sprich es wird ein neues Iterator-Objekt erzeugt und mittels `hasNext`/`next` iteriert.

9.2.6 Der Iterator der Reihungslisten

Ein Iterator für die Reihungsliste ist ein Objekt, das ein **`int`**-Feld hat, in dem die aktuelle Position abgelegt wird.

```
public class ArrayList<E> {
```

```
  ...
  private class It implements Iterator<E> {
    private int i = 0;
    public boolean hasNext() { return i < size; }
    public E next() { return elements[i++]; }
  }

  public Iterator<E> iterator() { return new It(); }
}
```

9.3 Mengen

Ein weiterer wichtiger abstrakter Datentyp ist die **Menge**. Die Menge unterscheidet sich zur Liste in folgenden Punkten:

- Die enthaltenen Elemente haben keine Reihenfolge. Folglich gibt es die Operationen `get` und `set` nicht.
- Ein Element kann nur einmal in einer Menge enthalten sein. Folglich wird es bei der Iteration nur einmal besucht. Das Einfügen eines Objekts, das bereits in der Menge enthalten ist, hat keinen Effekt.

```
interface Set<E> {
    int size();
    boolean add(E e);
    boolean remove(E e);
    boolean contains(E e);
    Iterator<E> iterator();
}
```

9.3.1 Indexleisten und Bitvektoren

Die schnellste Implementierung des abstrakten Datentyps `Set<E>` erreicht man über eine **Indexleiste**. Hierzu geht man jedoch davon aus, dass man jedes Objekt der Klasse `E` **injektiv** auf einen **`int`** abgebildet werden kann. Sei für ein Objekt o `:` `E` diese Zahl gegeben durch $h(o)$. Kennt man das größte $h(\cdot)$, das in einer Menge vorkommt, so kann eine Reihung vom Typ **`boolean`**`[]` anlegen, die an Stelle $h(o)$ **`true`** ist, wenn o in der Menge enthalten sein soll und dort `falsch` ist, wenn o nicht in der Menge ist.

Da ein **`boolean`** meist mehr Speicher verbraucht als ein Bit, verwendet man in der Praxis Bitvektoren. Hierbei legt man eine Reihung von Wörtern der Breite $w = 2^p$ Bits[1] an und signalisiert durch Setzen des n-ten Bits, dass das Element o mit $n = h(o)$ in der Menge enthalten ist. Hierzu berechnet man den Index i des Wortes in dem sich das n-te Bit befindet und die Nummer des Bits b innerhalb des Worts. Da $w = 2^p$, kann man beide Berechnungen effizient mit Bitoperationen ausdrücken:

[1] In Java würde man hier einen **`int`** nehmen, sprich $p = 5$ und $w = 32$

$$\begin{aligned} i &:= \lfloor n/w \rfloor &&= n >> p \\ b &:= n - w \cdot i &&= n \,\&\, ((1 << p) - 1) \end{aligned}$$

Mit einer Bitmaske $m := 2^b = 1 << b$ kann man nun das b-te Bit im i-ten Wort …

setzen: `words[i] |= m`
löschen: `words[i] &= ~m`
auslesen: `isSet = (words[i] & m) != 0`

In Java umgesetzt sieht das wie folgt aus:

```
public class BitSet {
```

```
    private int[] words;
    private static final int p = 5;
    private static final int w = 1 << p; // Ein int hat 2^5 Bits

    /* ... Konstruktor und andere Methonden */

    public boolean contains(int h) {
        int i = h >> p;
        int b = h & (w - 1);
        int m = 1 << b;
        return (words[i] & m) != 0;
    }

    public boolean add(int h) {
        int i = h >> p;
        int b = h & (w - 1);
        int m = 1 << b;
        boolean res = (words[i] & m) == 0;
        words[i] |= m;
        return res;
    }
}
```

Nun ist im Allgemeinen nicht zu erwarten, dass die Werte $h(x)$ für alle $x \in S$ einer Menge S dicht in dem Intervall $[1, \ldots, |S|]$ liegen. Insbesondere kann man h nicht so konstruieren, dass zwei nicht gleiche Mengen dicht abgebildet werden. Es kommt also zwangsläufig zu Verschnitt. Beispielsweise könnten die Elemente der Menge $\{x, y\}$ auf die Indizes 5, 182736 abgebildet werden. Um diesem Verschnitt entgegenzuwirken, lässt man in der Praxis nicht-injektive Funktionen h, sogenannte **Hashfunktionen**, zu.

Hinzu kommt, dass man die Objekte einer Klasse meist nicht **injektiv** auf **`int`** abbilden kann. So gibt es beispielsweise mehr als 2^{32} verschiedene Zeichenketten. Somit kann man keine Indexleiste (oder Bitvektor) einsetzen, da zwei Objekte o_1, o_2 existieren könnten mit $h(o_1) = h(o_2)$. Man sagt auch, dass o_1 und o_2 bzgl. h **kollidieren**. Im folgenden besprechen wir mehrere Techniken um mit Kollisionen umzugehen.

9.4 Kollisionslisten

Um Kollisionen aufzulösen, erweitert man die Indexleiste zur **Hashtabelle** (deutsch auch Streutabelle). Die Einträge der Hashtabelle sind keine Wahrheitswerte (Element in Menge enthalten oder nicht), sondern verkettete Listen, die alle Objekte mit gleichem **Hashwert** $h(\cdot)$ beinhalten.

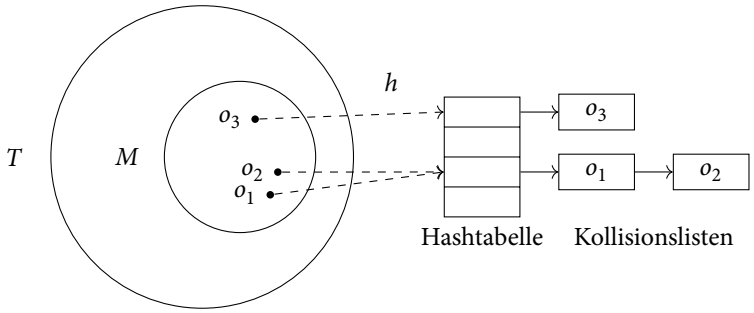


Abbildung 9.1: Kollisionsliste einer Hashtabelle. Die Hashtabelle stellt die Menge $M = \{o_1, o_2, o_3\}$ dar, die Teilmenge aller Objekte des Typs T ist.

In Java wird der Hashwert $h(o)$ eines Objektes o durch aufrufen der Methode o.`hashCode()` ermittelt. Objekte, die in Hashtabellen gehalten werden sollen, müssen diese Methode unter Umständen geeignet überschreiben. Betrachten wir die Implementierung einer Hashtabelle mit Kollisionslisten:

```
public class HashSet<E> implements Set<E> {
    private LinkedList<E>[] table;
    private int size;
    /* ... */
    public HashSet(int initCapacity) {
        this.table = (LinkedList<E>[]) new LinkedList[initCapacity];
        this.size  = 0;
    }

    private int index(E e) {
        return e.hashCode() % table.length;
    }

    public boolean add(E obj) {
        int idx = index(obj);
        if (table[idx] == null)
            table[idx] = new LinkedList<E>();
        else if (table[idx].contains(obj))
            return false;
        table[idx].add(obj);
        size++;
        return true;
    }
}
```

`remove` und `contains` gehen analog.

Man sieht hier, warum es wichtig ist, dass der Vertrag (siehe Abschnitt 8.7.6) zwischen `equals` und `hashCode` eingehalten wird. Ein Objekt, das gleich einem anderen ist, das in der Hashtabelle enthalten ist, aber einen anderen Hashwert hat, wird von `add` schlicht nicht gefunden und erneut eingefügt. Somit ist die Invariante, dass eine Menge ein Element nur einmal enthält, verletzt.

9.4.1 Lastfaktor

Nehmen wir an, wir möchten n Objekte in eine Hashtabelle aufnehmen, die m lang ist. Wenn wir weiter annehmen, dass die Hashfunktion $h(o)$ % m die Daten gleichmäßig über die Tabelle verteilt hat[2], dann hat jede Kollisionsliste $\alpha = n/m$ Einträge. α heißt der **Lastfaktor** der Hashtabelle. Die Operationen Einfügen, Löschen und Test auf Enthaltensein können also in $O(1 + \alpha)$ durchgeführt werden[3]. In der Praxis ist man daran interessiert, dass $\alpha < 1$ bleibt, um die Effizienz dieser Operationen zu gewährleisten. Daher muss, wenn der Lastfaktor zu groß wird, eine neue, größere Hashtabelle angelegt werden, in die alle Elemente neu eingefügt werden. Die Implementierung der Java-Standardbibliothek verdoppelt die Größe der Hashtabelle wenn α den Wert 0,75 übersteigt.

[2] Dies ist eine wünschenswerte Eigenschaft der Hashfunktion, die in der Praxis jedoch meist nicht zutrifft, bzw. gezeigt werden kann. Im Allgemeinen hängt es von den konkreten Daten ab, die in die Tabelle eingetragen werden sollen, ob eine Hashfunktion diese Eigenschaft erfüllt. Wir diskutieren dies in Abschnitt 9.6 genauer.

[3] Der Summand 1 kommt daher, dass wir auf jeden Fall die Hashfunktion auswerten müssen.

```
public class HashSet<E> implements Set<E> {
    private LinkedList<E>[] table;
    private int size;
    private double loadFactor;

    public HashSet(int initCapacity, double loadFactor) {
        this.table = (LinkedList<E>[]) new LinkedList[initCapacity];
        this.size  = 0;
        this.loadFactor = loadFactor;
    }

    public boolean add(E obj) {
        int idx = index(obj);
        if (table[idx] == null)
            table[idx] = new LinkedList<E>();
        else if (table[idx].contains(obj))
            return false;
        if ((double) size / table.length > loadFactor)
            rehashTable();
        idx = index(obj);
        table[idx].add(obj);
        size++;
        return true;
    }
    /* ... */
}
```

9.4.2 Hashtabellen und veränderliche Objekte

Ein Problem tritt auf, wenn der Hashwert eines Objektes von veränderlichen Feldern des Objektes abhängt. Betrachten wir folgende Klasse, die Brüche repräsentiert:

```
public class Fraction {
    private int z, n;
    // Konstruktoren, Methoden, etc.

    public int hashCode() {
        return (z + n) * (z + n + 1) / 2 + n;
    }
}
```

Der Bruch 2/7 hat somit den Hashwert 52. Wird ein Objekt, das den Bruch 2/7 darstellt, nun in eine Hashtabelle eingetragen (die mehr als 52 Einträge hat), so wird das Objekt in die Kollisionsliste am Eintrag 52 angehängt. Ändert man nun den Wert der Felder des Objekts, zum Beispiel auf 2/9, so ist das Objekt nicht mehr in der richtigen Kollisionsliste eingehängt. Fügt man nun ein Objekt ein, das gleich 2/9 ist, so wird das bereits vorhandene nicht gefunden, da es in der falschen Kollisionsliste hängt. Es empfiehlt sich also, den Hashwert eines Objektes nur aus unveränderlichen Feldern zu berechnen.

9.5 Sondieren

Kollisionslisten haben den Nachteil, dass man für die Listeneinträge extra Speicher anfordern und freigeben muss. Insbesondere wenn die Tabelle vergrößert werden soll, müssen alle Listeneinträge zerstört, und neue konstruiert werden. In diesem Abschnitt besprechen wir daher das **Sondieren (engl. open adressing, probing)**, um Kollisionen ohne Kollisionslisten aufzulösen. Im linearen und quadratischen Sondieren verwendet man die Hashtabelle selbst als Kollisionsliste. Hierzu definiert man sich eine neue Hashfunktion

$$h' : T \times \mathbb{N} \to \{0, \ldots, m-1\},$$

die zusätzlich zum Schlüssel noch einen Sondierungsindex bekommt. Beim Einfügen (und Nachschlagen) eines Schlüssels x wird zunächst geprüft, ob der Platz $h'(x, 0)$ den Schlüssel x enthält. Ist der Platz leer, so kann x dort eingefügt werden, befindet sich dort ein anderer Schlüssel, so wird $h'(x, 1)$ geprüft, und so weiter:

```
public abstract class ProbingHashTable<T> {
    private T[] table;

    // The exception that is thrown if search runs into a cycle
    private static class AllOccupied extends Exception { }

    // The probing hash function h'
    protected abstract int hash(T elm, int i);

    private int search(T elm) throws AllOccupied {
        for (int i = 0; i < table.length; i++) {
            int h = hash(elm, i);
            if (table[h] == null || table[h].equals(elm))
                return h;
        }
        throw new AllOccupied();
    }
}
```

Die Methode `search` gibt den Index in der Tabelle zurück, an dem ein `elm` gleichendes Objekt steht, falls die Tabelle ein solches enthält. Findet es in der Sequenz $h'(x, 0), h'(x, 1), \ldots$ eine Stelle, an der die Tabelle nicht belegt ist (`table[h] == null`), so liefert sie diesen Index zurück, da hier das Objekt eingefügt werden kann. Letztlich löst `search` eine Ausnahme aus, wenn alle Einträge $h'(x, 0), h'(x, 1), \ldots$ mit Objekten belegt sind, die ungleich `elm` sind. In diesem Fall kann das Objekt nicht mehr eingefügt werden, da alle möglichen Plätze in der Hashtabelle bereits belegt sind. Die spannende Frage ist nun, ob die Hashtabelle in diesem Fall tatsächlich voll ist. Dies hängt davon ab, wie die Hashfunktion h' das Sondieren realisiert.

9.5.1 Lineares Sondieren

Ein einfacher, erster Ansatz nennt sich **lineares Sondieren (engl. linear probing)**:

$$h'(x,i) := \big(h(x) + i\big) \,\%\, m$$

Er gewährleistet, dass alle Zellen der Hashtabelle als Platz für den Schlüssel x in Betracht gezogen werden können. Sprich, hat die Hashtabelle noch einen freien Platz, so kann auch noch ein Schlüssel eingefügt werden. Allerdings bilden sich beim linearen Sondieren schnell Häufungen (engl. cluster) (Abbildung 9.3) von aufeinanderfolgenden Einträgen, was die Leistungsfähigkeit beeinträchtigen kann. Je größer diese Häufungen sind, desto weiter liegt ein zu suchendes Element e von $h'(e,0)$ weg und die Laufzeit für das Suchen steigt.

9.5.2 Quadratisches Sondieren

Quadratisches Sondieren versucht die Bildung von Häufungen zu vermeiden. Hierzu wendet man eine quadratische Funktion auf den Sondierungsindex an:

$$h'(x,i) := \big(h(x) + ci + di^2\big) \,\%\, m$$

Beim quadratischen Sondieren stellt sich aber nun die Frage, ob h' auf der Menge der Tabelleneinträge $\{0,\dots,m-1\}$ **surjektiv** ist. Ist sie es nicht, kann ein Schlüssel womöglich nicht in die Tabelle eingefügt werden, obwohl die Tabelle noch leere Einträge hat. Durch geschickte Wahl von c, d und m kann man h' jedoch surjektiv machen. Hierzu nutzen wir aus, dass eine injektive Funktion von A nach A auch surjektiv ist und zeigen folgenden Satz:

Satz 9.1. *$h'_a(i) := (a + i(i+1)/2) \,\%\, 2^k$ ist injektiv auf $\{0,\dots,2^k-1\}$.*

Beweis. Beweis durch Widerspruch. Nehmen wir an, $h'_a(i) \equiv h'_a(j) \mod 2^k$ für zwei $i \neq j \in \{0,\dots,2^k-1\}$. Dann ist:

$$h'_a(i) - h'_a(j) \equiv 0 \mod 2^k$$

Es existiert also ein $q \in \mathbb{Z}$, so dass

$$\begin{aligned} & h'_a(i) - h'_a(j) &&= q \cdot 2^k \\ \iff\ & i(i+1) - j(j+1) &&= q \cdot 2^{k+1} \\ \iff\ & (i+j+1)(i-j) &&= q \cdot 2^{k+1} \end{aligned}$$

Zunächst stellt man fest, dass für $i \neq j$, das Produkt $(i+j+1)(i-j)$ auch ungleich 0 ist, sprich $q \neq 0$. Für alle $i,j \in \{0,\dots,m-1\}$ gilt, dass wenn $i-j$ gerade, dann ist $i+j+1$ ungerade und wenn $i-j$ ungerade, dann ist $i+j+1$ gerade. Das heißt, alle Vorkommen des Primfaktors 2 sind entweder in $i+j+1$ oder in $i-j$. Beide sind aber[4] kleiner 2^{k+1}. Daher kann es q nicht geben. □

[4] Nach Voraussetzung ist $0 \leq i,j < m-1$

Beispiel 9.1. Im Folgenden ein Vergleich zwischen Kollisionslisten, linearen und quadratischem Sondieren. Wir möchten folgende Strings hashen:

`Hash` `Ausnahme` `Heap` `Hoare` `Binär`

Hierzu verwenden wir eine Hashfunktion, die jede Zeichenkette auf die Position ihres ersten Buchstabens im Alphabet abbildet. Folgende Abbildungen zeigen die Hashtabllen mit den verschiedenen Verfahren zur Kollisionsauflösung.

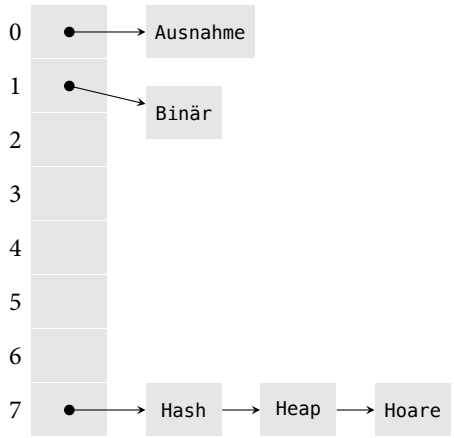


Abbildung 9.2: Hashing mit Kollisionslisten.

0	Ausnahme
1	Heap
2	Hoare
3	Binär
4	
5	
6	
7	Hash

Abbildung 9.3: Hashing durch lineares Sondieren.

0	Ausnahme
1	Binär
2	Heap
3	
4	
5	Hoare
6	
7	Hash

Abbildung 9.4: Hashing durch quadratisches Sondieren.

9.5.3 Löschen

Beim Löschen eines Eintrags in einer Sondierungstabelle tritt ein interessantes Problem auf. Betrachten wir das Beispiel in Abbildung 9.3. Nehmen wir an, wir wollen den Eintrag `Heap` aus der Tabelle wieder entfernen (nachdem wir alle Einträge eingefügt haben). Entfernen wir `Heap` einfach aus der Tabelle, entsteht eine Lücke in der Sondierfolge für den Eintrag `Hoare`: Suchen wir nach dem Entfernen von `Heap` nach `Hoare`, werden wir also nicht fündig, da die Suche nach `Ausnahme` abbricht und meldet, dass `Hoare` nicht in der Tabelle enthalten ist. Wir lösen dieses Problem indem wir beim Löschen von `Heap` in dessen Zelle Tabelle einen **Grabstein (engl. Tombstone)** hinterlassen (siehe Abbildung 9.5).

Suchen wir nach einem Element, um zu überprüfen, ob es in der Tabelle enthalten ist, interpretieren wir den Grabstein wie eine belegte Zelle. Suchen wir jedoch beim Einfügen eines Elements nach einem freien Platz, so ignorieren wir den Grabstein und sehen die Zelle als leer an.

0	Ausnahme
1	×
2	Hoare
3	Binär
4	
5	
6	
7	Hash

Abbildung 9.5: Tabelle nach Ersetzen von `Heap` durch Grabstein.

9.6 Hashfunktionen

Hashfunktionen müssen selbstverständlich deterministisch sein. Um zu gewährleisten, dass die Mengenoperationen nur konstante Zeit benötigen, bzw. nur vom Lastfaktor abhängen, muss eine Hashfunktion die Daten gleichmäßig über die Hashtabelle streuen. Allerdings kann man diese Eigenschaft von einer **gegebenen** Funktion h nicht nachweisen, **ohne** Annahmen über die Eingabedaten zu machen.

Für eine beliebige gegebene Funktion $h : A \to B$ beschreibt die Menge $K_x := \{y \mid h(x) = h(y)\}$ alle die $|K_x| - 1$ Elemente, die mit x kollidieren. Im Falle einer Implementierung durch Kollisionslisten würden all diese Elemente in der gleichen Kollisionsliste landen. Beispielsweise kollidieren unter folgender Funktion mindestens

```
public int hash(String s) {
    int res = 0;
    for (int i = 0; i < s.length; i++)
        res += s.charAt(i);
    return res;
}
```

alle Zeichenketten, die aus den gleichen Zeichen bestehen, zum Beispiel `otto` und `toto`. Will man also eine Aussage über die Anzahl der Kollisionen für eine gegebene Hashfunktion machen, so muss man Annahmen über die Eingabedaten machen. Das ist aber im Allgemeinen nicht möglich. Um dennoch Garantien zur Kollisionsanzahl beweisen zu können, betrachten wir im Folgenden Hashing als **randomisierten Algorithmus** bei dem wir die Hashfunktion aus einer geeigneten Menge zufällig ziehen. Die Aussage, die wir zeigen werden ist, dass die Anzahl der erwarteten Kollisionen beim Einfügen von m Elementen in eine Hashtabelle der Größe m dann kleiner 1 ist. Eine Menge von Hashfunktionen, die uns diesen Schluss erlaubt, heißt **universell (engl. universal hash function)**.

Definition 9.1 (Universelle Hashfunktionen)**.** Sei H eine Menge von Funktionen $T \to \{0, \dots, m-1\}$. H heißt **universell**, wenn für jedes Paar von Schlüsseln $x \neq y \in T$ gilt:

$$\left|\left\{h \in H \mid h(x) = h(y)\right\}\right| \leq \frac{|H|}{m}$$ ⌟

Sprich, jedes Schlüsselpaar $x \neq y$ kollidiert unter maximal $1/m$ aller Hashfunktionen in H. Universelle Hashfunktionen sind deshalb interessant, da die Wahrscheinlichkeit einer Kollision von x und y unter einer zufällig aus einer universellen Menge gezogenen Hashfunktion gleich der Wahrscheinlichkeit ist, für x und y zwei Werte aus $\{0, \dots, m-1\}$ zufällig zu ziehen. Folgender Satz gibt uns nun die gewünschte Aussage über den Erwartungswert der Anzahl der Kollisionen.

Satz 9.2. *Sei $h : T \to \{0, \dots, m-1\}$ eine zufällig aus einer Menge universeller Hashfunktionen gezogene Hashfunktion. Wird eine Menge S von $|S| < m$ Schlüsseln in die Hashtabelle der Größe m eingefügt, so*

ist die Anzahl der erwarteten Kollisionen der Elemente von S mit einem gegebenen $x \in T \setminus S$ kleiner als 1.

Beweis. Für jedes Element $y \in S$ gibt die Zufallsvariable

$$K_y(h) := \begin{cases} 1 & h(y) = h(x) \\ 0 & \text{andernfalls} \end{cases}$$

an, ob y mit x kollidiert. Die Zufallsvariable

$$X(h) := \left|\{y \in S \mid h(x) = h(y)\}\right| = \sum_{y \in S} K_y(h)$$

beschreibt die Anzahl der mit x kollidierenden Schlüssel unter h. Aufgrund der Linearität des Erwartungswertes gilt:

$$\mathrm{E}\left[X(h)\right] = \mathrm{E}\left[\sum_{y \in S} K_y(h)\right] = \sum_{y \in S} \mathrm{E}\left[K_y(h)\right]$$

Da h eine universelle Hashfunktion ist (siehe Definition 9.1), ist für $y \neq x$

$$\mathrm{E}\left[K_y(h)\right] \leq 1/m$$

und somit

$$\mathrm{E}\left[X(h)\right] \leq |S|/m < 1$$ □

Im folgenden werden wir nun einige Mengen von Hashfunktionen auf ihre Universalität hin untersuchen. Die Beweise die wir dazu führen werden, laufen immer darauf hinaus, abzuschätzen, wieviele Hashfunktionen unter zwei frei gewählten Elementen $x \neq y$ den gleichen Wert haben. Alle Hashfunktionen, die wir im Folgenden untersuchen werden, sind distributiv. Daher interessieren wir uns für die Menge $\{h \mid h(z) = 0 \text{ und } z \neq 0\}$ für $z := x - y \neq 0$.

Die Hashfunktionen verwenden Multiplikation und basieren im Prinzip darauf, dass es in geeigneten Faktorringen multiplikative Inverse gibt, was das Vorhandensein von Nullteilern ausschließt, die obige Mengen zu groß werden ließen. Um das genauer zu betrachten, zitieren wir noch einige Grundlagen aus der Theorie der Ringe.

9.6.1 Ein Einschub: Ringe

Satz 9.3. *Sind k und n teilerfremd, so hat k in $\mathbb{Z}/n\mathbb{Z}$ ein multiplikatives Inverses.*

Beweis. Da k und n teilerfremd sind, gilt $\mathrm{ggT}(k, n) = 1$. Der erweiterte euklidische Algorithmus liefert $s, t \in \mathbb{Z}$, so dass

$$1 = \mathrm{ggT}(k, n) = s \cdot k + t \cdot n$$

Somit ist $s \cdot k \equiv 1 \mod n$ und das Inverse k^{-1} von k ist s. □

Korollar 9.4. *Ist p prim, so hat jedes Element in $\mathbb{Z}/p\mathbb{Z} \setminus \{0\}$ ein multiplikatives Inverses. $\mathbb{Z}/p\mathbb{Z}$ ist dann ein Körper.*

Korollar 9.5. *Ist k ungerade, so hat es im Ring $\mathbb{Z}/2^n\mathbb{Z}$ ein multiplikatives Inverses.*

Satz 9.6. *Hat ein Element $a \in R$ ein multiplikatives Inverses $a^{-1} \in R$ mit $aa^{-1} = 1$, so existiert kein **Nullteiler** $b \in R \setminus \{0\}$ mit $ab = 0$.*

Beweis. Angenommen b existiert. Dann gilt $ab = 0 \iff a^{-1}ab = a^{-1}0 \iff b = 0$, was der Wahl von b widerspricht. □

9.6.2 Eine einfache universelle Menge von Hashfunktionen

Nehmen wir an, die Länge unserer Hashtabelle ist eine Primzahl p. Nehmen wir weiter an, ein Element der Menge T der einzuhashenden Schlüssel kann in $r + 1$ Teile zerteilt werden (zum Beispiel 4 Bytes für $T =$ `int`). Wir definieren nun eine Menge von Hashfunktionen, die dadurch entsteht, dass wir $r+1$ Werte $a_0, \ldots, a_r$ zufällig aus $\mathbb{Z}/p\mathbb{Z}$ ziehen. Für eine solche Sequenz $a := a_0, \ldots, a_r$ definieren wir:

$$h_a : x_0, \ldots, x_r \mapsto \sum_{i=0}^{r} a_i x_i \mod p \tag{9.1}$$

Die Menge der Hashfunktionen ist dann definiert durch

$$H := \{h_a \mid a = a_0, \ldots, a_r \text{ mit } a_i \in \mathbb{Z}/p\mathbb{Z} \text{ für } 0 \leq i \leq r\}$$

Das „Ziehen“ einer Hashfunktion besteht somit daraus, die Werte $a_0, \ldots, a_r$ zufällig zu ziehen. Die Menge H enthält p^{r+1} Funktionen.

Satz 9.7. *H ist universell.*

Beweis. Betrachten wir zwei unterschiedliche Schlüssel $x = x_0, \ldots, x_r$ und $y = y_0, \ldots, y_r$ und nehmen an, dass ohne Beschränkung der Allgemeinheit $x_0 \neq y_0$. Wir ermitteln nun, für wieviele Werte von a gilt, dass $h_a(x) \equiv h_a(y) \mod p$. Da h_a linear ist, gilt

$$\begin{aligned} h_a(x) \equiv h_a(y) \mod p &\iff \sum_{i=0}^{r} a_i(x_i - y_i) \equiv 0 \mod p \\ &\iff -\sum_{i=1}^{r} a_i(x_i - y_i) \equiv a_0(x_0 - y_0) \mod p \end{aligned} \tag{9.2}$$

Da p prim ist, hat $(x_0 - y_0)$ nach Korollar 9.4 ein multiplikatives Inverses. Somit gibt es für jedes $a_1, \ldots, a_r$ genau ein a_0 für das (9.2) gilt. Da es p^r verschiedene $a_1, \ldots, a_r$ gibt, Ist die Menge der h_a, für die $x \neq y$ kollidieren, genau $p^r = p^{r+1}/p = |H|/p$ Funktionen groß. □

Ein Nachteil der Funktion aus (9.1) ist, dass für man soviele a_i benötigt, wie der Schlüssel lang ist. Wir betrachten nun eine Abwandlung von h, in der wir nur noch einen Wert a und nicht mehr $r + 1$ Werte ziehen. Hierzu betrachten wir die Funktion

$$g_a(x_0, \ldots, x_r) = \sum_{i=0}^{r} x_i a^i \mod p \tag{9.3}$$

und untersuchen, für wieviele a zwei unterschiedliche Schlüssel $x := x_0, \ldots, x_r$ und $y := y_0, \ldots, y_r$ kollidieren. Sei wieder $z := x - y \neq 0$. Der Grad des Polynoms g_a ist r und, da $\mathbb{Z}/p\mathbb{Z}$ ein Körper ist, hat g_a höchstens r Nullstellen. Das heißt, dass die Anzahl der Funktionen, unter denen x und y kollidieren kleiner gleich r ist. Allerdings gibt es nur p Funktionen und der Term $|H|/m$ aus Definition 9.1 ist hier 1. Somit ist g_a für $r > 1$ **nicht** universell.

Ein mögliches Vorgehen ist jedoch, p sehr groß zu wählen und dann den Wert g_a durch eine weitere Hashfunktion auf den Bereich der Tabelle abzubilden. Für $r \ll p$ ist die Wahrscheinlichkeit, ein a zu ziehen, unter dem x und y kollidieren r/p, also nahe 0.

9.6.3 Eine fast universelle Hashfunktion

Die Hashfunktion h_a hat den Nachteil, dass die Größe der Tabelle eine Primzahl sein muss und wir modulo dieser Primzahl rechnen müssen. Die Division ist im Vergleich zu anderen arithmetischen und Bitoperationen allerdings sehr langsam, weswegen man in der Praxis Divisionen durch nicht-Zweierpotenzen gerne vermeidet.

Im folgenden besprechen wir eine Hashfunktion, die beinahe universell ist und effizient durch eine Multiplikation und einen Shift berechnet werden kann.

$$\begin{array}{rcll} f_a : \{0, \ldots, 2^w - 1\} & \to & \{0, \ldots, 2^k - 1\} & \\ x & \mapsto & (a \cdot x \,\%\, 2^w) \,/\, 2^{w-k} & a \text{ ungerade} \end{array} \tag{9.4}$$

Das Produkt zweier w Bit langen Zahlen ist $2w$ Bit lang. Die Division durch 2^{w-k} extrahiert die obersten k Bits der unteren w Bits des Produkts und wird effizient durch einen Rechtsshift um $w - k$ implementiert, wie die folgende Abbildung illustriert:

Typischerweise wählt man für w die Wortbreite des Prozessors. Dann rechnet der Prozessor implizit modulo 2^w, da die unteren w Bit des Produkts typischerweise in einem Register abgelegt werden.

Wir definieren die für gegebene $0 < k \leq w$ die Menge von Hashfunktionen

$$H_{w,k} := \{f_a \mid 0 < a < 2^w \text{ ungerade}\}$$

Wir werden nun zeigen, dass es höchstens

$$\frac{2 \cdot |H_{w,k}|}{2^k} = \frac{2 \cdot 2^{w-1}}{2^k} = 2^{w-k}$$

Werte für a gibt, für die zwei ungleiche x und y kollidieren. Somit ist die Menge $H_{w,k}$ „fast“ universell.

Satz 9.8. *Seien $1 \leq k \leq w \in \mathbb{N}$. Für alle $x \neq y \in \{0, \ldots, 2^w - 1\}$ gilt: Es gibt höchstens 2^{w-k} Werte für a für die $f_a(x) = f_a(y)$.*

Beweis. Seien $x, y \in \{0, \ldots, 2^w\}$ mit $x \neq y$ und $f_a(x) \equiv f_a(y) \mod 2^w$. Da nach Voraussetzung $f_a(x) \equiv f_a(y) \mod 2^w$ ist, sind die oberen k Bits von $ax \,\%\, 2^w$ und $ay \,\%\, 2^w$ gleich. Zieht man nun $ay \,\%\, 2^w$ von $ax \,\%\, 2^w$ ab, so erhält man eine Zahl $c = a(x - y) \,\%\, 2^w$, bei der die oberen k Bits entweder alle 0 (wenn $x \geq y$) oder 1 sind (je nachdem ob x größer ist als y). Sei s die Position des niederwertigsten gesetzten Bits in $x - y$. Die Bitdarstellung von c sieht nun wie folgt aus:

$$\underbrace{b \cdots b}_{k} \cdot c_{w-k-1} \cdots c_{s+1} \cdot 1 \underbrace{0 \cdots 0}_{s}$$

Um den Satz zu beweisen, müssen wir beantworten, wieviele a eine solche Bitdarstellung produzieren. Betrachten wir die Zahl z' mit $x - y = z'2^s$, die daraus resultiert, dass man $x - y$ um s nach rechts shiftet. Diese Zahl ist ungerade, da s gerade die Position des niederwertigsten gesetzten Bits in $x - y$ ist. Nach Korollar 9.5 ist somit $az' \neq 0$. Daraus folgt auch, dass die Funktion $a \mapsto az' \,\%\, 2^w$ injektiv ist. Die Bits $c_{w-k-1} \cdots c_s$ sind gerade die unteren $w-k-s$ Bits von $az' \,\%\, 2^w$. Das heißt, dass es höchstens 2^{w-k-s} mögliche Werte von a gibt, die ein solches c produzieren. Ist nun $s > w - k$, so gibt es gar kein a und ist $s = 0$ so gibt es höchstens 2^{w-k} unterschiedliche Werte für a. □

Man beachte, dass Korollar 9.5 eine zentrale Rolle in dem Beweis spielt, da es die Anzahl der möglichen Werte von a auf mit der Anzahl der möglichen Zahlen c gleichsetzt. Hätte man die Eigenschaft, dass $a \mapsto az'$ injektiv ist nicht, so könnte es **noch mehr** Werte für a geben. Ein etwas formalerer Beweis findet sich in der Originalarbeit [Die+97].

9.7 Zusammenfassung und Literaturhinweise

Reihungslisten und Hash-Tabellen mit linearer Sondierung haben in der Praxis deutliche Vorteile, da die Daten dort eng beieinander liegen, was zu besserem Cache-Verhalten führt und die Speicherzugriffstechnik moderner Prozessoren (Prefetching) begünstigt.

Ein ausführliche experimentelle Studie [RAD15] untersucht verschiedene Hashing-Techniken (Kollisionslisten, Sondieren, Cuckoo-Hashing) und Hash-Funktionen. Sie zeigt, dass die Hash-Funktion aus Abschnitt 9.6.3 in der Praxis sehr schnell ist, sehr gute Ergebnisse hinsichtlich der Streuung der zu hashenden Werte liefert und somit ein guter Kompromiss aus Laufzeit und Streuung ist. Des Weiteren zeigt die Arbeit, dass lineare Sondierung verkettete Kollisionslisten schlagen, wenn der Lastfaktor unter 0,5 bleibt. Insbesondere das Robin-Hood-Hashing [CLM85], eine Erweiterung der linearen Sondierung, zeigt sich besonders leistungsfähig. Hierbei werden beim Abschreiten der Kollisionen andere Einträge nochmals neu eingetragen, mit dem Ziel, die mittlere Entfernung eines Eintrags x zu seinem eigentlichen Eintragungsort $h'(x, 0)$ zu minimieren. Dies führt zu einer leicht höheren Laufzeit beim Eintragen (aufgrund des komplexeren Codes), verkürzt aber Suchanfragen, da die Kollisionslisten im Mittel kürzer werden.

10 Ein einfacher C0-Übersetzer

In diesem Kapitel beschäftigen wir uns mit einem einfachen **Übersetzer (engl. Compiler)** von C0 nach MIPS. Dies soll zum einen Einblicke in die Funktionsweise eines einfachen Übersetzers geben, andererseits einen Eindruck davon vermitteln, wie höhere Programmiersprachen in Maschinensprache implementiert werden können. Zudem ist ein Übersetzer ein etwas größeres Softwareprojekt, bei dem wir Vererbung, wie wir sie in Abschnitt 8.7 kennengelernt haben, sinnvoll einsetzen können.

Ein Übersetzer wandelt ein Programm, gegeben als Strom von Textzeichen in ein Maschinensprach-Programm um, das die gleiche Semantik hat wie das Eingabeprogramm. Abbildung 10.1 zeigt den schematischen Aufbau des einfachen Übersetzers, den wir in diesem Kapitel besprechen. Zunächst wird das Eingabeprogramm der Syntaxanalyse

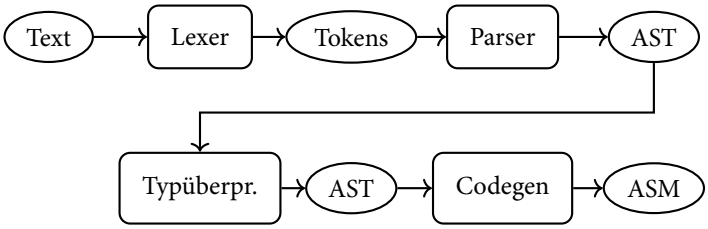


Abbildung 10.1: Schematischer Aufbau eines sehr einfachen Übersetzers. Moderne Übersetzer verwenden noch andere Programmdarstellungen außer dem AST. Des Weiteren „optimieren" sie das Programm, indem sie es (semantikerhaltend) umschreiben um bessere Laufzeit zu erzielen.

unterzogen, die feststellt, ob das Programm der **konkreten Syntax** der Programmiersprache genügt. Diese muss so definiert sein, dass jedem syntaktisch korrekten Programmtext genau ein abstrakter Syntaxbaum (AST) zugeordnet werden kann (siehe Abschnitt 5.1). Für den syntaktisch korrekten Aufbau eines Programms spielen konkrete Bezeichnernamen und die Werte von Konstanten keine Rolle. Daher wird der Programmtext vom **Lexer** in eine Folge von sogenannten Tokens unterteilt. Jedes Token besteht aus einer Kategorie (zum Beispiel „Bezeichner", „Additionsoperator", „Runde Klammer auf", etc.), dem Originaltext (wenn er nicht aus der Kategorie hervorgeht) und den Koordinaten (Zeilen- und Spaltennummer, Dateiname) wo das Token auftrat. Für die syntaktische Struktur überflüssige Merkmale wie Leerzeichen, Zeilenumbrüche und Kommentare werden vom Lexer entfernt.

Der Parser analysiert den Tokenstrom anhand der konkreten Syntax und erzeugt, wenn möglich, den abstrakten Syntaxbaum des eingelesenen Programms. Entspricht der Tokenstrom nicht der konkreten

```
q = 0;
r = x;
while (y <= r) {
    r = r - y;
    q = q + 1;
}
```

```
ID("q") ASSIGN INT_CONST("0") SEMI
ID("r") ASSIGN ID("x") SEMI
WHILE LPAREN VAR("y") LE VAR("r") RPAREN LBRACE
ID("r") ASSIGN ID("r") MINUS ID("y") SEMI
ID("q") ASSIGN ID("q") PLUS INT_CONST("1") SEMI
RBRACE
```

Abbildung 10.2: Ein Beispielprogramm und die Sequenz von Tokens (bestehend aus dem Namen der Kategorie und ggf. dem Programmtext des Tokens), die ein Lexer daraus erzeugt. Die Namen der Kategorien sind beispielhaft und frei gewählt.

Syntax der Sprache, so erzeugt der Parser eine (oder mehrere) Fehlermeldungen und beendet den Übersetzer. Wir diskutieren in diesem Rahmen keine Techniken zum Parsing und verweisen hierzu auf gängige Übersetzerbau-Lehrbücher [WSH13].

Die Typüberprüfung und Code-Generierung diskutieren wir in Abschnitt 10.1 und Abschnitt 10.2. In beiden Abschnitten definieren wir jeweils mehrere Funktionen, die Typen von Ausdrücken ableiten, überprüfen, ob Anweisungen wohlgetypt sind und Code für Ausdrücke und Anweisungen generieren. Diese Funktionen sind rekursiv über der abstrakten Syntax definiert.

Es bietet sich daher an, bei der Implementierung des abstrakten Syntaxbaumes für jede Regel der abstrakten Syntax eine Klasse, die das entsprechende syntaktische Konstrukt repräsentiert, vorzusehen. Es ist hilfreich, für jede syntaktische Kategorie (in C0 sind das *Stmt* und *Expr*) eine Schnittstelle einzuführen, von der diese Klassen dann erben. In jeder Schnittstelle vereinbart man die für die Kategorie entsprechenden Funktionen (hier Typüberprüfung, Code-Generierung).

```
public interface Statement {
    void checkType(...);
    void genCode(...);
}
```

```
public interface Expression {
    Type checkType(...);
    void genCode(...);
}
```

Es kann sinnvoll sein für bestimmte Regeln mehrere Unterklassen zu erstellen: Zum Beispiel ist die Typ-Regel für einen Ausdruck mit einem Vergleichsoperator unterschiedlich zu der eines binären Ausdrucks mit arithmetischem Operator (siehe Abschnitt 10.1). Hier ist es zum Beispiel sinnvoll, eine Klasse für Ausdrücke mit binärem Operator zu erstellen, die dann zwei Unterklassen (arithmetischer Operator, Vergleichsoperator) hat.

10.1 Typüberprüfung

Die statische Semantik von C0 wird in Abschnitt 5.6 diskutiert. Sie ist deterministisch, was bedeutet, dass für jede Typumgebung und jeden Ausdruck genau ein oder kein Typ abgeleitet werden kann (→ Aufgabe 5.17). Aus der induktiven Definition der statischen Semantik kann man somit ein Programm gewinnen, das die Typen der (Teil-) Ausdrücke berechnet und feststellt, ob ein C0-Programm wohlgetypt ist. In einer konkreten Implementierung des Übersetzers[1] hat man (grosso modo) für jedes Element der abstrakten Syntax eine Klasse. Jede Klasse, die einen Ausdruck implementiert, hat dann eine Methode, die den Typ des Ausdrucks ermittelt oder einen Fehler produziert wenn der Ausdruck nicht wohlgetypt ist. Diese Methode bekommt als Argument die Typumgebung (oft Environment oder Scope genannt) in welcher der Typ einer Variablen nachgeschlagen werden kann.

[1] wir nehmen, dass wir den Übersetzer in Java implementieren

```
public interface Expression {
    Type checkType(Diagnostic d, Scope s);
    // ...
}
```

Betrachten wir beispielsweise die Implementierung eines arithmetischen binären Operators:

```
public class ArithBinary implements Expression, Locatable {
    // ...
    public Type checkType(Diagnostic d, Scope s) {
        Type l = getLeft().checkType(d, s);
        Type r = getRight().checkType(d, s);
        if (!l.isIntType())
            d.printError(this, "...");
        if (!r.isIntType())
            d.printError(this, "...");
        return Types.getIntType();
    }
}
```

Die Typumgebung wird durch eine Klasse `Scope` implementiert, die alle aktuell sichtbaren Bezeichner auf den Knoten des abstrakten Syntaxbaums abbildet, der ihre Vereinbarung beschreibt. Da Blöcke geschachtelt, und Vereinbarungen einander verdecken können, wird die Typumgebung als Stapel von Abbildungen realisiert. Betritt man während der Typüberprüfung einen Block, wird eine neue Abbildung (ein Objekt der Klasse `Scope`) auf den Stapel gelegt. Beim Verlassen des Blocks verwirft man die für diesen Block erzeugte Tabelle, da die dort vereinbarten lokalen Variablen außerhalb des Blockes nicht sichtbar sind:

```
public class Block implements Statement, Locatable {
    private final Statement body;

    public void checkType(Diagnostic d, Scope parent) {
        Scope scope = parent.newNestedScope();
        body.checkType(d, scope);
```

```
    }
}
```

Realisiert wird der Stapel in der Klasse `Scope` durch einen Verweis auf eine „Elterntabelle“, die die Typumgebung des umgebenden Blockes ist. Beim Nachschlagen eines Bezeichners muss in dem Fall, dass der Bezeichner im aktuellen Block nicht vereinbart ist, der umgebende Block gefragt werden. Dieser muss eventuell seinen umgebenden fragen, und so weiter. Beim Hinzufügen einer Vereinbarung muss überprüft werden, ob ein entsprechender Bezeichner schon vereinbart wurde; die erneute Vereinbarung des gleichen Bezeichners im selben Block ist in C0 verboten. Abbildung 10.3 zeigt ein C0-Programm und die Typumgebungs-Stapel, die bei der Typüberprüfung entstehen.

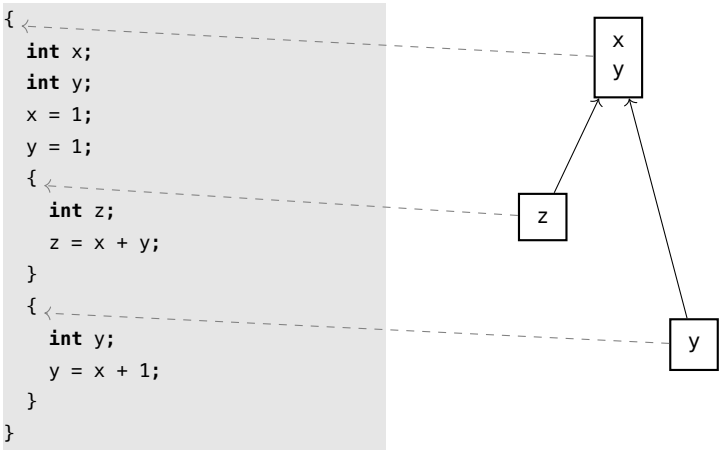


Abbildung 10.3: Ein C0-Programm mit mehreren Blöcken. Für jeden Block gibt es eine Typumgebung (Scope). Die Zugehörigkeit ist durch die gestrichelte Kante dargestellt. Die Schachtelung der Blöcke induziert einen **Baum** von Typumgebungen. Die Typumgebung eines inneren Blockes führt nur die in dem Block vereinbarten Variablen auf und verweist auf den äußeren, umschließenden Block. Sucht man die Vereinbarung zu einem verwendenden Auftreten, so schlägt man den Variablennamen zunächst in der Typumgebung des innersten Blockes nach, in dem sich das Auftreten befindet. Ist die Variable dort nicht vereinbart, werden alle Typumgebungen bis zur Äußersten konsultiert.

Folgender Code zeigt die Implementierung der Typumgebung eines Blocks. Besucht man bei der Typüberprüfung den AST-Knoten, der das Auftreten einer Variable darstellt, so empfiehlt es sich, in diesem Knoten einen Verweis auf die entsprechende Vereinbarung der auftretenden Variable zu speichern, da dieser AST-Knoten die Variable eindeutig identifiziert.[2] Dies ist sinnvoll, da man alle für diese Variable relevanten Informationen dort findet, bzw. dort ablegt. Bei der Typüberprüfung ist nur der Typ einer Variable von Interesse. Bei der Code-Erzeugung, die wir im folgenden Kapitel besprechen, werden wir noch weitere Informationen für die Variable berechnen und dort ablegen. Die Implementierung der Typüberprüfung für den AST-Knoten, der das Auftreten einer Variable darstellt, könnte also wie folgt aussehen:

[2] der Name identifiziert die Variable nicht eindeutig, da in unterschiedlichen Blöcken zwei Variablen den gleichen Namen haben können, siehe Abbildung 10.3.

```
public class Scope {
    private final Map<Identifier, Declaration> table;
    private final Scope parent;

    public Scope() {
        this(null);
    }

    private Scope(Scope parent) {
        this.parent = parent;
        this.table  = new HashMap<Identifier, Declaration>();
    }

    public Scope newNestedScope() {
        return new Scope(this);
    }

    public void add(Identifier id, Declaration d)
        throws IdAlreadyExists {
        if (table.contains(id))
            throw new IdAlreadyExists(id);
        table.put(id, d);
    }

    public Declaration lookup(Identifier id) throws IdUndeclared {
        // ...
    }
}
```

```
public class Var implements Expression, Locatable {
    private Identifier id;
    private Declaration decl = null;

    public Type checkType(Diagnostic d, Scope s) {
        try {
            this.decl = s.lookup(id);
            return this.decl.getType();
        }
        catch (IdUndeclared e) {
            d.printError(this, ”Variable ” + id + ” not declared”);
            return Types.getErrorType();
        }
    }
}
```

10.2 Syntaxgesteuerte Code-Erzeugung

Eine einfache Art und Weise ein C0-Programm in Maschinencode zu übersetzen, ist die syntaxgesteuerte Übersetzung. Hierbei erzeugt man den Code jedes Knoten des abstrakten Syntaxbaums rekursiv. Sprich man erzeugt erst Code für die Kinder eines Knotens und baut den Code des Knotens aus dem Code der Kinder zusammen. Diese Technik ist einfach zu implementieren, liefert aber im allgemeinen keinen besonders guten (im Sinne von schnell) Code. Um „guten" Code zu erzeugen, braucht man globale Techniken, die dann aber nicht mehr induktiv definiert werden können und somit nicht mehr syntaxgesteuert sind.

Die lokalen Variablen eines Unterprogramms legen wir im Keller (stack) ab. Wir verwenden die Register nur zur Ausdrucksauswertung. Das heißt, dass die Register zwischen zwei Anweisungen keine Werte enthalten, die noch benötigt werden.

Ähnlich wie bei der statischen Semantik definieren wir den zu erzeugenden Code wieder induktiv über der abstrakten Syntax. Hierzu definieren wir in den folgenden Abschnitten drei Funktionen (induktiv über der abstrakten Syntax): `codeS` produziert Code für eine Anweisung, `codeR` Code, der einen Ausdruck R-auswertet und `codeL` Code, der einen Ausdruck L-auswertet.

10.2.1 Anweisungen

Betrachten wir zunächst die Funktion `codeS`, die den Code für Anweisungen erzeugt:

$$\texttt{codeS} : (\mathit{Id} \rightharpoonup \mathbb{N}) \times \mathbb{N} \times \mathit{Stmt} \to \mathit{Code} \times \mathbb{N}$$

Die Parameter von `codeS` sind:

- Eine Abbildung, die Bezeichnern Adressen in der Aufrufschachtel (engl. stack frame) zuordnet.
- Die bisherige Größe der Aufrufschachtel.
- Eine Anweisung oder Anweisungsliste.

`codeS` liefert dann den Code und die maximale Größe der Aufrufschachtel, die für den Code benötigt wird. Wir definieren die Funktion `nun` induktiv über der abstrakten Syntax von C0:

- Der Code der While- und If-Anweisung setzt sich aus der Code-Sequenz des/der Rumpfes/Rümpfe und der Code-Sequenz der Bedingung zusammen. Wir verwenden hier eine Funktion `codeR`, die Code erzeugt, der einen Ausdruck R-auswertet. Dieser Funktion übergeben wir eine Liste von Registern, die sie für den Code verwenden kann. Der Code soll so gebaut sein, dass das Ergebnis der Ausdrucksauswertung im ersten Register dieser Liste liegt. Die Zahl n, die zur Benennung der Sprungmarken verwendet wird, soll bei jeder Regelinstantiierung anders sein, um zu vermeiden, dass zwei Sprungmarken

unterschiedlicher Schleifen den gleichen Namen haben.

$$\text{While}\ \frac{\texttt{codeR}\ \Omega\ (r :: rs)\ e \blacktriangleright c_1 \qquad \texttt{codeS}\ \Omega\ \delta\ (s) \blacktriangleright c_2, S \qquad n \text{ frisch}}{\texttt{codeS}\ \Omega\ \delta\ (\textbf{while}\ (e)\ s) \blacktriangleright \begin{array}{l}\texttt{b}\quad \texttt{T}n \\ \texttt{L}n\texttt{:} \\ \quad c_2 \\ \texttt{T}n\texttt{:} \\ \quad c_1 \\ \quad \texttt{bnez}\ r\ \texttt{L}n\end{array}, \delta}$$

$$\text{If}\ \frac{\begin{array}{c}\texttt{codeR}\ \Omega\ (\texttt{\$v0} :: rs)\ e \blacktriangleright c \\ \texttt{codeS}\ \Omega\ \delta\ (s_1) \blacktriangleright c_1, S_1 \qquad \texttt{codeS}\ \Omega\ \delta\ (s_2) \blacktriangleright c_2, S_2\end{array} \qquad n \text{ frisch}}{\texttt{codeS}\ \Omega\ \delta\ (\textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2) \blacktriangleright \begin{array}{l} c \\ \texttt{beqz \$v0 F}n \\ \quad c_1 \\ \quad \texttt{b}\quad \texttt{N}n \\ \texttt{F}n\texttt{:} \\ \quad c_2 \\ \texttt{N}n\texttt{:}\end{array}, \max(S_1, S_2)}$$

- Für die Zuweisung l `=` x`;` muss die linke L- und die rechte Seite R-ausgewertet werden. Danach muss das Ergebnis der Ausdrucksauswertung in die Speicherzelle geschrieben werden, deren Adresse die Auswertung der linken Seite ergeben hat.

$$\text{Assign}\ \frac{\begin{array}{c}\texttt{codeL}\ \Omega\ (r_1 :: r_2 :: rs)\ (l : k_1) \blacktriangleright c_1 \\ \texttt{codeR}\ \Omega\ (r_2 :: rs)\ (e : k_2) \blacktriangleright c_2 \\ S = \text{Store, der Wert vom Typ } k_1 \text{ speichert}\end{array}}{\texttt{codeS}\ \Omega\ \delta\ (l\ \texttt{=}\ r\texttt{;}) \blacktriangleright \begin{array}{l} c_1 \\ c_2 \\ S\quad r_2\ \ (r_1)\end{array}, \delta}$$

- Der Code der Hintereinanderausführung besteht einfach daraus, den Code der beiden Anweisungen hintereinander zu schreiben.

$$\text{Seq}\ \frac{\texttt{codeS}\ \Omega\ \delta\ (s_1) \blacktriangleright c_1, S_1 \quad \texttt{codeS}\ \Omega\ \delta\ (s_2) \blacktriangleright c_2, S_2}{\texttt{codeS}\ \Omega\ \delta\ (s_1 \_ s_2) \blacktriangleright \begin{array}{l} c_1 \\ c_2\end{array}, \max(S_1, S_2)}$$

- Zuletzt betrachten wir noch den Block. Bei der Vereinbarung müssen wir der vereinbarten Variable eine Adresse innerhalb der Aufrufschachtel zuweisen. Die Abbildung weist jeder Variablen ein Offset in der Aufrufschachtel zu. Da man in C0 nur lokale Variablen der Typen **`int`**, **`char`** und $t*$ vereinbaren kann, nehmen wir vereinfachend

an, dass wir für jede vereinbarte lokale Variable pauschal 4 Bytes veranschlagen. δ bezeichnet das nächste in der Schachtel verfügbare Offset. Alle Offsets kleiner als δ sind von den Variablen der Blöcke belegt, die diesen Block umschließen. Daher werden die Offsets des geschachtelten Blocks jenseits von δ gebildet.

$$\text{Block}\ \frac{\Omega' = \Omega[x_1 \mapsto \delta, \ldots, x_n \mapsto \delta + 4(n-1)] \quad \texttt{codeS}\ \ \Omega'\ \ (\delta + 4n)\ \ s\ \blacktriangleright\ c, S}{\texttt{codeS}\ \ \Omega\ \ \delta\ \ (\{t_1 x_1; \ldots t_n x_n; s\})\ \blacktriangleright\ c, S}$$

10.2.2 Ausdrücke

Der Code für Anweisungen wird durch die beiden Funktionen `codeL` und `codeR` erzeugt. Erstere implementiert die L-Auswertung eines Ausdrucks, letztere die R-Auswertung. Nach Definition 5.2 sind in C0 nur zwei Elemente der abstrakten Syntax L-auswertbar: Die Variable und die Indirektion. Ein L-auswertbarer Ausdruck wird immer zu einer Adresse eines Behälters ausgewertet. In unserem einfachen Übersetzer resultiert jede L-Auswertung zu einer Speicheradresse.[3] Die Funktion

$$\texttt{codeR} : (Id \rightharpoonup \mathbb{N}) \times [R] \times Exp \times Ty \to Code$$

und analog auf codeL nimmt folgende Argumente:

1. Eine Funktion, die Bezeichner auf Adressen in der Aufrufschachtel abbildet.
2. Eine Liste von Registern, die zur Ausdrucksauswertung verwendet werden können.
3. Den Ausdruck selbst,
4. und den Typ des Ausdrucks, wie er von der statischen Semantik ermittelt wurde.

Wir definieren `codeL` und `codeR` nun wieder induktiv für jeden Ausdruck:

- Die R-Auswertung einer Konstante entspricht dem Laden der Konstante in das gewünschte Register (das erste in der Liste).

$$\text{Const}\ \frac{}{\texttt{codeR}\ \ \Omega\ \ (r :: rs)\ \ (c : \texttt{int})\ \blacktriangleright\ \texttt{li}\ r\ n}$$

- Die L-Auswertung einer (lokalen) Variable produziert die Adresse der Variable in der Aufrufschachtel in ein Register. Das Offset der Variable wird in der Abbildung Ω nachgeschlagen. Wir erwarten, dass auf dem Keller ausreichend Platz angefordert wurde (durch Subtraktion der Schachtelgröße vom Kellerpegel) und der Kellerpegel auf den Beginn der Schachtel zeigt. Daher bilden wir die Adressen nun mit dem Offset **relativ** zum Kellerpegel.

$$\text{Var}\ \frac{\Omega\ x = \delta}{\texttt{codeL}\ \ \Omega\ \ (r :: rs)\ \ (x : k)\ \blacktriangleright\ \texttt{addiu}\ r\ \texttt{\$sp}\ \delta}$$

[3] Im Allgemeinen ist das aber nicht so: Ein ausgeklügelterer Übersetzer könnte lokale Variablen auch in Registern halten.

- Die beiden Operatoren & und $*$ schalten zwischen L- und R-Auswertung hin und her. $*$ konvertiert sozusagen den Wert eines Zeigers in eine Adresse und & konvertiert eine Adresse in einen Wert, den man in dem Behälter eines Zeigers ablegen kann: Beide Operatoren produzieren keinen zusätzlichen Code!

$$\text{Indir}\ \frac{\texttt{codeR}\ \Omega\ (r :: rs)\ (e : k*) \blacktriangleright c}{\texttt{codeL}\ \Omega\ (r :: rs)\ (*e : k) \blacktriangleright c} \qquad \text{Addr}\ \frac{\texttt{codeL}\ \Omega\ (r :: rs)\ (l : k) \blacktriangleright c}{\texttt{codeR}\ \Omega\ (r :: rs)\ (\&l : k*) \blacktriangleright c}$$

- Ist ein Ausdruck L-auswertbar, so besteht die R-Auswertung dieses Ausdrucks aus dem Laden von der Adresse, die die L-Auswertung produziert hat. Mit der folgenden Regel müssen wir für alle L-auswertbaren Ausdrücke keine zusätzliche Regel definieren, die ihre R-Auswertung beschreibt.

$$\text{LToR}\ \frac{\begin{array}{c}\texttt{codeL}\ \Omega\ (r :: rs)\ (l : k) \blacktriangleright c \\ L = \text{Load, der Wert vom Typ } k \text{ lädt}\end{array}}{\texttt{codeR}\ \Omega\ (r :: rs)\ (l : k) \blacktriangleright \begin{array}{l} c \\ L\ \ r\ r \end{array}}$$

- Bei der Auswertung eines binären Ausdrucks muss der eine Teilausdruck vor dem anderen ausgewertet werden. Das Register, in dem der Wert des Ausdrucks liegt, kann dann nicht mehr für die Auswertung des zweiten verwendet werden. Die Funktion `codeR` ist für binäre Ausdrücke also nur für Registerlisten definiert, die mindestens zwei Register enthalten. Gehen dem Übersetzer die Register aus, muss er eine Fehlermeldung produzieren und den Übersetzungsvorgang abbrechen.

$$\text{Binary}\ \frac{\begin{array}{c}\texttt{codeR}\ \Omega\ (r_1 :: r_2 :: rs)\ (e_1 : k_1) \blacktriangleright c_1 \\ \texttt{codeR}\ \Omega\ (r_2 :: rs)\ (e_2 : k_2) \blacktriangleright c_2\end{array}}{\texttt{codeR}\ \Omega\ (r_1 :: r_2 :: rs)\ (e_1\, o\, e_2) : k_1 \blacktriangleright \begin{array}{l} c_1 \\ c_2 \\ a\ \ r_1\ r_1\ r_2 \end{array}}$$

10.2.3 Nichtdeterminismus bei der Code-Erzeugung binärer Ausdrücke

Bei binären Ausdrücken hat man die Wahl ob man den linken oder rechten Ausdruck zuerst auswertet. Sprich, man kann folgende Regel hinzufügen.

$$\text{Binary}'\ \frac{\begin{array}{c}\texttt{codeR}\ \Omega\ (r_2 :: rs)\ (e_1 : k_1) \blacktriangleright c_1 \\ \texttt{codeR}\ \Omega\ (r_1 :: r_2 :: rs)\ (e_2 : k_2) \blacktriangleright c_2\end{array}}{\texttt{codeR}\ \Omega\ (r_1 :: r_2 :: rs)\ (e_1\, o\, e_2) : k_1 \blacktriangleright \begin{array}{l} c_2 \\ c_1 \\ a\ \ r_1\ r_1\ r_2 \end{array}}$$

Dadurch wird die Relation codeR nichtdeterministisch, da sie nun für die gleichen Argumente zwei unterschiedliche Code-Sequenzen produzieren kann. Der Übersetzer muss sich also beim Übersetzen bei jedem binären Ausdruck für die Regeln Binary oder Binary′ entscheiden. Diese Entscheidung hat keinen Einfluss auf die Korrektheit des Codes, aber auf den Registerverbrauch des Codes, der die Ausdrücke auswertet. Betrachten wir hierzu einen binären Ausdruck $l\ o\ r$ mit zwei Teilausdrücken l und r. Wertet man zuerst l aus, wo benötigt man ein Register, um sich das Ergebnis von l zu merken, während man r auswertet. Dieses Register kann also nicht verwendet werden um r auszuwerten. Man verbraucht also weniger Register, wenn man den Ausdruck zuerst auswertet, dessen Auswertung mehr Register konsumiert. Diese Vorgehensweise ist optimal [SU70] bezüglich des Registerverbrauchs der dann durch die Funktion *regs* bestimmt werden kann:

$$
\begin{aligned}
regs(x) &= 1 \\
regs(c) &= 1 \\
regs(e_1\ o\ e_2) &= \begin{cases} regs(e_1) + 1 & \text{falls } regs(e_1) = regs(e_2) \\ \max(regs(e_1), regs(e_2)) & \text{andernfalls} \end{cases}
\end{aligned}
$$

Unser C0-Übersetzer kann die Funktion *regs* also verwenden, um den Nichtdeterminismus von codeR aufzulösen, indem er den Teilausdruck mit höherem Registerverbrauch zuerst auswertet.

10.2.4 Die Funktionsdefinition: Prolog und Epilog

Die Code-Erzeugung für Ausdrücke und Anweisungen kann nun dazu verwendet werden, den Code eines Unterprogramms *f* zu bauen. Der Code des Rumpfes der Funktion muss nun noch durch den **Prolog** und **Epilog** erweitert werden. Der Prolog erstellt die Aufrufschachtel, indem er auf dem Laufzeitkeller durch Verringern des Kellerpegels entsprechend Platz schafft. Die Aufrufschachtel wird um 4 Byte größer angefordert, als für die lokalen Variablen benötigt, da noch Platz für die Rücksprungadresse benötigt wird. Dies ist notwendig, wenn man die Sprache C0 um den Aufruf von Unterprogrammen erweitert. Den Code für den Rumpf erzeugen wir durch die Regel [Block], wobei wir die Parameter der Funktion als lokale Variable für den Rumpf vereinbaren. Dadurch wird diesen ein Offset in der Aufrufschachtel zugewiesen. Der Prolog legt den Inhalt der Argumentregister $a0–$a3 in den entsprechenden Speicherplätzen der Aufrufschachtel ab. So können Parameter als herkömmliche lokale Variable betrachtet werden.

Der Epilog restauriert die Rückkehradresse und gibt die Schachtel wieder frei. Erweitert man die Sprache C0 entsprechend, so kann die Sprungmarke end als Ziel von Sprüngen, die aus return-Anweisungen resultieren, verwenden. Hierbei ist dann auch zu berücksichtigen, dass der Rückgabewert im Register $v0 abgelegt wird.

Wir zeigen die Code-Generierung für eine Funktionsvereinbarung hier anhand einer Regel, die eine Funktion codeFunc definiert, die den Code einer Funktion liefert. codeFunc nimmt als Argument lediglich die abstrakte Syntax einer Funktionsvereinbarung, konstruiert aus den

Parametern der Funktion und deren Rumpf einen Block, für den dann mittels codeS der Code generiert wird. Die Größe der nötigen Aufrufschachtel wird von codeS entsprechend mit zurück geliefert.

$$\text{Function}\ \frac{\texttt{codeS}\ \{\}\ 0\ \{k_1 x_1; \ldots k_n x_n; s\} \blacktriangleright c, S}{\texttt{codeFunc}\ (t\ f(k_1\,x_1, \ldots, k_n\,x_n)\ \{\ s\ \}) \blacktriangleright \begin{array}{l} f\texttt{:} \\ \quad \texttt{subiu \$sp \$sp } (S+4) \\ \quad \texttt{sw \$ra } S\texttt{(\$sp)} \\ \quad \texttt{sw \$a0 0(\$sp)} \\ \quad \vdots \\ \quad \texttt{sw \$a}n\ 4n\texttt{(\$sp)} \\ \quad c \\ f\texttt{\_end:} \\ \quad \texttt{lw \$ra } S\texttt{(\$sp)} \\ \quad \texttt{addiu \$sp \$sp } (S+4) \\ \quad \texttt{jr \$ra} \end{array}}$$

10.2.5 Ein kleines Beispiel

Betrachten wir den Code für die Anweisung `y = *x + 1;` unter $\Omega = \{\texttt{x} \mapsto 0, \texttt{y} \mapsto 4\}$.

$$\text{Assign}\ \frac{\text{Var}\ \dfrac{\Omega\,\texttt{y} = 4}{\texttt{codeL}\ \Omega\ \texttt{\$v1} :: rs\ (\texttt{y}) \blacktriangleright \texttt{addiu \$v1 \$sp 4}} \quad \texttt{codeR}\ \Omega\ \texttt{\$v0} :: \texttt{\$v1} :: rs\ \texttt{*x+1} \blacktriangleright c_1}{\texttt{codeS}\ \Omega\ 0\ \texttt{y = *x + 1;} \blacktriangleright \begin{array}{l} c_1 \\ \texttt{addiu \$v1 \$sp 4} \\ \texttt{sw \$v0 (\$v1)} \end{array}}$$

Die Generierung des Codes c_1 des rechten Teils des Ausdrucks führen wir aus Platzgründen als Nebenrechnung aus.

$$\text{BinArith}\ \frac{\texttt{codeR}\ \Omega\ \texttt{\$v1} :: rs\ \texttt{*x} \blacktriangleright c_2 \qquad \text{Const}\ \dfrac{}{\texttt{codeR}\ \Omega\ \texttt{\$v0} :: \texttt{\$v1} :: rs\ 1 \blacktriangleright \texttt{li \$v0 1}}}{\texttt{codeR}\ \Omega\ \texttt{\$v0} :: \texttt{\$v1} :: rs\ \texttt{*x+1} \blacktriangleright \begin{array}{l} \texttt{li \$v0 1} \\ c_2 \\ \texttt{addiu \$v0 \$v0 \$v1} \end{array} =: c_1}$$

Die Generierung des Codes c_2 des linken Teilausdrucks $*x$ führen wir wieder als Nebenrechnung aus:

$$\text{Var}\ \frac{\Omega\, x = 0}{\texttt{codeL}\ \Omega\ \$v1 :: rs\ x \blacktriangleright \texttt{addiu \$v1 \$sp 0}}$$

$$\text{LToR}\ \frac{\texttt{codeL}\ \Omega\ \$v1 :: rs\ x \blacktriangleright \texttt{addiu \$v1 \$sp 0}}{\texttt{codeR}\ \Omega\ \$v1 :: rs\ x \blacktriangleright \begin{array}{l}\texttt{addiu \$v1 \$sp 0}\\ \texttt{lw \quad \$v1 (\$v1)}\end{array}}$$

$$\text{Indir}\ \frac{\texttt{codeR}\ \Omega\ \$v1 :: rs\ x \blacktriangleright \begin{array}{l}\texttt{addiu \$v1 \$sp 0}\\ \texttt{lw \quad \$v1 (\$v1)}\end{array}}{\texttt{codeL}\ \Omega\ \$v1 :: rs\ *x \blacktriangleright \begin{array}{l}\texttt{addiu \$v1 \$sp 0}\\ \texttt{lw \quad \$v1 (\$v1)}\end{array}}$$

$$\text{LToR}\ \frac{\texttt{codeL}\ \Omega\ \$v1 :: rs\ *x \blacktriangleright \begin{array}{l}\texttt{addiu \$v1 \$sp 0}\\ \texttt{lw \quad \$v1 (\$v1)}\end{array}}{\texttt{codeR}\ \Omega\ \$v1 :: rs\ *x \blacktriangleright \underbrace{\begin{array}{l}\texttt{addiu \$v1 \$sp 0}\\ \texttt{lw \quad \$v1 (\$v1)}\\ \texttt{lw \quad \$v1 (\$v1)}\end{array}}_{=:c_2}}$$

Der generierte MIPS-Code für `y = *x + 1;` ist also:

```
li    $v0 1       ; | c1
addiu $v1 $sp 0   ; |     | c2
lw    $v1 ($v1)   ; |     |
lw    $v1 ($v1)   ; |     |
addiu $v0 $v0 $v1 ; |
addiu $v1 $sp 4   ;
sw    $v0 ($v1)   ;
```

A Verwendete Notationen

Symbol	Bedeutung	Kommentare
$t[x/t']$		Term, der ensteht, indem jedes Vorkommen von x durch t' in t ersetzt wird.
$x \mapsto t$		Anonyme Funktion, die x nach t abbildet
$\mathrm{dom}\, f$		Urbildbereich der Funktion f
$\mathrm{ran}\, f$	$\{y \mid \exists x.y = f(x)\}$	Bildbereich der Funktion f
if c *then* x *else* y		x falls c gilt, ansonsten y
$f[x \mapsto y]$	$z \mapsto$ *if* $z = x$ *then* y *else* $f(z)$	Funktion, die auf x den Wert y liefert und ansonsten so definiert ist wie f.
$\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_1\}$		Funktion, die für x_i den Wert v_i liefert (für $1 \leq i \leq n$).

B MIPS Assembler Kurzreferenz

B.1 Systemaufrufe

Aktion	`$v0`	Kommentare
print integer	1	`$a0` = Zahl, die ausgegeben werden soll
print string	4	`$a0` = Basisadresse der ASCIIZ Zeichenkette
read integer	5	`$v0` = Enthält gelesene Zahl
read string	8	`$a0` = Basisadresse des Puffers
		`$a1` = Maximale Anzahl der Zeichen, die gelesen werden sollen
exit	10	Beendet die Programmausführung
print character	11	`$a0` = ASCII-Code des Buchstabens
read character	12	`$v0` = Enthält den ASCII-Code des gelesenen Buchstabens

B.2 Assemblerdirektiven

Direktive	Bedeutung
`.align n`	Richte nächstes Datum an einer durch 2^n teilbaren Adresse aus
`.ascii str`	Speichere Zeichenkette `str`, hänge keine 0 an
`.asciiz str`	wie oben aber nullterminiert
`.byte b1, ..., bn`	Lege Bytes `b1` bis `bn` sukzessive im Speicher ab
`.data`	Beginne Datensegment
`.globl sym`	Deklariere Symbol `sym` als global (sichtbar aus anderen Objekt-Dateien)
`.half h1, ..., hn`	Lege Halbwörter `h1` bis `hn` sukzessive im Speicher ab
`.space n`	Alloziere `n` Bytes im aktuellen Segment
`.text`	Beginne Code-Segment
`.word w1, ..., wn`	Lege Wörter `w1` bis `wn` sukzessive im Speicher ab

B.3 Befehle

Eine vollständige Liste der MIPS-Befehle findet sich in der offiziellen Dokumentation [Pri95].

Soweit nicht anders angegeben, erhöht jeder Befehl den Befehlszeiger um 4. In den Befehlen, die Immediates verwenden, kommt es, je nach Befehl, zu einer vorzeichenbehafteten/-losen Erweiterung des 16-Bit Immediates i auf 32 Bit. Wir verwenden dafür die folgenden Abkürzungen: $si \coloneqq \texttt{sext}_{16}^{32}(i)$ und $zi \coloneqq \texttt{zext}_{16}^{32}(i)$. Die Symbole $\overset{s}{<}$ und $\overset{u}{<}$ stehen für den vorzeichenbehafteten bzw. vorzeichenlosen Vergleich. $W[a]$, $H[a]$ und $B[a]$ stehen für den Speicherinhalt an Adresse a in jeweils Form eines Worts, Halbworts, Bytes. $a\ ?\ b\ :\ c$ bedeutet: wenn a dann b sonst c.

Art	Mnemonic	Argumente	Bedeutung	Kommentar
Arithmetik	`addu`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s + \$t$	Addiere zwei Registerinhalte
	`addiu`	$\$d\ \$s\ i$	$\$d \leftarrow \$s + si$	Addiere Registerinhalt zu Immediate
	`subu`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s - \$t$	Subtrahiere zwei Registerinhalte
	`mul`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s \cdot \$t$	Multipliziere zwei Registerinhalte
	`div`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\,/\,\$t$	Dividiere zwei Registerinhalte
	`rem`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\,\%\,\$t$	Rest bei der Division
Bitoperationen	`and`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\ \&\ \$t$	Bitweises Und
	`andi`	$\$d\ \$s\ i$	$\$d \leftarrow \$s\ \&\ zi$	Bitweises Und mit Immediate
	`or`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\ \vert\ \$t$	Bitweises Oder
	`ori`	$\$d\ \$s\ i$	$\$d \leftarrow \$s\ \vert\ zi$	Bitweises Oder mit Immediate
	`xor`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\ \hat{}\ \$t$	Bitweises exkl. Oder
	`xori`	$\$d\ \$s\ i$	$\$d \leftarrow \$s\ \hat{}\ zi$	Bitweises exkl. Oder mit Immediate
	`nor`	$\$d\ \$s\ \$t$	$\$d \leftarrow \overline{\$s\ \vert\ \$t}$	Nicht-Oder, NOR
	`lui`	$\$d\ \ i$	$\$d \leftarrow i_{15} \dots i_0 \cdot 0^{16}$	Lade obere 16 Bits eines Registers
Shifts	`sll`	$\$d\ \$s\ n$	$\$d \leftarrow \$s_{31-n} \dots \$s_0 \cdot 0^n$	Shifte nach links um n
	`srl`	$\$d\ \$s\ n$	$\$d \leftarrow 0^n \cdot \$s_{31} \dots \$s_n$	Shifte vorzeichenlos nach rechts um n
	`sra`	$\$d\ \$s\ n$	$\$d \leftarrow (\$s_{31})^n \cdot \$s_{31} \dots \$s_n$	Shifte vorzeichenbeh. nach rechts um n
	`sllv`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s_{31-n} \dots \$s_0 \cdot 0^n$	$n \coloneqq \$t$ (s.o.)
	`srlv`	$\$d\ \$s\ \$t$	$\$d \leftarrow 0^n \cdot \$s_{31} \dots \$s_n$	$n \coloneqq \$t$ (s.o.)
	`srav`	$\$d\ \$s\ \$t$	$\$d \leftarrow (\$s_{31})^n \cdot \$s_{31} \dots \$s_n$	$n \coloneqq \$t$ (s.o.)
Vergleiche	`slt`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s \overset{s}{<} \$t\ ?\ 1\ :\ 0$	Vorzeichenbeh. Vergl. zweier Registerinhalte
	`sltu`	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s \overset{u}{<} \$t\ ?\ 1\ :\ 0$	Vorzeichenloser Vergl. zwei Registerinhalte
	`slti`	$\$d\ \$s\ i$	$\$d \leftarrow \$s \overset{s}{<} si\ ?\ 1\ :\ 0$	Vorzeichenbeh. Verg. von Reg mit Immediate
	`sltiu`	$\$d\ \$s\ i$	$\$d \leftarrow \$s \overset{u}{<} si\ ?\ 1\ :\ 0$	Vorzeichenloser Verg. von Reg mit Immediate
Laden	`lw`	$\$d\ i(\$s)$	$\$d \leftarrow W[\$s + si]$	Lade Wort
	`lh`	$\$d\ i(\$s)$	$\$d \leftarrow \texttt{sext}_{16}^{32}(H[\$s + si])$	Lade zwei Bytes mit Vorzeichenerw.
	`lb`	$\$d\ i(\$s)$	$\$d \leftarrow \texttt{sext}_{8}^{32}(B[\$s + si])$	Lade Byte mit Vorzeichenerw.
	`lhu`	$\$d\ i(\$s)$	$\$d \leftarrow \texttt{zext}_{16}^{32}(H[\$s + si])$	Lade zwei Bytes ohne Vorzeichenerw.
	`lbu`	$\$d\ i(\$s)$	$\$d \leftarrow \texttt{zext}_{8}^{32}(B[\$s + si])$	Lade Bytes ohne Vorzeichenerw.
Speichern	`sw`	$\$d\ i(\$s)$	$W[\$s + si] \leftarrow \$d$	Speichere Wort
	`sh`	$\$d\ i(\$s)$	$H[\$s + si] \leftarrow \$d[15\ :\ 0]$	Speichere zwei Bytes
	`sb`	$\$d\ i(\$s)$	$B[\$s + si] \leftarrow \$d[7\ :\ 0]$	Speichere Byte
Sprung	`beq`	$\$s\ \$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$s = \$t\ ?\ si\ :\ 0)$	
	`bne`	$\$s\ \$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$s \neq \$t\ ?\ si\ :\ 0)$	
	`blez`	$\$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$t \leq 0\ ?\ si\ :\ 0)$	
	`bgtz`	$\$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$t > 0\ ?\ si\ :\ 0)$	
	`bltz`	$\$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$t < 0\ ?\ si\ :\ 0)$	
	`bgez`	$\$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$t \geq 0\ ?\ si\ :\ 0)$	
	`jal`	$addr$	$\$ra \leftarrow pc + 4, pc \leftarrow addr$	Unterprogrammaufruf
	`jr`	$\$s$	$pc \leftarrow \$s$	Indirekter Sprung
	`syscall`			Rufe Betriebssystem
Pseudo	`li`	$\$d\ i$	$\$d \leftarrow i$	Lade Konstante in Register
	`la`	$\$d\ l$	$\$d \leftarrow addr$	Lädt Adresse der Marke l
	`move`	$\$d\ \$s$	$\$d \leftarrow \$s$	Kopiere Registerinhalt
	`not`	$\$d\ \$s$	$\$d \leftarrow \overline{\$s}$	Bitweise Negation
	`neg`	$\$d\ \$s$	$\$d \leftarrow -\$s$	Vorzeichen ändern
	`b`	i	$pc \leftarrow pc + 4 \cdot (1 + si)$	Unbedingter Sprung
	`b`cc	$\$s\ \$t\ i$	$pc \leftarrow pc + 4 \cdot (1 + \$s\ cc\ \$t\ ?\ si\ :\ 0)$	$cc \in \{\texttt{lt}, \texttt{gt}, \texttt{le}, \texttt{ge}\}$ (vorzeichenbehaftet)

C ASCII-Tabelle

Dez	Hex		Dez	Hex		Dez	Hex		Dez	Hex	
0	00	NUL	32	20	␣	64	40	@	96	60	‘
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	”	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	’	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	TAB	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	–	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

D Index

Literatur

[08] **IEEE Standard for Floating-Point Arithmetic**. 3 Park Avenue, New York, NY 10016-5997, USA: Microprocessor Standards Committee of the IEEE Computer Society, Aug. 2008. DOI: 10.1109/ieeestd.2008.4610935. URL: http://dx.doi.org/10.1109/ieeestd.2008.4610935.

[BM07] Aaron R. Bradley und Zohar Manna. **The calculus of computation - decision procedures with applications to verification**. Springer, 2007. DOI: 10.1007/978-3-540-74113-8. URL: https://doi.org/10.1007/978-3-540-74113-8.

[CLM85] Pedro Celis, Per-Åke Larson und J. Ian Munro. „Robin Hood Hashing (Preliminary Report)". In: **26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985**. IEEE Computer Society, 1985, S. 281–288. ISBN: 0-8186-0644-4. DOI: 10.1109/SFCS.1985.48. URL: https://doi.org/10.1109/SFCS.1985.48.

[Cor+09] Thomas H. Cormen u. a. **Introduction to Algorithms (3. ed.)** MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[Die+97] Martin Dietzfelbinger u. a. „A Reliable Randomized Algorithm for the Closest-Pair Problem". In: **J. Algorithms** 25.1 (1997), S. 19–51.

[Dij75] Edsger W. Dijkstra. „Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: **Commun. ACM** 18.8 (1975), S. 453–457. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

[Dij82] Edsger W. Dijkstra. „Why numbering should start at zero". circulated privately. Aug. 1982. URL: http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

[Gol91] David Goldberg. „What Every Computer Scientist Should Know About Floating-Point Arithmetic“. In: **ACM Comput. Surv.** 23.1 (1991), S. 5–48. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163.

[Gor15] Mike Gordon. „Background Reading on Hoare Logic“. Vorlesungsskript. 2015. URL: http://www.cl.cam.ac.uk/~mjcg/Teaching/2015/Hoare/Notes/Notes.pdf.

[Gos+05] James Gosling u. a. **Java(TM) Language Specification, The 3rd Edition**. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html. Addison-Wesley, 2005. ISBN: 0321246780.

[Gri81] David Gries. **The Science of Programming**. Texts and Monographs in Computer Science. Springer, 1981. ISBN: 978-0-387-96480-5. DOI: 10.1007/978-1-4612-5983-1. URL: https://doi.org/10.1007/978-1-4612-5983-1.

[Har13] Robert Harper. **Practical Foundations for Programming Languages**. New York, NY, USA: Cambridge University Press, 2013. ISBN: 978-1-107-02957-6.

[Hig02] Nicholas J. Higham. **Accuracy and stability of numerical algorithms, Second Edition**. SIAM, 2002. ISBN: 978-0-89871-521-7. DOI: 10.1137/1.9780898718027. URL: https://doi.org/10.1137/1.9780898718027.

[Hoa69] C. A. R. Hoare. „An Axiomatic Basis for Computer Programming“. In: **Commun. ACM** 12.10 (1969), S. 576–580. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[ISO99] ISO. **ISO/IEC 9899:1999 Draft, ISO C Standard 1999**. Techn. Ber. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. 1999.

[Kre15] Robbert Krebbers. „The C standard formalized in Coq“. Diss. Radboud University Nijmegen, 2015. URL: http://robbertkrebbers.nl/research/thesis.pdf.

[LB08] Xavier Leroy und Sandrine Blazy. „Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations“. In: **J. Autom. Reasoning** 41.1 (2008), S. 1–31. DOI: 10.1007/s10817-008-9099-0. URL: http://dx.doi.org/10.1007/s10817-008-9099-0.

[Mul+10] Jean-Michel Muller u. a. **Handbook of Floating-Point Arithmetic**. Birkhäuser, 2010. ISBN: 978-0-8176-4704-9. DOI: 10.1007/978-0-8176-4705-6. URL: https://doi.org/10.1007/978-0-8176-4705-6.

[Neu93] John von Neumann. „First Draft of a Report on the EDVAC". In: **IEEE Ann. Hist. Comput.** 15.4 (1993), S. 27–75. ISSN: 1058-6180. DOI: 10.1109/85.238389.

[NN92] Hanne Riis Nielson und Flemming Nielson. **Semantics with applications: a formal introduction**. http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html. New York, NY, USA: John Wiley & Sons, Inc., 1992. ISBN: 0-471-92980-8.

[Nor98] Michael Norrish. „C formalised in HOL". Diss. University of Cambridge, 1998. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf.

[Pri95] Charles Price. **The MIPS IV Instruction Set**. http://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf. Silicon Graphics Computer Systems, Jan. 1995.

[PY07] Mauro Pezzè und Michal Young. **Software Testing and Analysis: Process, Principles and Techniques**. Wiley, 2007. ISBN: 0471455938.

[RAD15] Stefan Richter, Victor Alvarez und Jens Dittrich. „A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing". In: **Proc. VLDB Endow.** 9.3 (2015), S. 96–107. DOI: 10.14778/2850583.2850585. URL: http://www.vldb.org/pvldb/vol9/p96-richter.pdf.

[Str00] Christopher Strachey. „Fundamental Concepts in Programming Languages". In: **Higher-Order and Symbolic Computation** 13.1/2 (2000), S. 11–49. DOI: 10.1023/A:1010000313106. URL: http://dx.doi.org/10.1023/A:1010000313106.

[SU70] Ravi Sethi und J. D. Ullman. „The Generation of Optimal Code for Arithmetic Expressions". In: **J. ACM** 17.4 (Okt. 1970), S. 715–728. ISSN: 0004-5411. DOI: 10.1145/321607.321620. URL: http://doi.acm.org/10.1145/321607.321620.

[Wan+12] Xi Wang u. a. „Undefined behavior: what happened to my code?" In: **Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012**. ACM, 2012, S. 9. ISBN: 978-1-4503-1669-9. DOI: 10.1145/2349896.

2349905. URL: http://doi.acm.org/10.1145/2349896.2349905.

[Win93] Glynn Winskel. **The Formal Semantics of Programming Languages: An Introduction**. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0-262-23169-7.

[Wir71] Niklaus Wirth. „Program Development by Stepwise Refinement“. In: **Commun. ACM** 14.4 (Apr. 1971), S. 221–227. ISSN: 0001-0782. DOI: 10.1145/362575.362577. URL: http://doi.acm.org/10.1145/362575.362577.

[WSH13] Reinhard Wilhelm, Helmut Seidl und Sebastian Hack. **Compiler Design - Syntactic and Semantic Analysis**. Springer, 2013. ISBN: 978-3-642-17539-8. DOI: 10.1007/978-3-642-17540-4. URL: https://doi.org/10.1007/978-3-642-17540-4.

[Zus90] Konrad Zuse. **Der Computer - Mein Lebenswerk**. Springer, 1990. ISBN: 978-3-540-16736-5.