Gliederung der Grundvorlesung

1. Grundlagen der Programmierung

~~1.1 Algorithmen und Sprachen~~
~~1.2 Aussagen über Algorithmen~~
~~1.3 Daten und ihre Strukturierung~~
~~1.4 / 1.5 Grundbegriffe der Programmierung und die Sprache Ada 95~~
~~1.6 Komplexität von Algorithmen und Programmen~~
1.7 Semantik von Programmen

Gliederung des Kapitels

1.7 Semantik von Programmen

1.7.1 Korrektheit von Programmen, Zusicherungen
1.7.2 Hoaresche Regeln
1.7.3 Beispiele
1.7.4 Schwächste Vorbedingung
1.7.5 Terminierung

1.7.1 Korrektheit von Programmen

Wie lässt sich beweisen, dass ein Algorithmus, dargestellt als ein Programm, genau das realisiert, was er realisieren soll?

Ein solcher Beweis besteht aus zwei Teilen:
- dem Nachweis, dass das Programm für alle Eingaben anhält (Terminierung),
- dem Nachweis, dass das Programm für alle Eingaben richtig arbeitet (Korrektheit).

Die Terminierung entspricht dem Halteproblem (siehe 1.2.2.1 und 1.6.2.7); sie ist daher nicht entscheidbar und muss für jedes Problem individuell nachgewiesen werden (wobei vor allem die while-Schleife und die Rekursion Probleme bereiten).

Wir betrachten in diesem Kapitel vor allem die Korrektheit eines Algorithmus.

Dazu muss man zweierlei kennen und auf Gleichheit testen:
- die Spezifikation, was zu realisieren ist,
- die vom Programm realisierte Abbildung.

Erinnerung an Abschnitt 1.1.3: Zu jedem Programm π gehört eine Abbildung f_π: $E \to A$. Hierbei ist E die Menge der zulässigen Eingabedaten und A die Menge der Ausgabedaten des Programms. f_π heißt die von π realisierte Abbildung.

Die Spezifikation kann also erfolgen, indem die angestrebte Funktion (z.B. durch Gleichungen, vgl. Abschnitt 1.2.4) oder deren Ein-Ausgabe-Verhalten genau beschrieben wird.

Spezifikation:
f: $E \to A$

oder

Spezifikation:
Anfangs-Formel ....
Am Ende Formel....

```
x, z: Integer;
begin get(x);
z:= x + x;
put(z); end;
```

Realisierte Abbildung
f": $\mathbf{Z} \to \mathbf{Z}$ mit
f"(n) = 2n für alle $n \in \mathbf{Z}$

Zu beweisen: f = f" oder f" erfüllt die Formeln.

Beispiel 1.7.1.1:
Spezifikation: Die Funktion f: $\mathbf{Z} \to \mathbf{Z}$ mit f(n) = 2n für alle $n \in \mathbf{Z}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;
begin get(x);
for i in 1..x loop x := x+1; end loop;
put(x);
end;
```

Dieses Programm liest eine ganze Zahl a in die Variable x ein, addiert a mal eine 1 zu ihr und druckt sie dann aus. Für $a \geq 0$ ist dieses Vorgehen sicher richtig, doch für $a < 0$ wird nur der Eingabewert wieder ausgegeben. Also wird die Multiplikation mit 2 nicht von diesem Programm realisiert.

Beispiel 1.7.1.2:
Spezifikation: Die Funktion $f: \mathbf{Z} \to \mathbf{Z}$ mit $f(n) = 2n$ für alle $n \in \mathbf{Z}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;
begin get(x);
put(x, Base=>2);
put("0");
end;
```

Dieses Programm liest eine ganze Zahl ein, druckt sie zur Basis 2 aus und fügt die Ziffer 0 am Ende an. Diese eventuell richtige Idee wird aber dadurch zunichte gemacht, dass Ada.Integer_Text_IO die Zahlen zu einer Basis, die nicht 10 ist, in "#" einschließt. Bei Eingabe von 13 wird also 2#1101#0 statt 11010 ausgegeben.

Beispiel 1.7.1.3:
Spezifikation: Die Funktion $f: \mathbf{Z} \to \mathbf{Z}$ mit $f(n) = 2n$ für alle $n \in \mathbf{Z}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;  Negativ: Boolean;
begin get(x);
  Negativ := x<0;
  if Negativ then x:=-x; end if;
  for i in 1..x loop x:=x+1; end loop;
  if Negativ then x:=-x; end if;
  put(x);
end;
```

In diesem Programm wird ein negativer Wert zunächst ins Positive umgewandelt, dann wird der Wert verdoppelt und im negativen Fall wieder negiert. Man vermutet, dieses Programm sei nun richtig?! Ist das schon ein Beweis?

Problem: Wie ermittelt man die Bedeutung eines Programms (also die realisierte Abbildung)? Wir wollen hier nur eine Methode betrachten, nämlich die prädikatenlogische; sie wird in der Literatur auch *"axiomatische Semantik"* genannt.

Man geht davon aus, dass zu jedem Zeitpunkt des Programmablaufs gewisse Bedingungen erfüllt sind; diese beschreibt man als prädikatenlogische Formeln. Eine solche Formel wird aufgebaut aus Booleschen Ausdrücken und den Quantoren $\forall$ und $\exists$ ("für alle" und "es existiert").

Typische prädikatenlogische Formeln sind:

$(x=y) \vee (x+1=y)$ d.h., x oder x+1 ist gleich y.

$\forall x : (x = x)$ d.h. für alle Werte x gilt, dass x gleich x ist.

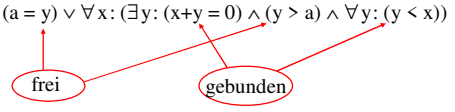
$\forall x : (\exists y : (x+y = 0))$
d.h., zu jedem x gibt es ein y, so dass x+y Null ist.

Solche Formeln können wahr oder falsch sein (man sagt auch, sie können *erfüllt* sein oder nicht), wobei stets der Wertebereich, aus dem die Werte der Variablen sein müssen, zu beachten ist. Die erste der obigen Formeln ist beispielsweise falsch, wenn man die ganzen Zahlen zugrunde legt; sie ist aber richtig, wenn man "modulo 2" auf der Menge {0, 1} rechnet. Die zweite Formel ist stets wahr. Die dritte Formel ist für ganze Zahlen wahr, für natürliche Zahlen falsch.

Definition: In solchen Formeln können Variablen frei oder gebunden vorkommen. Ein Vorkommen der Variablen x heißt *gebunden*, wenn dies im Bereich eines Quantors steht, also wenn das x irgendwo in einem Teilausdruck der Form $\exists x :(...)$ oder $\forall x : (...)$ ist. Steht x außerhalb jedes solchen möglichen Teilausdrucks, so heißt dieses Vorkommen *frei*.

Zum Beispiel ist kommt die Variable y in folgender Formel zweimal frei und zweimal gebunden vor:

$(a = y) \vee \forall x : (\exists y : (x+y = 0) \wedge (y > a) \wedge \forall y : (y < x))$



1.7.1.4: Solche Bedingungen / prädikatenlogische Formeln nennt man ***Zusicherungen*** oder ***assertions*** und schreibt sie in geschweifte Klammern zwischen die Anweisungen.
Jede Anweisung verändert diese Zusicherungen. Wenn vor der Durchführung einer Anweisung c die Zusicherung A erfüllt war und nach der Ausführung von c die Zusicherung B erfüllt ist, so schreibt man hierfür

{A} c {B}.

Wir betrachten zunächst ein triviales Beispiel, wobei wir die prädikatenlogischen Formeln umgangssprachlich formulieren:

```
x: Integer;
{Der Wert von x ist noch undefiniert}
get(x);
{Der Wert von x ist eine ganze Zahl a}
x := x+2;
{Der Wert von x ist a+2}
put(x);
{Der Wert a+2 wurde ausgegeben}
```

Als Wertebereich für die prädikatenlogischen Formeln wählen wir hier stets die ganzen Zahlen. Hinzu kommt der Fall "undefiniert", den wir durch das Zeichen ⊥ darstellen.

Die Aussage "Der Wert von x ist noch undefiniert" beschreiben wir nun durch die Zusicherung

$x = \perp$

Die Eingabe get(x) beschreiben wir durch die Zusicherung

$a \in \mathbf{Z} \wedge x = a$

Schließlich wird "der Wert von x ist a+2" dargestellt durch

$x = a + 2$

Damit erhalten wir aus der umganssprachlichen Version folgendes Programm mit zusätzlichen Zusicherungen zwischen je zwei aufeinander folgenden Anweisungen:


```
x: Integer;
{x = ⊥}
get(x);
{a ∈ Z ∧ x = a}
x := x+2;
{x = a + 2}
put(x);
```

Die Ausgabe lässt sich hiermit nicht beschreiben; man erhält sie, indem man die Werte aller Variablen, die ausgegeben werden, in der Anweisung davor betrachtet.

Die realisierte Abbildung dieses Programms lässt sich nun schrittweise an jeder Anweisung beweisen (in obigem Fall muss man nur die Bedeutung der Anweisungen einsetzen).

1.7.2 Hoaresche Regeln

Dies sieht so aus, als ob der Mensch als Beweiser benötigt wird. Das stimmt aber nur teilweise; denn große Teile solcher Beweise lassen sich automatisieren.

Es gelten nämlich folgende sechs Regeln von C.A.R.Hoare (englischer Informatiker). Hierbei werden die Ein- und Ausgabe nicht berücksichtigt; sie müssen zusätzlich hinzugefügt werden.

Vorbemerkung: Jede Regel $\frac{X}{Y}$ bedeutet:

Wenn X zutrifft, dann trifft auch Y zu.

1.7.2.1: Die sechs Regeln von Hoare:

Für alle Zusicherungen A, B, C, für alle Wertzuweisungen x:=ß , für alle Booleschen Ausdrücke b und für alle Anweisungen c, d:

1. $\frac{\text{true}}{\{A\}\ \varepsilon\ \{A\}}$, wobei ε die leere Anweisung (null) ist.

2. $\frac{\text{true}}{\{A'\}\ \mathbf{x:=ß}\ \{A\}}$, wobei A' aus A entsteht, indem man in A alle freien Vorkommen von x durch ß ersetzt (außer: x war zuvor undefiniert -- klar).

3. $\frac{\{A\}\ \mathbf{c}\ \{B\} \wedge \{B\}\ \mathbf{d}\ \{C\}}{\{A\}\ \mathbf{c;d}\ \{C\}}$

Für alle Zusicherungen A, A", B, B", für alle Booleschen Ausdrücke b und für alle Anweisungen c, d:

4. $\frac{\{A \wedge b\}\ \mathbf{c}\ \{B\} \wedge \{A \wedge \underline{\text{not}}(b)\}\ \mathbf{d}\ \{B\}}{\{A\}\ \mathbf{if}\ b\ \mathbf{then\ c\ else\ d\ end\ if}\ \{B\}}$

5. $\frac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \mathbf{while}\ b\ \mathbf{loop\ c\ end\ loop}\ \{A \wedge \underline{\text{not}}(b)\}}$ Ein solches A heißt Schleifen-invariante.

6. $\frac{A \Rightarrow A'' \wedge \{A''\}\ \mathbf{c}\ \{B''\} \wedge B'' \Rightarrow B}{\{A\}\ \mathbf{c}\ \{B\}}$ Konsequenzregel

Man überlege sich an einigen Beispielen, dass diese Regeln korrekt sind:

$\{X \geq 0\}\ X := X + 1;\ \{X > 0\}$ nach Regel 2

$\{X \geq 0\}\ X := X+1;\ X:=X+1;\ \{X \geq 2\}$ nach Regel 3

$\{X > 0\}$ if X mod 2 = 0
then X := X+1;
else X := X+2; end if; $\{X > 1\}$ nach Regel 4

$\{X \geq 0\}$ while X > 1 loop
X := X-2; end loop; $\{X \geq 0 \wedge X \leq 1\}$ nach Regel 5 mit A = (X ≥ 0)

$\{X+Y=a\}$ if X > Y then X := X-2;
else Y := Y-2; end if; $\{X+Y=a-2\}$ nach Regel 4

1.7.3 Beispiele

Beispiel 1.7.3.1: Was macht folgendes Programmstück?

```
x, y, z: Natural;
begin
    get(x); get(y); z:=0;
    while y > 0 loop
        if (y mod 2 = 0) then
            y:=y div 2;
            x:=x+x;
        else
            y:=y-1;
            z:=z+x;
        end if;
    end loop;
    put(z);
end;
```

Füge geeignete gültige Zusicherungen ein. Dann erst weiterlesen.


```
x, y, z: Natural;
{x=⊥ ∧ y=⊥ ∧ z=⊥}
begin get (x); get (y);
{a∈IN0∧a≥0∧x=a ∧ b∈IN0∧b≥0∧y=b ∧ z=⊥}
z:=0;
{a∈IN0∧a≥0∧x=a ∧ b∈IN0∧b≥0∧y=b ∧ z=0}
while y > 0 loop
    if (y mod 2 = 0) then
        y:=y div 2;
        x:=x+x;
    else
        y:=y-1;
        z:=z+x;
    end if;
end loop;
put(z); end;
```

Wir betrachten nun nur die while-Schleife.


```
{a∈IN0∧a≥0∧x=a ∧ b∈IN0∧b≥0∧y=b ∧ z=0} ⇒
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
while y > 0 loop
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
    if (y mod 2 = 0) then
    {x≥0 ∧ y>0 ∧ ∃k: y=2k ∧ z≥0 ∧ z+x·y=a·b}
        y := y div 2;
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·2y=a·b}
        x := x + x;
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
    else
        y:=y-1;
        z:=z+x;
    end if;
end loop;
```

then-Zweig betrachten


```
{a∈IN0∧a≥0∧x=a ∧ b∈IN0∧b≥0∧y=b ∧ z=0} ⇒
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
while y > 0 loop
    {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
    if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
    else
    {x≥0 ∧ y>0 ∧ ∃k: y=2k+1 ∧ z≥0 ∧ z+x·y=a·b}
        y:=y-1;
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y+x=a·b}
        z:=z+x;
    {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
    end if;
end loop;
```

else-Zweig betrachten

Es gilt also

```
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
else
        y:=y-1;
        z:=z+x;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
end if;
```


Mit der Konsequenzregel folgt hieraus

```
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
else
        y:=y-1;
        z:=z+x;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
end if;
```

Also gilt nach der vierten Regel:

```
{x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
if (y mod 2 = 0) then
        y := y div 2;
        x := x + x;
else
        y:=y-1;
        z:=z+x;
end if;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
```

Wir setzen dies ein:


```
{a∈IN0∧a≥0∧x=a ∧ b∈IN0∧b≥0∧y=b ∧ z=0} ⇒
        {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
while y > 0 loop
        {x≥0 ∧ y>0 ∧ z≥0 ∧ z+x·y=a·b}
        if (y mod 2 = 0) then
                y := y div 2;
                x := x + x;
        else
                y:=y-1;
                z:=z+x;
        end if;
        {x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
end loop;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b ∧ y≤0} ⇒
        {y=0 ∧ z=a·b}
put(z);
```



```
x, y, z: Natural;
begin  get (x),  get(y);  z:=0;
{x≥0 ∧ y≥0 ∧ z≥0 ∧ z+x·y=a·b}
while y > 0 loop
        if (y mod 2 = 0) then
                y:=y div 2;
                x:=x+x;
        else
                y:=y-1;
                z:=z+x;
        end if;
end loop;
{y=0 ∧ z=a·b}
put(z); end;
```

Also wird das Produkt zweier nicht-negativer Zahlen berechnet.

Beispiel 1.7.3.2: ***Plateau-Problem***

Gegeben: n ≥ 1 und ein sortiertes Feld ganzer Zahlen
a: array (1..n) of Integer;

Gesucht ist die maximale Anzahl gleicher Zahlen, also
Max {d≥1 | ∃ i mit a(i) = a(i+1) = ... = a(i+d-1)}.

Dieser Wert heißt auch "maximale Plateaulänge" von a.

Lösungsvorschlag:

" Lies n und das array a ein. Sortiere a. Danach:"

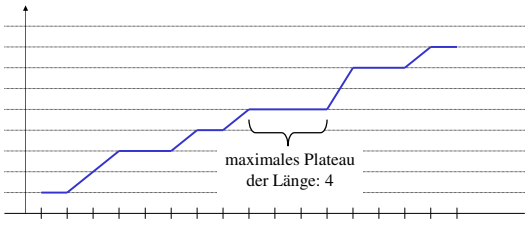
```
j := 1; p := 1;
while j /= n loop
    j := j+1;
    if a(j) = a(j-p) then  p := p+1; end if;
end loop;
```

} abgekürzt durch **μ**

" Ergebnis ist p. "

Erläuterung des Namens "Plateau-Problem" am Beispiel:

Stelle das sortierte Feld mit der maximalen Plateaulänge 4
1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 7, 7, 7, 8
grafisch dar:



Behauptung: Dieses Programmstück **μ** berechnet die maximale Plateaulänge des sortierten Feldes a.

Beweis: Zur Abkürzung definieren wir:
MP(j,p) ⇔ p ist die maximale Plateaulänge des Teilfelds a(1..j).

Wenn für alle j gilt: MP(j,p), wobei p der aktuelle Wert von p ist nach Durchlauf der Schleife für den Wert j, dann gilt MP(n,p) für den letzten Wert von p, womit die Behauptung bewiesen wäre. Zu Beginn von **μ** gilt:
n ≥ 1 ∧ a ist sortiert.

Hieraus folgt, dass anfangs gilt: MP(1,1) ∧ a ist sortiert.

Nun gehen wir die einzelnen Anweisungen durch:

{MP(1,1) ∧ a ist sortiert}
j := 1;
{MP(j,1) ∧ a ist sortiert}
p := 1;
{MP(j,p) ∧ a ist sortiert}
while j /= n loop
{MP(j,p) ∧ a ist sortiert ∧ j ≠ n}
    j := j+1;
    {MP(j-1,p) ∧ a ist sortiert (∧ j-1 ≠ n) }
    if a(j) = a(j-p) then p := p+1; end if;
    Behauptung: **{MP(j,p) ∧ a ist sortiert}**
end loop;

Regel 2:

$$\frac{\text{true}}{\{A'\}\ \mathbf{x:=ß}\ \{A\}}$$

, wobei A' aus A entsteht, indem man in A alle freien Vorkommen von x durch ß ersetzt.

{MP(j-1,p) ∧ a ist sortiert}
if a(j) = a(j-p) then
{MP(j-1,p) ∧ a ist sortiert ∧ a(j) = a(j-p)}
⇒ {MP(j,p+1) ∧ a ist sortiert} -- es gibt eine um 1 längere Folge
  p := p+1;
  {MP(j,p) ∧ a ist sortiert} -- Regel 2
else
{MP(j-1,p) ∧ a ist sortiert ∧ a(j) ≠ a(j-p)}
⇒ {MP(j,p) ∧ a ist sortiert} -- es ist keine längere Folge enstanden
  null;
  {MP(j,p) ∧ a ist sortiert} -- Regel 1
end if;
{MP(j,p) ∧ a ist sortiert} -- Regel 4

{MP(1,1) ∧ a ist sortiert}
j := 1;
{MP(j,1) ∧ a ist sortiert}
p := 1;
{MP(j,p) ∧ a ist sortiert}
while j /= n loop
{MP(j,p) ∧ a ist sortiert ∧ j ≠ n}
    j := j+1;
    {MP(j-1,p) ∧ a ist sortiert}
    if a(j) = a(j-p) then p := p+1; end if;
    {MP(j,p) ∧ a ist sortiert}
end loop;
{MP(j,p) ∧ a ist sortiert ∧ j=n} ⇒ {MP(n,p)}

µ berechnet also tatsächlich die maximale Plateaulänge, sofern **µ** terminiert. Dies geschieht nach genau n Schleifendurchläufen.

was zu beweisen war

Bei solchen Beweisen muss man stets darauf achten, dass man alle Voraussetzungen auch verwendet.

Frage an Sie: Wo genau wurden im Beweis die beiden Voraussetzungen

" n >= 1 " sowie
" a ist sortiert "

benutzt?

Gehen Sie den Beweis nochmals im Detail durch. Machen Sie sich auch die Terminierung klar.

Beispiel 1.7.3.3: ***Quicksort***
Gegeben: n ≥ 1 und ein Feld ganzer Zahlen
A: array (1..n) of Integer;
Dieses Feld A soll sortiert werden. Das Verfahren Bubblesort (siehe 1.4.4.4) wiederholt für j = 1, 2, ..., n-1 folgendes: durchlaufe A(j+1..n) und vertausche zwei aufeinander folgende Zahlen genau dann, wenn die größere vor der kleineren steht.
Bubblesort benötigt im Mittel $O(n^2)$ Vergleiche; es ist in der Praxis nur für kleinere Felder geeignet (bis etwa n=50; für größere n gibt es deutlich schnellere Verfahren).
Eines der schnellsten Verfahren für in der Praxis auftretende Probleme ist "Quicksort" (1962 von C.A.R.Hoare vorgestellt). Es wählt ein "Pivot-Element" p aus und ordnet das Feld so in zwei Teilfelder A(1..m) und A(m+1..n) um, dass im ersten Teil alle Elemente kleiner oder gleich p und im zweiten Teil alle Elemente größer oder gleich p sind. Das Verfahren arbeitet rekursiv.

... *type Index is 1..n; ... A: array (Index) of Integer;* ...
procedure Aufteilen (L, R: Index) is -- A ist global, A(L..R) wird sortiert
i, j, m: Index; p, h: Integer; -- p wird das Pivot-Element
begin
if L < R then
    i := L; j := R; m := (L+R)/2; p := A(m);
    while i <= j loop -- die Indizes i und j laufen aufeinander zu
        while A(i) <= p loop i := i+1; end loop;
        while A(j) >= p loop j := j-1; end loop;
        if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                i := i+1; j := j-1; end if;
    end loop;
    Aufteilen(L, j); Aufteilen(i, R);
end if;
end Aufteilen;
... *Aufteilen(1,n);* ... -- *Aufruf des Sortierverfahrens*

Vorsicht: Dieses Programm enthält einen Fehler!

```
procedure Aufteilen (L, R: Index) is
i, j, m: Index; p, h: Integer;
{A(L..R) ist ein Feld ganzer Zahlen}
if L < R then
{A(L..R) ist ein Feld mit mindestens 2 ganzen Zahlen (R-L >=1)}
    i := L; j := R; m := (L+R)/2; p := A(m);
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in
A(L..i-1) sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}
   while i <= j loop

   end loop;
   Aufteilen(L, j); Aufteilen(i, R);
end if;
end Aufteilen;
```

Vorsicht: Dieses Programm enthält einen Fehler!


```
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}
   while i <= j loop
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p ∧ i <= j}
       while A(i) <= p loop i := i+1; end loop;
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ A(i) > p ∧ alle Elemente in A(j+1..R) sind >= p}
       while A(j) >= p loop j := j-1; end loop;
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ A(i) > p ∧ alle Elemente in A(j+1..R) sind >= p ∧ A(j) < p}
       if i <= j then  h:=A(i); A(i):=A(j); A(j):=h;
                       i := i+1; j := j-1;  end if;
   end loop;
```

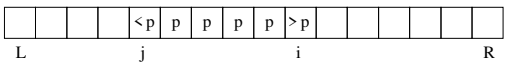


```
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ A(i) > p ∧ alle Elemente in A(j+1..R) sind >= p ∧ A(j) < p}
       if i <= j then  h:=A(i); A(i):=A(j); A(j):=h;
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i)
sind <= p ∧ alle Elemente in A(j..R) sind >= p ∧ i <= j}
                       i := i+1; j := j-1;
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}
       end if;
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}
   end loop;
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p ∧ i > j}  ⇒ (Wunsch!)
{R-L >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..j)
sind <= p ∧ alle Elemente in A(i..R) sind >= p ∧ i > j}
```


Diese letzte Folgerung ist gesondert zu beweisen. Hierfür berechnen wir, welche Werte i und j am Ende der äußeren while-Schleife besitzen können.

Fall 1: Unmittelbar vor dem Ende der while-Schleife traf die if-Bedingung zu. Falls hierbei i=j war, so muss A(i) > p und A(j) < p gegolten haben. Dies ist aber unmöglich. Folglich müssen i<j und A(i) > p und A(j) < p gewesen sein. Da anschließend die while-Schleife abbricht, muss "neues i" = i+1 > j-1 = "neues j" sein; zusammen mit i<j folgt hieraus j - i = 1, d.h., nach Abarbeiten des then-Zweigs muss i - j = 1 gelten, wobei A(i) > p und A(j) < p ist. Folglich sind alle Element in A(L..j) kleiner oder gleich p und alle Elemente in A(i..R) größer oder gleich p.

Fall 2: Unmittelbar vor dem Ende der while-Schleife traf die if-Bedingung nicht zu. Dann muss die Abbruchbedingung i > j der while-Schleife durch die inneren while-Schleifen erreicht worden sein. Es muss A(i) > p und A(j) < p gelten, weil dies die Abbruchbedingungen der inneren while-Schleifen sind. Dann kann wiederum nicht i=j gelten. Da aber A(i-1) <= p gewesen sein muss (sonst wäre die innere while-Schleife bereits mit i-1 terminiert), müssen alle Elemente in A(j+1..i-1) gleich p sein:



Es folgt: Alle Elemente in A(L..j) sind <= p ∧ alle Elemente in A(i..R) sind >= p ∧ alle Elemente in A(j+1..i-1) sind = p ∧ i > j .

Fazit: Wenn wir nun die Rekursion aufrufen, so genügt es, dies für den Bereich von L bis j und von i bis R zu tun. Unter der Annahme, dass durch

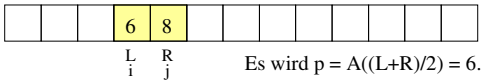
Aufteilen(L, j); Aufteilen(i, R);

die Bereiche A(L..j) und A(i..R) sortiert werden, und zusammen mit der Tatsache, dass A(j+1..i-1) nur aus Werten gleich p bestehen kann (sofern dieser Bereich nicht leer ist), folgt, dass die Prozedur "Aufteilen" den Bereich von L bis R korrekt sortiert.

Aber nur, sofern i und j den Bereich L..R nicht verlassen und der rekursive Aufruf stets mit kleineren Bereichen erfolgt!!

Und genau dieser Fehlerfall tritt bei obigem Programm auf !

Irgendwann stößt die Rekursion auf einen kleinen, z.B. einen zweielementigen Teilbereich (L+1=R):



Es wird p = A((L+R)/2) = 6.

In der ersten inneren while-Schleife läuft i nach rechts, bis i=R ist, wegen A(R) = 8 > p. Danach läuft der Index j absteigend von R über die linke Grenze L hinaus, weil jedes Mal A(j) >= p ist. Fehler!

Diesen Fehler kann man beseitigen, indem man entweder zusätzliche Abfragen bzgl. der Bereichsgrenzen einfügt oder indem man in den inneren while-Schleifen auch bei Gleichheit mit p die Vertauschung durchführt; das heißt, die Bedingungen müssen dann lauten

```
while A(i) < p loop i := i+1; end loop;      -- statt A(i) <= p
while A(j) > p loop j := j-1; end loop;      -- statt A(i) >= p
```


Wegen der jetzt geltenden Zusicherung

{R-L >=1 ∧ p ist ein Element aus A(L..R)
∧ alle Elemente in A(L..i-1) sind <= p ∧ **A(i) >= p**
∧ alle Elemente in A(j+1..R) sind >= p ∧ **A(j) <= p**}

können die Indizes i und j nun nicht mehr übereinander hinweglaufen. Vielmehr kann der zweite Index j höchstens um eine Stelle nach links über den Index i gelangen, d.h., beide innere while-Schleifen enden stets mit Werten i und j, für die gilt

i - j <= 1.

Da p ein Element von A(L..R) ist, müssen beide Indizes spätestens stehen bleiben, wenn sie auf p treffen, d.h., die if-Bedingung ist mindestens einmal erfüllt und damit gilt anschließend i > L und j < R. Die Rekursion erfolgt also stets mit kleineren Grenzen. Dies sichert die Terminierung.

Wir formulieren auf der nächsten Folie das korrekte Verfahren Quicksort und führen auf den anschließenden Folien den Nachweis der Korrektheit.

Hinweis 1: zur Laufzeit des Verfahrens. Im Mittel benötigt Quicksort ca. 1,39·n·log(n) Vergleiche, wie wir im Kapitel "Sortieren" in Teil III beweisen werden.

Hinweis 2: zur Laufzeit des Verfahrens. Im schlechtesten Fall, wenn zufällig p immer das Maximum oder Minimum des Bereichs A(L..R) ist, benötigt Quicksort $O(n^2)$ Schritte (man sagt, das Verfahren wird zum "Slowsort").

Hinweis 3: zum Pivotelement. Wir benötigen im Beweis nur, dass p im Bereich A(L..R) liegt. Man kann p daher beliebig im Feld A wählen (z.B. auch A(L) oder A(R) oder als Minimum der drei Werte A(L), A((L+R)/2) und A(R); bzgl. des Pivotelements ist Quicksort also nichtdeterministisch).

1.7.3.4: ... *type Index is 1..n; ... A: array (Index) of Integer; ...*

```
procedure Quicksort (L, R: Index) is   -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;            -- p wird das Pivot-Element
begin
if L < R then
    i := L; j := R; "wähle p beliebig aus A(L..R)";
   while i <= j loop                   -- die Indizes i und j laufen aufeinander zu
        while A(i) < p loop i := i+1; end loop;
        while A(j) > p loop j := j-1; end loop;
        if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                       i := i+1; j := j-1;  end if;
   end loop;                           -- auch bei Gleichheit A(i)=p oder A(j)=p vertauschen!
   Quicksort(L, j); Quicksort(i, R);
end if;
end Quicksort;
```

... Quicksort(1,n); ... *-- Aufruf des Sortierverfahrens*

Korrektheit dieses Verfahrens:

```
procedure Quicksort (L, R: Index) is
i, j, m: Index; p, h: Integer;
{A(L..R) ist ein Feld ganzer Zahlen}
if L < R then
{A(L..R) ist ein Feld mit mindestens 2 ganzen Zahlen (L-R >=1)}
    i := L; j := R; m := (L+R)/2; p := A(m);
{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in
A(L..i-1) sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}
   while i <= j loop

   end loop;
   Quicksort(L, j); Quicksort(i, R);
end if;
end Quicksort;
```



```
{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}
   while i <= j loop
{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ alle Elemente in A(j+1..R) sind >= p ∧ i < j}
        while A(i) < p loop i := i+1; end loop;
{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ A(i) >= p ∧ alle Elemente in A(j+1..R) sind >= p}
        while A(j) > p loop j := j-1; end loop;
{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1)
sind <= p ∧ A(i) >= p ∧ alle Elemente in A(j+1..R) sind >= p ∧ A(j) <= p}
        if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                       i := i+1; j := j-1;  end if;
   end loop;
```

{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind <= p ∧ A(i) >= p ∧ alle Elemente in A(j+1..R) sind >= p ∧ A(j) <= p}

if i <= j then h:=A(i); A(i):=A(j); A(j):=h;

{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i) sind <= p ∧ alle Elemente in A(j..R) sind >= p}

i := i+1; j := j-1;

{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}

end if;

{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind <= p ∧ alle Elemente in A(j+1..R) sind >= p}

end loop;

{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..i-1) sind <= p ∧ alle Elemente in A(j+1..R) sind >= p ∧ i > j} ⇒

{L-R >=1 ∧ p ist ein Element aus A(L..R) ∧ alle Elemente in A(L..j) sind <= p ∧ alle Elemente in A(i..R) sind >= p ∧ i > j}

Die weiteren Folgerungen können von vorne übernommen werden.
Man beachte, dass Quicksort terminiert, da die if-Bedingung mindestens einmal erfüllt sein muss, die Prozedur also den then-Zweig ausführt und daher i um mindestens 1 erhöht und j um mindestens 1 erniedrigt wird.

Hinweis 4: zum Speicherplatzbedarf. Quicksort benötigt Platz für die Rekursion. Wenn man die Rekursion jeweils mit dem größeren Bereich zuerst durchführt, so kann der zusätzliche Platz bis zu 2n Speicherplätze betragen, um jedes Mal das Paar (i,R) abzulegen. Dies lässt sich auf 2·log(n) verringern, indem man stets zuerst den kürzeren Bereich nimmt, d.h., indem man die beiden rekursiven Aufrufe ersetzt durch:
if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
else Quicksort(i, R); Quicksort(L, j); end if;

1.7.4 Schwächste Vorbedingung

1.7.4.1: Vorgehensweise zum Nachweis der Korrektheit:
Schreibe zwischen je zwei Anweisungen eine prädikatenlogische Formel und versuche mit Hilfe der Hoareschen Regeln zu bewiesen, dass jeweils {A} c {B} gilt.
Da die Hoareschen Regeln sich nur auf die Anweisungen "leere Anweisung", "Wertzuweisung", "Hintereinanderausführung von Anweisungen", "Alternative" und "while-Schleife" beziehen, können wir dieses Schema bisher auch nur auf Programme anwenden, die höchstens aus diesen Bestandteilen bestehen.
Will man beliebige Ada-Programme untersuchen, so muss man weitere Regeln einführen.
(Solche weiteren Regeln kann man aufstellen. Wir tun dies nicht, weil es hier um das Prinzip geht und weil weitere Regeln rasch recht kompliziert werden.)

Kann man dieses Vorgehen automatisieren, d.h., kann man ein Programm schreiben, das
- ein Programm einliest,
- die Spezifikation einliest,
- schrittweise die erforderlichen Zusicherungen ermittelt und
- den Beweis der Korrektheit führt?

Nein, denn das Problem, die Korrektheit eines Programms zu beweisen, ist algorithmisch nicht lösbar. Es lässt sich auf das Halteproblem zurückführen, vgl. Abschnitt 1.2.2. Formale Beweise hierzu werden in der Theoretischen Informatik geliefert (z.B. der Satz von Rice).

Aber: Man kann natürlich ein interaktives Programm, also ein "Unterstützungssystem" bauen, welches
- ein Programm einliest,
- die Spezifikation einliest,
- diese Spezifikation als letzte Zusicherung verwendet und versucht, die vor der letzten Anweisung einzufügende Zusicherung zu konstruieren oder den Benutzer aufzufordern, einen Vorschlag einzugeben,
- einen Beweis für die Korrektheit dieses einen Schrittes zu führen oder den Benutzer aufzufordern, einen solchen Beweis einzugeben und diesen nachzuvollziehen,
- das Gleiche für die Anweisung davor zu wiederholen usw.

Auf diese Weise kann für viele Programme ein Beweis der Korrektheit ermöglicht und teilweise sogar automatisiert werden. Entsprechende "Programmbeweiser" sind verfügbar.

Wie muss ein Programmbeweiser vorgehen?
Der wichtigste Schritt ist:

Ausgehend von einer Zusicherung und der davor stehenden Anweisung muss man versuchen die Zusicherung zu konstruieren, die vor der letzten Anweisung einzufügen ist.

Also: Finde zu der Situation

c {B}

eine (möglichst schwache) Zusicherung A, so dass

{A} **c** {B}

gilt.

(Das geht in vielen Fällen tatsächlich mittels folgender "wp".)

1.7.4.2 Triviales Beispiel: Gegeben sei

x := x +1;
{x > 7}

Dann wird eine Zusicherung vor der Wertzuweisung gesucht:

{x > 6}
x := x +1;
{x > 7}

Dies entspricht der Hoareschen Regel: {A'} **x:=ß** {A}, wobei A' aus A durch Ersetzen von x durch ß hervorgeht.

Man hätte auch die Zusicherung {x > 20} nehmen können:

{x > 20}
x := x +1;
{x > 7}

Welche Zusicherung soll man nehmen?

Man denke an die Konsequenzregel. Es gilt:

{x > 20} ⇒ {x > 6}
x := x +1;
{x > 7}

Die Zusicherung {x>6} ist "schwächer" als die Zusicherung {x>20}. "Schwächer" bedeutet: Die Zusicherung {x>6} trifft auf mehr Werte zu als die Zusicherung {x>20}.

{x>0} wäre eine noch schwächere Bedingung. Leider gilt aber die Formel

{x > 0}
x := x +1; **falsch**
{x > 7}

nicht mehr, da x ja einen der Werte 1, 2, 3, 4, 5 oder 6 haben könnte. Gibt es zu **c** und B vielleicht eine "schwächste" Zusicherung A, für die {A} **c** {B} zutreffend ist? **Ja!**

1.7.4.3 Bezeichnungen: Wir müssen nun in der Formel

{A} **c** {B}

zwischen den Zusicherungen A und B unterscheiden. Statt "Zusicherung" sagt man oft auch "Bedingung", und daher nennt man

A die Vorbedingung zur Anweisung **c**
(englisch: *precondition*)

und

B die Nachbedingung zur Anweisung **c**
(englisch: *postcondition*).

Unsere Aufgabe lautet also: Suche zu der Situation

c {B}

eine Vorbedingung A, so dass

{A} **c** {B}

gilt. Wenn es mehrere solche Vorbedingungen gibt, dann suche die "schwächste" Vorbedingung, d.h., suche ein A mit:

1. {A} **c** {B}
2. wenn für irgendeine Zusicherung A' gilt: {A'} **c** {B}, dann ist A eine Folgerung aus A', d.h., dann gilt A' ⇒ A.

Definition 1.7.4.4:
Eine solche Zusicherung A heißt "schwächste Vorbedingung" zu **c** und B (englisch: *weakest precondition,* abgekürzt wp).

Man schreibt dann: **A = wp(c,B)**.

Als erstes müssen wir fragen, ob unsere Definition in sich stimmig ist (man sagt auch, ob sie "wohldefiniert" ist). Es könnte ja sein, dass es zu **c** und B in der Regel keine schwächste Vorbedingung gibt oder dass es mehrere gibt.

Ohne Beweis notieren wir hier:
Es gibt stets eine schwächste Vorbedingung zu **c** und B.

Dass diese immer eindeutig ist, ist leicht zu sehen: Wenn es zwei solche schwächsten Vorbedingungen A und A' gäbe, dann müssen gleichzeitig A' ⇒ A und A ⇒ A' gelten. Zwei solche Zusicherungen sind aber gleichwertig, d.h., es gilt dann A' ⇔ A.

Beispiel 1.7.4.5:
c ≡ if (X mod 2 = 1) then X:=X+3; end if;
B ≡ X mod 6 = 0
Gesucht ist wp(**c**,B).

Betrachte hierzu alle Belegungen der Variablen X, für die nach Durchführung der Anweisung **c** die Zusicherung B gilt:
{a∈ **Z** |falls a ungerade ist, dann muss a+3 durch 6 teilbar sein;
falls a gerade ist, dann muss a durch 6 teilbar sein}
= {a∈ **Z** | a ist durch 3 teilbar}.

Eine Zusicherung, die genau für die Werte dieser Menge erfüllt ist, muss zur schwächsten Vorbedingung gehören (also zur größten Menge, die die Zusicherung erfüllt). Also gilt hier:

wp(**c**,B) ≡ X mod 3 = 0

Wenn **c** eine Wertzuweisung ist, dann ist wp(**c**,B) in der Regel leicht zu berechnen und dies kann auch automatisiert werden. Die Hintereinanderausführung und die Alternative sind auch noch gut zu handhaben. Als Beispiel betrachte man:

c ≡ if X<0 then X:=-X+1; else X:=X-1; end if;
B ≡ X > 5

Betrachte zunächst den then-Zweig:
c_1 ≡ X := -X + 1;
B ≡ X > 5
wp(c_1, B) lautet X < -4

Dies muss man noch koppeln mit der Bedingung b ≡ X < 0 für den then-Zweig. Dies ergibt die Zusicherung
(X < -4) ∧ (X < 0) , also X < -4.

c ≡ if X<0 then X:=-X+1; else X:=X-1; end if;
B ≡ X > 5

Betrachte nun den else-Zweig:
c_2 ≡ X := X - 1;
B ≡ X > 5
wp(c_2, B) lautet X > 6

Dies muss man koppeln mit der Bedingung not(b) ≡ X ≥ 0. So erhält man die Zusicherung
(X > 6) ∧ (X ≥ 0) , also X > 6 für den else-Zweig.

Aus beiden Vorbedingungen erhält man die schwächste Vorbedingung der gesamten Alternative als Disjunktion:
wp(**c**, B) ≡ (X < -4) ∨ (X > 6).

1.7.4.6: Wesentlich schwieriger ist die Behandlung der while-Schleife. Als Beispiel betrachte man:

c ≡ while (X /= 0) loop X := X-2; end loop;
B ≡ X mod 3 = 0
wp(**c**, B) lautet ?

"X mod 3 = 0" ist keine Schleifeninvariante, da die Formel {(X mod 3 = 0) ∧ (X /= 0)} **X := X-2;** {X mod 3 = 0} nicht erfüllt ist (vgl. Hoaresche Regel 5, sie steht auch auf der nächsten Folie).

Dagegen sind sowohl "X mod 2 = 0" als auch "X mod 2 = 1" Schleifeninvarianten.

Diese Information nützt uns aber nichts. Betrachten wir stattdessen nochmal die 5. Hoaresche Regel:

$$\frac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \textbf{while b loop c end loop}\ \{A \wedge \underline{not}(b)\}}$$

In unserem Fall ist not(b) ≡ (X = 0), so dass B ≡ X mod 3 = 0 stets erfüllt ist, egal mit welchem Wert von X die Schleife begonnen wurde. Die Frage, ob die Schleife terminiert, spielt nach der Definition von {A} **c** {B} keine Rolle: Denn es soll B nach der Ausführung von **c** erfüllt sein (vgl. 1.7.4.4); wenn jedoch die Schleife nicht endet, "dann ist danach alles erfüllt".

Somit erhalten wir als die schwächste Vorbedingung obiger Schleife die Zusicherung: X∈ **Z**, d.h., die Nachbedingung ist für jede ganze Zahl, mit der man startet, erfüllt.

Im Allgemeinen lässt sich die schwächste Vorbedingung für eine Schleife nicht algorithmisch berechnen. Wer jedoch ein Programm entwickelt, kennt mindestens eine Schleifeninvariante, nämlich die, die er bei der Formulierung der Schleife im Sinn hatte. Mit einem interaktiven System kann diese Zusicherung eingegeben werden und das System kann versuchen zu beweisen, dass sie tatsächlich eine Schleifeninvariante ist.

Hinweis: Statt der schwächsten Vorbedingung kann man auch die "stärkste Nachbedingung" einführen und ermitteln.

Hier brechen wir die Erläuterung der Korrektheit mit Hilfe der axiomatischen Semantik ab. Es wurden die Idee vorgestellt und ein mögliches Unterstützungssystem angedeutet. (Vertiefungen hierzu siehe Vorlesungen über "Semantik" und "sichere Systeme" und z.T. in Vorlesungen über Interaktive Systeme, Wissensverarbeitung, Programmiersprachen/Übersetzer.)

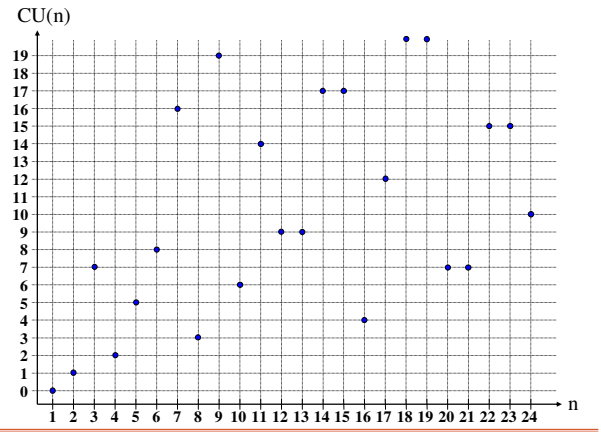
1.7.5 Terminierung

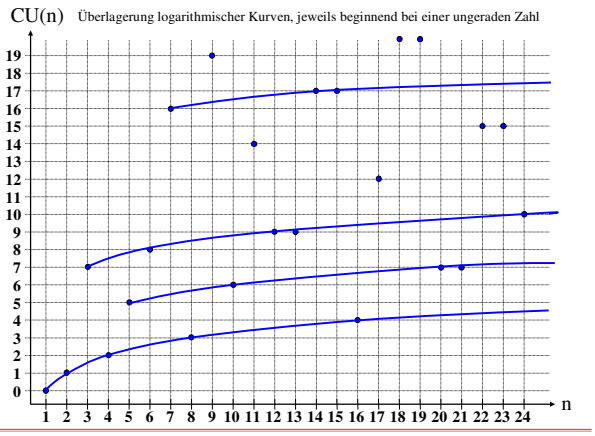
Die Terminierung und die Korrektheit eines Programms werden getrennt nachgewiesen.

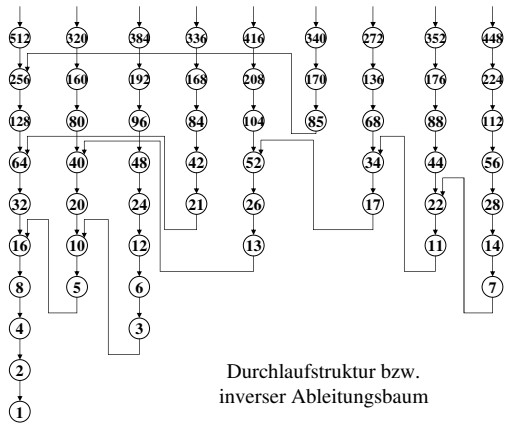
1.7.5.1 Standardbeispiel: Collatz- oder Ulam-Funktion (vgl. 1.4.2.7):

```
function CU (x: Positive) return Natural is
begin
    if x=1 then return 0;
    elsif x mod 2 = 0 then return CU(x/2) + 1;
    else return CU(3*x+1) + 1;
    end if;
end CU;
```

Veranschaulichung auf den nächsten Folien. Auffällig sind die häufigen Pärchen im Funktionsgraphen.







Durchlaufstruktur bzw.
inverser Ableitungsbaum

Es ist bis heute unbekannt, ob diese Funktion für alle Eingaben terminiert. (Man weiß dies aber für unendlich viele Werte, nur "einige Zahlen" entziehen sich bisher hartnäckig einem Beweis.)
Für alle Zahlen, für die man mit heutigen Rechnern diese Funktion testen kann (insbesondere für alle Zahlen bis 10^{16}), liefert die Funktion CU ein Ergebnis, sodass man vermutet, die von CU berechnete Funktion sei total.

Diese Funktion wird meist mit Stanislaw Ulam, 1909-1984, Prof. an der Univ. Florida, in Verbindung gebracht, der sich erst in den 1950er Jahren hiermit beschäftigte. Älter sidn Untersuchungen des Hamburger Mathematikprofessors Lothar Collatz, weshalb sie oft als Collatz-Funktion in der Literatur erscheint; manchmal auch als Hasse's Problem oder Thwaite's Problem oder "the syracuse algorithm" (nach der Syracuse University, wo sie genauer untersucht wurde).

1.7.5.2: Die Terminierung ist bei rekursiven Definitionen nicht leicht nachzuweisen. Relativ einfach geht dies noch bei folgender doppelt-rekursiv dargestellten, genannt McCarthys 91-Function:

```
function MC91 (x: Natural) return Natural is
begin if x > 100 then return x-10
      else return MC91(MC91(x+11)); end if;
end;
```

Übung: Beweisen Sie, dass die hiervon realisierte Funktion lautet:
f(x) = if x > 100 then x-10 else 91 fi.

1.7.5.3: Terminierung und Unverständnis.

Wer eine Programmeinheit plant, hat in der Regel eine klare Vorstellung von dessen Invarianten und den Veränderungen der Variablen. Zugleich sorgt man dafür, dass keine unendlichen Schleifen oder Verklemmungen auftreten. Die Terminierung ist daher vor allem dann ein Problem, wenn kein Verständnis über die zu lösende Aufgabe oder über die Struktur des Lösungsalgorithmus vorhanden ist.

Als Beispiel geben wir auf der nächsten Folie ein undurchschaubares Verfahren an. Hier werden irgendwelche Veränderungen auf den Elementen eines arrays vorgenommen, die vielleicht zu einem Abbruch führen können, vielleicht aber auch eine unendliche Schleife entstehen lassen. Vollziehen Sie das Verfahren für verschiedene Konstanten N nach und machen Sie sich eine Vorstellung von der Schwierigkeit, für ein Programm, dessen Ablaufidee unbekannt ist, die Terminierung zu beweisen.

```
procedure TerminierungsTest is
N: constant integer := 10;
type TupelNat is array (1..N) of Natural;
A: TupelNat := (others => 1);
procedure unklar (A: in out TupelNat) is
Max: Natural; Pos: 1..N; weiter: Boolean:=true;
begin Pos := 1; Max := A(Pos);
      while weiter loop
            weiter := false;
            for i in 1..N loop
                  if Max < A(i) then weiter := true;
                        Max := A(i); A(i) := A(Pos); A(Pos) := Max;
                        if Pos = N then Pos:=1; else Pos := Pos +1; end if;  end if;
                  if (i+Pos) mod 2 = 0  then A(i) := A(i) + A(Pos);
                                        else A(i) := A(i) - A(pos);  end if;
            end loop;
      end loop;
end unklar;

begin unklar (A);
      for i in 1..N loop put(A(i)); end loop;
end;
```


1.7.6 Historische Anmerkungen

Das Problem, die Richtigkeit von Programmen nachzuweisen, besteht seit dem Beginn der praktisch verwendbaren Programmierung, also seit den 1950er Jahren. Dies führte rasch auf die Frage nach der Bedeutung ("Semantik") eines Programms und dessen präziser Formulierung in Kalkülen.
Die ersten Arbeiten hierzu stammen von R. Floyd (Einführung von festen Stellen im Programm, an denen die Bedeutung ermittelt wird), C.A.R. Hoare (Aufstellung eines Regelsystems) und W. Dijkstra (Einführung der weakest precondition) in den 1960er Jahren. Ab 1970 entwickelt sich eine Fülle von Arbeiten zu diesem Gebiet (denotationale Semantik, Semantik nebenläufiger Systeme, Entwicklung von Kalkülen und Beweissystemen, konkrete Methoden wie model checking usw.).

Die Terminierung ist ein eher mathematisches Problem. Seine Unentscheidbarkeit lässt sich zum einen aus dem Gödelschen Unvollständigkeitssatz (1931) ableiten, zum anderen war sie Gegenstand der ersten Arbeiten über berechenbare Funktionen (Church, Kleene, Turing, 1936; später: Post, Markov).

Die terminierenden Eingaben eines Programms sind aufzählbar, im Allgemeinen aber nicht die Menge der Eingaben, für die ein Programm divergiert. Zur Terminierung gibt es viele äquivalente Probleme: Busy Beaver, Postsches Korrespondenzproblem, Maximalität kontextfreier Grammatiken, Wortproblem für Halbgruppen, Game of Life (Conway) usw. Man spricht auch vom "Arbeitsplatzerhaltungstheorem" für Informatiker(innen): Im Prinzip muss man für jedes Programm und jede Eingabe einen eigenen Beweis führen, ob es mit dieser Eingabe anhält oder nicht - und hierfür braucht man eine Informatikausbildung.