Einführung in die Informatik

Prof. Dr. H.-D. Ehrich

TU Braunschweig

Auszug aus dem Skript: Seiten 9 - 93

2. Algorithmische Grundkonzepte

2.1 Intuitiver Algorithmenbegriff

In nahezu allen Lebensbereichen haben wir es mit Anweisungen (Vorschriften, Rezepten, Regeln) zu tun, nach denen man etwas ausführen muß oder soll.

Beispiel 2.1:

1. Bedienungsanleitungen, z.B. für Münzfernsprecher (Hörer abnehmen; mindestens 3 x 10 Pfennig einwerfen; wählen; sprechen; Hörer aufhängen; Rest Geld entnehmen)
2. Gebrauchsanweisungen
3. Bauanleitungen, z.B. Flug- und Schiffsmodelle
4. Koch- und Backrezepte
5. Anweisungen zur Einnahme von Medikamenten
   etc.

Wenn solche Anweisungen gewisse Mindestanforderungen an Präzision, Eindeutigkeit, Vollständigkeit und Ausführbarkeit erfüllen, spricht man von einem Algorithmus.

Intuitive Begriffsbestimmung

Ein *Algorithmus* ist eine präzise (d.h. in einer festgelegten Sprache abgefaßte), endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

Beispiel 2.2: Aufgaben, für die Algorithmen bekannt sein dürften:

a) Addition zweier positiver Dezimalzahlen p,q.
b) Test, ob eine gegebene natürliche Zahl eine Primzahl ist.
c) Berechnung der Zahl e = 2.7182. . .
d) Einsortieren einer Karte in eine sortierte Kartei.
e) Sortieren einer unsortierten Kartei.

Man verlangt von einem Algorithmus häufig, daß er zu einem Ende kommt, d.h. daß er nach endlich vielen Schritten abbricht. Ein solcher Algorithmus heißt *terminierend*. Für die Beispiele a, b, d, e gibt es terminierende Algorithmen, für Beispiel c nur nichtterminierende Algorithmen.

Ein Algorithmus braucht ferner keinen eindeutig vorgeschriebenen Ablauf zu haben.

Beispiel 2.3:

1. Nehmen Sie eine Zahl x;
2. Addieren Sie 5 hinzu oder multiplizieren Sie mit 3;
3. Schreiben Sie das Ergebnis auf.

Ein solcher Algorithmus heißt *nicht-deterministisch*. Auch nicht-deterministische Algorithmen können ein eindeutiges Ergebnis liefern. Sie heißen dann *determiniert*.

Beispiel 2.4:

1. Nehmen Sie eine Zahl $x \neq 0$;
2. Entweder: addieren Sie das Dreifache von x zu x und teilen das Ergebnis durch x
   Oder: subtrahieren Sie 4 von x und subtrahieren das Ergebnis von x;
3. Schreiben Sie das Ergebnis auf.

(Das Endergebnis ist immer 4)

Wir fassen die Begriffe zusammen:

Definition 2.5: ein Algorithmus heißt

(1) *terminierend*, wenn er für jede Wahl von Eingabewerten nach endlich vielen Schritten anhält,
(2) *determiniert*, wenn bei wiederholtem Ablauf mit den gleichen Eingabewerten stets das gleiche Ergebnis erzielt wird,
(3) *deterministisch*, wenn bei wiederholtem Ablauf mit den gleichen Eingabewerten stets dieselbe Berechnung (als Folge der Einzelschritte) durchgeführt wird (wobei an denselben Zwischenstellen immer dieselben Zwischenergebnisse erscheinen).

Natürlich sind deterministische Algorithmen stets determiniert.

Im Vordergrund unseres Interesses stehen

determinierte terminierende Algorithmen.

Diese geben eine konstruktive ausführbare Beschreibung einer Ein-/Ausgabefunktion und definieren somit eine Ein-/Ausgabefunktion, d.h. eine Abbildung

$$f: \text{Eingabewerte} \longrightarrow \text{Ausgabewerte}$$

Diese Funktion wird auch *Bedeutung* (Semantik) des Algorithmus genannt. Selbstverständlich kann es mehrere verschiedene Algorithmen mit der gleichen Bedeutung geben.

Beispiel 2.6: Es gibt verschiedene Algorithmen mit den folgenden Bedeutungen (s. Beispiele 2.2, a bis e) :

a) $f: Q \times Q \longrightarrow Q$ $\quad f(p,q) = p + q$

b) $f: N \longrightarrow \{ja, nein\}$ $\quad f(n) = \begin{cases} \text{ja, falls } n \text{ Primzahl} \\ \text{nein sonst} \end{cases}$

c) (nicht terminierend)

Seien K = Menge von Karteikarten
SK = Menge von sortierten Karteien über K
USK = Menge von unsortierten Karteien über K

d) $f: SK \times K \longrightarrow SK$ $\quad f(sk,k) = sk' =$ (sk nach Einsortieren von k)

e) $f: USK \longrightarrow SK$ $\quad f(usk) = sk =$ (usk sortiert)

2.2 Elementare Datentypen

Determinierte terminierende Algorithmen lassen sich demnach auffassen als konstruktive, operationale ("ausführbare") Beschreibungen von (Ein-/Ausgabe-)Funktionen. Sie setzen sich aus

"ausführbaren elementaren (Verarbeitungs-) Schritten"

zusammen. Diese basieren in der Regel auf den *Grundoperationen eines Datentyps*, d.h. auf elementaren Funktionen, die als "gegeben" und "direkt ausführbar" angesehen werden können.

Wir behandeln daher zunächst einige einfache elementare Datentypen mit den auf ihnen definierten Grundoperationen.

Bemerkung: "Daten" ist die Mehrzahl von "datum" (lat.): "Das Gegebene".

1. boolean (bool) Der Datentyp der Wahrheitswerte

Werte: {true, false}
Operationen: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, ...

$\neg$	
false	true
true	false

$\wedge$	false	true
false	false	false
true	false	true

$p \vee q = \neg(\neg p \wedge \neg q)$, $\quad p \Rightarrow q = \neg p \vee q$, $\quad p \Leftrightarrow q = (p \Rightarrow q) \wedge (q \Rightarrow p)$, . . .

2. <u>integer</u> (<u>int</u>) Der Datentyp der ganzen Zahlen

Werte: {. . ., −2, −1, 0, 1, 2, . . .}
Operationen: +, −, ·, ÷ , mod, abs, sign, . . .
$=, \neq, \leq, >, <, \geq$, even, odd, . . .

Bedeutung:

$+ : int \times int \longrightarrow int$	Addition	$7+9 \longmapsto 16$
$- : int \times int \longrightarrow int$	Subtraktion	$7-9 \longmapsto -2$
$\cdot : int \times int \longrightarrow int$	Multiplikation	$7 \cdot 9 \longmapsto 63$
$\div : int \times int \longrightarrow int$	ganzz. Division	$19 \div 7 \longmapsto 2$
$mod : int \times int \longrightarrow int$	Rest bei Division	$19 \bmod 7 \longmapsto 5$
$abs : int \longrightarrow int$	Absolutbetrag	$abs(7) \longmapsto 7$
		$abs(-7) \longmapsto 7$
$sign : int \longrightarrow \{-1,0,1\}$	Signum	$sign(7) \longmapsto 1$
	(Vorzeichen)	$sign(0) \longmapsto 0$
		$sign(-7) \longmapsto -1$
. . .		
$= : int \times int \longrightarrow bool$	gleich	$7=7 \longmapsto true$
		$7=9 \longmapsto false$
$\neq : int \times int \longrightarrow bool$	ungleich	$7 \neq 9 \longmapsto true$
$\leq : int \times int \longrightarrow bool$	kleiner-gleich	$7 \leq 9 \longmapsto true$
		$9 \leq 7 \longmapsto false$
$> : int \times int \longrightarrow bool$	größer	$7 > 9 \longmapsto false$
$\geq$ (größer-gleich), $<$ (kleiner)	entsprechend	
$even : int \longrightarrow bool$	gerade	$even(6) \longmapsto true$
		$even(7) \longmapsto false$
$odd : int \longrightarrow bool$	ungerade	$odd(6) \longmapsto false$
		$odd(7) \longmapsto true$
. . .		

Weitere elementare Datentypen sind u.a.

3. <u>real</u> Der Datentyp der reellen Zahlen

4. <u>character (char)</u> Der Datentyp der "darstellbaren" Zeichen

5. <u>string</u> Der Datentyp der Zeichenfolgen
etc.

Wir beschränken uns hier zunächst auf <u>bool</u> und <u>int</u> zur Erläuterung algorithmischer Grundkonzepte.

Bemerkung: In einem Rechner sind stets nur endlich viele verschiedene int-Werte darstellbar, z.B.

$$\{ -2^{31}+1, \ldots, -1, 0, +1, \ldots, 2^{31}-1 \}$$

Wir sehen hier von dieser (maschinenabhängigen) Beschränkung und den dadurch verursachten Problemen ab und betrachten den "idealisierten" Datentyp *aller* (unendlich vielen) ganzen Zahlen.

Bemerkung: Operationen auf Datentypen sind nicht immer vollständig definiert. Zum Beispiel ist in int die Division durch 0 nicht erklärt. Als Ergebnis der Auswertung verwenden wir in solchen Fällen das Zeichen $\bot$ für *undefiniert*, z.B.

$$19 \div 0 \longmapsto \bot$$
$$19 \; \mathbf{mod} \; 0 \longmapsto \bot$$

Weiterhin vereinbaren wir, daß die übrigen Operationen auf int den Wert $\bot$ ergeben, wenn mindestens einer der Operanden $\bot$ ist, z.B.

$$\bot + 3 \longmapsto \bot$$
$$2 \cdot \bot \longmapsto \bot$$
$$\bot \le 3 \longmapsto \bot$$
etc.

2.3 Ausdrücke (Terme)

Wie kann man nun solche Grundoperationen zu Algorithmen "zusammensetzen"? Eine wohl hinreichend geläufige Art der Zusammensetzung ist die Bildung von Ausdrücken (Termen), wie z.B.

$$7 + (9 + 3) \cdot 6 - 18 \;, \quad \mathrm{abs}(13 - \mathrm{sign}(-7) \cdot 17) \;, \quad \ldots$$
$$\neg \, \mathrm{true} \wedge (\mathrm{false} \vee (\neg \, \mathrm{false} \wedge \mathrm{true})), \ldots$$

Die erste Zeile gibt Beispiele für int-Terme, die sich zu je einem int-Wert auswerten lassen, und die untere Zeile gibt ein Beispiel eines bool-Termes, der sich zu einem bool-Wert auswerten läßt.

Solchen "konventionellen" Termen fügen wir die für die nachfolgenden Erörterungen sehr wichtige Klasse der *bedingten Terme* hinzu :

if b then t else u fi

Hierbei ist b ein bool-Term, und t,u sind entweder zwei int- oder zwei bool-Terme. Die *Auswertung* bedingter Terme wird bestimmt durch

$$\text{if true then } t \text{ else } u \text{ fi} \longmapsto t$$
$$\text{if false then } t \text{ else } u \text{ fi} \longmapsto u$$

Dies gilt auch, wenn t oder u undefiniert ist, z.B.

$$\text{if true then } 3 \text{ else } \bot \text{ fi} \longmapsto 3 \quad (!)$$
$$\text{if false then } 3 \text{ else } \bot \text{ fi} \longmapsto \bot$$

Weiterhin setzen wir für undefiniertes b

$$\text{if } \bot \text{ then } t \text{ else } u \text{ fi} \longmapsto \bot$$

(s.u. Beispiel 2.26).

Obwohl die Bildung und Auswertung von bool- und int-Termen damit hinreichend klar sein dürfte, soll dies noch präziser definiert werden (nicht zuletzt, um das präzise Definieren zu üben):

Definition 2.7: bool-Terme

B1: Die Wahrheitswerte true und false sind bool-Terme.

B2: Sind b,c,d bool-Terme, so sind auch
$\neg b$, $(b \wedge c)$, $(b \vee c)$, $(b \Rightarrow c)$, $(b \Leftrightarrow c)$, . . . , if b then c else d fi
bool-Terme.

B3: Sind t,u int-Terme, so sind auch
$(t = u)$, $(t \neq u)$, $(t \leq u)$, $(t > u)$, $(t \geq u)$, $(t < u)$, even(t), odd(t), . . .
bool-Terme.

B4: Nur die durch B1, B2, B3 bestimmten Zeichenfolgen sind bool-Terme, keine anderen.

Definition 2.8: int-Terme

I1: Die int-Werte . . . , -2, -1, 0, 1, 2, . . . sind int-Terme.

I2: Sind t,u int-Terme, so sind auch $(t + u)$, $(t - u)$, $(t \cdot u)$, $(t \div u)$, $(t \bmod u)$, abs(t), sign(t), . . . int-Terme.

I3: Ist b ein bool-Term und sind t,u int-Terme, so ist auch
if b then t else u fi
ein int-Term.

I4: Nur die durch I1, I2, I3 bestimmten Zeichenfolgen sind int-Terme, keine anderen.

Bemerkung: Sowohl die Definition der bool-Terme als auch die der int-Terme bezieht sich auf sich selbst (ist "rekursiv"). Darüber hinaus bezieht sich die Definition der bool-Terme auf die der int-Terme und umgekehrt (die Definitionen sind "wechselseitig rekursiv"). Derartige Definitionsmethoden sind weit verbreitet für die endliche Definition unendlicher Mengen, insbesondere auch für die Definition der Syntax von Programmiersprachen (wenn auch in kompakterer Notation).

Klammereinsparungsregeln

Die obige Definition von Termen erfordert "vollständige Klammerung" aller Teilterme, z.B.

$(((abs((7 \cdot 9) + 7) - 6) + 8) - 17)$
$(((true \wedge true) \vee false) \Rightarrow ((false \vee true) \vee \neg false))$

Üblicherweise erspart man sich viele Klammern durch

a) Vorrangregeln zwischen Operationen:
   - "Punktrechnung vor Strichrechnung"
   - $\neg$ vor $\wedge$ vor $\vee$, $\Rightarrow$, $\Leftrightarrow$

b) Ausnutzung von Assoziativgesetzen

Die obigen Beispielterme lassen sich so übersichtlich und trotzdem eindeutig wie folgt notieren:

$abs(7 \cdot 9 + 7) - 6 + 8 - 17$
$(true \wedge true \vee false) \Rightarrow (false \vee true \vee \neg false)$

Auswertung von Termen

Hierfür gibt es einen allgemeinen Algorithmus. Die Auswertung elementarer Terme der Art $7 + 9$, $19 \bmod 7$, . . . , $true \wedge false$, $\neg true$, . . . wird als *direkt ausführbare Grundoperation* angenommen.

Man geht nun (in der Klammerstruktur) *von innen nach außen* und ersetzt solche elementaren Terme durch ihren Wert. Dies wiederholt man, bis nur noch ein Wert dasteht, und dieser ist der Wert des Terms. Ausnahme: bei bedingten Termen if b then t else u fi wertet man erst die Bedingung (i.e. den bool-Term) b aus und ersetzt ihn dann durch den gesamten Term t bzw. u, je nachdem der Wert von b true oder false ist.

Beispiel 2.9:

$$
\begin{array}{lllll}
 & 1 + \underline{if}\ true \vee \neg false & \underline{then}\ 7 \cdot 9 + 7 - 6 & \underline{else}\ abs(3 - 8) & \underline{fi} \\
\mapsto & 1 + \underline{if}\ true \vee true & \underline{then}\ 7 \cdot 9 + 7 - 6 & \underline{else}\ abs(3 - 8) & \underline{fi} \\
\mapsto & 1 + \underline{if}\ true & \underline{then}\ 7 \cdot 9 + 7 - 6 & \underline{else}\ abs(3 - 8) & \underline{fi}
\end{array}
$$

$\longmapsto\ 1 + 7 \cdot 9 + 7 - 6$
$\longmapsto\ 1 + 63 + 7 - 6$
$\longmapsto\ 64 + 7 - 6$
$\longmapsto\ 71 - 6$
$\longmapsto\ 65$

Bemerkung: Das obige Verfahren zur Auswertung von Termen ist ein *nichtdeterministischer*, jedoch *determinierter* und *terminierender* Algorithmus (wenn auch informell und daher nicht sehr präzise beschrieben): Die Reihenfolge der einzelnen Schritte ist nicht immer festgelegt, jedoch ist das Endergebnis unabhängig davon und wird nach endlich vielen Schritten erreicht.

Beispiel 2.10:

$$\begin{array}{ccc} & (7+9)\cdot(6+10) & \\ / & & \backslash \\ 16\cdot(6+10) & & (7+9)\cdot 16 \\ \backslash & & / \\ & 16 \cdot 16 & \\ & | & \\ & 256 & \end{array}$$

2.4 Applikative Algorithmen

Man kann Terme dazu benutzen, um auf der Grundlage gegebener Grundoperationen neue, zusammengesetzte Funktionen zu definieren, wie z.B.

$f(x) = 2 \cdot x + 1$ vom Typ int
$g(p,q) = p \wedge \neg q$ vom Typ bool

Hierzu reichen jedoch die im vorigen Abschnitt definierten Terme nicht aus. Wir brauchen *Terme mit Unbestimmten*.

Gegeben seien zwei (unendliche, abzählbare) Mengen von Symbolen, genannt "Unbestimmte":

$x, y, z, \ldots, x_0, x_1, x_2, \ldots$ vom Typ int
$p, q, r, \ldots, p_0, p_1, p_2, \ldots$ vom Typ bool

Ein *Term mit Unbestimmten* wird gebildet wie ein Term im vorigen Abschnitt, nur daß eine Unbestimmte vom Typ int bzw. bool an der Stelle eines Wertes vom gleichen Typ stehen kann.

Beispiel 2.11: $x,\ x-2,\ 2 \cdot x + 1,\ x \bmod y,\ (x+t)\cdot(y-t)\cdot z, \ldots$
$p,\ p \wedge true,\ p \wedge q,$
$((p \vee true) \Rightarrow (q \wedge false)) \Leftrightarrow r, \ldots$

Im folgenden wird unter "Term" immer ein Term verstanden, der Unbestimmte enthalten kann.

Definition 2.12: Sei $n \in N = \{0,1,2,...\}$, und seien $u_1, \dots, u_n$ Unbestimmte vom Typ $\tau_1, \dots, \tau_n$ (jeweils bool oder int). Sei ferner $t(u_1, \dots, u_n)$ ein Term mit den Unbestimmten $u_1, \dots, u_n$ vom Typ τ (bool oder int). Dann ist

(*) $\quad f(u_1, \dots, u_n) = t(u_1, \dots, u_n)$

eine *Funktionsdefinition* vom Typ τ. f ist der *Funktionsname*, $u_1, \dots, u_n$ heißen *formale Parameter*, und $f(u_1, \dots, u_n)$ heißt der *Funktionsausdruck* der Funktionsdefinition.

Beispiel 2.13:

a) $f(p,q,x,y)$ = if $p \vee q$ then $2 \cdot x + 1$ else $3 \cdot y - 1$ fi
b) $g(x)$ = if even(x) then $x+2$ else $3 \cdot x + 1$ fi
c) $h(p,q)$ = if p then q else false fi

a und b sind Funktionsdefinitionen vom Typ int, c ist eine vom Typ bool.

Auswertung von definierten Funktionen

Eine Funktionsdefinition der Art (*) definiert eine Funktion

$$f: \tau_1 \times \tau_2 \times \dots \times \tau_n \longrightarrow \tau$$

und zwar wie folgt. Seien $a_1, \dots, a_n$ Werte vom Typ $\tau_1, \dots, \tau_n$.
Dann ist

$$f(a_1, \dots, a_n)$$

derjenige Wert vom Typ τ, der sich folgendermaßen errechnet:

1. Ersetze im definierenden Term von (*), $t(u_1, \dots, u_n)$, jedes Vorkommen von u_i durch a_i, $i = 1, \dots, n$. Der entstehende Term sei $t(a_1, \dots, a_n)$. ($t(a_1, \dots, a_n)$ enthält keine Unbestimmten mehr)
2. Werte $t(a_1, \dots, a_n)$ mit dem Algorithmus des vorigen Abschnitts aus.

Dies ist wiederum ein nichtdeterministischer, determinierter, terminierender Algorithmus zur Auswertung von *Funktionsanwendungen (Applikationen)* $f(a_1, \dots, a_n)$.

Beispiel 2.14: (siehe obige Beispiele 2.13)

a) f : bool × bool × int × int ⟶ int
f(true,true,3,4) = 7
f(false,false,3,4) = 11
etc.

b) g : int ⟶ int
g(1) = 4
g(2) = 1
g(3) = 10
g(4) = 2
etc.

c) h : bool × bool ⟶ bool
h(false,false) = false
h(false,true) = false
h(true,false) = false
h(true,true) = true

Bemerkung: $h(p,q) = p \wedge q$

Bemerkung: Der Fall $n=0$ ist zugelassen. In diesem Fall hat die definierte Funktion f keine Parameter. Die Auswertung von f liefert dann immer ("bei jeder Anwendung von f auf 0 Werte") den gleichen Wert als Ergebnis. Funktionen ohne Parameter sind also i.w. dasselbe wie Konstante.

Die bisher betrachteten Möglichkeiten der Definition neuer Funktionen sind noch recht beschränkt. Neue Möglichkeiten gewinnen wir durch die Beobachtung, daß sich Funktionsausdrücke wiederum als Terme auffassen lassen.

Wir *erweitern die Klasse der Terme* beträchtlich dadurch, daß definierte Funktionen wie die Grundoperationen bei der Termbildung "mitmachen" dürfen.

Beispiel 2.15: Seien f,g,h die im obigen Beispiel 2.13 definierten Funktionen. Dann sind z. B. Terme in diesem erweiterten Sinne:

$f(p,q,x,y) + 1$
$2 \cdot f(\, p \wedge \text{false} \,,\, q \Rightarrow \text{false} \,,\, x+1 \,,\, y-1 \,) - g(\, f(\, \text{true} \,,\, \text{true} \,,\, x \,,\, x+y \,))$
etc.

Die *Auswertung* solcher Terme geschieht wie bisher von innen nach außen (Ausnahme: if-then-else-fi), wobei elementare Ausdrücke mit definierten Funktionen (d.h. alle Parameter sind ausgewertet) nach dem obigen Algorithmus ausgewertet werden.

Die Möglichkeiten der Definition neuer Funktionen erweitern sich stark, wenn wir solche erweiterten Terme *wieder zur Funktionsdefinition verwenden.* Dadurch können sich Funktionsdefinitionen auf andere Funktionsdefinitionen beziehen.

<u>Beispiel 2.16:</u> f(x,y) = <u>if</u> g(x,y) <u>then</u> h(x+y) <u>else</u> h(x-y) <u>fi</u>
g(x,y) = (x = y) ∨ odd(y)
h(x) = j(x+1) · j(x-1)
j(x) = 2 · x − 3

f bezieht sich auf g und h.
h bezieht sich auf j.
Damit bezieht sich f (indirekt über h) auch auf j.

Dies Beispiel zeigt, daß sich Funktionsdefinitionen auf diese Weise übersichtlich strukturieren lassen: f ist mit Hilfe der "Hilfsfunktionen" g und h definiert, h wiederum mit Hilfe der "Hilfsfunktion" j. Im Prinzip ließe sich dies f jedoch auch ohne Hilfsfunktionen definieren, und zwar durch Einsetzen der Hilfsfunktionen:

f(x,y) = <u>if</u> (x = y) ∨ odd(y)
<u>then</u> (2 · (x+y+1) − 3) · (2 · (x+y−1) − 3)
<u>else</u> (2 · (x−y+1) − 3) · (2 · (x−y−1) − 3) <u>fi</u>

Dies geht jedoch prinzipiell nicht mehr, wenn sich die Funktionsdefinition - direkt oder indirekt - *auf sich selbst bezieht!*

Solche Funktionsdefinitionen heißen *rekursiv*.

<u>Beispiel 2.17:</u> f(x,y) = <u>if</u> x=0 <u>then</u> y <u>else</u>
<u>if</u> x>0 <u>then</u> f(x−1,y) + 1
<u>else</u> −f(−x,−y) <u>fi</u> <u>fi</u>

<u>Bemerkung:</u> −x steht für 0−x.

Einige Auswertungen :

f(0,y) = y für alle y ∈ <u>Int</u>
f(1,y) ⟼ f(0,y) + 1 ⟼ y+1 für alle y ∈ <u>Int</u>
f(2,y) ⟼ f(1,y) + 1 ⟼ f(0,y) +1 +1 ⟼ y+2
. . .
f(x,y) ⟼ . . . ⟼ x+y falls x≥0
f(−1,y) ⟼ −f(1,−y) ⟼ −(−y+1) = y−1
f(−2,y) ⟼ −f(2,−y) ⟼ . . . ⟼ −(−y+2) = y−2
. . .
f(x,y) ⟼ . . . ⟼ x+y falls x<0

Dies zeigt: $f(x,y) = x+y$ für alle $x,y \in$ <u>int</u>.

<u>Bemerkung</u>: Rekursive (Mengen von) Funktionsdefinitionen sind ein sehr mächtiges Mittel zur Definition neuer Funktionen. Man kann zeigen, daß sich *alle* (mit irgendwelchen Algorithmen bzw. Maschinen, und seien sie noch so mächtig!) *berechenbaren* Funktionen so definieren lassen (mehr darüber im Abschnitt 3.1). Sie bilden daher die Grundlage für eine Klasse universeller Programmiersprachen, der sog. "applikativen Sprachen" (z.B. LISP, APL).

<u>Definition 2.18</u>: Ein *applikativer Algorithmus* ist eine Menge von Funktionsdefinitionen

$$
\begin{aligned}
f_1(u_{1,1}, \ldots, u_{1,n_1}) &= t_1(u_{1,1}, \ldots, u_{1,n_1}) \\
f_2(u_{2,1}, \ldots, u_{2,n_2}) &= t_2(u_{2,1}, \ldots, u_{2,n_2}) \\
&\ldots \\
f_m(u_{m,1}, \ldots, u_{m,n_m}) &= t_m(u_{m,1}, \ldots, u_{m,n_m}) \quad .
\end{aligned}
$$

bei der eine (die als erste hingeschriebene, also f_1) "die" durch den Algorithmus definierte Funktion ist.

Ein "Aufruf" des Algorithmus erfolgt durch Eingabe von n_1 Parameterwerten $a_1, \ldots, a_{n_1}$ passenden Typs. Berechnet und ausgegeben wird der Wert von $f_1(a_1, \ldots, a_{n_1})$. Zur Auswertung kann unsere oben beschriebene Methode (oder eine Variante davon) dienen.

Die "Bedeutung" eines applikativen Algorithmus ist die definierte Funktion f_1. Sind $\tau_1, \ldots, \tau_{n_1}$ die Typen von $u_{1,1}, \ldots, u_{1,n_1}$ und τ der Typ des Terms t_1, so ist

$$f_1 : \tau_1 \times \ldots \times \tau_{n_1} \longrightarrow \tau .$$

<u>Terminierung</u>

Mit dem Übergang zu rekursiven Definitionen haben wir uns ein Problem eingehandelt, wie das folgende Beispiel zeigt.

<u>Beispiel 2.19</u>: $f(x) =$ <u>if</u> $x=0$ <u>then</u> 0 <u>else</u> $f(x-1)$ <u>fi</u>

Einige Auswertungen :

$f(0) \longmapsto 0$

$f(1) \longmapsto f(0) \longmapsto 0$

$f(2) \longmapsto f(1) \longmapsto f(0) \longmapsto 0$

...

$f(x) \longmapsto \ldots \longmapsto 0$ für alle $x \geq 0$

Aber: $f(-1) \longmapsto f(-2) \longmapsto f(-3) \longmapsto \ldots\ldots$

diese Auswertung terminiert nicht!

Insgesamt ist $f(x) = \begin{cases} 0 & \text{falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$

Bei rekursiven Funktionsdefinitionen kann es vorkommen, daß der Algorithmus der Termersetzung und -auswertung nicht terminiert. (Trivialbeispiel: $f(x) = f(x)$).

Sei $f(u_1, \ldots, u_n)$ eine rekursiv definierte Funktion, und seien $a_1, \ldots, a_n$ Argumentwerte für f.

Terminiert die Auswertung von $f(a_1, \ldots, a_n)$ nicht, so betrachten wir f an der Stelle $(a_1, \ldots, a_n)$ als *undefiniert*. In Zeichen

$$f(a_1, \ldots, a_n) = \perp$$

Eine möglicherweise an manchen Stellen undefinierte Funktion heißt *partielle Funktion*. Eine überall definierte Funktion heißt *totale Funktion* (und ist ein Spezialfall einer partiellen Funktion!).

Demnach gilt :

Die Bedeutung eines applikativen Algorithmus ist eine partielle Funktion.

Einige Beispiele für applikative Algorithmen

Applikative Algorithmen ergeben sich häufig direkt aus der Problemstellung.

Beispiel 2.20: Fakultät-Funktion $x! = x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 2 \cdot 1$ für $x \geq 0$

Bekannte Definition : $0! = 1$, $x! = x \cdot (x-1)!$
Problem: Wert für negative x ?

1. Lösung:

fac(x) = if x=0 then 1 else x · fac(x-1) fi

Bedeutung: $fac(x) = \begin{cases} x! & \text{falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$

2. Lösung

fac(x) = if x≤0 then 1 else x · fac(x-1) fi

Bedeutung: $fac(x) = \begin{cases} x! & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$

Andere Festlegungen für x<0 sind möglich.

Beispiel 2.21: Fibonacci-Zahlen $f_0 = f_1 = 1$, $f_i = f_{i-1} + f_{i-2}$ für $i>1$

```
fib(x) = if (x=0) ∨ (x=1) then 1 else fib(x-2) + fib(x-1) fi
```

Bedeutung: $\mathrm{fib}(x) = \begin{cases} x\text{-te Fibonacci-Zahl} & \text{falls } x \geq 0 \\ \bot & \text{sonst} \end{cases}$

Beispiel 2.22: **Produkt** $\mathrm{prod}(x,y) = x \cdot y$, **nur mit + als Grundoperation.**

Es ist

$$0 \cdot y = 0$$
$$x \cdot y = (x-1) \cdot y + y \quad \text{für } x>0$$
$$x \cdot y = -((-x) \cdot y) \quad \text{für } x<0$$

```
prod(x,y) = if x=0 then 0 else
              if x>0 then prod(x-1,y) + y
                     else - prod(-x,y)      fi fi
```

Beispiel 2.23: Größter gemeinsamer Teiler ggT(x,y)

Für $0<x$ und $0<y$ gilt: $\mathrm{ggT}(x,x) = x$
$\mathrm{ggT}(x,y) = \mathrm{ggT}(y,x)$
Für $x<y$ gilt: $\mathrm{ggT}(x,y) = \mathrm{ggT}(x,y-x)$

```
ggT(x,y) = if (x≤0) ∨ (y≤0) then ggT(x,y) else
             if x=y then x else
             if x>y then ggT(y,x)
                    else ggT(x,y-x)      fi fi fi
```

Für positive x und y berechnet der Algorithmus den größten gemeinsamen Teiler korrekt. Ist $x \leq 0$ oder $y \leq 0$, so ergeben sich nicht abbrechende Berechnungen, d.h. ggT(x,y) ist an diesen Stellen undefiniert.
Dies ist die formale Niederschrift eines schon von Euklid beschriebenen Algorithmus (Euklid: Elemente, 7.Buch, Satz 2; ca. 300 v.Chr.)

Beispiel 2.24: Test, ob eine Zahl gerade ist : even(x)

Es gilt even(0) = true, even(x+1) = odd(x),
odd(0) = false, odd(x+1) = even(x)

```
even(x) = if x=0 then true else
            if x>0 then odd(x-1)
                   else odd(x+1)    fi fi
```

```
odd(x)  =  if x=0 then false else
              if x>0 then even(x-1)
                     else even(x+1)  fi fi
```

Dieser applikative Algorithmus besteht aus zwei Funktionsdefinitionen: definiert wird even mit odd als Hilfsfunktion. Durch Vertauschen der Reihenfolge erhält man einen applikativen Algorithmus für odd mit even als Hilfsfunktion.

Beispiel 2.25: Test, ob x eine Primzahl ist: prim(x)

Es gilt $\text{prim}(x) \Leftrightarrow (x>1) \wedge \text{pr}(2,x)$ für $x \geq 0$,
wobei $\text{pr}(x,y) \Leftrightarrow$ y ist durch keine Zahl z, $x \leq z < y$, teilbar
Wir setzen $\text{prim}(x) \Leftrightarrow \text{prim}(-x)$ für $x<0$

```
prim(x) = if abs(x)≤1 then false else
            if x<0 then prim(-x)
                   else pr(2,x)      fi fi

pr(x,y) = if x≥y then true
                 else (y mod x)≠0 ∧ pr(x+1,y) fi
```

Diese Lösung für pr ist nicht sehr effizient.

Bessere Lösung: ersetze $x \geq y$ durch $x \cdot x > y$. pr(x,y) testet dann, ob es keinen Teiler von y zwischen x und z gibt, wobei z die kleinste Zahl $\geq x$ ist mit $z^2 > y$. Gäbe es einen Teiler q zwischen z und y, so müßte es auch einen Teiler p zwischen x und z geben, damit $p \cdot q = y$ sein kann.

Weitere Verbesserung: ersetze pr(x+1,y) durch if x=2 then pr(3,y) else pr(x+2,y) fi.

Beispiel 2.26: Ein seltsames Beispiel

```
f(x) = if f(x) = 0 then 1 else 0 fi
```

Gemäß unserer Definition der Bedeutung über die Termauswertung ist

$$f(x) = \perp \text{ für alle } x \in \text{int}$$

Denn die Auswertung von z.B. f(7) verläuft wie folgt:

```
f(7) ↦ if f(7)=0 then 1 else 0 fi
     ↦ if (if f(7)=0 then 1 else 0 fi)=0 then 1
                                          else 0 fi
     ↦ . . .
```

d.h. die Auswertung terminiert nicht, woraus $f(7)=\perp$ folgt.

Dies scheint aber der Funktionsdefinition zu widersprechen: da $f(7)=\perp$ ist, ist $f(7)\neq 0$ und somit muß $f(7)=0$ sein!

Lösung des Rätsels: Vorsicht beim Rechnen mit $\perp$! Es ist falsch, $\perp=0$ als falsch anzusehen (Natürlich ist es auch nicht wahr.). Alles wird konsistent, wenn wir $\perp=0$ als *undefiniert* ansehen und ferner setzen:

$$\underline{\text{if}}\ \perp\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u\ \underline{\text{fi}} = \perp$$

(s.o. Abschnitt 2.3)

Dies entspricht auch der Interpretation von $\perp$ als "unendlich langer Berechnung".

Beispiel 2.27: Ein kniffliges Beispiel

$$f(x) = \underline{\text{if}}\ x>100\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ f(f(x+11))\ \underline{\text{fi}}$$

Es ist klar, daß $f(x) = x-10$ ist, falls $x>100$ ist. Was ist aber für $x\leq 100$?

$$\begin{array}{llllll}
f(100) & \mapsto f(f(111)) & \mapsto f(101) & \mapsto 91 & & \\
f(99) & \mapsto f(f(110)) & \mapsto f(100) & \mapsto \ldots & \mapsto 91 & \\
f(98) & \mapsto f(f(109)) & \mapsto f(99) & \mapsto \ldots & \mapsto 91 & \\
\ldots & \ldots & \ldots & \ldots & & \\
f(90) & \mapsto f(f(101)) & \mapsto f(91) & \mapsto \ldots & \mapsto 91 & \\
f(89) & \mapsto f(f(100)) & \mapsto \ldots & \mapsto f(91) & \mapsto \ldots & \mapsto 91 \\
f(88) & \mapsto f(f(99)) & \mapsto \ldots & \mapsto f(91) & \mapsto \ldots & \mapsto 91 \\
\ldots\ldots & & & & &
\end{array}$$

Tatsächlich ist

$$f(x) = \underline{\text{if}}\ x>100\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ 91\ \underline{\text{fi}}$$

die allgemein Lösung. Der Beweis hierfür ist nicht ganz trivial (s.u. Beispiel 3.30).

Beispiel 2.28: Ein noch kniffligeres Beispiel

$$\begin{aligned}
f(x) &= \underline{\text{if}}\ x=1\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(g(x))\ \underline{\text{fi}}\\
g(x) &= \underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ x\div 2\ \underline{\text{else}}\ 3\cdot x+1\ \underline{\text{fi}}
\end{aligned}$$

Man sieht leicht, daß die Termauswertung für $f(x)$, $x\leq 0$, nicht terminiert. Also ist

$$f(x) = \perp \qquad \text{für } x\leq 0.$$

Was aber ist mit $x>0$?

$f(1) \mapsto 1$

$f(2) \mapsto f(1) \rightarrow 1$

$f(3) \mapsto f(10) \mapsto f(5) \mapsto f(16) \mapsto f(8) \longrightarrow f(4) \mapsto f(2) \mapsto f(1) \mapsto 1$

$f(4) \mapsto \ldots \mapsto 1$ s.o.

$f(5) \mapsto \ldots \mapsto 1$ s.o.

$f(6) \mapsto f(3) \mapsto \ldots \mapsto 1$

$f(7) \mapsto f(22) \mapsto f(11) \mapsto f(34) \rightarrow f(17) \rightarrow f(52) \mapsto f(26) \rightarrow f(13) \longrightarrow f(40) \mapsto f(20)$
$\quad \mapsto f(10) \mapsto f(5) \mapsto \ldots \mapsto 1$

Es ist eine ***bisher unbewiesene Vermutung***, daß

$$f(x) = 1 \qquad \text{für alle } x>0$$

Bis zu sehr großen x hat man dies mit Computern verifiziert, aber das ist natürlich kein Beweis.

<u>Beispiel 2.29:</u> Ein gewaltiges Beispiel

$f(x,y) = \underline{\text{if}}\ x\le 0\ \underline{\text{then}}\ y+1\ \underline{\text{else}}$
$\qquad \underline{\text{if}}\ y\le 0\ \underline{\text{then}}\ f(x-1,1)$
$\qquad\qquad \underline{\text{else}}\ f(x-1,f(x,y-1))\ \underline{\text{fi}}\ \underline{\text{fi}}$

Diese Funktion ist als "Ackermann-Funktion" bekannt und spielt eine große Rolle in der Theorie der Berechenbarkeit. Ihre Bedeutung rührt daher, daß ihre Funktionswerte – und damit die Länge ihrer Berechnung – unglaublich schnell anwachsen:

$f(0,0) \mapsto 1$

$f(1,0) \mapsto f(0,1) \mapsto 2$

$f(1,1) \mapsto f(0,f(1,0)) \mapsto f(0,f(0,1)) \mapsto f(0,2) \mapsto 3$

$f(2,2) \mapsto f(1,f(2,1)) \rightarrow f(1,f(1,f(2,0))) \mapsto f(1,f(1,f(1,1))) \mapsto \ldots \mapsto f(1,f(1,3))$
$\quad \mapsto f(1,f(0,f(0,3))) \mapsto f(1,f(0,4)) \mapsto f(1,5) \mapsto f(0,f(1,4)) \longrightarrow f(0,f(0,f(1,3)))$
$\quad \mapsto f(0,f(0,f(1,2)))) \mapsto f(0,f(0,f(0,f(0,f(1,1))))) \rightarrow \ldots \mapsto f(0,f(0,f(0,f(0,3))))$
$\quad \mapsto f(0,f(0,5)) \mapsto f(0,6) \mapsto 7$

$f(3,3) \mapsto 61$ Vorsicht! Herleitung sehr lang!

$$f(4,4) \mapsto \ldots \mapsto 2^{2^{2^{2^{2^{2}}}}} -3 = 2^{2^{2^{16}}} -3 = 2^{2^{65536}} -3$$

Diese Zahl ist so unglaublich groß, daß für die Zeit zu ihrer Berechnung (i.w. mit +1!) das Alter des Universums nicht ausreichen dürfte.

2.5 Imperative Algorithmen

Dies ist die verbreitetste Art, Algorithmen für Computer zu formulieren. Man bedient sich dazu einer "imperativen Programmiersprache" wie z.B. PASCAL, ALGOL68, MODULA, BASIC, FORTRAN, PL/1, COBOL, . . . Zwar kommen in diesen Sprachen auch Terme (oder Ausdrücke, s.u.) vor; charakteristisches Grundkonzept ist hier jedoch nicht die auf Termen basierende Funktionsdefinition und -applikation (obwohl es sie z.B. in ALGOL68 auch gibt), sondern die *Anweisung* (Befehl), die auf der Verwendung von (Programm-) *Variablen* beruht.

Mit Hilfe von *Variablen* lassen sich Zwischenergebnisse *benennen* und für spätere Wiederverwendungen *aufbewahren*. Als Idee liegt ihnen das Konzept des *Speicherplatzes* zugrunde. Demgemäß besteht eine Variable aus

- einem Namen
- einem veränderlichen Wert.

Der Wert einer Variablen wird durch eine *Wertzuweisung* verändert.

Zur Präzisierung sei X der Name einer Variablen. Mit $|X|$ bezeichnen wir ihrer (momentanen) Wert. Sei t ein Term ohne Unbestimmte, und sei w(t) sein Wert. Eine *Wertzuweisung* ist dann ein Paar

$$X := t \ .$$

Sie stellt eine *ausführbare Grundanweisung* dar. Ihre Bedeutung ist festgelegt durch die Regel:

"*nach Ausführung von* $X{:=}t$ *ist* $|X|{=}w(t)$."

Vor Ausführung irgendeiner Wertzuweisung ist der Wert $|X|$ von X undefiniert: $|X| = \perp$.

<u>Beispiel 2.30:</u>

$X{:=}7$

$X{:=}(3-7)\cdot 9$

$P := \text{true}$

$Q{:=}\neg(\text{true} \vee \neg\text{false}) \vee \neg\neg\text{true}$

Eine genaue und angemessene Darstellung der Bedeutung von Variablen und (Grund-)Anweisungen sowie ihrer Ausführung ist verwickelter, als es zunächst scheinen mag. Eine vertiefende Behandlung dieses Themas soll dem Programmierkurs und Vorlesungen über die Semantik von Programmiersprachen vorbehalten bleiben.

Grundlage dazu ist der Begriff des *Zustands*, der die Information widerspiegelt, welche Variable gerade welchen Wert hat, und der durch die Ausführung von Anweisungen *transformiert* wird.

Ein einfaches (für wirkliche Programmiersprachen *zu* einfaches) Modell wollen wir dennoch betrachten, um das Prinzip anzudeuten:

Sei $\underline{X} = \{X_1, X_2, \ldots\}$ eine Menge von Variablen(-namen), von denen jede Werte aus der Wertemenge W haben kann (alle Variablen sind also "vom gleichen Typ"). Dann wollen wir als *Zustand* eine partielle Abbildung

$$Z : \underline{X} \longrightarrow W$$

verstehen. Z ordnet anschaulich jeder Variablen ihren "momentanen" Wert zu. Zu $Z:\underline{X}\longrightarrow W$, $X \in \underline{X}$ und $w \in W$ sei

$$Z\langle X \longleftarrow w \rangle : \underline{X} \longrightarrow W$$

die folgendermaßen definierte partielle Abbildung:

$$Z\langle X \longleftarrow w \rangle (Y) = \begin{cases} w, \text{ falls } X=Y \\ Z(Y) \text{ sonst} \end{cases}$$

Die *Bedeutung eines Anweisung* α ist eine *Zustandstransformation*, d.h. eine Abbildung

$$[\![\alpha]\!] : \ Z \longrightarrow Z$$

Die Bedeutung einer Wertzuweisung X:=t ist in diesem Modell darstellbar durch

$$[\![X:=t]\!] = Z\langle X \longleftarrow w(t) \rangle$$

Um Werte, die Variablen als Zwischenergebnisse zugewiesen wurden, später wiederverwenden zu können, muß man die Werte von Variablen "aufrufen" bzw. auslesen können. Dies geschieht im Rahmen von *Ausdrücken*, die i.w. den Termen einer applikativen Sprache entsprechen, jedoch mit folgendem wesentlichen Unterschied:

an der Stelle von Unbestimmten stehen Variable.

Die *Auswertung* solcher Terme ist *zustandsabhängig*: an die Stelle der Variablen wird ihr jeweiliger Wert im gegebenen Zustand gesetzt. Dann wird der Term wie üblich ausgewertet.

<u>Beispiel 2.31</u>: $2 \cdot X + 1$ Wert in Z: $2 \cdot Z(X) + 1$
etc.

Der so bestimmte Wert eines Terms $t(X_1, \ldots, X_n)$ mit Variablen sei mit $Z(t(X_1, \ldots, X_n))$ bezeichnet.

<u>Beispiel 2.32</u>: $Z(2 \cdot X + 1) = 2 \cdot Z(X) + 1$
etc.

Nun kann man auch Wertzuweisungen verwenden, auf deren rechter Seite ein Ausdruck mit Variablen steht:

$$X := t(X_1, \ldots, X_n)$$

Die Bedeutung ist dann allgemein:

$$[\![X := t(X_1, \ldots, X_n)]\!](Z) = Z\langle X \longleftarrow Z(t(X_1, \ldots, X_n))\rangle$$

Beispiel 2.33: $\alpha_1 = (X := 2 \cdot Y + 1)$ Bedeutung: $[\![\alpha_1]\!](Z) = Z\langle X \longleftarrow 2 \cdot Z(Y) + 1\rangle$

$\alpha_2 = (X := 2 \cdot X + 1)$ Bedeutung: $[\![\alpha_2]\!](Z) = Z\langle X \longleftarrow 2 \cdot Z(X) + 1\rangle$
ist wohldefiniert! nicht rekursiv!

Wertzuweisungen lassen sich nun auf verschiedene Art und Weise zu größeren, komplexeren Anweisungen zusammensetzen.

1. Folge (Sequenz): Sind α_1, α_2 Anweisungen, so ist $\alpha_1;\alpha_2$ eine Anweisung.

   Bedeutung: $[\![\alpha_1;\alpha_2]\!](Z) = [\![\alpha_2]\!]([\![\alpha_1]\!](Z))$

2. Auswahl (Selektion): Sind α_1, α_2 Anweisungen und B ein boolescher Ausdruck, so ist

   <u>if</u> B <u>then</u> α_1 <u>else</u> α_2 <u>fi</u>

   eine Anweisung.

   Bedeutung: $[\![\underline{\text{if}}\ B\ \underline{\text{then}}\ \alpha_1\ \underline{\text{else}}\ \alpha_2\ \underline{\text{fi}}]\!] = \begin{cases} [\![\alpha_1]\!](Z) & \text{falls } Z(B)=\text{true} \\ [\![\alpha_2]\!](Z) & \text{falls } Z(B)=\text{false} \end{cases}$

   Hier ist angenommen, daß Z(B) definiert ist; ansonsten ist die Bedeutung der Auswahl-Anweisung undefiniert.

3. Wiederholung (Iteration): Ist α eine Anweisung und B ein boolescher Ausdruck, so ist

   <u>while</u> B <u>do</u> α <u>od</u>

   eine Anweisung.

   Bedeutung: $[\![\underline{\text{while}}\ B\ \underline{\text{do}}\ \alpha\ \underline{\text{od}}]\!](Z) = \begin{cases} Z & \text{falls } Z(B)=\text{false} \\ [\![\underline{\text{while}}\ B\ \underline{\text{do}}\ \alpha\ \underline{\text{od}}]\!]([\![\alpha]\!](Z)) & \text{falls } Z(B)=\text{true} \end{cases}$

   Ist Z(B) undefiniert, so ist die Bedeutung der Wiederholungsanweisung wiederum undefiniert.

Wir beschränken uns hier auf diese Anweisungsarten, obwohl es in realen Programmiersprachen noch einige mehr gibt. Diese Anweisungen sind in gewisser Weise grundlegend: man kann damit "im Prinzip" alle Zustandstransformationen programmieren, die sich überhaupt programmieren lassen, d.h. unsere simple Programmiersprache ist bereits "universell" (s.u. 3.1).

Bemerkung: Man beachte die rekursive applikative Definition der Bedeutung der Wiederholungsanweisung. Insbesondere folgt, daß solche <u>while</u>-Schleifen nicht zu terminieren brauchen. In diesem Fall ist die Bedeutung undefiniert.

Wir beschränken uns hier wie im vorigen Abschnitt auf die beiden Datentypen <u>bool</u> und <u>int</u>. Das heißt, daß es Variable vom Typ <u>bool</u> und solche vom Typ <u>int</u> gibt, denen jeweils nur Werte ihres Typs zugewiesen werden können, und daß es Terme vom jeweiligen Typ gibt.

Die Variablen eines Programms vereinbaren wir am Anfang des Programms wie folgt:

<u>var</u> X,Y, . . . :<u>int</u>; P,Q, . . . :<u>bool</u>;

Außerdem legen wir am Anfang eines Programms fest, welche dieser Variablen wir als *Eingabevariablen* ansehen wollen, und am Ende legen wir fest, welche Variablen als *Ausgabevariablen* aufgefaßt werden sollen. Entsprechend notieren wir imperative Algorithmen in der folgenden Grundform:

⟨Programmname⟩:
    <u>var</u> X,Y, . . . :<u>int</u> ; P,Q, . . . :<u>bool</u>
    <u>input</u> $X_1, \ldots, X_n$;
        α;
    <u>output</u> $Y_1, \ldots, Y_m$.

Die *Bedeutung* eines solchen Programms sei wie folgt definiert. Dabei seien PROG der Programmname, $W_1, \ldots, W_n$ die Wertebereiche der Typen von $X_1, \ldots, X_n$, und $V_1, \ldots, V_m$ die Wertebereiche der Typen von $Y_1, \ldots, Y_m$.

$$[\![\mathrm{PROG}]\!]: \quad W_1 \times \ldots \times W_n \longrightarrow V_1 \times \ldots \times V_m$$

$$[\![\mathrm{PROG}]\!](w_1, \ldots, w_n) = (Z(Y_1), \ldots, Z(Y_m)),$$
wobei $Z = [\![\alpha]\!](Z_0)$,
wobei $Z_0(X_i) = w_i$, $1 \le i \le n$
und $Z_0(Y) = \bot$ für alle vereinbarten Variablen $Y \ne X_i$, $1 \le i \le n$.

Die Bedeutung eines imperativen Algorithmus ist also - wie im Falle applikativer Algorithmen - wiederum eine partielle Abbildung, nämlich von Eingabewerten in Ausgabewerte. (Bei den applikativen Algorithmen hatten wir uns auf m=1 (nur ein Ausgabewert) beschränkt.)

Nach diesen theoretischen Erörterungen betrachten wir nun einige Beispiele (vgl. ggf. applikative Lösungen):

<u>Beispiel 2.34</u>: Fakultät-Funktion $0! = 1$, $x! = x \cdot (x-1)!$ für $x > 0$

FAC: <u>var</u> X,Y:<u>int</u>;
    <u>input</u> X;
    Y:=1; <u>while</u> X>1 <u>do</u> Y:=Y·X; X:=X-1 <u>od</u>
    <u>output</u> Y.

Hier ist $[\![FAC]\!](x) = \begin{cases} x! \text{ für } x \geq 0 \\ \bot \text{ sonst} \end{cases}$

Setzt man als Bedingung für die <u>while</u>-Schleife "$X \neq 0$", so erhält man die Version

$$[\![FAC]\!](x) = \begin{cases} x! \text{ für } x \geq 0 \\ \bot \text{ sonst} \end{cases}$$

<u>Beispiel 2.35</u>: Fibonacci-Zahlen

```
FIB:  var X,A,B,C: int;
      input X;
      A:=1; B:=1;
      while X>0 do
            C:=A+B; A:=B; B:=C; X:=X-1 od
      output A
```

$$[\![FIB]\!](x) = \begin{cases} \text{die x-te Fibonacci-Zahl, falls } x \geq 0 \\ \bot \qquad \text{sonst} \end{cases}$$

<u>Beispiel 2.36</u>: Größter gemeinsamer Teiler, Version 1:

```
GGT1: var X,Y: int
      input X,Y;
      while X≠Y do
            if X>Y then X:=X-1
                   else Y:=Y-X fi od;
      output X.
```

$$[\![GGT1]\!](x,y) = \begin{cases} ggT(x,y), & \text{falls } x,y>0 \\ x & \text{falls } x=y \\ \bot & \text{falls } x \neq y \text{ und } x \leq 0 \text{ oder } y \leq 0 \end{cases}$$

<u>Beispiel 2.37</u>: Größter gemeinsamer Teiler, Version 2:

```
GGT2: var X,Y: int;
      input X,Y;
      while X≠Y do
            while X>Y do X:=X-Y od;
            while X<Y do Y:=Y-X od od;
      output X.
```

$[\![GGT2]\!] = [\![GGT1]\!]$, aber GGT2 ist "meistens" effizienter (warum?)

Beispiel 2.38: Größter gemeinsamer Teiler, Version 3:

```
GGT3: var X,Y,R: int;
      input X,Y;
      R:=1;
      while R≠0 do
            R:=X mod Y; X:=Y; Y:=R od;
      output X.
```

Einige Berechnungen für GGT3:

X	Y	R
19	5	1
5	4	4
4	1	1
$\boxed{1}$	0	0

X	Y	R	
2	100	1	ist X>Y, so wird erst
100	2	2	vertauscht : $X \longleftrightarrow Y$
$\boxed{2}$	0	0	

Wie ist das Verhalten für negative X oder Y?

$$[\![GGT3]\!](x,y) = \begin{cases} ggT(x,y), & \text{falls } x,y>0 \\ y, & \text{falls } x=y\neq 0 \text{ oder } x=0, y\neq 0 \\ \perp, & \text{falls } y=0 \\ ggT(|x|,y), & \text{falls } x<0 \text{ und } y>0 \\ -ggT(|x|,|y|), & \text{falls } y<0 \end{cases}$$

Ist GGT3 effizienter als die Versionen 1 und 2?

Beispiel 2.39: Was tut der folgende Algorithmus?

```
XYZ: var W,X,Y,Z: int;
     input X;
     Z:=0; W:=1; Y:=1;
     while W≤X do
           Z:=Z+1; W:=W+Y+2; Y:=Y+2 od;
     output Z.
```

Einige Berechnungen für XYZ:

W	X	Y	Z
1	0	1	$\boxed{0}$
1	1	1	0
4		3	$\boxed{1}$
1	2	1	0
4		3	$\boxed{1}$
1	3	1	0
4		3	$\boxed{1}$

Im Abschnitt 3.2 wird verraten und dann auch bewiesen, was [[XYZ]] ist (bis dahin: raten!).

Es gibt viele weitere Konzepte zur Formulierung imperativer Algorithmen, z.B.

- Blockkonzept
- Prozedurkonzept, Rekursion
- Modulkonzept

  etc.

Diese sollen in anderen Veranstaltungen (wie z.B. dem Programmierkurs) behandelt werden. Wir beschränken uns hier auf einfache Grundprinzipien.

2.6 Deduktive Algorithmen

Dies ist nach den applikativen und den imperativen die dritte Art von Algorithmen, die als Grundlage von Programmiersprachen wesentliche Bedeutung erlangt haben. Deduktive Algorithmen bilden den Kern der Programmiersprache PROLOG ("PROgramming in LOGic") und ihrer Erweiterungen, die ihrerseits wiederum eine große Rolle spielen beim Aufbau von wissensbasierten Systemen und Expertensystemen. PROLOG beruht auf formaler Logik, d.h. auf Aussagen und ihren logischen Verknüpfungen, sowie auf der Herleitung (Deduktion) neuer Aussagen aus vorhandenen aufgrund logischer Herleitungsregeln.

Aussagen werden in natürlicher Sprache durch Sätze beschrieben, von denen es sinnvoll ist zu sagen, sie seien *wahr* oder *falsch*.

Beispiel 2.40: Folgende Sätze beschreiben Aussagen:

1. Susi ist eine Tochter von Petra.
2. Susi ist eine Enkelin von Rita.
3. Töchter von Töchtern sind Enkelinnen.
4. Mütter sind Vorfahren ihrer Töchter.
5. Vorfahren der Mütter sind auch Vorfahren ihrer Töchter.
   usw.

Folgende Sätze beschreiben keine Aussagen:

6. Hat Petra eine Tochter? (Dies ist eine *Frage*)
7. Sage mir, ob Petra eine Tochter hat! (Dies ist eine *Aufforderung*)
   usw.

Wie bei den applikativen Algorithmen verwenden wir *Unbestimmte*

$$X,Y,Z, \ldots, X_0,X_1,X_2, \ldots,$$

um "Aussagen mit Unbestimmten" auch *Aussageformen* genannt, zu beschreiben.

Beispiel 2.41: Folgende Sätze beschreiben Aussageformen:

1. X ist eine Tochter von Y.
2. X ist eine Enkelin von Z.
3. Wenn X eine Tochter von Y und Y eine Tochter von Z ist, dann ist X eine Enkelin von Z.
4. Wenn X eine Tochter von Y ist, dann ist Y ein Vorfahr von X.
5. Wenn X eine Tochter von Y und Z ein Vorfahr von Y, so ist Z ein Vorfahr von X.
   usw.

Aus Aussageformen entstehen Aussagen, wenn man Unbestimmte mit konkreten Objekten *belegt* (d.h. letztere für die ersteren einsetzt).

Beispiel 2.42: Folgende Belegungen überführen die ersten beiden Aussageformen im Beispiel 2.41 in die entsprechenden Aussagen im Beispiel 2.40:

1. $X \mapsto$ Susi, $Y \mapsto$ Petra
2. $X \mapsto$ Susi, $Z \mapsto$ Rita

In den Fällen 3 bis 5 entstehen die entsprechenden Aussagen jedoch nicht durch Belegung mit konkreten Objekten, sondern durch *Generalisierung* der Unbestimmten: die Aussagen drücken allgemeine Regeln aus, und diese gewinnt man aus den Aussageformen, indem man die Floskel "Für alle X,Y, . . . gilt:" voranstellt, und zwar für alle vorkommenden Unbestimmten.

Beispiel 2.43: Aus "X ist sterblich" entsteht durch Generalisierung "Für alle X gilt: X ist sterblich" oder, in sprachlich gefälligerer Form, "Alle (Menschen?) sind sterblich".

Bemerkung: Belegt bzw. generalisiert man nur einen Teil der Unbestimmten, so entsteht wieder eine Aussageform. Zum Beispiel entsteht aus "X ist eine Tochter von Y" durch die Belegung $X \mapsto$ Susi die Aussageform "Susi ist eine Tochter von Y" mit nur mehr einer Unbestimmten. Aus "Wenn X ein Kind von Y ist, dann ist X eine Tochter von Y" entsteht durch Generalisierung von X - nach Umformung in einen sprachlich gefälligeren Satz - die Aussageform "alle Kinder von Y sind Töchter". Man kann auch einen Teil der Unbestimmten belegen und einen anderen Teil generalisieren. Zum Beispiel entsteht aus "X ist Vorfahr von Y" durch die Belegung $X \mapsto$ Eva und Generalisierung von Y die Aussage "Eva ist Vorfahr aller (Menschen?)" (wiederum nach sprachlicher Umformung).

Unklar ist nun, was "konkrete Objekte" eigentlich sind, mit denen man die Unbestimmten belegen darf, bzw. welche Objekte mit "Für alle X . . ." eigentlich gemeint sind. Um dies zu präzisieren, ordnen wir (wie bei den applikativen Algorithmen) jeder Unbestimmten X einen *Typ* zu und vereinbaren, daß X nur durch die Werte ihres Typs belegt werden darf, bzw. daß bei "Für alle X . . . " genau die Werte des Typs von X gemeint sind.

Als Typen kommen grundsätzlich int, bool, real, character, string, etc. in Frage (s.o. Abschnitt 2.2). Um die Dinge jedoch nicht unnötig kompliziert zu machen und um uns leichter auf das wesentliche konzentrieren zu können, kümmern wir uns hier nicht weiter um die Typen. In konkreten Beispielen verwenden wir wie oben Namen oder Worte der deutschen Sprache, je nach Bedarf jedoch auch Zahlen, Wahrheitswerte, usw.

Eine nähere Betrachtung der obigen Beispiele zeigt, daß es einfachere und kompliziertere Aussagen und Aussageformen gibt. Wir nennen eine Aussage(form) *zusammengesetzt*, wenn sie eine andere Aussage(form) als echten Bestandteil enthält. Andernfalls nennen wir sie *atomar*. In den obigen Beispielen beschreiben nur die Fälle 1 und 2 atomare Aussagen bzw. Aussageformen, alle anderen sind zusammengesetzt.

In der formalen Logik werden Aussagen und Aussageformen durch Formeln beschrieben, deren Syntax und Semantik präzise definiert wird. Wir beschränken uns hier auf eine einfache "Logik der Fakten und Regeln" (Horn-Logik), die den Kern von PROLOG bildet.

Definition 4.44: Logik der Fakten und Regeln (Syntax):

Alphabet:	Unbestimmte $X,Y,Z,\ldots,X_0,X_1,\ldots$
	Konstante $a,b,c,\ldots,a_0,a_1,\ldots$,
	Prädikatensymbole $P,Q,R,\ldots,P_0,P_1,\ldots$,
	jedes P hat eine Stelligkeit $\sigma(P)\in\mathbb{N}$,
	Logische Operatoren $\wedge$ $\Rightarrow$
	Sonderzeichen ( ) .
Atomare Formeln	sind Zeichenreihen der Art $P(t_1,\ldots,t_n)$, wobei die t_i, $1\le i\le n$, Unbestimmte oder Konstante sind, und wobei P ein Prädikatensymbol der Stelligkeit $\sigma(P)=n$ ist.
Fakten	sind atomare Formeln ohne Unbestimmte, bei denen also alle t_i, $1\le i\le n$, Konstante sind.
Regeln	sind Zeichenreihen der Art $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m \Rightarrow \alpha_0$, wobei die α_j, $0\le j\le m$, atomare Formeln sind.

Bemerkung: In der formalen Sprache dieser Logik stellen atomare Formeln die atomaren Aussageformen und Fakten speziell die atomaren Aussagen dar. In den Regeln liest man "$\wedge$" als "und" und "$\Rightarrow$" als "wenn ... , dann". Regeln stellen recht spezielle zusammengesetzte Aussageformen dar. In der formalen Logik werden weitaus mehr Arten der Zusammensetzung von Formeln betrachtet. Wir gehen hier nicht weiter darauf ein.

Beispiel 2.45: Gegeben seien

X,Y,Z	als Unbestimmte,
Susi, Petra, Rita	als Konstante,
TOCHTER, ENKELIN, VORFAHR	als 2-stellige Prädikatensymbole.

Dann sind

Fakten: 1. TOCHTER(Susi,Petra)

2. ENKELIN(Susi,Rita)

Regeln: 3. TOCHTER(X,Y) $\wedge$ TOCHTER(Y,Z) $\Rightarrow$ ENKELIN(X,Z)

4. TOCHTER(X,Y) $\Rightarrow$ VORFAHR(Y,X)

5. TOCHTER(X,Y) $\wedge$ VORFAHR(Z,Y) $\Rightarrow$ VORFAHR(Z,X)

Diese Fakten und Regeln formalisieren die Aussagen im Beispiel 2.40, wenn wir die Regeln im Geiste mit "Für alle . . . " generalisieren.

Die *Semantik* einer formalen Logik wird allgemein mit Hilfe mathematischer Strukturen präzisiert. Wir verzichten hier darauf. Für unsere einfache Logik der Fakten und Regeln genügt ein intuitives Verständnis: wir nehmen an, daß wir die Bedeutung jedes gegebenen Prädikatensymbols und der damit gebildeten atomaren Formeln und Fakten kennen, und wir nehmen an, daß uns die Bedeutung von $\wedge$ ("und") und $\Rightarrow$ ("wenn . . . ,dann") sowie der damit gebildeten Regeln hinreichend gut bekannt sind.

Regeln stellen nun nicht nur – durch Generalisierung – Aussagen dar, sondern bilden auch einen Mechanismus, mit dem man aus bekannten Fakten neue Fakten *folgern* oder *herleiten* kann. Man belegt dazu alle vorkommenden Unbestimmten mit konkreten Objekten. Stehen dann auf der linken Seite von "$\Rightarrow$" nur bekannte Fakten, so darf man das Faktum auf der rechten Seite folgern (diese Schlußweise heißt in der Logik "Modus Ponens").

Beispiel 2.46: Aus den Fakten

TOCHTER(Susi,Petra)
TOCHTER(Petra,Rita)

und der Regel

TOCHTER(X,Y) $\wedge$ TOCHTER(Y,Z) $\Rightarrow$ ENKELIN(X,Z)

läßt sich mittels der Belegung X $\longrightarrow$ Susi, Y $\longmapsto$ Petra, Z $\longmapsto$ Rita das Faktum

ENKELIN(Susi,Rita)

herleiten.

Die linke Seite einer Regel $\alpha_1 \wedge \ldots \wedge \alpha_m \Rightarrow \alpha_0$, d.h. der Teil $\alpha_1 \wedge \ldots \wedge \alpha_m$ vor dem $\Rightarrow$, kann auch leer sein (m=0). Auch können in α_0 Unbestimmte auftreten, die in der linken Seite nicht vorkommen. Diese sind dann als generalisiert aufzufassen.

Beispiel 2.47: Die Regel

$\Rightarrow$ VORFAHR(Eva,X)

besagt, daß Eva Vorfahr aller (Menschen) X ist.

Fakten lassen sich somit als spezielle Regeln auffassen: das Faktum $P(a_1, \ldots, a_n)$ bedeutet dasselbe wie die Regel $\Rightarrow P(a_1, \ldots, a_n)$.

Durch Angabe der Unbestimmten, Konstanten und Prädikatensymbole wird eine bestimmte Logik L festgelegt.

Definition 2.48: Ein *deduktiver Algorithmus D in L* ist eine Menge von Fakten und Regeln in L.

Bemerkung: Ein deduktiver Algorithmus wird zuweilen auch "Wissensbasis" genannt.

Aus einem deduktiven Algorithmus D lassen sich, wie oben angedeutet, neue Fakten herleiten. Dazu werden "Anfragen" an D gestellt, die dann durch einen allgemeinen Algorithmus zur Auswertung von Anfragen "beantwortet" werden. Bevor wir diesen behandeln können, sind einige Vorbereitungen nötig.

Seien F_0 die Fakten in D. Aus D lassen sich nun (mittels "Modus Ponens", s.o.) neue Fakten herleiten. Die Menge dieser neuen Fakten sei F_1. Nun lassen sich aus $D \cup F_1$ wieder neue Fakten herleiten. Diese Menge sei F_2. Aus $D \cup F_1 \cup F_2$ leiten wir wiederum neue Fakten F_3 her, aus $D \cup F_1 \cup F_2 \cup F_3$ neue Fakten F_4 usw. Diesen Prozeß führen wir fort, bis keine neuen Fakten mehr entstehen (dies muß nicht nach endlich vielen Schritten der Fall sein). Auf diese Weise erweitern wir F_0 zur Menge aller aus D (direkt oder indirekt über mehrere Zwischenschritte) herleitbaren Fakten.

Definition 2.49: Mit F(D) bezeichnen wir alle (direkt oder indirekt) aus D herleitbaren Fakten,

$$F(D) := F_0 \cup F_1 \cup F_2 \cup \ldots$$

Bemerkung: Es ist vielleicht nicht offensichtlich, daß der obige Prozeß zu einer wohldefinierten Menge F(D) führt. Ohne dies näher auszuführen, geben wir hier den Hinweis, daß F(D) sich auch als kleinste Menge von Fakten definieren läßt, die F_0 enthält und gegen Herleitung in D "abgeschlossen" ist (d.h. daß aus F(D) nur Fakten hergeleitet werden können, die bereits in F(D) sind).

Definition 2.50: Sei D ein deduktiver Algorithmus in der Logik L. Eine *Anfrage* γ *an D* ist eine Konjunktion ($\wedge$-Verknüpfung)

$$\gamma = \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m$$

von atomaren Formeln α_i, $1 \le i \le m$.

Bemerkung: Im allgemeinen werden wesentlich komplexer zusammengesetzte Formeln als Anfragen zugelassen, aber wir beschränken uns hier auf diesen einfachen Fall.

Definition 2.51: Sei $\gamma = \alpha_1 \wedge \ldots \wedge \alpha_m$ eine Anfrage an D. Eine *Antwort* auf γ ist dann eine Belegung β der Unbestimmten in γ, mit der aus allen α_i, $1 \le i \le m$, Fakten in F(D) entstehen (die also aus D herleitbar sind).

Es kann keine, eine oder auch mehrere Antworten auf eine Anfrage geben, auch unendlich viele. Enthält γ keine Unbestimmten, so gibt es höchstens eine Antwort, nämlich die leere Belegung. Ist diese eine Antwort, so nennen wir sie auch "ja", andernfalls "nein".

Beispiel 2.52: Wir legen die natürlichen Zahlen $\mathbb{N}=\{0,1,2, \ldots\}$ als Konstante zugrunde. Die vorkommenden Unbestimmten können nur mit Werten aus $\mathbb{N}$ belegt werden. Als Prädikatensymbole seien

SUC ("Successor", 2-stellig) und
ADD (Addition, 3-stellig)

gegeben. Dies legt die Logik L (hinreichend, d.h. bis auf die Namen der Unbestimmten) fest.

Der deduktive Algorithmus (die "Wissensbasis") NAT sei gegeben durch

Fakten SUC(n,n+1) für alle $n \in \mathbb{N}$

Regeln (1) $\Rightarrow$ ADD(X,0,X)

(2) ADD(X,Y,Z) $\wedge$ SUC(Y,V) $\wedge$ SUC(Z,W) $\Rightarrow$ ADD(X,V,W)

ADD(X,Y,Z) steht für X+Y=Z. Wir stellen einige Anfragen.

1. Anfrage ADD(3,2,5)?

Zunächst folgt aus Regel (1) mittels der Belegung $X \mapsto 3$:

ADD(3,0,3).

Hieraus, aus der Regel (2) mittels der Belegung $X \mapsto 3$, $Y \mapsto 0$, $Z \to 3$ sowie den Fakten SUC(0,1), SUC(3,4) folgt

ADD(3,1,4).

Hieraus, aus der Regel (2) mittels der Belegung $X \mapsto 3$, $Y \mapsto 1$, $Z \to 4$ sowie den Fakten SUC(1,2), SUC(4,5) folgt

ADD(3,2,5).

Die Antwort auf die Anfrage ist also "ja".

2. Anfrage ADD(3,2,X)?

Die obige Herleitung zeigt, daß die Belegung $X \mapsto 5$ ein Faktum in F(NAT) ergibt, und zwar nur diese. Die einzige Antwort ist also $X \mapsto 5$.

3. Anfrage ADD(3,X,5)?

Die obige Herleitung zeigt, daß die Belegung $X \mapsto 2$ ein Faktum in F(NAT) ergibt, und zwar nur diese. Die einzige Antwort ist also $X \mapsto 2$.

4. Anfrage ADD(X,Y,5)?

Die obige Herleitung zeigt, daß die Belegung $X \mapsto 3$, $Y \mapsto 2$ ein Faktum in F(NAT) ergibt, aber nicht nur diese! Antworten sind genau die folgenden Belegungen:

$X \mapsto$	0	1	2	3	4	5
$Y \mapsto$	5	4	3	2	1	0

5. Anfrage ADD(X,Y,Z)?

Die Antworten sind alle Belegungen $X \mapsto a$, $Y \mapsto b$, $Z \mapsto c$, so daß $a+b=c$ ist. Diese Anfrage hat also unendlich viele Antworten.

Der deduktive Algorithmus NAT "berechnet" also die Addition auf den natürlichen Zahlen, anders ausgedrück: diese "Wissensbasis" versorgt uns mit dem "Wissen" über die Addition natürlicher Zahlen. Darin ist implizit auch das "Wissen" über die Subtraktion natürlicher Zahlen enthalten, wie z.B. die 3. Anfrage zeigt.

Auswertung von Anfragen

Die obigen Beispiele geben keinen Hinweis darauf, wie man die Antworten auf eine Anfrage systematisch berechnen kann. Hierzu verfolgen wir die Herleitung gewissermaßen rückwärts und gehen von der gestellten Anfrage aus. Wir zeigen die Grundidee anhand des obigen Beispiels, bevor wir den allgemeinen Algorithmus skizzieren.

Beispiel 2.53: *Fakten* SUC(n,n+1) für alle $n \in \mathbb{N}$

Regeln (1) $\Rightarrow$ ADD(X,0,X)

(2) ADD(X,Y,Z) $\wedge$ SUC(Y,V) $\wedge$ SUC(Z,W) $\Rightarrow$ ADD(X,V,W)

1. Anfrage ADD(3,2,5)?

Dies könnte nur mit Regel (2) *und der*

Belegung $X \mapsto 3$, $V \mapsto 2$, $W \mapsto 5$

hergeleitet werden, sofern die linke Seite der Regel mit dieser Belegung gilt, d.h. sofern die

Anfrage ADD(3,Y,Z) $\wedge$ SUC(Y,2) $\wedge$ SUC(Z,5)?

die Antwort "ja" ergibt. Wegen der bekannten SUC-Fakten müßte dazu die

Belegung $Y \mapsto 1$, $Z \mapsto 4$

genommen werden, und dies heißt, daß die

Anfrage ADD(3,1,4)

mit "ja" beantwortet werden müßte. Dies könnte wieder nur mit Regel (2) *und der*

Belegung $X \mapsto 3$, $V \mapsto 1$, $W \mapsto 4$

hergeleitet werden, sofern die Antwort auf die

Anfrage ADD(3,Y,Z) ∧ SUC(Y,1) ∧ SUC(Z,4)

"ja" lauten würde. Dazu müßte die

Belegung Y ↦ 0, Z ↦ 3

genommen werden, d.h. die

Anfrage ADD(3,0,3)

müßte mit "ja" beantwortet werden.

Dies ist aufgrund der Regel (1) der Fall. Daraus folgt nun rückwärts, daß auch alle Zwischenanfragen mit "ja" zu beantworten sind und damit auch die anfangs gestellte Anfrage.

<u>2. Anfrage</u> ADD(3,2,X)?

Wir kürzen die Zwischentexte durch das Symbol "↑" ab, das andeuten soll, daß die folgende (Zwischen-)Anfrage beantwortet werden muß, um dann mit der angegebenen Belegung eine Antwort auf die vorherige (Zwischen-)Anfrage herzuleiten. Hinter dem Symbol ↑ notieren wir die angewandten Regeln bzw. die geltenden Fakten. Namenskollisionen bei Unbestimmten werden durch Umbenennung beseitigt.

↑ Regel (2)

Belegung X ↦ 3, V ↦ 2, W ↦ X'

Anfrage ADD(3,Y,Z) ∧ SUC(Y,2) ∧ SUC(Z,X')

↑ SUC(1,2)

Belegung Y ↦ 1

Anfrage ADD(3,1,Z) ∧ SUC(Z,X')

↑ Regel (2)

Belegung X ↦ 3, V ↦ 1 W ↦ Z'

Anfrage ADD(3,Y,Z) ∧ SUC(Y,1) ∧ SUC(Z,Z') ∧ SUC(Z',X')

↑ SUC(0,1)

Belegung Y ↦ 0

Anfrage ADD(3,0,Z) ∧ SUC(Z,Z') ∧ SUC(Z',X')

↑ Regel (1), SUC(3,4), SUC(4,5)

Belegung Z ↦ 3, Z' ↦ 4, X' ↦ 5

Damit ist die Antwort "ja".

Eine Antwort auf die Anfrage ADD(3,2,X) ist somit X ↦ 5 (in diesem Fall die einzige).

<u>3. Anfrage</u> ADD(3,X,5)?

↑ Regel (2)

Belegung X ↦ 3, V ↦ X', W ↦ 5

Anfrage ADD(3,Y,Z) ∧ SUC(Y,X') ∧ SUC(Z,5)

↑ SUC(4,5)

Belegung Z ↦ 4

Anfrage $ADD(3,Y,4) \wedge SUC(Y,X')$

↑ **Regel** (2)

Belegung $X \rightarrow 3,\ V \mapsto Y',\ W \mapsto 4$

Anfrage $ADD(3,Y,Z) \wedge SUC(Y,Y') \wedge SUC(Z,4) \wedge SUC(Y',X')$

↑ $SUC(3,4)$

Belegung $Z \mapsto 3$

Anfrage $ADD(3,Y,3) \wedge SUC(Y,Y') \wedge SUC(Y',X')$

↑ **Regel** (1), $SUC(0,1)$, $SUC(1,2)$

Belegung $Y \rightarrow 0,\ Y' \mapsto 1,\ X' \rightarrow 2$

Damit ist die Antwort "ja".

Eine Antwort auf die Anfrage $ADD(3,X,5)$ ist somit $X \rightarrow 2$ (in diesem Fall die einzige).

<u>4. Anfrage</u> $ADD(X,Y,5)$?

↑ **Regel** (1)

Belegung $X \rightarrow 5,\ Y \rightarrow 0$

Dies ist eine Antwort, jedoch nicht die einzige.

Anstelle von Regel (1) *läßt sich auch* Regel (2) *anwenden:*

Regel (2)

Belegung $V \mapsto Y',\ W \rightarrow 5$

Anfrage $ADD(X,Y,Z) \wedge SUC(Y,Y') \wedge SUC(Z,5)$

↑ $SUC(4,5)$

Belegung $Z \mapsto 4$

Anfrage $ADD(X,Y,4) \wedge SUC(Y,Y')$

↑ ↑ Regel (1), $SUC(0,1)$ *(2 Schritte zusammengefaßt)*

Belegung $X \mapsto 4,\ Y \mapsto 0,\ Y' \mapsto 1$

Dies ergibt eine Antwort ($X \mapsto 4,\ Y \mapsto 1$), jedoch gibt es noch weitere:

Regel (2), $SUC(3,4)$ *(2 Schritte zusammengefaßt)*

Belegung $V \mapsto Y'',\ W \mapsto 4,\ Z \rightarrow 3$

Anfrage $ADD(X,Y,3) \wedge SUC(Y,Y'') \wedge SUC(Y'',Y')$

↑ ↑ Regel (1), $SUC(0,1)$, $SUC(1,2)$ *(3 Schritte)*

Belegung $X \rightarrow 3,\ Y \mapsto 0,\ Y'' \mapsto 1,\ Y' \mapsto 2$

Dies ergibt eine Antwort ($X \mapsto 3,\ Y \mapsto 2$), jedoch gibt es noch weitere:

Regel (2), $SUC(2,3)$ *(2 Schritte)*

Belegung $V \mapsto Y''',\ W \rightarrow 3,\ Z \mapsto 2$

Anfrage $ADD(X,Y,2) \wedge SUC(Y,Y''') \wedge SUC(Y''',Y'') \wedge SUC(Y'',Y')$

↑ ↑ Regel (1), $SUC(0,1)$, $SUC(1,2)$, $SUC(2,3)$ *(4 Schritte)*

Belegung $X \mapsto 2,\ Y \mapsto 0,\ Y''' \mapsto 1,\ Y'' \rightarrow 2,\ Y' \mapsto 3$

Dies ergibt eine Antwort ($X \rightarrow 2,\ Y \rightarrow 3$), jedoch gibt es noch weitere:

Regel (2), $SUC(1,2)$ *(2 Schritte)*

Belegung $V \mapsto Y^{IV},\ W \mapsto 2,\ Z \mapsto 1$

Anfrage $ADD(X,Y,1) \wedge SUC(Y,Y^{IV}) \wedge SUC(Y^{IV},Y''') \wedge SUC(Y''',Y'') \wedge SUC(Y'',Y')$

$\wedge$ Regel (1), SUC(0,1), ... SUC(3,4) *(5 Schritte)*

Belegung $X \mapsto 1$, $Y \mapsto 0$, $Y^{iv} \mapsto 1$, $Y''' \mapsto 2$, $Y'' \mapsto 3$, $Y' \mapsto 4$

Dies ergibt eine Antwort ($X \mapsto 1$, $Y \mapsto 4$), jedoch gibt es noch weitere:

Regel (2), SUC(0,1) *(2 Schritte)*

Belegung $V \mapsto Y^{v}$, $W \mapsto 1$, $Z \mapsto 0$

Anfrage $ADD(X,Y,0) \wedge SUC(Y,Y^{v}) \wedge SUC(Y^{v},Y^{iv}) \wedge SUC(Y^{iv},Y''') \wedge SUC(Y''',Y'')$
$\wedge\ SUC(Y'',Y')$

$\wedge$ Regel (1), SUC(0,1), ... SUC(4,5) *(6 Schritte)*

Belegung $X \mapsto 0$, $Y \mapsto 0$, $Y^{v} \mapsto 1$, $Y^{iv} \mapsto 2$, $Y''' \mapsto 3$, $Y'' \mapsto 4$, $Y' \mapsto 5$

Dies ergibt eine Antwort ($X \mapsto 0$, $Y \mapsto 5$). Gibt es noch weitere?

Regel (2)

Belegung $V \mapsto Y^{vi}$, $W \mapsto 0$

Anfrage $ADD(X,Y,Z) \wedge SUC(Y,Y^{vi}) \wedge SUC(Z,0) \wedge SUC(Y^{vi},Y^{v}) \wedge SUC(Y^{v},Y^{iv}) \wedge$
$\ldots \wedge SUC(Y'',Y')$

Dies ergibt keine weiteren Antworten, da SUC(Z,0) für keine Belegung von Z ein Faktum in F(NAT) ergibt.

5. Anfrage ADD(X,Y,Z)?

Eine Schar von Antworten gibt unmittelbar Regel (1), nämlich alle Belegungen $X \mapsto a$, $Y \mapsto 0$, $Z \mapsto a$ für $a \in N$. Mit Regel (2) können weitere Antworten hergeleitet werden, insgesamt unendlich viele. Wir verzichten auf die Ausführung.

Der allgemeine *Algorithmus zur Auswertung von Anfragen* läßt sich wie folgt skizzieren. Hierzu ist es vorteilhaft, die Fakten $P(a_1, \ldots, a_n)$ als Regeln $\Rightarrow P(a_1, \ldots, a_n)$ mit leerer linker Seite aufzufassen. "Von hinten rückwärts verfolgte" Herleitungen bezeichnen wir als *Reduktionen*.

```
begin γ:= Anfrage
      while weitere Reduktionen möglich do
            angleichen;
            reduzieren
      od
end
```

Die Bedingung "weitere Reduktionen möglich" läßt sich durch "γ ist nicht leer" präzisieren, d.h. $\gamma = \alpha_1 \wedge \ldots \wedge \alpha_m$ mit $m \geq 1$. Dies wurde in unserem Beispiel nicht deutlich: wir haben bereits abgebrochen, wenn in γ nur noch bekannte Fakten vorkamen. Diese Bedingung ist jedoch gleichwertig, denn wenn nur noch m bekannte Fakten vorkommen, so lassen sich diese mit m weiteren Schritten (s.u.) vollkommen entfernen, so daß dann γ leer ist.

Die Operation "angleichen" besteht darin, eine Belegung β zu wählen, die eine atomare Formel in γ, etwa α_k, mit der rechten Seite einer Regel gleich macht. Dieser Schritt ist nichtdeterministisch!

Die Operation "reduzieren" besteht darin, γ durch β zu belegen und dann α_k durch die rechte Seite der im Schritt "angleichen" gewählten Regel zu ersetzen. Dadurch entsteht ein neues γ für den nächsten Schritt. Ist die Regel ein Faktum, so wird α_k durch die leere linke Seite ersetzt, also entfernt.

Die Antworten zur Anfrage γ gewinnt man nun aus all den Reduktionen (Herleitungen rückwärts)

$$\gamma = \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_p,$$

die zu einer leeren Konjunktion γ_p führen, und zwar aus der bei dieser Reduktion gewählten Belegung für die Unbestimmten in γ. Enthält γ keine Unbestimmten, so ist die Antwort genau dann "ja", wenn es eine solche Herleitung mit leerem γ_p gibt.

Bemerkung: Es kann auch unendlich lange Reduktionen geben, die niemals zu einer leeren Konjunktion γ_p führen, so daß die Auswertung der Anfrage nicht terminiert. Im obigen Beispiel 2.53 trat dies bei der 4. Anfrage am Schluß auf, als die atomare Formel SUC(Z,0) auftrat, die mit keiner Belegung für Z mehr entfernt werden konnte. Wünschenswert wäre eine verfeinerte Abbruchbedingung in unserem Auswerte-Algorithmus, der solche Fälle erkennt.

Bemerkung: Wir haben als Argumente für die atomaren Formeln nur Unbestimmte und Konstante zugelassen. Unser Algorithmus funktioniert im Prinzip ganz ähnlich, wenn wir hier beliebige Terme mit Funktionen (wie bei den applikativen Algorithmen) zulassen, die Operation "angleichen" (allgemein "unifizieren" genannt) wird dann jedoch viel komplizierter.

Der Auswerte-Algorithmus in der obigen Form ist weder deterministisch noch determiniert, und er braucht nicht unbedingt zu terminieren. Ein terminierender Durchlauf durch diesen Algorithmus ergibt jeweils nur *eine* Antwort. Um *alle* Antworten zu bekommen, wäre ein *determinierter* Algorithmus nötig, der systematisch alle terminierenden Durchläufe durch unseren nichtdeterminierten Algorithmus verfolgt. Dieser determinierte Algorithmus müßte - soweit möglich - auch nichtterminierende Durchläufe durch unseren Algorithmus rechtzeitig erkennen und als erfolglos abbrechen. Ein solcher Algorithmus ist zu schwierig und komplex, um ihn hier zu behandeln. Wie jeder determinierte Algorithmus bezeichnet er eine partielle Ein-/Ausgabefunktion

$$f\colon \text{Anfragen} \longrightarrow \text{Antwortmengen},$$

wobei "Anfragen" die Menge der syntaktisch korrekten Anfragen ist und "Antwortmengen" die Menge der Antwortmengen: jeder Anfrage γ wird die Menge aller Belegungen β zugeordnet, so daß γ belegt mit β nur herleitbare Fakten enthält.

3. Theorie der Algorithmen

3.1 Berechenbarkeit

Wir gehen hier zunächst der Frage nach, welche Funktionen sich überhaupt durch Algorithmen – applikative, imperative, deduktive oder andere – berechnen lassen.

Gibt es z.B. Funktionen, die sich nicht durch irgendeinen Algorithmus berechnen lassen?

Daß dies tatsächlich der Fall sein muß, läßt sich unschwer einsehen. Wir benutzen hierzu nur eine – sicherlich unabdingbare – Eigenschaft eines jeden Algorithmus, nämlich:

Jeder Algorithmus läßt sich durch einen endlichen Text
über einem festen, endlichen Alphabet beschreiben.

Sei $A=\{a_1, \ldots, a_n\}$ ein Alphabet mit der alphabetischen Ordnung $a_1<a_2<\ldots<a_n$. Mit A^* bezeichnen wir die Menge der *Texte* (Zeichenketten, endlichen Folgen, Worte) über A:

$$A^* = \{ \varepsilon, a_1, \ldots, a_n, a_1a_1, a_1a_2, \ldots, a_5a_3a_3a_1a_5a_2, \ldots \}$$

Hierbei ist ε der leere Text der Länge 0

Wir können A^* nun auflisten, und zwar

1. der Länge nach. Zu jeder Länge l gibt es n^l verschiedene Texte über A, also endlich viele.

2. lexikographisch innerhalb der Text gleicher Länge:

$$\begin{aligned} b_1b_2\ldots b_l < c_1c_2\ldots c_l \;:\Leftrightarrow\; & b_1 < c_1 \\ & \vee (b_1=c_1 \wedge b_2<c_2) \\ & \vee (b_1=c_1 \wedge b_2=c_2 \wedge b_3<c_3) \\ & \ldots \\ & \vee (b_1=c_1 \wedge \ldots \wedge b_{l-1}=c_{l-1} \wedge b_l< c_l) \end{aligned}$$

Hierdurch ist eine Ordnung auf A^* eindeutig bestimmt, so daß wir von dem ersten, zweiten, dritten, etc. Text über A^* (gemäß dieser Ordnung) reden können. Daraus folgt:

<u>Lemma 3.1:</u> A^* ist abzählbar.

Da es nicht mehr berechenbare Funktionen als sie berechnende Algorithmen und nicht mehr Algorithmen als sie beschreibende Texte geben kann, gilt offenbar folgende Aussage.

<u>Korollar 3.2:</u> Die Menge der berechenbaren Funktionen ist abzählbar.

Betrachten wir nun speziell einstellige Funktionen auf Z, $f: Z \rightarrow Z$. Wie bewiesen, gibt es nur abzählbar viele berechenbare solche Funktionen. Funktionen insgesamt gibt es jedoch weit mehr, wie das folgende Lemma zeigt.

Lemma 3.3: Die Menge $F = \{ f \mid f: \mathbb{Z} \longrightarrow \mathbb{Z} \}$ der einstelligen Funktionen auf $\mathbb{Z}$ ist überabzählbar.

Beweis: Wir nehmen an, F sei abzählbar, so daß wir alle $f \in F$ auflisten können, $F = \{f_0, f_1, \ldots\}$, d.h. jedes f hat eine Nummer i in dieser Folge. Sei nun $g: \mathbb{Z} \longrightarrow \mathbb{Z}$ definiert durch

$$g(x) = f_{abs(x)}(x)+1$$

Dann ist für $i=1,2,\ldots$ $g(i) \neq f_i(i)$, also ist für $i=1,2,\ldots$ immer $g \neq f_i$. g kommt demnach in der obigen Folge nicht vor. Offensichtlich ist aber g eine einstellige Funktion auf $\mathbb{Z}$ und müßte somit in der obigen Folge *aller* dieser Funktionen vorkommen. Der Widerspruch läßt sich nur lösen, wenn wir die Annahme fallenlassen, F sei abzählbar.

Berechenbare Funktionen sind also unter allen Funktionen genauso "seltene" Ausnahmen wie ganze (oder natürliche oder rationale) Zahlen unter den reellen.

Bemerkung: Die obige Beweismethode ist unter dem Namen "Cantorsches Diagonalverfahren" bekannt. Auf ähnliche Weise hat Cantor erstmals bewiesen, daß die reelen Zahlen überabzählbar sind.

Nachdem wir wissen, daß berechenbare Funktionen eher seltene Ausnahmen sind, ist die Frage naheliegend, ob sich nicht-berechenbare Funktionen *konkret angeben* lassen. Um zu beweisen, daß eine gegebene Funktion berechenbar ist, braucht man nur einen Algorithmus anzugeben, der sie berechnet. So schwierig dies im Einzelfall sein mag, so ist es doch prinzipiell schwieriger zu beweisen, daß eine gegebene Funktion *nicht* berechenbar ist. Dies erfordert nämlich eine Beweisführung über *alle denkbaren und möglichen Algorithmen!*

Solche - und andere - Ergebnisse über Algorithmen lassen sich nur gewinnen, wenn man den Begriff des Algorithmus mit hinreichender mathematischer Genauigkeit präzisiert. Wir definieren daher ein recht allgemeines mathematisches Modell für deterministische Algorithmen, "abstrakte Maschine" genannt.

Definition 3.4: Eine ***abstrakte Maschine*** M ist ein 7-Tupel

$$M = (X, Y, K, \alpha, \omega, \tau, \sigma),$$

wobei gilt:

X ist eine Menge von *Eingabewerten*
Y ist eine Menge von *Ausgabewerten*
K ist eine Menge von *Konfigurationen*
$\alpha: X \longrightarrow K$ ist die *Eingabefunktion*
$\omega: K \longrightarrow Y$ ist die *Ausgabefunktion*
$\tau: K \longrightarrow K$ ist die *Transitionsfunktion*
$\sigma: K \longrightarrow \underline{\text{bool}}$ ist die *Stopfunktion*

Definition 3.5: Die Menge der *Endkonfigurationen* zu M ist $E = \{ k \in K \mid \sigma(k)=\text{true} \}$

Eine abstrakte Maschine M läßt sich folgendermaßen durch ein Diagramm darstellen:

$$\begin{tikzcd} X \arrow[r, "\alpha"] & K \arrow[loop above, "\tau"] \arrow[r, "\omega"] \arrow[d, "\sigma"] & Y \\ & \underline{\text{bool}} & \end{tikzcd}$$

Eine solche Maschine *arbeitet* folgendermaßen:

1. Ein Eingabewert $x \in X$ bestimmt die Anfangskonfiguration $k_0=\alpha(x) \in K$.

2. Wir überführen mittels τ Konfigurationen in Folgekonfigurationen, also
   $k_1=\tau(k_0), k_2=\tau(k_1), \ldots$
   bis zum erstenmal eine Endkonfiguration $k_t \in E$ erreicht wird. Dies braucht natürlich niemals einzutreten.

3. Wird eine Endkonfiguration $k_t \in E$ erreicht, so wird der Ausgabewert $\omega(k_t) \in Y$ ausgegeben.

Bei Eingabe von $x \in X$ gibt es also zwei Möglichkeiten:

a) Die Maschine hält nie an.

b) Die Maschine hält und gibt einen eindeutig bestimmten Ausgabewert $y \in Y$ aus.

Auf diese Weise berechnet die Maschine M eine partielle Funktion

$$f_M: X \longrightarrow Y$$

Wir präzisieren diese Verhaltensweise mathematisch.

Definition 3.6: Die *Laufzeit* einer abstrakten Maschine M für die Eingabe $x \in X$ ist

$$t_M(x) = (\mu n)[\sigma(\tau^n(\alpha(x)))]$$

Hierbei ist $(\mu n)[B]$ die kleinste natürliche Zahl $n \in \mathbb{N} = \{0,1,2, \ldots\}$, so daß die Bedingung B=true wird und B für alle $m \leq n$ definiert ist. Gibt es keine solche Zahl $n \in \mathbb{N}$, so ist $t_M(x)=\bot$ (undefiniert).

<u>Definition 3.7</u>: Die von einer abstrakten Maschine M *berechnete Funktion*

$$f_M: X \longrightarrow Y$$

ist gegeben durch

$$f_M(x) = \omega(\tau^{t_M(x)}(\alpha(x))) \ ; \qquad \text{Ist } t_M(x) = \bot, \text{ so ist } f_M(x) = \bot.$$

Applikative, imperative und deduktive Algorithmen sind Beispiele dieses allgemeinen Algorithmenmodells, d.h. sie lassen sich als abstrakte Maschinen auffassen. Wir skizzieren den Zusammenhang grob, ohne ins Detail zu gehen.

<u>Beispiel 3.8:</u> Applikative Algorithmen (nur ganzzahlige E/A)

$$f_i(u_{i,1}, \ldots, u_{i,n_i}) = t_i(u_{i,1}, \ldots, u_{i,n_i}) \ , \qquad i=1, \ldots, m.$$

$X = \mathbb{Z}^{n_1}$
$Y = \mathbb{Z}$
K = Terme ohne Unbestimmte
$\alpha(a_1, \ldots, a_{n_1})$ = der Term "$f_1(a_1, \ldots, a_{n_1})$"
$\omega(t)$ = der Wert von t
τ: Termauswertung aufgrund der Funktionsdefinitionen.
Da diese nichtdeterministisch ist, müssen wir sie durch eine zusätzliche "Berechnungsvorschrift" deterministisch machen, z.B. durch die Forderung, das erste Auftreten von links eines Funktionsaufrufs mit Konstanten, $f_i(b_1, \ldots, b_{n_i})$ mit $b_j \in \mathbb{Z}$, $j=1, \ldots, n_i$, durch die rechte Seite $t_i(b_1, \ldots, b_{n_i})$ zu ersetzen.

$$\sigma(t) = \begin{cases} \text{true, falls } t=b \in \mathbb{Z} \text{ (Konstante) ist} \\ \text{false sonst} \end{cases}$$

<u>Beispiel 3.9:</u> Imperative Algorithmen (nur ganzzahlige E/A)

PROG: <u>var</u> V,W, . . . : <u>int</u>;
       P,Q, . . . : <u>bool</u>;
    <u>input</u> $X_1, \ldots, X_n$;
       β;
    <u>output</u> $Y_1, \ldots, Y_m$.

$X = \mathbb{Z}^n$
$Y = \mathbb{Z}^m$
$K = \{ (Z,\beta) \mid Z \text{ Zustand}, \beta \text{ Anweisung} \}$
   Z bezeichnet den aktuellen Zustand, β die noch auszuführende Anweisung.
$\alpha(a_1, \ldots, a_n) = (Z_0, \beta)$, wobei
   $Z_0(X_i) = a_i$, $i=1, \ldots, n$, und
   $Z_0(Y) = \bot$ für $Y \neq X_i$, $i=1, \ldots, n$.
$\omega(Z,\beta) = (Z(Y_1), \ldots, Z(Y_m))$

$\tau(Z,\beta) = (Z',\beta')$, wobei

Z' = Zustand nach Ausführung der nächsten Anweisung

β' = Rest der noch auszuführenden Anweisung

$$\sigma(Z,\beta) = \begin{cases} \text{true, falls } \beta \text{ leer ist} & \textit{(keine Anweisungen mehr)} \\ \text{false sonst} \end{cases}$$

Beispiel 3.10: Deduktive Algorithmen (Fakten und Regeln)

Wir legen eine deterministische Version des (determinierten) Algorithmus zugrunde, der zu jeder Anfrage *alle* Antworten ermittelt. Da wir den Algorithmus im Abschnitt 2.6 nicht im Detail behandelt haben, verzichten wir auch hier auf die Details. Eingabewerte X, Ausgabewerte Y und Konfigurationen K wären folgendermaßen festzulegen:

X = Anfragen

Y = Antwortmengen

K = Reduktionen mit zugehörigen Belegungen

Diese Algorithmenmodelle sind - nach der nötigen Präzisierung - als mathematische Strukturen zu komplex und unhandlich, um damit umzugehen, z.B. Beweise darüber zu führen. Wir führen daher ein mathematisch wesentlich simpleres Algorithmenmodell ein, das gleichwohl hinsichtlich der Berechnungsfähigkeiten den übrigen in nichts nachsteht (mehr darüber folgt weiter unten).

Sei $A=\{a_1, \ldots, a_n\}$ ein Alphabet, und sei A^* die Menge der Worte (Texte) über A. Wir definieren die Klasse der *Markov-Algorithmen* (Markov 1954). Die elementaren Einzelschritte dieser Algorithmen beruhen auf *Regeln* der Form $\varphi \longrightarrow \psi$, wobei $\varphi,\psi \in A^*$ sind. Diese bedeuten, daß das Wort φ durch das Wort ψ *ersetzt* werden soll. Angewendet auf ein Wort $\xi \in A^*$ entsteht auf eindeutige Weise ein neues Wort $g[\varphi \longrightarrow \psi](\xi)$, das wie folgt definiert ist:

1. Ist φ ein Teilwort von ξ, also $\xi=\mu\varphi\nu$ für $\mu,\nu \in A^*$, und ist φ an dieser Stelle (nach μ) das erste Auftreten (von links) von φ in ξ, so ist $g[\varphi \longrightarrow \psi](\xi) = \mu\psi\nu$, d.h. φ wird (nur!) an dieser Stelle durch ψ ersetzt.

2. Ist φ kein Teilwort von ξ, so ist $g[\varphi \longrightarrow \psi](\xi) = \xi$, d.h. es passiert nichts.

Beispiel 3.11: Sei $A=\{0,1\}$. Wir wenden die Regel $01 \longrightarrow 10$ sukzessive an.

$$\begin{aligned} 1\,1\,0\,\underline{0\,1}\,0\,1\,0\,1\,1 &\longrightarrow 1\,1\,\underline{0\,1}\,0\,0\,1\,0\,1\,1 \\ &\longrightarrow 1\,1\,1\,0\,0\,\underline{0\,1}\,0\,1\,1 \\ &\longrightarrow 1\,1\,1\,0\,\underline{0\,1}\,0\,0\,1\,1 \\ &\longrightarrow 1\,1\,1\,\underline{0\,1}\,0\,0\,0\,1\,1 \\ &\text{etc.} \end{aligned}$$

Ein Markov-Algorithmus besteht nun aus einer Menge solcher Regeln, zusammen mit einer "Kontrollstruktur", die eindeutig festlegt, wann welche Regel anzuwenden ist. Diese Information wird übersichtlich in einer Tabelle, der *Markov-Tafel*, zusammengefaßt. Diese hat fünf Spalten und beliebig viele (endlich viele) Zeilen. Eine Zeile ist ein 5-Tupel der Form

$$k \quad \varphi \quad \psi \quad i \quad j,$$

wobei $i,j,k \in \mathbb{N}$ sind. k ist die Zeilennummer, φ und ψ stellen die Regel $\varphi \longrightarrow \psi$ dar. i ist die Nummer der nächsten Zeile, falls das Teilwort φ gefunden (die Regel also angewandt) wurde, und j ist die Nummer der nächsten Zeile, falls φ nicht auftrat.

Die *Ausführung* der Markov-Tafel beginnt in der ersten Zeile (Nummer 0) und stoppt, sobald zu einer nicht vorhandene Zeilennummer (i oder j) gegangen werden soll. Dazwischen werden die Regeln der angelaufenen Zeilen auf das Eingabewort nacheinander angewandt.

Bevor wir Beispiele betrachten, definieren wir Markov-Algorithmen zunächst als abstrakte Maschinen:

<u>Definition 3.12:</u> Ein *Markov-Algorithmus* über dem Alphabet A ist gegeben durch

$X \subseteq A^*$	Eingabemenge
$Y \subseteq A^*$	Ausgabemenge
$K \subseteq \{(z,n) \mid z \in A^*, n \in \mathbb{N}\}$	Konfigurationen
$\alpha(x) = (x,0)$	Eingabefunktion
$\omega(z,n) = z$	Ausgabefunktion
$\tau: K \longrightarrow K$	Transitionsfunktion

$\tau(z,n) = (g[\varphi \longrightarrow \psi](z), n')$, wobei
$\varphi \longrightarrow \psi$ die Regel in der Zeile n ist und
n' die Folgenummer (s.o., i bzw. j)

$$\sigma(z,n) = \begin{cases} \text{true}, & \text{falls } n \text{ keine Zeilennummer ist} \\ \text{false}, & \text{sonst} \end{cases}$$

<u>Beispiel 3.13:</u> "Addiere |", $A=\{|\}$; $X=A^*$; $Y=A^+=A^*-\{\varepsilon\}$

k	φ	ψ	i	j
0	ε	|	1	–

berechnet die Funktion $f(|^n) = |^{n+1}$, $n \in \mathbb{N}$.

<u>Bemerkung</u>: Das leere Wort ε kommt in jedem Wort vor. Das erste Auftreten ist ganz am Anfang. Der Algorithmus schreibt also ein | *vor* das Eingabewort $|^n = |\,|\ldots|$. Der |-Eintrag der Tabelle kann niemals angelaufen werden und ist daher irrelevant. Dies deuten wir durch das Zeichen – in der Tabelle an.

Beispiel 3.14: "Addition". $A_0 = \{|\}$; $A = A_0 \cup \{+\}$
$X = A_0^* + A_0^*$ $(= \{\mu + \nu \mid \mu, \nu \in A_0^*\})$
$Y = A_0^*$

k	φ	ψ	i	j
0	+	ε	1	-

Der Algorithmus löscht das erste + im Eingabewort, also:

$$f(|^n + |^m) = |^{n+m}.$$

Beispiel 3.15: "Verdopplung". $A_0 = \{|\}$; $A = A_0 \cup \{*\}$
$X = Y = A_0^*$

Das Zeichen * spielt die Rolle einer Markierung, die einmal von links nach rechts durchwandert und dabei die Zeichen | verdoppelt.

k	φ	ψ	i	j	Kommentar
0		*|	1	3	$\|^n \longrightarrow *\|^n$
1	*|	||*	1	2	$\|^{2p}*\|^q \longrightarrow \|^{2(p+1)}*\|^{q-1}$ wiederholen bis q=1
2	*	ε	3	-	$\|^{2n}* \longrightarrow \|^{2n}$

Eine Berechnung mit dem Eingabewort ||| ergibt:

$$||| \xrightarrow[0]{} *||| \xrightarrow[1]{} ||*|| \xrightarrow[1]{} ||||*| \xrightarrow[1]{} ||||||* \xrightarrow[2]{} ||||||$$

Allgemein wird die Funktion $f(|^n) = |^{2n}$ berechnet.

Beispiel 3.16: "Multiplikation". $A_0 = \{|\}$; $A = A_0 \cup \{*, \#\}$
$X = A_0^* * A_0^*$; $Y = A_0^*$

Der folgende Markov-Algorithmus berechnet die Funktion $f(|^n * |^m) = |^{n \cdot m}$:

k	φ	ψ	i	j	Kommentar	
0	*	**	1	-	$\longrightarrow \|^n**\|^m$	
1	ε	#	2	-	$\longrightarrow \#\|^n**\|^m$	
2	#*|	*#*	3	6		nächster Faktor
3	|#	#|	4	5	Faktor vorn kopieren	
4	ε	|	3	-		
5	#	ε	2	-		
6	*|	*	6	7	1. Faktor löschen	
7	***	ε	8	-	Hilfsmarkierungen löschen	

Mit der Eingabe | | * | | ergibt sich z.B. folgende Berechnungsfolge:

$$| \; * \; | \xrightarrow[0]{} \; * \; * \; \xrightarrow[1]{} * \; | \; | \; * \; * \; |$$
$$\xrightarrow[2]{} * \; | \; \# \; * \; * \xrightarrow[3,4]{} * \; | \; \# \; | \; * \; * \xrightarrow[3,4]{} | \; * \; | \; \# \; | \; * \; * \; |$$
$$\xrightarrow[3,4,5]{} | \; | \; * \; | \; | \; * \; * \; |$$
$$\xrightarrow[2]{} | \; * \; | \; * \; * \; * \xrightarrow[3,4]{} | \; * \; | \; \# \; | \; * \; * \xrightarrow[3,4]{} | \; * \; * \; \# \; * \; *$$
$$\xrightarrow[3,4,5]{} | \; | \; | \; * \; | \; * \; *$$
$$\xrightarrow[(2)6]{} | \; | \; | \; | \; | \; * \; | \; | \; * \; * \xrightarrow[6]{} | \; | \; | \; | \; * \; | \; * \; * \xrightarrow[6]{} | \; | \; | \; * \; * \; * \xrightarrow[7]{} | \; | \; |$$

<u>Beispiel 3.17:</u> "Kopieren". $A_0 = \{0,1\}$; $A = A_0 \cup \{*,\#\}$
$X = A_0^*$; $Y = A_0^* * A_0^*$

Der folgende Markov-Algorithmus berechnet die Funktion $f(\mu) = \mu * \mu$, $\mu \in A_0^*$

k	φ	ψ	i	j	Kommentar
0	ε	*#	1	-	
1	#0	0#	2	3	kopiere nächstes Zeichen 0 vor *
2	*	0*	1	-	und wiederhole
3	#1	1#	4	5	kopiere nächstes Zeichen 1 vor *
4	*	1*	1	-	und wiederhole
5	#	ε	6	-	

Mit der Eingabe 0 1 0 ergibt sich folgende Berechnung:

$$0\,1\,0 \xrightarrow[0]{} *\,\#\,0\,1\,0 \xrightarrow[1]{} *\,0\,\#\,1\,0 \xrightarrow[2]{} 0\,*\,0\,\#\,1\,0$$
$$\xrightarrow[(1)3]{} 0\,*\,0\,1\,\#\,0 \xrightarrow[4]{} 0\,1\,*\,0\,1\,\#\,0$$
$$\xrightarrow[1]{} 0\,1\,*\,0\,1\,0\,\# \xrightarrow[2]{} 0\,1\,0\,*\,0\,1\,0\,\#$$
$$\xrightarrow[(1,3)5]{} 0\,1\,0\,*\,0\,1\,0$$

Welche Funktionen können nun durch Markov-Algorithmen berechnet werden? Leisten Markov-Algorithmen mehr oder weniger als applikative bzw. imperative bzw. deduktive Algorithmen? Wie kann man überhaupt die Leistungen von Algorithmenmodellen miteinander vergleichen?

Einem direkten Leistungsvergleich steht zunächst der Umstand im Wege, daß die Algorithmenmodelle auf ganz verschiedenen Argument- und Wertebereichen definiert sind:

- applikative Algorithmen: $f: \mathbb{Z}^n \longrightarrow \mathbb{Z}$, n variabel
- imperative Algorithmen: $f: \mathbb{Z}^n \longrightarrow \mathbb{Z}^m$, m,n variabel
- deduktive Algorithmen: f: Anfragen $\longrightarrow$ Antwortmengen
- Markov-Algorithmen: $f: X \longrightarrow Y$, $X,Y \subseteq A^*$

Um vergleichen zu können, muß man zunächst die Argument- und Wertebereiche "vereinheitlichen", d.h. man muß einen Argument- und Wertebereich finden, mit dem sich alle Ein- und Ausgaben aller Algorithmenmodelle eindeutig und effektiv (i.e. berechenbar) *codieren* lassen.

Einen solchen für die Zwecke der Berechenbarkeit "universellen" Bereich bilden u.a. Texte A^* über einem festen Alphabet $A = \{a_1, \ldots, a_n\}$.

Zu jedem Datenbereich D eines Algorithmenmodells ($\mathbb{Z}^n, A^*$, etc.) lassen sich bijektive berechenbare Funktionen auf A^* (Codierungen) angeben:

$$cod_D: D \longrightarrow A^*$$

Wegen der Bijektivität gibt es eine Umkehrfunktion

$$decod_D: A^* \longrightarrow D,$$

so daß gilt:

$$decod_D(cod_D(d)) = d \text{ für alle } d \in D$$
$$cod_D(decod_D(\xi)) = \xi \text{ für alle } \xi \in A^*$$

Auch $decod_D$ muß berechenbar sein.

Wir wollen hier nicht im einzelnen untersuchen, wie solche Codierungen für $\mathbb{Z}^n$, B^*, etc. aussehen könnten. Wir nehmen an, wir hätten sie zur Verfügung. Dann berechnet jeder Algorithmus, der die Funktion $f: X \longrightarrow Y$ berechnet, per Codierung und Decodierung die einstellige Textfunktion

$$f': A^* \longrightarrow A^*$$

gegeben durch

$$f'(n) = cod_Y(f(decod_X(n))).$$

$$\begin{tikzcd}
X \arrow[r, "f"] & Y \\
A^* \arrow[u, "decod_X"] \arrow[r, dashed, "f'"'] & A^* \arrow[u, dashed, "cod_Y"']
\end{tikzcd}$$

Damit lassen sich nun verschiedene Algorithmenmodelle bzgl. ihrer Leistung, Textfunktionen der Art $f: A^* \longrightarrow A^*$ zu berechnen, miteinander vergleichen.

Zu festem Alphabet A seien:

$$F_M = \{f: A^* \longrightarrow A^* \mid f \text{ ist Markov-berechenbar}\}$$
$$F_a = \{f: A^* \longrightarrow A^* \mid f \text{ ist applikativ berechenbar}\}$$
$$F_i = \{f: A^* \longrightarrow A^* \mid f \text{ ist imperativ berechenbar}\}$$
$$F_d = \{f: A^* \longrightarrow A^* \mid f \text{ ist deduktiv berechenbar}\}$$

Hierbei bedeutet "Markov-berechenbar" Berechenbarkeit im obigen Sinne durch einen Markov-Algorithmus. Für die anderen Algorithmenmodelle gilt entsprechendes.

Nun gilt das folgende wichtige, grundlegende (und vielleicht überraschende) Ergebnis:

Satz 3.18: $F_M = F_a = F_i = F_d$

Alle diese Algorithmenmodelle leisten also prinzipiell gleich viel.

Den Beweis dieses Satzes wollen wir hier nicht durchführen. Er wird konstruktiv geführt, indem man allgemeine "Übersetzer" angibt, die jeden Algorithmus des einen Modells in einen gleichwertigen des anderen Modells überführen, der dieselbe Funktion berechnet. Die Konstruktion solcher Übersetzer ist – auch für einfache mathematische Algorithmenmodelle – sehr aufwendig und langwierig.

Bemerkung: Auch noch viele andere Algorithmenmodelle führen auf diese gleiche Klasse berechenbarer Funktionen, z.B. Rekursive Funktionen, λ-Kalkül, Turing-Maschinen, Post'sche Normalsysteme, Random-Access-Maschinen, Universelle-Register-Maschinen, goto-Programme, etc.

Derartige Ergebnisse haben 1936 den Logiker und Mathematiker A. Church veranlaßt, folgende – prinzipiell nicht beweisbare – These aufzustellen:

Church'sche These: Die Klasse der *intuitiv berechenbaren* Funktionen stimmt mit dieser Klasse der (Markov-, applikativ, imperativ, etc.) berechenbaren Funktionen überein.

Daß jede in einem der Algorithmenmodelle berechenbare Funktion auch intuitiv berechenbar ist, ist klar. Die wesentliche Aussage der These besagt, daß jede Funktion, die man intuitiv als berechenbar ansehen würde, tatsächlich auch in jedem dieser Algorithmenmodelle berechenbar ist. Das besagt im Grunde, daß der intuitive Begriff der Berechenbarkeit durch die mathematische Präzisierung korrekt und vollständig erfaßt ist.

Definition 3.19: Ein Algorithmenmodell heißt *universell*, wenn es alle berechenbaren Funktionen zu beschreiben gestattet.

Bemerkung: Nahezu alle Programmiersprachen, höhere sowie niedere, sind universell.

Nun kehren wir zu dem Problem zurück, eine nicht berechenbare Funktion konkret anzugeben. Wir benutzen dazu ein beliebiges Algorithmenmodell, welches folgenden Kriterien genügt:

1. Berechnet werden partielle Funktionen $f: A^* \longrightarrow A^*$ über einem festen Alphabet A.

2. Auch die Algorithmen selbst lassen sich als Texte über A darstellen.

Damit können Algorithmentexte wiederum als Eingaben für Algorithmen genommen werden!

Bemerkung: Z.B. lassen sich Markov-Algorithmen leicht als Text über einem geeigneten Alphabet A codieren. Codierungen von Algorithmen durch natürliche Zahlen gehen auf K. Gödel (1931) zurück. Man spricht dann von "Gödelisierung".

Definition 3.20: Sei $x \in A^*$. Dann bezeichnet φ_x die *vom Algorithmus mit Text x berechnete Funktion*. Ist x kein sinnvoller Algorithmentext, so sei φ_x überall undefiniert.

Definition 3.21: Sei $f: A^* \to A^*$ eine partielle Funktion. Dann ist

$$\mathrm{dom}\, f := \{ x \in A^* \mid f(x) \text{ ist definiert} \}$$

der *Urbildbereich* von f("domain").

Nun sind wir endlich in der Lage, eine nicht berechenbare Funktion konkret anzugeben. Dazu sei $a \in A$ ein fest gewählter Buchstabe.

Satz 3.22: Die (totale) Funktion $h: A^* \to A^*$,

$$h(x) = \begin{cases} \varepsilon, & \text{falls } x \in \mathrm{dom}\varphi_x \\ a & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Die obige Funktion h ist mathematischer Ausdruck eines recht anschaulichen Problems, des sog. *Halteproblems*. h entscheidet durch die Ausgabe ε oder a, ob $x \in \mathrm{dom}\varphi_x$ ist oder nicht. Nun bedeutet $y \in \mathrm{dom}\varphi_x$, daß $\varphi_x(y)$ definiert ist, und das heißt, daß der Algorithmus mit Text x bei Eingabe von y irgendwann anhält. Die Funktion h drückt also die Frage aus:

"Hält ein Algorithmus irgendwann an, wenn man ihm seinen eigenen Text eingibt ?"

Entscheidungsprobleme, deren Entscheidungsfunktion nicht berechenbar ist, nennt man *nicht entscheidbar*. Es gibt eine große Fülle nicht entscheidbarer Probleme. Das Problem $x \in \mathrm{dom}\varphi_x$ ist ein spezielles Halteproblem, auch *Selbstanwendungsproblem* genannt. Das *allgemeine Halteproblem* ist $y \in \mathrm{dom}\varphi_x$, also:

"Hält Algorithmus x bei Eingabe von y ?"

Korollar 3.23: Das allgemeine Halteproblem ist nicht entscheidbar.

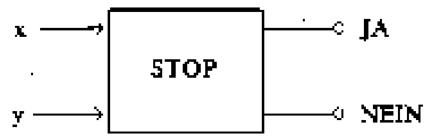
Beweis: Gäbe es einen Entscheidungsalgorithmus für $y \in \mathrm{dom}\varphi_x$, so könnte man damit auch speziell $x \in \mathrm{dom}\varphi_x$ entscheiden.

Bemerkung: So einfach dieser Beweis ist, er zeigt ein typisches Grundmuster, nämlich "Reduktion auf ein bereits als nicht entscheidbar bewiesenes Problem". Fast alle Nicht-Entscheidbarkeits-Beweise sind so konstruiert: "Wäre das (neue) Problem Y entscheidbar, so auch das (alte) Problem X". Das Halteproblem spielt hierbei die Rolle einer Wurzel, auf die letzten Endes alle Nicht-Entscheidbarkeits-Probleme reduziert werden.

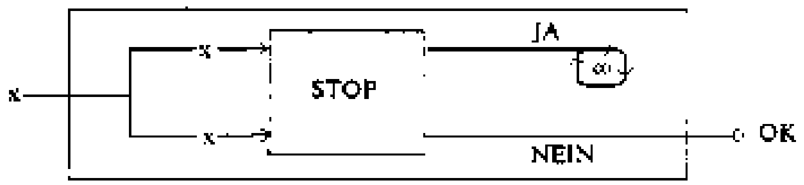
Wegen der grundlegenden Bedeutung des Halteproblems für die Theorie der Entscheidbarkeit wollen wir zunächst einen "anschaulichen" Beweis für seine Nicht-Entscheidbarkeit liefern: Danach geben wir einen "mathematischen" Beweis von Satz 3.22.

Angenommen, wir hätten eine Maschine (Algorithmus) STOP mit zwei Eingaben, nämlich einem Algorithmentext x und einer Eingabe y für x, sowie zwei Ausgaben,

JA: x stoppt bei Eingabe von y
NEIN: x stoppt nicht bei Eingabe von y



Mit dieser Maschine STOP könnten wir dann eine neue Maschine SELTSAM konstruieren:



Bei Eingabe von x wird getestet, ob x bei Eingabe von x stoppt. Im JA-Falle wird in eine Endlosschleife gegangen, die nie anhält. Im NEIN-Falle hält SELTSAM an mit der Anzeige OK.

Nun geben wir SELTSAM als Eingabe für sich selbst ein und fragen:

Hält SELTSAM bei Eingabe von SELTSAM?

1. Wenn ja, so wird der JA-Ausgang von STOP angelaufen, und SELTSAM gerät in die Endlosschleife, hält also nicht. Widerspruch!

2. Wenn nein, so wird der NEIN-Ausgang von STOP angelaufen, und SELTSAM stoppt mit OK. Widerspruch!

Diese Widersprüche folgen aus der Annahme, daß eine STOP-Maschine existiert, was daher verneint werden muß.

Mathematischer Beweis von Satz 3.22: Wir nehmen an, h sei berechenbar. Dann ist aufgrund der Church'schen These auch die Funktion $g: A^* \longrightarrow A^*$,

$$g(x) = \begin{cases} \varphi_x(x)a, & \text{falls } h(x) = \varepsilon \\ \varepsilon & \text{sonst} \end{cases}$$

berechenbar. Nun ist sicherlich $g(x) \neq \varphi_x(x)$ für alle $x \in A^*$. Da g berechenbar ist, gibt es einen Algorithmus mit einem Text $y \in A^*$, der g berechnet, d.h. es ist $g = \varphi_y$. Damit folgt nun:

$$\varphi_y(y) = g(y) \neq \varphi_y(y)$$

Dies ist offenbar ein Widerspruch, der sich nur dadurch lösen läßt, daß wir die Annahme aufgeben, h sei berechenbar.

Bemerkung: Dies ist wiederum ein Diagonalbeweis.

Das Halteproblem ist ein Beispiel für ein *semantisches* Problem von Algorithmen, nämlich ein Problem der Art:

Kann man anhand eines Programmtextes (Syntax) entscheiden,
ob die berechnete Funktion (Semantik) eine bestimmte Eigenschaft hat ?

Beim Halteproblem ist dies die Eigenschaft, ob die berechnete Funktion an einer bestimmten Stelle definiert ist. Wie steht es nun mit anderen semantischen Eigenschaften von Algorithmen?

Aus einem tiefliegenden Satz der Algorithmentheorie (Satz von Rice) folgt die folgende bemerkenswerte Aussage:

Jede nichttriviale semantische Eigenschaft von Algorithmen ist nicht-entscheidbar!

Dabei ist eine Eigenschaft genau dann trivial, wenn entweder jede oder keine berechnete Funktion sie hat. Nichttriviale semantische Eigenschaften sind demnach solche, die manche berechneten Funktionen haben und manche nicht.

Nicht entscheidbar sind also u.a. folgende Probleme:

1. Ist die berechnete Funktion total? überall undefiniert? injektiv? surjektiv? bijektiv? etc. etc.

2. Berechnen zwei gegebene Algorithmen dieselbe Funktion?

3. Ist ein gegebener Algorithmus korrekt, d.h. berechnet er die gegebene (gewünschte) Funktion ?

etc.

Diese Ergebnisse bedeuten nicht, daß man solche Fragen nicht *im Einzelfall* entscheiden könnte! Dies ist durchaus möglich. So behandeln wir z.B. im nächsten Abschnitt das Problem, die Korrektheit von Algorithmen nachzuweisen. Es ist jedoch prinzipiell unmöglich, eine allgemeine Methode hierfür zu finden, also z.B. *einen* Algorithmus, der die Korrektheit *aller* Algorithmen nachweist (und damit auch seine eigene!).

Ein besonders einfaches und verblüffendes nicht-entscheidbares Problem ist das *Post'sche Korrespondenzproblem* (E. Post): Gegeben seien ein Alphabet A und zwei gleichlange Listen von nichtleeren Worten über A:

$$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$$
$$\beta = (\beta_1, \beta_2, \ldots, \beta_n)$$

wobei $\alpha_i, \beta_i \in A^+ = A^* - \{\varepsilon\}$ und $n \geq 1$. Das Problem besteht darin, eine "Korrespondenz" zu finden, d.h. eine endliche Folge $(i_1, i_2, \ldots, i_k)$, $i_j \in \{1, \ldots, n\}$ für $j = 1, 2, \ldots, k$, so daß gilt

$$\alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_k} = \beta_{i_1} \beta_{i_2} \ldots \beta_{i_k}$$

<u>Beispiel 3.24</u>: $A = \{0,1\}$ $\quad \alpha = (1\ ,\ 1\,0\,1\,1\,1\ ,\ 1\,0)$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \beta = (1\,1\,1\ ,\ 1\,0\ ,\ 0)$

Dies Post'sche Korrespondenzproblem besitzt eine Lösung, nämlich (2,1,1,3):

$$1\,0\,1\,1\,1 . 1 . 1 . 1\,0 = 1\,0 . 1\,1\,1 . 1\,1\,1 . 0$$

<u>Beispiel 3.25</u>: $A = \{0,1\}$ $\quad \alpha = (1\,0\ ,\ 0\,1\,1\ ,\ 1\,0\,1)$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \beta = (1\,0\,1\ ,\ 1\,1\ ,\ 0\,1\,1)$

Dies Post'sche Korrespondenzproblem besitzt keine Lösung. Gäbe es nämlich eine, so müßte sie mit 1 anfangen:

$$1\,0 \ldots \stackrel{?}{=} 1\,0\,1 \ldots$$

Als zweites müßte ein Index i mit $\alpha_i = 1 \ldots$ gewählt werden, also 1 oder 3. Aber (1, 1, . . .) ergibt

$$1\,0\,1\,0 \ldots \neq 1\,0\,1\,1\,0\,1 \ldots,$$

und (1, 3, . . .) ergibt

$$1\,0\,1\,0\,1 \ldots \stackrel{?}{=} 1\,0\,1\,0\,1\,1 \ldots$$

Die gleiche Überlegung wie oben führt dazu, daß wieder 3 gewählt werden muß (1, 3, 3, . . .) ergibt

$$1\,0\,1\,0\,1\,1\,0\,1 \ldots \stackrel{?}{=} 1\,0\,1\,0\,1\,1\,0\,1\,1 \ldots$$

Wieder müßte 3 gewählt werden, usw. Es ist nun zu sehen, daß niemals eine Lösung entstehen kann, da die rechte Seite der linken immer um 1 voraus ist.

Während wir in diesen Beispielen die Lösbarkeit haben entscheiden können, ist das allgemeine Problem, nämlich zu entscheiden, ob eine Lösung für ein gegebenes Post'sches Korrespondenzproblem über einem Alphabet mit mindestens *zwei* Buchstaben existiert, nicht entscheidbar.

3.2 Korrektheit

Es ist ganz wesentlich, daß Algorithmen, die wir für eine bestimmte Aufgabe entworfen haben, korrekt sind, d.h. daß sie sich genauso verhalten, wie wir es beabsichtigt hatten. Bei einigen hochsensiblen und kritischen Anwendungen (z.B. in der medizinischen Informatik, bei der Steuerung von Waffensystemen, aber auch in der Baustatik, etc.) können durch inkorrekte Algorithmen hervorgerufene Fehlfunktionen schwere Schäden anrichten oder gar Katastrophen auslösen.

Nun haben wir im vorigen Abschnitt gesehen, daß die Korrektheit von Programmen im allgemeinen nicht entscheidbar ist. In vielen Anwendungsfällen wird man mit pragmatischem Austesten hinreichende Sicherheit erreichen können. Will man jedoch die Korrektheit eines Algorithmus' mit mathematischer Strenge und Exaktheit nachweisen, so ist man auf ad-hoc-Beweisführung im Einzelfall angewiesen. Dies ist nicht nur bei sicherheitskritischen Anwendungen angezeigt, sondern z.B. auch bei Standard-Algorithmen, die hardwaremäßig, etwa als VLSI-Chip, realisiert und in großer Serie produziert werden sollen. Ein großer Aufwand bei dem Nachweis der Korrektheit lohnt sich hier, da Fehler große wirtschaftliche Verluste bewirken können.

In diesem Abschnitt wollen wir einige grundlegende Konzepte und Vorgehensweisen für Korrektheitsbeweise für Algorithmen erörtern. Die Problematik kann hier nicht tief und ausführlich abgehandelt werden. Wir beschränken uns darauf, Problematik und Vorgehensweise anhand einfacher Beispiele zu demonstrieren.

Der Nachweis der Korrektheit eines Algorithmus bezieht sich immer auf eine *Spezifikation* dessen, was er tun *soll*. Insofern handelt es sich also um eine *relative* Korrektheit.

Wir behandeln zunächst imperative Algorithmen. Eine verbreitete Methode, gewünschte Eigenschaften von Algorithmen zu spezifizieren, ist die Angabe von Vor- und Nachbedingungen:

$$(*) \qquad \{VOR\} \quad ANW \quad \{NACH\}$$

VOR und NACH sind dabei Aussagen über den Zustand vor bzw. nach Ausführung der Anweisung ANW. Genauer bedeutet die Aussage (*):

Gilt VOR unmittelbar vor Ausführung von ANW und terminiert ANW,
so gilt NACH unmittelbar nach Ausführung von ANW.

Terminiert ANW nicht, so ist (*) trivialerweise wahr, wie auch immer VOR und NACH aussehen!

Entsprechend ist (*) trivialerweise wahr, wenn VOR nicht gilt, gleichgültig ob ANW terminiert oder nicht und ob NACH gilt oder nicht!

Beispiel 2.26: Anweisungen über $\mathbb{Z}$

$\{X=0\}$ $X:=X+1$ $\{X=1\}$ ist wahr

$\{true\}$ $X:=Y$ $\{X=Y\}$ ist wahr

$\{Y=a\}$ $X:=Y$ $\{X=a \wedge Y=a\}$ ist wahr für alle $a \in \mathbb{Z}$

$\{X=a \wedge Y=b \wedge a \neq b\}$ $X:=Y;\ Y:=X$ $\{X=b \wedge Y=a\}$ ist falsch für alle $a,b \in \mathbb{Z}$

$\{X=a \wedge Y=b\}$ $Z:=X;X:=Y;Y:=Z$ $\{X=b \wedge Y=a\}$ ist wahr für alle $a,b \in \mathbb{Z}$

$\{false\}$ ANW $\{NACH\}$ ist wahr für alle ANW,NACH

$\{true\}$ ANW $\{false\}$ ist genau dann wahr, wenn ANW nicht terminiert

$\{X=a\}$ <u>while</u> $X \neq 0$ <u>do</u> $X:=X-1$ <u>od</u> $\{X=0\}$ ist wahr für alle $a \in \mathbb{Z}$! *(Auch für $a<0$, da dann die <u>while</u> - Schleife nicht terminiert!)*

Bei der Korrektheit *imperativer* Algorithmen bzgl. gegebener Vor- und Nachbedingungen unterscheidet man zwischen *partieller* und *totaler* Korrektheit. Zur Definition dieser Begriffe sei ein Algorithmus PROG gegeben:

PROG: <u>var</u> $X,Y, \ldots$: <u>int</u>;
$P,Q, \ldots$: <u>bool</u>;
<u>input</u> $X_1, \ldots, X_n$;
α;
<u>output</u> $Y_1, \ldots, Y_m$.

Ferner seien eine Vorbedingung VOR und eine Nachbedingung NACH für die Anweisung α von PROG gegeben.

Definition 3.27: PROG heißt *partiell korrekt bzgl. VOR und NACH* gdw. $\{VOR\}\ \alpha\ \{NACH\}$ wahr ist (s.o.). PROG heißt *total korrekt* bzgl. VOR und NACH gdw. PROG partiell korrekt ist bzgl. VOR und NACH, und wenn α darüberhinaus immer dann terminiert, wenn vorher VOR gilt.

Beispiel 3.28: Für den Algorithmus GGT3 (s.o. 2.5) gilt

$$[\![GGT3]\!](x,y) = \begin{cases} ggT(x,y), & \text{falls } x,y>0 \\ y, & \text{falls } x=y\neq 0 \text{ oder } x=0,y\neq 0 \\ \perp, & \text{falls } y=0 \\ ggT(|x|,y), & \text{falls } x<0 \text{ und } y>0 \\ -ggT(|x|,|y|), & \text{falls } y<0 \end{cases}$$

Also gilt:

GGT3 ist *partiell* korrekt bzgl. VOR = $(X=a\geq 0 \wedge Y=b\geq 0)$
NACH = $(X=ggT(a,b))$ $a,b\in\mathbb{Z}$

GGT3 ist *total* korrekt bzgl. VOR = $(X=a\geq 0 \wedge Y=b>0)$
NACH = $(X=ggT(a,b))$ $a,b\in\mathbb{Z}$

Wir untersuchen jetzt die Korrektheit des Algorithmus XYZ aus Beispiel 2.39 .

Beispiel 3.29: XYZ: var W,X,Y,Z: int;
input X
Z:=0; W:=1; Y:=1;
while W≤X do Z:=Z+1; W:=W+Y+2; Y:=Y+2 od;
output Z.

Seien VOR = $(X\geq 0)$ und NACH = $(Z^2\leq X<(Z+1)^2) = (Z=\lfloor\sqrt{X}\rfloor)$.

Wir beweisen, daß XYZ total korrekt ist bzgl. VOR und NACH.

Dazu sei

$\alpha = (Z:=0\,;\ W:=1\,;\ Y:=1)$
$\beta = (Z:=Z+1\,;\ W:=W+Y+2\,;\ Y:=Y+2)$
$P = (Z^2\leq X \wedge (Z+1)^2=W \wedge 2\cdot Z+1=Y \wedge Y>0)$

P ist eine "Zwischenbedingung", von der wir zeigen, daß sie am Anfang jedes Durchlaufs durch die while-Schleife gilt ("Schleifeninvariante"), und daß sie nach Ablauf der while-Schleife die Nachbedingung NACH impliziert. Das heißt, wir zeigen:

1. $\{VOR\}\ \alpha\ \{P\}$
2. $\{P \wedge W\leq X\}\ \beta\ \{P\}$
3. $P \wedge W>X \Rightarrow NACH$

Haben wir dies bewiesen, so folgt daraus, daß XYZ partiell korrekt ist bzgl. VOR und NACH.

zu 1. Mit den Werten Z=0, W=Y=1 nach α gilt dort P.

zu 2. Offenbar gilt:
$\{(Z+1)^2=W \wedge W\leq X\}\ \beta\ \{Z^2\leq X\}$
$\{(Z+1)^2=W \wedge 2\cdot Z+1=Y\}\ \beta\ \{W=Z^2+2\cdot(Z-1)+1+2 = Z^2+2\cdot Z+1 = (Z+1)^2\}$
$\{2\cdot Z+1=Y\}\ \beta\ \{Y=2\cdot(Z-1)+1+2 = 2\cdot Z+1\}$
$\{Y>0\}\ \beta\ \{Y>0\}$
Hieraus folgt insgesamt $\{P \wedge W\leq X\}\ \beta\ \{P\}$

zu 3. $P \wedge W>X \Rightarrow Z^2\leq X \wedge X<(Z+1)^2 \Rightarrow NACH$

Damit ist die partielle Korrektheit bewiesen. Man beachte, daß hierfür die Bedingung $Y>0$ nicht benötigt wurde. Sie wird erst zum *Beweis der Terminierung* benötigt.

Hierzu zeigen wir, daß für den Wert von $u = X-W$ gilt:

1. $W \le X \Rightarrow u \ge 0$, d.h. u bleibt bei allen Schleifendurchläufen nichtnegativ.

2. u wird bei jedem Schleifendurchlauf echt kleiner.

Haben wir dies bewiesen, so folgt daraus, daß es nur endlich viele Schleifendurchläufe geben kann, daß XYZ also (bzgl. VOR) terminiert.

zu 1. Dies ist offensichtlich.

zu 2. Sei u der Wert unmittelbar vor dem Ablauf von β, und sei u' der Wert unmittelbar danach. Dann gilt:
$u = X-W$
$u' = X-(W+Y+2) = X-W-Y-2$
Da $Y>0$ ist (!), ist $u'<u$.

Dies vervollständigt den Beweis der totalen Korrektheit von XYZ bzgl. VOR und NACH.

Für den Nachweis von Eigenschaften - wie z.B. Korrektheit - *applikativer* Algorithmen verwendet man typischerweise *Induktionsbeweise*, die der Struktur der rekursiven Funktionsdefinition folgen.

Beispiel 3.30: Mc'Carthy's 91-Funktion:

$f(x) = \textbf{if } x>100 \textbf{ then } x-10 \textbf{ else } f(f(x+11)) \textbf{ fi}.$

Sei $g(x) = \text{if } x>100 \text{ then } x-10 \text{ else } 91 \text{ fi}.$

Wir beweisen: $f(x)=g(x)$ für alle $x \in \mathbf{Z}$.

1. Induktionsanfang: für $x>100$ ist $f(x) = x-10 = g(x)$.

Die Induktion verläuft "rückwärts", d.h. wir zeigen:
gilt $f(y)=g(y)$ für $y \ge x$, so gilt auch $f(x-1) = g(x-1)$; also:

2. Induktionsannahme: Es gelte $f(y)=g(y)$ für $y \ge x$

3. Induktionsschritt:

1. Fall: sei $101 \ge x \ge 91$. Dann gilt:

$f(x-1) = f(f(x-1+11)) = f(f(x+10)) = f(x+10-10) = f(x) = g(x) = 91 = g(x-1)$

2. Fall: sei $90 \geq x$. Dann gilt: $f(x-1) = f(f(x+10))$

Aufgrund der Induktionsannahme ist $f(x+10) = g(x+10) = 91$, also folgt:

$f(x-1) = f(91) = g(91) = 91 = g(x-1)$

Dies beweist $f(x)=g(x)$ für alle $x \in Z$.

Für den Nachweis von Eigenschaften *deduktiver* Algorithmen stehen, da es sich hier um Formelmengen handelt, die Methoden der formalen Logik direkt zur Verfügung. Wir gehen hier nicht weiter darauf ein.

3.3 Komplexität

Für die algorithmische Lösung eines gegebenen Problems ist es *unerläßlich*, daß der gefundene Algorithmus das Problem *korrekt* löst. Darüber hinaus ist es *wünschenswert*, daß er dies mit einem möglichst geringen *Aufwand* tut. Die Theorie der Komplexität von Algorithmen beschäftigt sich damit, gegebene Algorithmen hinsichtlich ihres Aufwandes abzuschätzen und - darüber hinaus - für gegebene Problemklassen anzugeben, mit welchem Mindestaufwand Probleme dieser Klasse gelöst werden können.

Beispiel 3.31: Suche in Listen

Gegeben seien:
- eine Zahl $n \geq 0$,
- n Zahlen $a_1, \ldots, a_n$, alle verschieden,
- eine Zahl b.

Gesucht wird der Index $i=1,2, \ldots, n$, so daß $b=a_i$ ist, sofern ein solcher Index existiert. Sonst soll $i=n+1$ ausgegeben werden.

Eine sehr einfache Lösung dieses Suchproblems ist die folgende (wobei standardmäßig $a_{n+1}=0$ gesetzt sei):

$i:=1$; <u>while</u> $i \leq n \wedge b \neq a_i$ <u>do</u> $i:=i+1$ <u>od</u>

Am Ende hat i den gesuchten Ausgabewert.

Diese Lösung hängt von der Eingabe ab, d.h. von $n, a_1, \ldots, a_n$ und b, und zwar gilt:

1. Bei *erfolgreicher* Suche, wenn $b=a_i$ ist, werden $S=i$ Schritte benötigt.
2. Bei *erfolgloser* Suche werden $S=n+1$ Schritte benötigt.

Die Aussage 1 hängt noch von zu vielen Parametern ab, um aufschlußreich zu sein. Man ist bemüht, globalere Aussagen zu finden, die nur von *einer* einfachen Größe abhängen. Hier bietet sich die Länge n der Eingabeliste an, und man beschränkt sich auf Fragen der Art

A: Wie groß ist S für gegebenes n *im schlechtesten Fall?*

B: Wie groß ist S für gegebenes n *im Mittel?*

Wir analysieren den Fall der erfolgreichen Suche:

zu A: Im schlechtesten Fall wird b erst im letzten Schritt gefunden, d.h. $b=a_n$. Also gilt:

S=n im schlechtesten Fall.

zu B: Um eine *mittlere* Anzahl von Schritten zu berechnen, muß man Annahmen über die Häufigkeit machen, mit der - bei wiederholter Verwendung des Algorithmus mit verschiedenen Eingaben - b an erster, zweiter, . . . , letzter Stelle gefunden wird. Wird b häufiger am Anfang der Liste gefunden, so ist die mittlere Anzahl von Suchschritten sicher kleiner, als wenn b häufiger am Ende der Liste gefunden wird. Als ein einfaches Beispiel nehmen wir Gleichverteilung an, d.h.:

läuft der Algorithmus N-mal, $N>>1$, so wird n gleich häufig an erster, zweiter, . . . , letzter Stelle gefunden, also jeweils $\frac{N}{n}$-mal.

Dann werden für all N Suchvorgänge insgesamt

$$M = \frac{N}{n} \cdot 1 + \frac{N}{n} \cdot 2 + \ldots + \frac{N}{n} \cdot n$$

Schritte benötigt. Die Auswertung ergibt:

$$M = \frac{N}{n} (1 + 2 + \ldots + n) = \frac{N}{n} \cdot \frac{n(n+1)}{2} = N \cdot \frac{n+1}{2}$$

Für *eine* Suche werden mit Mittel demnach $S = \frac{M}{N}$ Schritte benötigt, also:

$S = \frac{n+1}{2}$ *im Mittel* (bei Gleichverteilung)

Bemerkung: Unser Algorithmus ist für das gestellte Problem keineswegs der schnellste, sondern eher einer der langsamsten. Es gibt Algorithmen, die die Suche in Listen (oder Tabellen) mit n Einträgen unter speziellen Voraussetzungen i.w. mit einer konstanten, nicht von n abhängigen, Anzahl von Schritten lösen! Die besten Suchverfahren für direkten Zugriff, die auch sequentielle Verarbeitung in der Ordnung der Schlüssel zulassen, benötigen eine logarithmische Anzahl von Schritten, d.h. $S = a \cdot \log_2(b \cdot n) + c$, wobei $a,b,c \in \mathbb{R}$ Konstante sind.

Meist geht es bei der Analyse der Komplexität von Algorithmen bzw. Problemklassen darum, als Maß für den Aufwand eine Funktion

$$f: \mathbb{N} \longrightarrow \mathbb{N}$$

anzugeben, wobei $f(n)=a$ bedeutet: "bei einem Problem der Größe n ist der Aufwand a". Die

"Problemgröße" n bezeichnet dabei in der Regel ein grobes Maß für den Umfang der Eingabe, z.B. die Anzahl der Elemente in einer Eingabeliste oder die Größe eines bestimmten Eingabewertes. Der *"Aufwand"* a ist in der Regel ein grobes Maß für die *Rechenzeit*. Jedoch ist auch der benötigte *Speicherplatz* zuweilen Gegenstand der Analyse. Die Rechenzeit wird meist dadurch abgeschätzt, daß man zählt, wie häufig eine bestimmte Art von Operationen ausgeführt wird, z.B. Speicherzugriffe, Multiplikationen, Additionen, Vergleiche, etc. Auf diese Weise erhält man ein maschinenunabhängiges Maß für die Rechenzeit.

Zur Illustration betrachten wir einige Beispiele. Wir benutzen dabei eine einfache <u>for</u>-Schleife, die folgendermaßen definiert ist:

<u>for</u> i:=1 <u>to</u> u <u>do</u> α <u>od</u> = i:=1 ; <u>while</u> i≤u <u>do</u> α ; i:=i+1 <u>od</u>

<u>Beispiel 3.32:</u> Wie oft wird die Wertzuweisung "x:=x+1" in folgenden Anweisungen ausgeführt?

1. x:=x+1	1mal
2. <u>for</u> i:=1 <u>to</u> n <u>do</u> x:=x+1 <u>od</u>	n-mal
3. <u>for</u> i:=1 <u>to</u> n <u>do</u> <u>for</u> j:=1 <u>to</u> n <u>do</u> x:=x+1 <u>od</u>	n^2-mal

Die Aufwandsfunktion $f: \mathbb{N} \to \mathbb{N}$ läßt sich in den wenigsten Fällen exakt bestimmen. Vorherrschende Analysemethoden sind

- Abschätzungen des *Aufwandes im schlechtesten Fall*
- Abschätzungen des *Aufwandes im Mittel*

Selbst hierfür lassen sich im allgemeinen keine exakten Angaben machen. Man beschränkt sich dann auf "ungefähres Rechnen in Größenordnungen". Das folgende Beispiel illustriert die Schwierigkeit von exakten Analysen, insbesondere bei Abschätzungen im Mittel, bereits in recht einfachen Fällen.

<u>Beispiel 3.33:</u> *Gegeben:* $n \geq 0$, $a_1, \ldots, a_n \in \mathbb{Z}$

Gesucht: der Index i der (ersten) größten Zahl unter den a_i, $i=1, \ldots, n$.

<u>Lösung:</u>

```
max:=1;
for i:=2 to n do
    if a_max < a_i then max:=i fi
od
```

Am Ende enthält max den gesuchten Index (bzw. die Zahl 1, falls n=0 ist).

<u>Analyse:</u> Wie oft wird die Anweisung "max:=i" im Mittel ausgeführt, abhängig von n?

Diese gesuchte mittlere Anzahl sei T_n. Offenbar gilt: $1 \leq T_n \leq n$.

Beobachtung: "max:=i" wird genau dann ausgeführt, wenn a_i das größte der Elemente $a_1, \ldots, a_i$ ist.

Annahme (Gleichverteilung): Für jedes i=1, . . . ,n hat jedes der Elemente $a_1, \ldots, a_n$ die gleiche Chance, das größte zu sein.

Das heißt, daß bei N Durchläufen durch den Algorithmus, N>>1, insgesamt $\frac{N}{i}$-mal die Anweisung "max:=i" ausgeführt wird für i=1, . . . ,n. Das heißt:

> max:=1 wird N-mal, d.h. immer, ausgeführt,
> max:=2 wird $\frac{N}{2}$-mal ausgeführt,
> etc.

Daraus folgt für die gesamte Anzahl $N \cdot T_n$ von Ausführungen von "max:=i" (für irgendein i) bei N Durchläufen:

$$\begin{aligned} N \cdot T_n &= N + \frac{N}{2} + \frac{N}{3} + \ldots + \frac{N}{n} \\ &= N \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \right) \end{aligned}$$

Damit haben wir $\quad T_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$,

und dies ist H_n, die n-te harmonische Zahl. Für H_n ist keine exakte geschlossene Formel bekannt, jedoch eine ungefähre Abschätzung:

$$T_n = H_n \approx \ln n + \gamma,$$

wobei γ=0.57721566 die Eulersche Konstante ist.

Da der Aufwand schon vom Ansatz her ohnehin nur grob abgeschätzt werden kann, macht eine Feinheit wie "$+\gamma$" im obigen Beispiel nicht viel Sinn. Interessant ist lediglich, daß T_n logarithmisch von n abhängt, nicht so sehr, wie dieser Zusammenhang im Detail aussieht. Man schreibt dafür

$$T_n = O(\log n),$$

und dies bedeutet, daß T_n "von der Ordnung log n" ist, wobei multiplikative und additive Konstante sowie die Basis des Logarithmus unspezifiziert bleiben.

Die O-Notation läßt sich mathematisch exakt definieren. Seien $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ gegeben.

Definition 3.34: $f(n) = O(g(n)) :\Leftrightarrow \exists c, n_0 \; \forall n \geq n_0 : f(n) \leq c \cdot g(n)$

Das heißt daß $\frac{f(n)}{g(n)}$ für genügend große n durch eine Konstante c beschränkt ist. Anschaulich bedeutet dies, daß f "nicht stärker wächst" als g.

Diese Begriffsbildung wendet man bei der Analyse von Algorithmen an, um Aufwandsfunktionen $f: \mathbb{N} \longrightarrow \mathbb{N}$ durch Angabe einer *einfachen Vergleichsfunktion* $g: \mathbb{N} \longrightarrow \mathbb{N}$ abzuschätzen, so daß $f(n) = O(g(n))$ gilt, also das Wachstum von f durch das von g beschränkt ist.

Beispiel 3.35: Für das obige Beispiel gilt genauer

$$T_n = H_n = \ln n + \gamma, \qquad \text{aber auch}$$
$$T_n = O(\log n)$$

Die Basis des Logarithmus ist unerheblich, da ein Basiswechsel gleichbedeutend ist mit der Multiplikation einer Konstanten: $\log_b n = \log_a n \cdot \log_b a$.

Gebräuchliche Vergleichsfunktionen sind die folgenden:

$O(1)$	konstanter Aufwand
$O(\log n)$	logarithmischer Aufwand
$O(n)$	linearer Aufwand
$O(n \cdot \log n)$	
$O(n^2)$	quadratischer Aufwand
$O(n^k)$ für ein $k \geq 0$	polynomialer Aufwand
$O(2^n)$	exponentieller Aufwand

Zur Veranschaulichung des Wachstums geben wir einige Werte ($ld = \log_2$).

f(n)	n = 2	$2^4=16$	$2^8=256$	$2^{10}=1024$	$2^{20}=1048576$
ld n	1	4	8	10	20
n	2	16	256	1024	1048576
n · ld n	2	64	1808	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$
n^3	8	4096	16777200	$\approx 10^9$	$\approx 10^{18}$
2^n	4	65536	$\approx 10^{77}$	$\approx 10^{308}$	$\approx 10^{315653}$

$\longrightarrow$ Grenze der "praktischen Berechenbarkeit"

Recht illustrativ ist es auch, sich vor Augen zu führen, welche Größenordnung ein Problem haben kann, wenn man die Zeit durch eine maximale Dauer T begrenzt. Wir nehmen dazu an, daß ein "Schritt" des Aufwandes genau eine Mikrosekunde (1 µs) dauert. Sei G das größte lösbare Problem in der Zeit T. Den Aufwand g(n) = log n lassen wir weg, da man hier praktisch unbegrenzt große Probleme in "vernünftiger" Zeit bearbeiten kann.

G	T = 1 Min.	1 Std.	1 Tag	1 Woche	1 Jahr
$g(n) = n$	$6 \cdot 10^7$	$3{,}6 \cdot 10^9$	$8.6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3 \cdot 10^{13}$
n^2	7750	$6 \cdot 10^4$	$2.9 \cdot 10^5$	$7.8 \cdot 10^5$	$5.6 \cdot 10^6$
n^3	391	1530	4420	8450	31600
2^n	25	31	36	39	44

Zu den einzelnen Komplexitätsklassen listen wir einige typische Klassen von Problemen auf, die sich mit diesem Aufwand, jedoch nicht mit geringerem, lösen lassen.

Aufwand	Problemklasse
$0(1)$, konstant	einige Suchverfahren für Tabellen ("Hashing")
$0(\log n)$, logarithmisch	allgemeinere Suchverfahren für Tabellen (Baum-Suchverfahren)
$0(n)$, linear	sequentielle Suche, Suche in Texten, syntaktische Analyse von Programmen (bei "guter" Grammatik)
$0(n \cdot \log n)$	Sortieren
$0(n^2)$, quadratisch	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume) Multiplikation Matrix-Vektor (einfach)
$0(n^3)$	Matrizen-Multiplikation (einfach)
$0(2^n)$, exponentiell	viele Optimierungsprobleme (z.B. optimale Schaltwerke) automatisches Beweisen (im Prädikatenkalkül 1. Stufe) etc.

Bemerkung: Es ist, streng genommen, ein offenes Problem, ob sich diese exponentiellen Probleme nicht doch mit polynomialem Aufwand lösen lassen. Es wäre jedoch eine große Überraschung, wenn sich dies herausstellen sollte.

4. Entwurf von Algorithmen

4.1 Strukturierung und Effizienz

Die Ziele beim Entwurf von Algorithmen haben sich im Verlauf der Geschichte der Informatik stark gewandelt. In den 50er und 60er Jahren ging es fast ausschließlich um *Effizienz*. Die Programmierer arbeiteten auf kleinen und langsamen Maschinen, und beträchtliche Mühe wurde darauf verwendet, hier ein paar Befehle und dort ein paar Byte Speicherplatz zu sparen, um so die teure Hardware optimal zu nutzen. Es war die große Zeit der Programmiertricks. Die Programme waren oft bewundernswerte Kunstwerke mit äußerst verschlungener und verwickelter Struktur, und es war oft nahezu unmöglich, ein derartiges Programm zu verstehen, auftretende Fehler zu korrigieren, das Programm für neue Anwendungen zu erweitern - oder zu prüfen, ob es korrekt ist.

In den 70er Jahren sanken die Hardwarekosten so dramatisch, daß plötzlich der Aufwand zur Entwicklung und Pflege von Programmen zum bestimmenden Kostenfaktor wurde. Die überschaubare *Struktur* eines Programms gewann neben seiner Effizienz zunehmend an Bedeutung.

Unter dem Stichwort "Strukturierte Programmierung" wurden Forderungen an den Programmierstil erhoben, die die Verständlichkeit von Programmen für den Menschen gegenüber der Effizienz in den Vordergrund stellten. Es wurden Methoden und Vorgehensweisen bei der Programmierung - insbesondere großer Programmsysteme ("Software Engineering") - vorgeschlagen, deren Hauptziele darin bestanden, möglichst systematisch und übersichtlich zu korrekten und zuverlässigen Programmen zu kommen, die sich später auch möglichst leicht an geänderte Anforderungen anpassen lassen. Systematische Programmentwicklung, Modularisierung, schrittweise Verfeinerung, Spezifikation vor der Implementierung, gute Dokumentation sowie viele Vorschläge für einen "guten" Programmierstil gewannen zunehmend an Bedeutung.

Natürlich ist Effizienz weiterhin ein wichtiges Entwurfsziel. Es gibt zunehmend Vorschläge, wie man die unbestreitbaren und unabdingbaren Vorteile einer guten Strukturierung bewahren kann, ohne die Aspekte der Effizienz zu vernachlässigen. Das Stichwort heißt *Programmtransformation*. In grober Näherung verläuft die Programmentwicklung nach dieser Idee in zwei Hauptphasen:

1. Entwurf eines gut strukturierten und korrekten Programms, ohne vorrangig auf Effizienz zu achten.
2. Anwendung von Programmtransformationen, die die Korrektheit bewahren und die Effizienz steigern.

Je nach Situation und Bedarf kann man mehr oder weniger Aufwand in die zweite Phase investieren. Man wird i.a. hier mehr Aufwand treiben, wenn das Programm sehr oft laufen soll, und weniger Aufwand, wenn es nur gelegentlich laufen soll.

Die Effizienzunterschiede zwischen den möglichen Lösungen eines Problems sind zuweilen recht dramatisch. Zur Illustration geben wir ein kleines Beispiel.

Beispiel 4.1: Gegeben sei ein Vektor von n reellen Zahlen $x_1, x_2, \ldots, x_n$. Gesucht ist der Wert der größten Summe aufeinanderfolgender Zahlen $x_i + x_{i+1} + \ldots + x_j$ für $1 \le i \le j \le n$, d.h.

$$\max \left\{ \sum_{k=i}^{j} x_k \mid 1 \le i \le j \le n \right\}$$

Sind z.B. x_i, $i=1,2, \ldots, 10$, als

$$31, -41, 59, 26, -53, 58, 97, -93, -23, 84$$

gegeben, so wäre die maximale Summe aufeinanderfolgender Zahlen

$$x_3 + \ldots + x_7 = 59+26-53+58+97 = 187$$

Keine andere solche Summe hat einen größeren Wert, z.B.

$$x_1 + \ldots + x_4 = 31-41+59+26 = 75$$

usw.

Auszugeben ist nur der Wert der größten Summe, also 187 in diesem Beispiel.

Eine offensichtliche, leicht verständliche und wohlstrukturierte Lösung ist die folgende (Deklarationen etc. weggelassen).

Lösung 1:

```
max:=0;
for i:=1 to n do
    for j:=i to n do
        sum:=0;
        for k=i to j do
            sum:=sum+x_k od;
    max:=MAX(max,sum)
    od;
od
```

Offenbar hat dieser Algorithmus eine Laufzeit von $O(n^3)$. Er ist also sehr langsam.

Nach einigem Nachdenken wird jeder Programmierer auf eine Idee kommen, wie man diesen Algorithmus wesentlich beschleunigen kann. Es gibt eine naheliegende Möglichkeit, einen $O(n^2)$-Algorithmus zu erreichen. Sie basiert auf der Idee, die innerste Schleife einzusparen und $x_i + \ldots + x_j$ in konstanter statt linearer Zeit zu berechnen. Hierzu kommt man durch die Beobachtung, daß man nacheinander x_i, x_i+x_{i+1}, $x_i+x_{i+1}+x_{i+2}$, $\ldots$, $x_i+\ldots+x_{j-1}$, $x_i+\ldots+x_j$ immer wieder neu berechnet. Es ist sicher intelligenter, Ergebnisse früherer Berechnungen für nachfolgende Berechnungen wiederzuverwenden.

Lösung 2:

```
max:=0;
for i:=1 to n do
        sum:=x_i;
        for j:=i+1 to n do
                sum:=sum+x_j;
                max:=MAX(max , sum)
        od
od
```

Dieser Algorithmus hat offensichtlich eine Laufzeit von $O(n^2)$. Er ist wesentlich schneller als der in Lösung 1. Ist er aber der schnellstmögliche?

Es ist eine Lösung bekannt (M. Shamos 1977), die mit einer Laufzeit von $O(n \log n)$ auskommt. Der Algorithmus ist sehr kompliziert und schwer verständlich. Entsprechend leicht macht man Fehler beim Programmieren, und entsprechend schwer sind Fehler zu finden und zu beseitigen. Wir wollen diese Lösung hier nicht behandeln, da sie nur kurze Zeit als schnellste gelten konnte. Bald danach gelang es J. Kadane (1977), einen *linearen* Algorithmus für unser Problem anzugeben!

Lösung 3:

```
max:=0; s:=0;
for i:=1 to n do
        s:=MAX(0,s+x_i);
        max:=MAX(max,s)
od
```

Offensichtlich ist die Laufzeit dieses Algorithmus $O(n)$. Es ist jedoch nicht ganz leicht einzusehen, daß er wirklich das Gewünschte leistet, daß er also korrekt ist.

Die Idee dieses Algorithmus ist die folgende: man geht einmal durch die Liste der gegebenen Zahlen $x_1, x_2, \ldots, x_n$ und merkt sich bei jedem x_i folgende Größen:

max_i: die maximale Summe $x_k + \ldots + x_h$ für $1 \le k \le h \le i$, also bis zur Stelle i.
s_i: die maximale Summe der Form $x_l + \ldots + x_i$, $1 \le l \le i+1$, die an der Stelle i endet (für $l=i+1$ ist dies die leere Summe mit dem Wert 0, d.h. es ist immer $s \ge 0$).

Haben wir diese Größen für die Stelle i, so bekommen wir sie für die nächste Stelle i+1 wie folgt:

Für max_{i+1} gibt es offenbar nur zwei Möglichkeiten: entweder ist x_{i+1} daran beteiligt oder nicht. Falls nicht, so ist $max_{i+1} = max_i$. Im anderen Fall ist $max_{i+1} = s_{i+1}$. Also gilt:

$$max_{i+1} = MAX(max_{i+1}, s_{i+1}).$$

Der Schlüssel zum Verständnis des Algorithmus liegt in der Berechnung von s_{i+1} aus s_i. Ist

$s_i+x_{i+1} \geq 0$, so ist $s_{i+1} = s_i+x_{i+1}$ die größte Summe, die an der Stelle i+1 endet, denn jede andere derartige Summe wäre von der Form $t+x_{i+1}$, wobei t eine Summe wäre, die an der Stelle i endet. Da s_i die größte an der Stelle i endende Summe ist, wäre $t \leq s_i$, und damit $t+x_{i+1} \leq s_i+x_{i+1}$. Ist $s_i+x_{i+1}<0$, so muß $x_{i+1}<0$ sein, denn es ist immer $s_i \geq 0$. In diesem Fall ist 0, d.h. die leere Summe, die größte an der Stelle i+1 endende Summe. Also gilt:

$$s_{i+1} = \mathrm{MAX}(0, s_i+x_{i+1}).$$

Damit haben wir bewiesen, daß die lineare Lösung 3 korrekt ist.

Ohne die Bedeutung der Effizienz für den Entwurf von Algorithmen in Abrede zu stellen, konzentrieren wir uns im folgenden auf strukturelle Aspekte. Für den Entwurf einer ersten Version eines Algorithmus ist es anzuraten, eher auf gute Strukturierung und Korrektheit zu achten als auf Effizienz. Ein zu frühzeitiges Streben nach Effizienz macht die Programmentwicklung langwieriger und fehleranfälliger. Außerdem birgt dies die Gefahr einer zu frühen Festlegung auf eine bestimmte konkrete Realisierung, die das Auffinden andersartiger – womöglich noch effizienterer – Lösungen behindert oder unmöglich macht. Es liegt daher durchaus im Sinne einer wohlverstandenen Berücksichtigung der Effizienz, wenn man sie in der *ersten* Phase des Entwurfs von Algorithmen nicht beachtet. Wir unterstreichen dies durch ein Zitat von Kernighan und Plauger (1978):

"Premature optimization is the root of all evil."

4.2 Abstraktion

Die Grundidee der strukturierten Programmierung läßt sich folgendermaßen formulieren:

> *Man entwerfe zunächst ein "abstraktes Programm" auf "abstrakten Daten" und konkretisiere dies durch "schrittweise Verfeinerung". Entscheidungen über konkrete Darstellungen - insbesondere der Daten - durch Mittel der Programmiersprache sind möglichst spät zu treffen.*

Um nämlich Entscheidungen über effiziente Realisierungen fundiert treffen zu können, sollte man möglichst viel über die Details des Algorithmus wissen, z.B. welche Operationen (am häufigsten) auf welchen Daten ausgeführt werden.

<u>Beispiel 4.2:</u> Wir entwerfen einen einfachen Sortieralgorithmus (Bubble Sort).

Gegeben sei eine Liste $X=\langle x_1, x_2, \ldots, x_n \rangle$ von natürlichen Zahlen, $x_i \in N$ für $i=1,2,\ldots,n$. Auszugeben sind diese Zahlen in sortierter Folge, d.h. eine Liste $Y=\langle x_{j_1}, x_{j_2}, \ldots, x_{j_n} \rangle$ derselben Zahlen, so daß gilt: $i \leq k \Rightarrow x_{j_i} \leq x_{j_k}$ für $i,k \in \{1, \ldots, n\}$.

Ansatz:

```
while [Liste nicht sortiert] do
      [Gehe durch Liste und vertausche benachbarte Zahlen, falls notwendig]
```

1. Verfeinerung: [Gehe durch . . .] ≙

```
for i:=1 to n-1 do
      if x_i>x_{i+1} then [Vertausche x_i und x_{i+1}] fi
od
```

2. Verfeinerung: [Liste nicht sortiert]

Hier ist es bei diesem sehr einfachen Beispiel wohl angebracht, einen Effizienzgesichtspunkt ins Spiel zu bringen: da wir die Liste ohnehin durchlaufen, empfiehlt es sich, bei dieser Gelegenheit gleich festzustellen, ob sie bereits sortiert ist und sie zu diesem Zweck nicht noch einmal getrennt zu durchlaufen.

Wir führen daher eine boolesche Variable B ein mit der Bedeutung

B = true:	die Liste ist sortiert
B = false:	die Liste ist nicht sortiert.

Damit ergibt sich folgender Stand für den Algorithmus:

```
B:=false;
while ¬B do
      B:=true;
      for i:=1 to n-1 do
            if x_i>x_{i+1} then [Vertausche x_i und x_{i+1}];
                                B:=false
            fi
      od
od
```

Wir hören an dieser Stelle auf. Zu entscheiden wäre jetzt die Darstellung der Liste. Die auszuführenden Grundoperationen sind

1. Sequentieller Zugriff auf die Listenelemente
2. Vergleich $x_i > x_{i+1}$
3. Vertauschen von x_i und x_{i+1}

Unter den vielen möglichen Darstellungen von Listen sollte eine gewählt werden, bei der diese Grundoperationen besonders effizient realisierbar sind (in diesem Falle ein Array).

In den nächsten Abschnitten werden wir größere - und sinnvollere - Beispiele für die Vorgehensweise bei der strukturierten Programmierung behandeln. Wesentlich ist, daß zunächst "abstrakte Programme" auf "abstrakten Daten" formuliert werden. Ein "abstraktes Programm" ist noch nicht direkt ausführbar. Es enthält noch "abstakte Operationen", die z.B. wie oben durch Prosa, in eckige Klammern eingeschlossen, notiert werden und noch weiter zu verfeinern sind. Es operiert auf "abstrakten Daten", wie z.B. Listen im obigen Beispiel, über deren Darstellung mit Mitteln der Programmiersprache noch nicht entschieden ist.

Das für diese Programmiermethodik typische Gemisch von Prosa und Programmiersprachelementen wird auch *Pseudocode* genannt.

Es ist häufig erstrebenswert, die bei der schrittweisen Verfeinerung entstehende Strukturierung des Programms auch im endgültigen Programmtext sichtbar zu machen. Hierzu - und zu manchen anderen nützlichen Zwecken - dienen Konzepte der *Abstraktion* in Programmiersprachen. Unter Abstraktionen versteht man Programmiersprachelemente, die es gestatten, größere Programmteile als Einheit zu definieren und sie als Einheit zu verwenden. Die grundlegenden Abstraktionen sind *Prozedur-Abstraktion* und *Daten-Abstraktion*.

Prozedur-Abstraktion ist in der Form des Prozedur-Konzepts in vielen Programmiersprachen verbreitet. Eine *Prozedur* ist ein (Unter-)Programm mit einem eigenen *Namen*, unter dem es in anderen Prozeduren oder im Hauptprogramm als eine "abstrakte" (anderswo konkretisierte) Operation aufgerufen werden kann. Dabei können *Parameter* übergeben werden. Mit einer Prozedur kann man es erreichen, daß gleiche oder "ähnliche" Berechnungen an verschiedenen Stellen eines Programms herausgezogen ("abstrahiert"), nur einmal als Prozedur formuliert und dann an den verschiedenen Programmstellen aufgerufen werden. Eine Prozedur kann andere Prozeduren aufrufen. Dabei kann sie auch - direkt oder auf Umwegen - sich selbst aufrufen. In diesem Fall heißt die Prozedur *rekursiv*. Rekursive Prozeduren bilden ein mächtiges Hilfsmittel bei vielen Techniken des Entwurfs von Algorithmen.

Wir gehen hier nicht auf die Details der Syntax und Semantik von Prozedur-Deklarationen, Prozedur-Aufrufen und der Parameterübergabe ein. Es sei hier auf den Programmierkurs verwiesen. In unseren Beispielen benutzen wir eine Ad-hoc-Syntax, deren Bedeutung aus dem Zusammenhang hinreichend deutlich wird.

<u>Beispiel 4.3:</u> Die abstrakte Operation [Vertausche x_i und x_{i+1}] im obigen Beispiel "Bubble Sort" könnte sehr wohl noch an vielen Stellen eines größeren Programms vorkommen. In diesem Fall könnte man an die Formulierung einer Prozedur denken:

```
proc Vertausche (x,y:int);
        begin z:int;
                z:=x; x:=y; y:=z
        end
```

Diese Prozedur kann dann bei Bedarf aufgerufen werden:

. . . ; Vertausche (a,b); . . . ; Vertausche (x_i,x_{i+1});

Beispiel 4.4: Als Beispiel für die Verwendung *rekursiver* Prozeduren entwerfen wir einen Algorithmus zum Umkehren von Texten. Die Aufgabe lautet: gegeben sei ein Text $x=x_1x_2 \ldots x_{n-1}x_n$ über einem Alphabet A, $x \in A^*$. Berechnet werden soll die Umkehrung (auch *Spiegelung* genannt) $x_nx_{n-1} \ldots x_2x_1$, z.B.

S A R G	$\longmapsto$	G R A S
R E G A L	$\longmapsto$	L A G E R
R O T	$\longmapsto$	T O R
O T T O	$\longmapsto$	O T T O
R E L I E F P F E I L E R	$\longmapsto$	R E L I E F P F E I L E R

Ein erster Ansatz zur Lösung diese Problems ist:

```
[Entferne den ersten Buchstaben];
[Kehre den Rest des Wortes um];
[Hänge den entfernten Buchstaben hinten an].
```

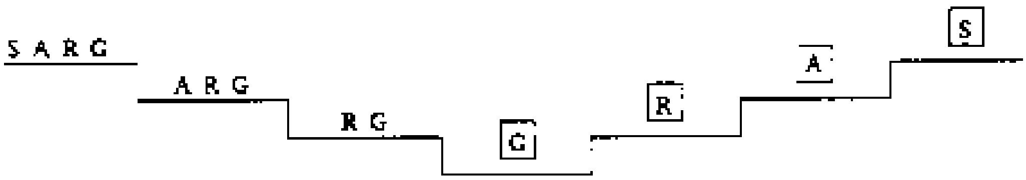
Dieser Ansatz führt unmittelbar auf eine rekursive Prozedur: das Problem wird auf ein "kleineres der gleichen Art" zurückgeführt:

```
proc Umkehrung (x: text);
    if [Länge(x) ≤ 1]
        then [Gib x aus]
        else [Entferne ersten Buchstaben von x];
            Umkehrung ([Rest von x]);
            [Gib entfernten Buchstaben aus]
    fi
```

die Aufruf-Struktur von

Umkehrung (S A R G)

läßt sich dann folgendermaßen veranschaulichen:



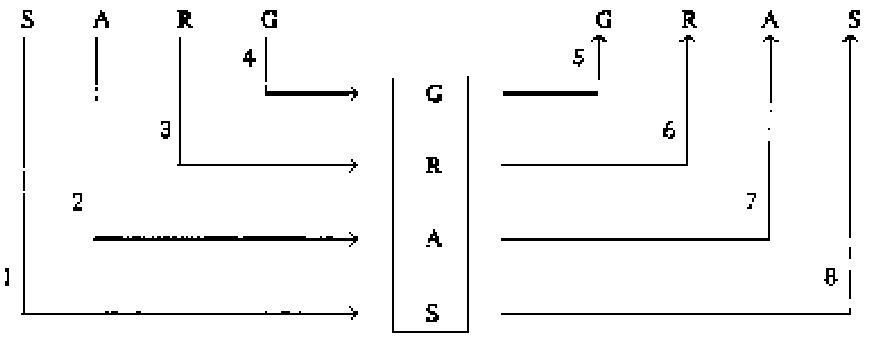
1. Aufruf mit S A R G ⟶ S entfernen und
2. Aufruf mit A R G ⟶ A entfernen und
3. Aufruf mit R G ⟶ R entfernen und
4. Aufruf mit G ⟶ $\boxed{G}$ ausgeben und
5. zurück zu 3. ⟶ $\boxed{R}$ ausgeben und
6. zurück zu 2. ⟶ $\boxed{A}$ ausgeben und
7. zurück zu 1. ⟶ $\boxed{S}$ ausgeben und
8. fertig

Man kann diese Aufgabe auch ohne Verwendung rekursiver Prozeduren, d.h. auf iterative Art und Weise, lösen. Das Aufbewahren der jeweils entfernten ersten Buchstaben bis zu ihrer Ausgabe muß dann selbst programmiert werden. Der hierfür geeignete "abstrakte" Datentyp ist der *Stapel*, auf den man Elemente herauflegen und von dem man Elemente herunternehmen kann. Heruntergenommen wird dabei jeweils das unter den Elementen im Stapel zuletzt aufgelegte Element (LIFO-Prinzip, "Last-In-First-Out").

Sei *S* ein solcher Stapel, und sei x der gegebene Text. Ein erster Ansatz für eine iterative Umkehrungs-Prozedur ist der folgende:

```
proc Umkehrung (x:text);
begin
      while [x nicht leer]
            do [Entferne ersten Buchstaben von x und lege ihn auf den Stapel S]
            od;
      while [S nicht leer]
            do [Nimm den obersten Buchstaben vom Stapel S und gib ihn aus]
            od
      end
```

Angewendet auf x = S A R G ergibt sich folgende Situation:



Stapel S

Die Zahlen 1, . . . ,8 bedeuten die Reihenfolge der Einlagerung bzw. Entnahme.

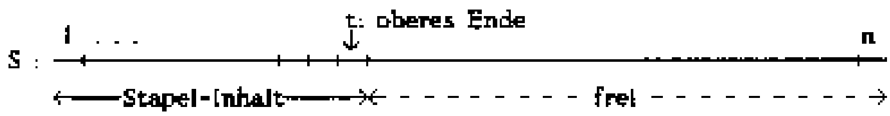
Der Stapel ist ein typisches Beispiel für einen "abstrakten" Datentyp, der in einer Programmiersprache nicht direkt vorhanden ist, sondern durch deren Sprachmittel dargestellt werden muß. Einige Programmiersprachen, z.B. SIMULA, CLU, MODULA, ADA, verfügen über Sprachmittel zur *Daten-Abstraktion*, mit denen sie zur Darstellung eines abstrakten Datentyps gehörenden Programmteile zu einer Einheit mit einem eigenen Namen zusammengefaßt werden können. solche Einheiten nennt man *(Daten-)Module* (oder auch Datenkapseln).

Um das Prinzip der Daten-Abstraktion zu erläutern, müssen wir zunächst den Begriff des abstrakten Datentyps etwas genauer fassen.

Ein *abstrakter Datentyp* wie z.B. "Stapel" wird zunächst bestimmt durch eine *Menge von Objekten*, z.B. den möglichen Stapelzuständen oder -inhalten. Dies ist jedoch nicht alles: wesentlich gehören dazu die *Operationen*, die auf diesen Objekten definiert sind. Das Prinzip der Daten-Abstraktion beruht u.a. auf der Vorschrift, daß die Objekte des Datentyps *nur über die definierten Operationen* des Datentyps zugegriffen bzw. manipuliert werden dürfen. Im Falle des Stapels sind dies z.B. folgende Operationen:

create:	schaffe einen leeren Stapel
push(x):	lege das Element x auf den Stapel
pop:	nimm das oberste Element vom Stapel
top:	Funktion, die das oberste Element des Stapels als Wert liefert
empty:	Abfrage, ob der Stapel leer ist.

Diese *Operationen* können durch *Prozeduren* bzw. *Funktionen* in einer Programmiersprache dargestellt werden. Die *Objekte* dagegen müssen durch die *Datenstrukturen*, die die Programmiersprache bietet, dargestellt werden. Im Falle des Stapels ist es naheliegend, ein Array S[1 . . . n] zu verwenden, wobei n eine festgelegte maximale Stapelgröße ist, sowie eine Integer-Variable t, die die augenblickliche Größe des Stapels anzeigt:



Eine mögliche Implementierung eines Stapel in einer hypothetischen Programmiersprache mit Datenabstraktion läßt sich dann folgendermaßen skizzieren:

```
module STACK . . . . .

representations          array S[1 . . . n] of elem;
                         int t;
                         . . .
proc create (a:rep); a.t:=0;
```

```
proc push (a:rep; e:elem);
        begin a.t:=a.t+1; a.S[a.t]:=e end;                (ohne Überlauf-Test!)
proc pop (a:rep);
        if empty(a) then [error] else a.t:=a.t-1 fi
fun top (a:rep) returns elem;
        if empty(a) then [error] else a.S[a.t] fi
fun empty (a:rep) returns bool; a.t=0

end module
```

Hierbei ist elem entweder ein fester gegebener Datentyp (wie z.B. int) oder eine "Datentyp-Variable", die es gestattet, beliebige Elementtypen als Parameter mitzugeben. Der Typ rep bezieht sich auf Objekte. Deren Komponenten werden mit Hilfe von "." angesprochen. Ist z.B. a vom Typ rep, so bezeichnet a.t die Integer-Variable t in der Struktur (der Stapel-Darstellung) a, und a.S bezeichnet deren Array.

4.3 Greedy-Algorithmen

Hier und in den nächsten beiden Abschnitten behandeln wir einige verbreitete Standard-Techniken für den Entwurf von Algorithmen, die zu charakteristischen Standard-Typen von Algorithmen führen. Auf diese - und noch einige weitere, hier nicht behandelte - Standardlösungen kann man beim Entwurf von Algorithmen häufig zurückgreifen.

Die wohl direkteste und geradlinigste Art von Algorithmen sind die *Greedy-Algorithmen* (greedy (engl.): gierig, gefräßig). Wir geben zunächst ein Beispiel, um deren Prinzip zu erläutern.

Beispiel 4.5: Auf Geldbeträge unter 1 DM soll Wechselgeld herausgegeben werden. Zur Verfügung stehen Münzen mit den Werten 50, 10, 5, 2 und 1 Pfennig, und zwar jeweils ein hinreichend großer Vorrat. Das Wechselgeld soll immer aus so wenig Münzen wie möglich bestehen.

So würde man z.B. 78 Pfennig in der Form 50, 2mal 10, 5, 2, 1 herausgeben. Der zu verwendende Algorithmus ist offensichtlich und entspricht verbreiteter Gepflogenheit: man nehme die größte Münze unter 78, also 50, und ziehe sie von 78 ab. Dies ergibt 28. Dann nehme man wieder die größte Münze unter 28, also 10, und ziehe sie ab. Das ergibt 18. So fahre man fort, bis sich der Rest 0 ergibt.

Der Algorithmus ist "gierig" (greedy) in dem Sinne, daß er immer ein lokales Optimum (die größte Münze) anstrebt, ohne sich um das globale Optimum zu kümmern. Im obigen Beispiel erreicht er dabei immer auch das verlangte globale Optimum (möglichst wenig Münzen). Das liegt jedoch an der speziellen Verteilung der Münzwerte. Hätten wir z.B. nur die Werte 11, 5 und 1 zur Verfügung und sollten 15 herausgeben, so würde der Greedy-Algorithmus 5 Münzen, nämlich 11 und 4mal 1, liefern. Das globale Optimum jedoch wären 3 Münzen, nämlich 3mal 5.

Greedy-Algorithmen haben einen weiten Anwendungsbereich. Für eine große Klasse von Problemen liefern sie optimale Lösungen, und wo sie es nicht tun, liefern sie doch häufig brauchbare Näherungslösungen. Insbesondere dort, wo exakte Lösungen nur mit sehr großem (z.B. exponentiellem) Aufwand zu bekommen sind.

Die allgemeine Problemstellung, auf die Greedy-Algorithmen anwendbar sind, läßt sich folgendermaßen charakterisieren:

1. Es gibt eine Menge von Eingabewerten.
2. Es gibt eine Menge von möglichen Lösungen, die aus Eingabewerten aufgebaut sind.
3. Die Lösungen lassen sich schrittweise aus partiellen Lösungen, beginnend bei der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen.
4. Es gibt eine Bewertungsfunktion für partielle und vollständige Lösungen.
5. Gesucht wird eine (die) optimale Lösung.

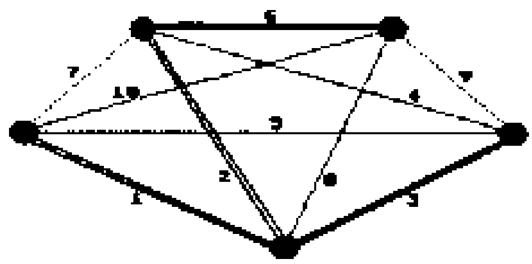
Das allgemeine Schema eines Greedy-Algorithmus sieht dann folgendermaßen aus:

```
proc GREEDY . . .
    begin . . .
        [Beginne mit leerer Lösung]
        while [Lösung noch unvollständig] do
            [Erweitere Lösung um eine Eingabe so, daß der Wert möglichst groß wird]
        od
    end
```

Beispiel 4.6: Zwischen n Knotenpunkten $P_1, \ldots, P_n$ soll ein möglichst billiges Kommunikationsnetz geschaltet werden, so daß jeder Knotenpunkt mit jedem anderen verbunden ist, ggf. auf einem Umweg über andere Knotenpunkte. Bekannt sind die Kosten d_{ij} für die direkte Verbindung zwischen P_i und P_j, $1 \le i,j \le n$.

Wir setzen voraus, daß alle Kosten d_{ij} verschieden und größer als Null sind. Dann ist es klar, daß die optimale Lösung keine Zyklen enthält (auf denen "Rundreisen" möglich wären). Denn sonst könnte man eine beliebige Verbindung im Zyklus weglassen; das Netz würde billiger werden und zusammenhängend bleiben.

Als konkretes Beispiel betrachten wir 5 Knotenpunkte mit folgenden Verbindungskosten:



Die Lösung besteht aus den Verbindungen mit den Kosten 1,2,3,6.

Das Problem läßt sich auch mathematisch formulieren. Eine Menge von *Knoten* mit Verbindungen, sog. *Kanten*, zwischen ihnen wird ein *Graph* genannt. Ein zusammenhängender Graph ohne Zyklen heißt *Baum*. Ein Graph mit n Knoten, die paarweise alle durch Kanten verbunden sind, heißt *vollständiger Graph* K_n.

Die Aufgabenstellung läßt sich nun graphentheoretisch folgendermaßen formulieren: für K_n mit Kantenbewertung $d_{i,j}$, $1 \le i, j \le n$, $d_{i,j} > 0$, alle $d_{i,j}$ verschieden, ist ein *aufspannender Baum* zu ermitteln, für den die Summe der gegebenen Kantenkosten minimal wird. Ein "aufspannender Baum" für K_n ist dabei einer, der aus Kanten von K_n besteht und *alle* Knoten von K_n umfaßt.

Man kann dies Problem für beliebige, nicht vollständige Graphen auf das Problem für K_n zurückführen, indem man die fehlenden Kanten mit derart hohen Kosten einfügt, daß sie in keiner optimalen Lösung vorkommen können.

Es gibt nur endlich viele verschiedene Bäume mit n Knoten. Man könnte daher alle generieren, von jedem die Kosten (als Summe der Kantenkosten) berechnen und den billigsten auswählen. Man weiß jedoch (Satz von Cayley), daß es n^{n-2} verschiedene Bäume mit n Knoten gibt. Diese Methode wird daher mit wachsendem n sehr schnell zu aufwendig. Daß unser Problem mit einem wesentlich effizienteren Greedy-Algorithmus lösbar ist, ist nicht ganz offensichtlich. Wir stellen hierzu einige Vorüberlegungen an.

Satz 4.7: Ein Baum mit $n > 0$ Knoten hat imer genau $n-1$ Kanten.

Beweis: vollständige Induktion nach n.

Satz 4.8: Sei B ein optimaler aufspannender Baum von K_n. Sei R ein Teilbaum von B. Sei b die billigste "von R ausgehende" Kante (d.h. ein Knoten innerhalb, der andere außerhalb von R). Dann liegt b in B.

Beweis: Wir nehmen an, b liege nicht in B. Dann konstruieren wir einen aufspannenden Baum B' von K_n, der billiger ist als B. Zunächst fügen wir b zum Baum B hinzu. Dadurch entsteht ein Zyklus, auf dem außer b noch eine weitere "von R ausgehende" Kante b' liegen muß, $b \neq b'$.

Nun entfernen wir b' aus B. Dadurch entsteht wieder ein Baum B', der K_n aufspannt und sich von B nur dadurch unterscheidet, daß die Kante b' gegen b ausgewechselt wurde. b und b' gehen beide von R aus, und da b die billigste solche Kante sein sollte, ist B' billiger als B.

Aus diesem Satz folgt, daß die optimale Lösung schrittweise aus Teilbäumen aufgebaut werden kann: man beginne bei einem beliebigen Knoten als Teilbaum und füge schrittweise jeweils die billigste vom Teilbaum ausgehende Kante hinzu, bis alle n Knoten verbunden sind.

Bemerkung: Die Annahme, daß alle $d_{i,j}$, $1 \le i, j \le n$, verschieden sein sollten, diente der Vereinfachung. Wie man jetzt sieht, erzwingt dies die Eindeutigkeit der optimalen Lösung.

Eine erste Grobversion unseres Algorithmus läßt sich nun aufstellen:

```
[Teilbaum R besteht anfangs aus einem beliebigen Knoten];
while [R noch nicht K_n aufspannt] do
      [Suche billigste von R ausgehende Kante];
      [Füge diese zu R hinzu]
od
```

Wir verfeinern den zentralen Schritt

```
      [Suche billigste von R ausgehende Kante]
```

weiter. Die anderen "abstrakten Operationen" sind leicht zu realisieren.

Hat der wachsende Teilbaum R gerade k Knoten, so liegen (n-k) Knoten außerhalb von R, und jeder Knoten von R ist mit jedem außerhalb R verbunden. Dies ergibt k(n-k) von R ausgehende Kanten. Sequentielles Durchsuchen all dieser Kanten würde $O(n^2)$ Schritte erfordern. Zusammen mit der äußeren while-Schleife, die genau (n-1)-mal durchlaufen wird, würde der Algorithmus dann eine Laufzeit von $O(n^3)$ haben. Die Frage ist, ob sich dies nicht wesentlich verbessern läßt.

Dies würde gelingen, wenn sich die Suche nach der billigsten von R ausgehenden Kante auf eine - möglichst kleine - Teilmenge der k(n-k) von R ausgehenden Kanten beschränken ließe. Wir nennen diese - noch unbekannte - Teilmenge V. Um nützlich zu sein, muß V den folgenden Forderungen genügen:

1. V muß immer die billigste von R ausgehende Kante enthalten.
2. V sollte wesentlich weniger als k(n-k) Kanten enthalten.
3. V sollte im Verlauf des Algorithmus leicht zu ändern sein, um stets der Forderung 1 zu genügen.

Für die Wahl von V bieten sich zwei naheliegende Möglichkeiten an:

A. V enthält für jeden Knoten P in R die billigste von P aus R herausführende Kante. Dann ist $|V| = k$.
B. V enthält für jeden Knoten P außerhalb R die billigste von P nach R hineinführende Kante. Dann ist $|V| = n-k$.

Das obige Kriterium 2 gibt keiner dieser Alternativen den Vorzug. Die Suche ließe sich in jedem Fall auf $O(n)$ beschleunigen, der gesamte Algorithmus also auf $O(n^2)$. So muß Kriterium 3, also die Effizienz der Änderung von V, den Ausschlag geben.

Ohne die Änderungsoperationen für beide Alternativen voll durchzudiskutieren, läßt sich eine Beobachtung machen, die der *Alternative B* den Vorzug gibt:

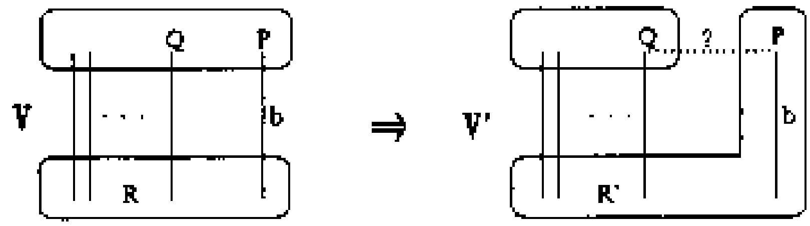
Bei der Alternative A können mehrere Kanten in V zum gleichen Knoten außerhalb R führen. Wir wissen dann von vornherein, daß höchstens eine von ihnen einmal zu R gehören wird. Die anderen müßten im Laufe der Zeit aus V verschwinden und gegen andere ausgetauscht werden.

Bei der Alternative B führt von jedem Knoten außerhalb R genau eine Kante nach R. Die Summe der Kanten in R und in V ist stets konstant n-1. Es ist möglich, daß alle Kanten in V im Laufe der Zeit zu R kommen, so daß dann nur Kanten aus V entfernt werden müßten.

Dies ist kein wirklicher Beweis, daß Alternative B besser ist, aber ein hinreichendes Indiz, um sich an dieser Stelle dafür zu entscheiden. Damit hat unser Algorithmus folgenden Stand erreicht:

```
[R:=ein beliebiger Knoten P];
[V:=alle n-1 nach P führenden Kanten];
for i:=1 to n-1 do
        [Suche billigste Kante b in V];
        [Füge b zu R hinzu];
        [Ändere V]
od
```

Wie muß nun die Operation [Ändere V] aussehen? Sei b die billigste Kante in V, und sei P der Endknoten von b außerhalb von R. Nun fügen wir b – und damit P – zu R hinzu und entfernen b aus V. Die Resultate seien R' und V'.



Die Anzahl der Kanten in V' ist korrekt, aber stimmen die Kanten in V' noch? Für jeden Knoten Q außerhalb R ist die bisherige Verbindung von Q nach R weiterhin die billigste Verbindung nach R', außer wenn die Kante Q–P billiger ist. In diesem Fall muß die bisherige Kante von Q nach R in V durch Q-P ersetzt werden, um die neue Kante von Q nach R' in V' zu ergeben. Die Änderung von V läßt sich demnach in (n–k) Schritten durchführen, also in O(n) Zeit. Damit hat sich die Einführung der Teilmenge V bewährt.

Der gesamte Algorithmus benötigt nun $O(n^2)$ Zeit und hat jetzt folgende Form:

```
[R:=ein beliebiger Knoten P];
[V:=alle n-1 nach P führenden Kanten];
for i:=1 to n-1 do
        [Suche billigste Kante b in V (Endknoten sei P)];
        [Füge b zu R hinzu]; [Entferne b aus V];
        for [alle Kanten c in V mit Endknoten Q] do
                if [Kosten von Q-P] < [Kosten von c]
                        then [Ersetze c durch Q-P]
                fi
        od
od
```

Jetzt ist es an der Zeit, über die Darstellung der "abstrakten" Datentypen Knoten, Kante, Baum, Kantenmenge V, Kosten, etc. zu entscheiden. Danach muß dann das Programm endgültig ausformuliert werden. Wir brechen unseren "logischen" Entwurf hier ab.

Bemerkung: Der obige Algorithmus stammt von R.C. Prim (1957). Für das allgemeine Problem mit beliebigen (nicht notwendig vollständigen) zusammenhängenden Graphen gibt es einen Algorithmus von J.B. Kruskal jr. (1956), der O(k log k) Schritte benötigt, wenn k die Anzahl der Kanten ist. Da der vollständige Graph K_n n^2 Kanten hat, ist Kruskals Algorithmus bei kleinen, Prims Algorithmus bei großen Kantenzahlen asymptotisch besser.

4.4 Divide-and-Conquer-Algorithmen

Eine der wichtigsten und am weitesten anwendbaren Entwurfstechniken für effiziente Algorithmen ist die Anwendung einer uralten Strategie, von den alten Römern "divide et impera" genannt (teile und herrsche, engl.: divide and conquer). Die Grundidee besteht darin, das gegebene Problem in mehrere getrennte Teilprobleme aufzuteilen, diese einzeln zu lösen, und die Lösung des ursprünglichen Problems aus den Teillösungen zusammenzusetzen. Dieselbe Technik läßt sich auf jedes der Teilprobleme anwenden, dann wieder auf deren Teilprobleme usw., bis die Teilprobleme klein genug sind, daß man eine Lösung explizit angeben kann.

Diese Technik führt dann zu besonders eleganten Algorithmen, wenn jedes Teilproblem wieder *von derselben Art* ist wie das ursprüngliche Problem, d.h. wenn es mit demselben Algorithmus gelöst werden kann. Dies führt auf natürliche Weise zu *rekursiven* Algorithmen.

<u>Beispiel 4.9:</u> Es sollen Spielpläne für Tennis-Turniere konstruiert werden. Es nehmen $n=2^k$, $k \in N$, Spieler teil. Im Verlauf von $n-1$ Tagen soll jeder Spieler täglich genau ein Spiel spielen, und dabei soll jeder Spieler gegen jeden anderen spielen.

Nehmen wir an, wir haben einen Turnierplan T_k für $n = 2^k$ Spieler. Daraus können wir dann einen Turnierplan T_{k+1} für $m = 2n = 2^{k+1}$ Spieler konstruieren.

T_k habe folgende Gestalt:

$$\begin{array}{c|ccccc} & 1 & 2 & \ldots\ j & \ldots & n-1 \\ \hline 1 & s_{11} & s_{12} & \ldots & & \\ 2 & s_{21} & \ldots & & & \\ \vdots & & & & & \\ i & & & s_{ij} & & \\ \vdots & & & & & \\ n=2^k & & & & & \end{array} \qquad T_k$$

s_{ij} ist der Spieler $(1,2,\ldots,n)$, gegen den Spieler i am Tage j des Turniers spielt.

Als konkretes Beispiel geben wir T_2:

	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

T_2

Aus T_k konstruieren wir nun T_{k+1} wie folgt:

$$\begin{array}{c|c|c|} & 1\ \ldots\ldots\ldots\ n-1 & n\ \ldots\ldots\ldots\ m-1 \\ \hline 1\ \vdots\ n=2^k & T_k & S_k \\ \hline n+1\ \vdots\ m=2^{k+1} & T_k+n & Z_k \\ \hline \end{array} \qquad T_{k+1}$$

Dabei ist T_k+n der Spielplan, der aus T_k entsteht, indem man zu jedem Matrixelement n hinzuaddiert. Z_k ist die $(n \cdot n)$-Matrix, $n=2^k$, die durch zyklisches Verschieben der Zeile $(1,2,\ldots,n)$ entsteht, z.B.

$$Z_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

S_k entsteht durch zyklisches Verschieben der Spalte $(n, \ldots, m-1)$, z.B.

$$S_2 = \begin{pmatrix} 5 & 8 & 7 & 6 \\ 6 & 5 & 8 & 7 \\ 7 & 6 & 5 & 8 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

Dieser Konstruktion von T_{k+1} liegt folgende Idee zugrunde: man *teile* die $m=2^{k+1}$ Spieler in zwei Gruppen $G1 = \{1, \ldots, n=2^k\}$, $G2 = \{n+1, \ldots, m\}$. In den ersten $n-1$ Tagen spielt jede Gruppe für sich nach dem Plan T_k. Danach spielen die Gruppen nach naheliegendem Plan gegeneinander.

Als Beispiele für die Konstruktion geben wir die Turnierpläne T_1, T_2 und T_3.

T_1

	1
1	2
2	1

T_2 (+2)

	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

T_3 (+4)

	1	2	3	4	5	6	7
1	2	3	4	5	8	7	6
2	1	4	3	6	5	8	7
3	4	1	2	7	6	5	8
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	3	4	1
7	8	5	6	3	4	1	2
8	7	6	5	4	1	2	3

Das allgemeine Schema einer nach dem "divide and conquer"-Prinzip aufgebauten rekursiven Prozedur sieht folgendermaßen aus:

```
proc DIVANDCONQ (P : problem);
    begin . . .
        if [P klein]
            then [explizite Lösung]
            else [Teile P auf in P1, . . . ,Pk];
                 DIVANDCONQ (P1);
                 . . . . .
                 DIVANDCONQ (Pk);
                 [Setze Lösung von P aus den Lösungen von P1, . . . ,Pk zusammen]
        fi
    end
```

Beispiel 4.10: Quicksort

Quicksort ist eines der leistungsfähigsten Sortierverfahren. Die Idee geht auf C.A.R. Hoare (1962) zurück. Das Problem besteht darin, eine gegebene Liste $X = \langle x_1,x_2, \ldots ,x_n\rangle$ von Größen, auf denen eine Ordnungsrelation $\leq$ definiert ist, zu sortieren. Wir nehmen der Einfachheit halber an, die x_i seien natürliche Zahlen, $x_i \in \mathbb{N}$ für $i=1, \ldots ,n$. Auszugeben ist dann eine Liste derselben Zahlen in sortierter Folge: $Y = \langle x_{j_1}, \ldots ,x_{j_n}\rangle$ mit $i \leq k \Rightarrow x_{j_i} \leq x_{j_k}$.

Hoares Idee besteht in einer Anwendung der "divide and conquer"-Strategie: man teile die Liste X so in zwei Teile

$$Y_l = \langle x_{j_1}, \ldots ,x_{j_{t-1}}\rangle, \qquad Y_r = \langle x_{j_{t+1}}, \ldots ,x_{j_n}\rangle,$$

daß für das ausgesparte "Mittelelement" x_{j_t} gilt:

1. $x_{j_i} \leq x_{j_t}$ für $i=1, \ldots ,t-1$
2. $x_{j_t} \leq x_{j_k}$ für $k=t+1, \ldots ,n$

Y_l enthält also keine größeren, Y_r dagegen nur größere Elemente als x_{j_t}. Dann ist das Element X_{j_t} bereits auf dem richtigen Platz in der endgültigen Sortierung. Das Problem läßt sich nun dadurch lösen, daß man die gleiche Methode auf Y_l und Y_r anwendet.

Der erste naive Ansatz besteht daher in einer rekursiven Prozedur der folgenden Form:

```
proc QUICKSORT (X : liste ; p , q : index);
    begin var t:index;
        [Teile X(p:q) = <xp, . . . ,xq> an der Stelle t];
        if p<t-1 then QUICKSORT (X,p,t-1);
        if t+1<q then QUICKSORT (X,t+1,q);
    end
```

Die Liste X wird dann durch einen Aufruf

QUICKSORT (X , 1 , n)

sortiert. Die Prozedur kann mit n=1 aufgerufen werden, generiert aber selbst nur Aufrufe für Teillisten mit mindestens 2 Elementen.

Der obige naive Ansatz läßt sich in seiner Aufrufstruktur wesentlich verbessern. Dies wird deutlicher werden, wenn wir die *Teilungsoperation* behandelt haben. Diese besteht darin, einen Index t sowie das zugehörige Element x_{j_t} der sortierten Liste zu finden. Dann werden die t-p ersten Elemente irgendwie (unsortiert) nach vorn nach X(p:t-1) gebracht, und die q-t letzten Elemente werden nach X(t+1:q) gebracht.

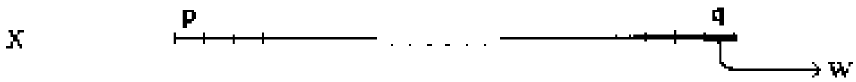
Für die Teilungsoperation sind zunächst zwei Möglichkeiten denkbar:

1. Man lege t fest und bestimme x_{j_t}.
2. Man lege ein Element x_s, $p \le s \le q$, fest und bestimme seinen Index t in der endgültigen Sortierung.

Die erste Alternative läßt sich mit vertretbarem Aufwand nicht realisieren (d.h. ohne komplett zu sortieren!). Daher bleibt nur die zweite Methode.

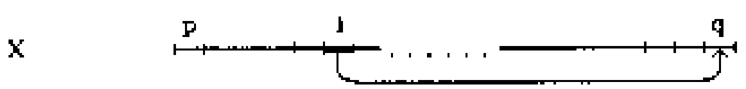
Ein wichtiger Aspekt bei "divide and conquer"-Methoden ist, daß die Teilprobleme möglichst den "gleichen Umfang" haben sollten. Nur eine solche *Ausgeglichenheit* garantiert die Effizienz des Verfahrens. In unserem Beispiel bedeutet dies, daß der Teilungspunkt t *möglichst in der Mitte* zwischen p und q liegen sollte. Um dies zu erreichen, könnte man daran denken, eine kurze Suche zu programmieren, um einen "halbwegs mittleren" Wert x_s zu wählen. Wir verzichten hier darauf und wählen willkürlich das *letzte* Element, also immer s=q.

Also ist x_q der Teilungswert. Wir müssen nun die kleineren Werte nach vorn und die größeren nach hinten bringen. Dazu benutzen wir folgendes Verfahren:



Wir setzen $W := x_q$ und geben die Position q in der Liste frei. Sie bildet ein *Loch*.

Nun gehen wir von links nach rechts, bis wir ein Element x_i gefunden haben, welches größer als W ist. Dieses bringen wir an die Stelle q, und das Loch ist an der Stelle i:



Jetzt gehen wir von rechts nach links, bis wir ein Element x_j gefunden haben, welches kleiner oder gleich W ist. Dieses bringen wir an die Stelle i, und das Loch ist an der Stelle j:

X 

Wir gehen jetzt abwechselnd von i weiter nach rechts, dann von j weiter nach links usw., bis sich i und j irgendwo treffen, d.h. das Loch seine endgültige Lage erreicht hat. Dies Loch kennzeichnet nun die Teilungsposition t, in die wir nun den Wert W hineinspeichern. Wir haben so gleichzeitig die Teilung vollzogen und den Teilungspunkt t bestimmt.

<u>Beispiel</u>:

⟶	7	9	2	1	5	4	3	8	*	⟶ $\boxed{6}$
⟵	*	9	2	1	5	4	3	8	7	
⟶	3	9	2	1	5	4	*	8	7	
⟵	3	*	2	1	5	4	9	8	7	
⟶	3	4	2	1	5	*	9	8	7	
<u>fertig</u>:	3	4	2	1	5	$\boxed{6}$	9	8	7	
	← – – –	– –	≤6	– –	– – →		← – –	>6	– – →	

Die wesentlichen Eigenschaften des Teilungs-Algorithmus sind damit skizziert, aber die Ablaufstruktur des Programms geht daraus noch nicht hervor. Naheliegend wäre vielleicht folgender Ansatz:

<u>while</u> [Loch noch nicht in Endposition]
<u>do</u> [Suche aufwärts ⟶ neues Loch];
[Suche abwärts ⟶ neues Loch] <u>od</u>

Dies würde jedoch zu einer häßlichen und verworrenen Programmstruktur führen, denn es gibt in den ineinandergeschachtelten Suchschleifen mehrere Abbruchkriterien, je nachdem, ob das Loch bei der Aufwärts- oder der Abwärtssuche erreicht wird. Dadurch wird es auch schwierig nachzuweisen, daß das Programm überhaupt anhält.

Ein eleganterer Ansatz benutzt *eine* Schleife, in der das Suchintervall j–i heruntergezählt wird. Eine boolesche Variable "ab" unterscheidet zwischen Ab- und Aufwärtssuche:

$$ab = \begin{cases} \text{true: abwärts} \\ \text{false: auswärts} \end{cases}$$

Der Teilungs-Algorithmus nimmt nun folgende Form an:

```
begin
    var i,j:index; W:elem; ab:bool;
    i:=p; j:=q; ab:=false; W:=x_q;
    while i<j do
        if ab
            then if x_j<W then x_i:=x_j; ab:=false fi
            else if W<x_i then x_j:=x_i; ab:=true fi
        fi;
        if ab then j:=j-1 else i:=i+1 fi
    od;
    x_j:=W; t:=j
end
```

Damit ist das Teilungs-Problem gelöst. Wir kehren nun zu unserem QUICKSORT-Algorithmus zurück. Unser erster Ansatz ist aus folgendem Grund naiv: die Teilungsoperation teilt die Liste möglicherweise in zwei ziemlich ungleiche Teile (z.B. wenn x_q das maximale Element in der Liste ist). Daraus ergibt sich, daß die "dynamische Tiefe" der rekursiven Aufrufe proportional mit n wachsen kann. Da rekursive Aufrufe auf der anderen Seite einen Speicherplatz beanspruchen, der zu ihrer dynamischen Tiefe proportional ist, könnte sich unser Programm als zu verschwenderisch in bezug auf Speicherplatz erweisen. Daraus könnte man folgern, daß diese rekursive Sortiermethode unpraktisch ist, aber das wäre voreilig. Tatsächlich können wir durch eine kleine Änderung erreichen, daß die maximale dynamische Tiefe den Wert $\log_2 n$ nicht übersteigt.

Dies erreichen wir dadurch, daß nach jeder Teilungsoperation die weitere Sortierung *nur für die kleinere der beiden Teillisten rekursiv* aufgerufen wird. Sind beide Teillisten gleich groß, so ist die Auswahl beliebig. Auf die andere Teilliste wird der Algorithmus dann *iterativ* angewendet.

In der folgenden QUICKSORT-Version werden zwei Hilfsvariablen u,v vom Typ index eingeführt. Sie zeigen auf die linke und rechte Grenze des noch nicht sortierten Abschnitts.

```
proc QUICKSORT (X:liste; p,q:index);
    begin var t,u,v,i,j:index ; W:elem ; ab bool;
        u:=p; v:=q;
        while u<v do
            [Teilungs-Operation, s. oben (ohne Deklarationen) ];
            if v-t < t-u
                then if t+1<v then QUICKSORT (X,t+1,v); v:=t-1 fi
                else if u<t-1 then QUICKSORT (X,u,t-1); u:=t+1 fi
            fi
        od
    end
```

Eine vollständige Analyse von QUICKSORT ist sehr verwickelt. Wir referieren hier nur die Ergebnisse.

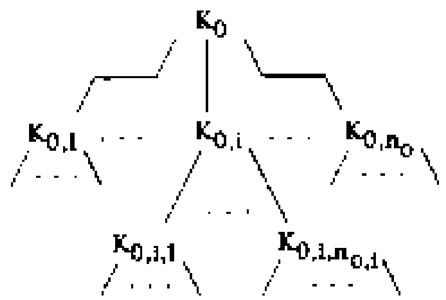
Zeit:	$O(n^2)$	im schlechtesten Fall
	$O(n \log n)$	im Mittel
Platz:	$O(\log n)$	im schlechtesten Fall und im Mittel

Es gibt Sortierverfahren mit einer Rechenzeit von $O(n \cdot \log n)$ im schlechtesten Fall. Obwohl QUICKSORT hier asymptotisch ungünstig ist, hat es sich im praktischen Einsatz häufig als überlegen erwiesen, so daß der Name gerechtfertigt ist. Trotzdem sollte man beachten: im entarteten Falle kann QUICKSORT zu einem "SLOWSORT" werden.

4.5 Backtracking-Algorithmen

Backtracking ("Zurückverfolgen") ist eines der grundlegenden Prinzipien des Algorithmen-Entwurfs. Es ist eine allgemeine systematische Suchtechnik, die auf Problemstellungen der folgenden Art angewendet werden kann.

Gegeben ist eine Menge KF von *Konfigurationen*. Gesucht ist eine Menge $L \subseteq KF$ von *Lösungen*. KF ist hierarchisch strukturiert: es gibt eine *Anfangskonfiguration* K_0, und zu jeder Konfiguration K_i gibt es eine Menge von *direkten Erweiterungen* $K_{i,1}, \ldots, K_{i,n_i}$:



Erweiterung von K_0

Erweiterung von $K_{0,i}$

Wir können bei jeder Konfiguration entscheiden, ob sie eine Lösung ist.

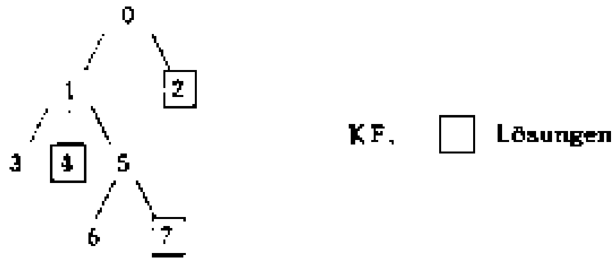
Ein Backtracking-Algorithmus zum Durchsuchen der Struktur der Konfigurationen und zum Auffinden aller Lösungen besteht aus einem Aufruf

$$\text{BACKTRACK}(K_0)\ .$$

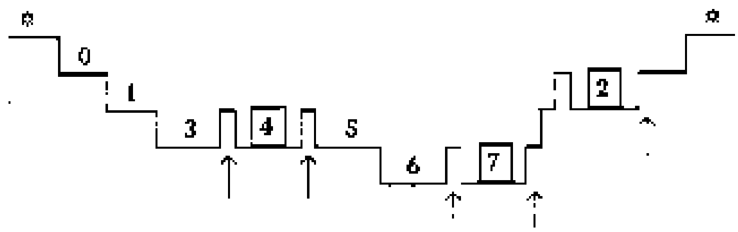
wobei BACKTRACK(K) eine rekursive Prozedur ist, die alle Lösungen unter den direkten und indirekten Erweiterungen von K findet:

```
proc BACKTRACK ( K: konfig );
    begin . . .
      for [jede direkte Erweiterung K' von K] do
        if [K' ist eine Lösung] then [Gib K' aus] fi;
        BACKTRACK ( K' )
      od
    end
```

Um die Arbeitsweise zu verstehen, betrachten wir ein konkretes Beispiel:



Bei einem Aufruf von BACKTRACK (0) ergibt sich folgende Aufrufstruktur:



Bei ^ geschieht "backtracking": in der Suche wird zurückgegangen auf eine höhere Ebene der Konfigurations-Hierarchie. Dies geschieht immer dann, wenn es keine direkten Erweiterungen der gerade betrachteten Konfiguration (mehr) gibt; dann wird die for-Schleife verlassen und das Ende der Prozedur erreicht.

Es werden – in dieser Reihenfolge – die Lösungen 4, 7 und 2 ausgegeben.

Typische Beispiele für Backtracking-Algorithmen finden sich in der Programmierung von Spielen wie Schach, Dame, usw. Hier geht es oft darum, mögliche Folgezüge und deren mögliche Konsequenzen abzusuchen, um aufgrund einer Bewertungsfunktion über den günstigsten nächsten Spielzug zu entscheiden.

Wir behandeln hier kein Spielprogramm, sondern ein klassisches kombinatorisches Problem in Anlehnung an das Schachspiel.

Beispiel 4.11: Acht-Damen-Problem

Es sollen alle Konfigurationen von 8 Damen auf einem 8x8-Schachbrett erzeugt werden, bei denen keine Dame eine andere "bedroht". Das heißt, daß keine zwei Damen in der gleichen Zeile, Spalte oder Diagonale stehen dürfen.

	1	2	3	4	5	6	7	8
1			♕					
2						♕		
3								♕
4	♕							
5				♕				
6							♕	
7					♕			
8		♕						

	1	2	3	4	5	6	7	8
1				♕				
2						♕		
3								♕
4		♕						
5							♕	
6	♕							
7			♕					
8					♕			

Zwei Lösungen

Es ist keineswegs offensichtlich, wie man die *Menge der Konfigurationen KF* am günstigsten festlegt. Sie sollte folgenden Bedingungen genügen:

1. L⊆KF für die gegebene Menge L von Lösungs-Konfigurationen.
2. Für jede Konfiguration k∈KF läßt sich leicht entscheiden, ob k eine Lösung ist.
3. Die Konfigurationen in KF lassen sich leicht schrittweise erweitern und bilden so eine hierarchische Struktur.
4. KF sollte nicht zu groß sein.

Ein naheliegendes Mittel, mögliche Kandidaten für KF zu finden, ist es, sich alle Bedingungen für L anzusehen und eine davon wegzulassen. Die Bedingungen für L sind:

(L1) Es sind 8 Damen auf dem Brett.

(L2) Keine zwei Damen bedrohen sich.

Läßt man eine dieser Bedingungen weg, so ergeben sich für KF folgende Alternativen:

KF1: alle Konfigurationen mit $n \geq 0$ Damen auf dem Brett, die sich nicht gegenseitig bedrohen
KF2: alle Konfigurationen mit 8 Damen auf dem Brett.

Beide Mengen sind aber derart riesig und unhandlich, daß sie zu äußerst unpraktischen Algorithmen führen würden.

In der Hoffnung, eine dieser Mengen auf ein praktikables Maß einschränken zu können, betrachten wir einige weitere offensichtliche Eigenschaften von Lösungen etwas genauer:

(a) Keine Zeile kann mehr als eine Dame enthalten. Da 8 Damen untergebracht werden müssen, muß jede Zeile genau eine Dame enthalten.

(b) Ähnlich schließen wir, daß jede Spalte genau eine Dame enthalten muß.

(c) Es gibt 15 "Aufwärts"-Diagonalen, von denen jede höchstens eine Dame enthält. 8 Aufwärtsdiagonalen enthalten also genau eine Dame, und 7 sind leer.

(d) Ähnlich schließen wir, daß 8 "Abwärts"-Diagonalen genau eine Dame enthalten und 7 leer sind.

(e) Haben wir irgendeine nicht-leere Konfiguration von Damen, die sich nicht gegenseitig bedrohen, und entfernen wir daraus eine beliebige Dame, so erhalten wir wieder eine Konfiguration von Damen, die sich nicht gegenseitig bedrohen.

Die letzte Eigenschaft (e) ist sehr bedeutsam. Dreht man sie um, so besagt sie, daß jede nicht-leere Konfiguration in KF1 (s.o.) *erzeugt* werden kann (auf n verschiedene Arten!), indem eine Konfiguration von KF1 mit n-1 Damen durch eine Dame *erweitert* wird. KF1 haben wir verworfen, weil es zu unhandlich ist, aber vielleicht finden wir eine passende Teilmenge $KF \subseteq KF1$, in der jede nicht-leere Konfiguration eine Erweiterung (um eine Dame) nur *einer einzigen* Konfiguration in KF ist? (Das Erweitern um eine Dame ist vermutlich eine leicht zu realisierende Operation.)

Eigenschaft (a) legt folgende Idee für die Teilmenge KF von KF1 nahe:

KF: alle Konfigurationen mit einer Dame in jeder der ersten n Zeilen, $0 \le n \le 8$, so daß je zwei Damen sich nicht bedrohen.

Nicht jede solche Konfiguration läßt sich zu einer Lösung erweitern, z.B. diese:

	1	2	3	4	5	6	7	8
1				♕				
2	♕							
3			♕					
4					♕			
5								
6								
7								
8								

Jedes Feld in Zeile 7 ist bereits bedroht, so daß hier keine Dame mehr gesetzt werden kann.

Offensichtlich gilt jedoch $L \subseteq KF$, und wir entscheiden uns hier dafür, unseren Backtracking-Algorithmus auf der Konfigurationenmenge KF aufzubauen:

```
proc PLAZIERE (i:[1 . . 8]);
   begin var h:[1 . . 8];
      for h:=1 to 8 do
         if [Feld in Zeile i, Spalte h nicht bedroht]
            then [Setze Dame auf dies Feld (i,h)];
               if [Brett voll] then [Gib Konfig. aus]
                               else PLAZIERE (i+1)
               fi
         fi
      od
   end
```

Ein Aufruf von

PLAZIERE (1)

wird alle gesuchten Lösungskonfigurationen ausdrucken. (Es gibt, nebenbei, 92 Lösungen.)

Erst jetzt, nachdem die wesentlichen Entwurfsentscheidungen für unseren Algorithmus gefallen sind, ist es an der Zeit, an die Darstellung der "abstrakten" Datentypen durch Mittel der Programmiersprache heranzugehen. (Dies sollte niemals zu früh geschehen!)

Die Darstellung der *Konfigurationen* ist kein großes Problem, aber auch nicht ganz offensichtlich. Es wäre eine große Ungeschicklichkeit, hier ein boolesches Array

Brett(1 . . 8, 1 . . 8)

zu wählen! Es würde unnötig Platz kosten, und die Operationen wie Setzen oder Entfernen einer Dame, Ermitteln der bedrohten Felder, etc. wären insgesamt nur mit großem Aufwand realisierbar.

Viel geschickter ist es, Konfigurationen in KF durch die Zahl $n \in (1, \ldots, 8)$ und ein Array X[1 . . 8] darzustellen, wobei $X[i] \in (1, \ldots, 8)$ ist für $1 \le i \le 8$. Dabei enthält x[i] die Spaltennummer h, in der sich die Dame in der i-ten Zeile befindet, $1 \le i \le h \le 8$.

Es gibt *keine logische Notwendigkeit*, weitere Variable einzuführen, wohl aber eine *praktische Zweckmäßigkeit*: mit den obigen Daten allein ist es sehr mühsam und zeitaufwendig herauszufinden, ob ein bestimmtes Feld in der nächsten Zeile bedroht ist. Wir führen daher weitere *redundante Daten* ein, um die Laufzeit des Algorithmus zu verbessern. (Sehr häufig kann man die Rechenzeit auf Kosten des Speicherplatzes verbessern und umgekehrt!)

Um zu sehen, welche Daten uns am besten weiterhelfen, betrachten wir die vorkommenden *Operationen*. Die Abfrage [Brett voll] wird leicht durch i=8 erledigt. Kritisch ist jedoch die Abfrage [Feld (i,h) nicht bedroht] in Abhängigkeit vom Setzen und Entfernen von Damen (letzteres passiert implizit beim "backtracking"!).

jedes Feld, an dessen Nicht-Bedroht-Sein wir interessiert sind, bestimmt je eine Zeile, eine Spalte, eine Aufwärts- und eine Abwärtsdiagonale durch dieses Feld. Das Feld ist genau dann nicht bedroht, wenn alle diese vier Linien frei von Damen sind. Demnach sollten wir *jederzeit Daten verfügbar* haben über die *leeren Spalten, Aufwärts- und Abwärtsdiagonalen*. Um die Zeilen brauchen wir uns nicht zu kümmern, denn die laufende Zeile i ist per definitionem leer.

Für die *Spalten* führen wir ein boolesches Array Spalte[1 . . 8] ein, so daß gilt:

$$\text{Spalte}[h] = \begin{cases} \text{true: Spalte h ist leer} \\ \text{false: sonst} \end{cases}$$

Für die *Abwärtsdiagonalen* ist die Differenz zwischen Zeilennummer i und Spaltennummer h konstant: $i-h = const$. Entlang einer *Aufwärtsdiagonalen* ist die Summe konstant: $i+h = const$. Wir führen daher ein:

- ein boolesches Array Auf[-7 . . 7]
- ein boolesches Array Ab[2 . . 16]

Hier halten wir fest, welche Diagonalen leer sind.

Nun wird [Feld (i,h) nicht bedroht] realisiert durch: $\text{Spalte}[h] \wedge \text{Auf}[i-h] \wedge \text{Ab}[i+h]$, und das Setzen und Wegnehmen einer Dame erfordert jeweils die Änderung von drei booleschen Werten, einem in jedem Array.

Nun ist unser Problem zufriedenstellend gelöst. Das Zusammenschreiben des Programms ist jetzt nur noch eine mechanische Tätigkeit, auf die wir hier verzichten wollen. Schließlich ist das Programm nicht wichtiger als das Acht-Damen-Problem selbst. Wichtig ist jedoch der Weg, auf dem wir es gefunden haben, und wichtig sind die Methoden, die wir dabei verwendet haben.