Einführung in die Informatik II

Sommersemester 2002, Volker Claus

0. Vorbemerkungen

0.1. Planung im Studium

Es gibt einen Plan der Vorlesung (siehe verteilte Unterlage).

Die Übungen sind durchorganisiert (ähnlich zum WS 01/02).

Die Prüfungen richten sich nach den jeweiligen Prüfungsordnungen. Leider darf ich Ihnen daher Leistungen aus den Übungsgruppen nicht anrechnen.

Wir werden zwei Tests (auch zur Selbstkontrolle) anbieten, die auf die Übungen angerechnet werden.

Planen Sie nun genau, wie Sie an der Vorlesung und an den Übungen teilnehmen wollen/können. Und planen Sie genau Ihre Arbeitszeit!

0.2. Ihre Arbeitsleitung

Wenn Sie die Veranstaltung erfolgreich bestehen wollen, dann müssen Sie mitarbeiten sowohl in der Vorlesung als auch in den Übungen. Zur Unterstützung bieten wir Ihnen zusätzliche Vortragsübungen an.

Wir bewegen uns in eine Zeit der Evaluationen: Alles und Jedes wird analysiert und bewertet. Auch Sie. Für die, die kontinuierlich mitmachen, wird es aber kaum Probleme geben.

Ihr Aufwand für die "Einführung in die Informatik II" alles inklusive: Während der Vorlesungszeit (also vom 15.4. bis zum 20.7.02) sind pro Woche 12 bis 14 Zeitstunden aufzuwenden. Wichtig ist: Halten Sie durch!

0.3. Ziele des Studiums, Ziele der Vorlesung

Details siehe verteilte Unterlagen.

Allgemeine Hinweise für diese spezielle Veranstaltung:

- Wissen und Fähigkeiten mit Theorieunterbau (ca. 65%)
- Fertigkeiten, Handwerk, Routinewissen (ca. 30%)
- Fachübergreifendes, Persönlichkeitsentwicklung (ca. 3%)
- Entwicklung eines Beurteilungsvermögens (ca. 2%)

0.4. Voraussetzung für diese Vorlesung

Einführung in die Informatik I

Programmierkenntnisse (in Ada 95)

Mathematikkenntnisse aus dem WS 01/02

Konkretisierung von Ideen/Formalismen an Beispielen

Siehe auch: verteilte Unterlagen.

0.5. Ablauf und Ihre Mitwirkung

Vorlesung, Übungen, Vortragsübungen: siehe Unterlagen.

Wie können Sie sich stärker äußern?

- In den Übungen (leider *nicht* in der Vorlesung)
- Kontakte zu Tutoren und Mitarbeitern
- Elektronische Medien (Email, Forum)
- Wahl von Vorlesungssprecher(inne)n und Rückmeldungen über diesen Personenkreis

0.6. Zusammensetzung der Vorlesung

Fragebogen

0.7. Bücher, Skript, Folienkopien, Mitschrift

Jede(r) kaufe sich mindestens ein Buch! (Bücher sind ausgereift, enthalten kaum Fehler, geben Zusatzhinweise in die Geschichte, die Literatur, weitere Übungsaufgaben usw.)

Das Skript von Prof. Plödereder (SS 01) ist prinzipiell für den größten Teil des Stoffes hinreichend. Die Vorlesung greift auf diese Folien auch teilweise direkt zurück.

Zusätzlich hierzu können die Beamerpräsentationen und Folien elektronisch abgerufen oder bei der Fachschaft kopiert werden.

Schreiben Sie dennoch Manches mit, vor allem was an die Tafel geschrieben wird. (Ideen kann man schriftlich kaum darstellen; aber Sie erinnern sich über Ihre Notizen daran.) Bereiten Sie Ihre Notizen umgehend auf.

0.8. Unerwünschtes

An einer vergleichbaren Universität in den USA müssten Sie rund 2000 Euro allein für diese 6-SWS-Veranstaltung bezahlen und dieser Gegenwert soll allen Interessierten zugute kommen. Das bedeutet vor allem:

Keine Störungen während der Veranstaltung.

Speziell: Kein Lärm, keine Unterhaltungen, kommen Sie pünktlich und gehen Sie erst am Ende der Veranstaltungen.

Wir wollen, dass Sie etwas lernen. Das erreicht man nicht durch Abschreiben. Wir wünschen daher **keine identischen oder sehr ähnlichen Abgaben** in den Übungen (hier: außer im Rahmen kleiner vorab festgelegter Gruppen), bei den Tests und schon gar nicht bei den Klausuren.

0.9. Fragen?

1. Algorithmen

Gliederung:

1.1. Charakteristika von Algorithmen

1.2. Grenzen der Algorithmen

1.3. Korrektheit von Algorithmen

1.4. Komplexität von Algorithmen

1.5. Gegenläufigkeiten

1.1. Charakteristika von Algorithmen

Vorbemerkung: Begriffe für Mengen M
"endlich", "unendlich", "abzählbar", "überabzählbar", "aufzählbar", "beschränkt".

M ist endlich $\Leftrightarrow$ M ist leer oder es gibt eine natürliche Zahl n und eine Bijektion f: M $\rightarrow$ {1, ..., n}.

M ist unendlich $\Leftrightarrow$ M ist nicht endlich.

M abzählbar (unendlich) $\Leftrightarrow$ Es gibt eine Bijektion zu den natürlichen Zahlen.

M ist höchstens abzählbar $\Leftrightarrow$
M ist endlich oder M ist abzählbar unendlich.

M ist überabzählbar ⇔ M ist nicht "höchstens abzählbar".

M ist aufzählbar ⇔ M ist leer oder
es gibt einen Algorithmus, der jeder natürlichen Zahl n ein Element der Menge zuordnet, und zu jedem Element von M gibt es mindestens eine hierdurch zugeordnete natürliche Zahl.

Der Begriff "beschränkt" ist ein relativer Begriff; er bezieht sich auf die jeweils betrachtete Umgebung. Hierzu muss man *vorher* eine Schranke k festlegen. Bei Zahlenmengen lautet eine solche Definition dann beispielsweise:
M ist beschränkt ⇔

Es gibt eine vorab festgelegte natürliche Zahl k, so dass M nicht mehr als k Elemente besitzt.

Beispiel 1.1:
Die leere Menge ∅ ist endlich. Sie besitzt 0 Elemente.

Für jede natürliche Zahl n ist die Menge {1, 2, ..., n} endlich.

Die Menge der natürlichen Zahlen ∠ = {1, 2, 3, ...} bzw. $\angle_0$= {0, 1, 2, 3, ...} und die Menge der ganzen Zahlen 9 = { ..., -2, -1, 0, 1, 2, 3, ... } sind abzählbar unendlich und selbstverständlich aufzählbar.

Die Menge der rationalen Zahlen Θ = {n/m | n ganze Zahl, m positive ganze Zahl, n und m teilerfremd} ist abzählbar unendlich und zugleich aufzählbar (wie zeigt man dies?).

Die Menge der reellen Zahlen 3 und die Menge der komplexen Zahlen ∀ sind überabzählbar (Beweis über das Cantorsche Diagonalverfahren, vgl. Mathematikvorlesungen).

Ein Algorithmus ist eine Vorschrift, die die Reihenfolge von durchzuführenden Handlungen (Operationen) auf Daten (Operanden) genau beschreibt. Hierbei muss gelten:

a) Die Daten sind "diskret" aufgebaut (z.B. nur aus Binärziffern). Genauer: Es gibt beschränkt viele Zeichen $\{a_1, ...,a_k\}$, so dass jedes Datum eine Folge dieser Zeichen ist.

b) Die Operationen sind "diskret" aufgebaut. Genauer: Es gibt beschränkt viele Zeichen $\{b_1, ...,b_m\}$, so dass jede Operation einschließlich ihrer Operanden hiermit beschrieben werden kann.

c) Die Vorschrift ist eine endliche Folge von Operationen. Die Vorschrift wird schrittweise abgearbeitet (diskrete Zeitskala).

d) Eine der Operationen ist als Startoperation ausgezeichnet.

e) Für jede Operation ist unmittelbar nach ihrer Ausführung bekannt, welches die Folgeoperation ist oder ob der Algorithmus abbricht (terminiert).

f) Die Eingabe für die Vorschrift ist eine (eventuell unendliche oder auch leere) Folge von Daten.

g) In jedem Schritt (d.h. zu jedem Zeitpunkt) gilt: Die bis dahin bearbeitete oder betrachtete Menge an Daten und durchgeführten Operationen ist endlich.

Hinweis: Der Algorithmus verfügt über eigene Speicherbereiche für die Daten und für die Vorschrift. Beide Bereiche kann der Algorithmus während seiner Abarbeitung verändern, aber in jedem Schritt nur einen endlichen Bereich. Beide Bereiche sind prinzipiell unendlich groß.

Hinweis: Obige Ausführungen sind keine richtige Definition, sondern nur eine Liste von umgangssprachlich formulierten Anforderungen.

Turingmaschinen erfüllen diese Anforderungen,

aber auch andere "Rechenmaschinen" oder Grammatiken.

Auch Programme beschreiben Algorithmen. Prüfen Sie die Anforderungen an der Ihnen geläufigen Programmiersprache und ihren Programmen nach.

Man beachte, dass Algorithmen beliebige Algorithmen als Eingabe besitzen können.

Zusatz: Wünschenswerte Anforderungen an Algorithmen:

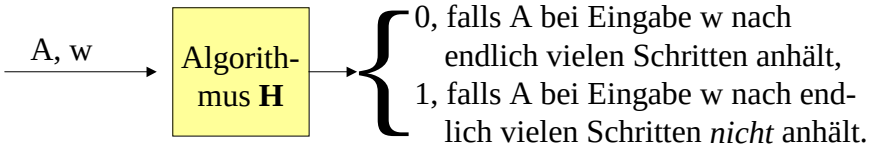
- Nichtdeterminismus: Oft weiß man nicht, welches die nachfolgende Operation sein soll, und möchte eine Menge möglicher Folgeoperationen angeben. Solange diese Menge bei jeder Operation endlich ist, lässt sich das Problem durch einen normalen Algorithmus simulieren (siehe später: BFS-Verfahren mit BFS = Breadth-First-Search). Im Falle unendlicher Mengen liegt jedoch kein Algorithmus mehr vor.
- Terminierung = ein Algorithmus muss für alle Eingaben nach endlich vielen Schritten anhalten.
  Diese Bedingung ist für Algorithmen *nicht notwendig* und wird auch nicht für alle Algorithmen gewünscht. Z.B. sollte ein Betriebssystem möglichst unendlich lange arbeiten ...

1.2. Grenzen der Algorithmen

Was kann man mit Algorithmen ***nicht*** beschreiben?

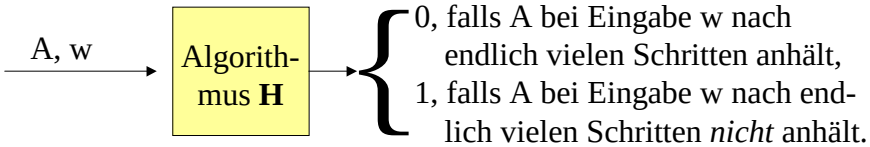
Betrachte folgende Aufgabe:
Konstruiere einen Algorithmus, der beliebige Algorithmen und Daten einlesen kann und der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A mit den Eingabedaten w nach endlich vielen Schritten anhält.

Gesucht wird also ein Algorithmus H

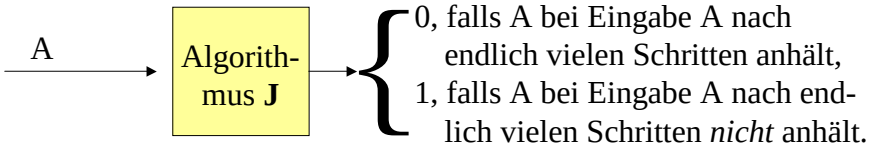


für alle Algorithmen A und für alle Eingabefolgen w.

Angenommen, es gibt solch einen Algorithmus H mit ∀A ∀w

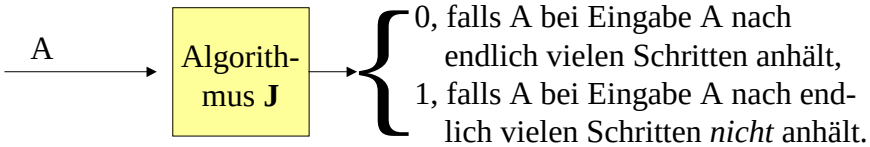


Speziell gibt es nun auch einen Algorithmus J mit ∀A

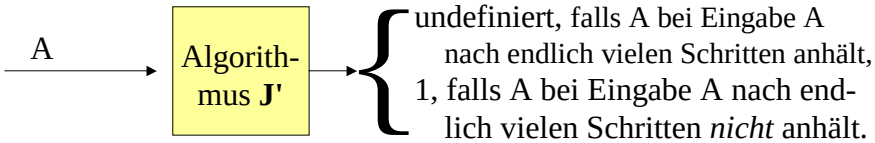


für alle Algorithmen A, indem man in H nur den Algorithmus A eingibt und dann H mit der Eingabe A und A (=w) laufen lässt.

Zu diesem Algorithmus J mit ∀A

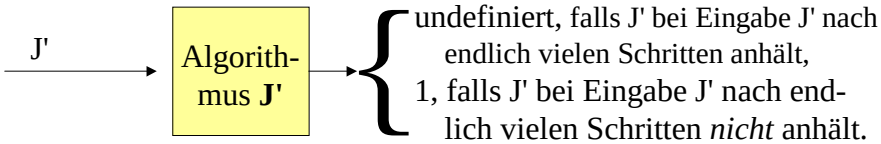


gibt es dann auch einen Algorithmus J' mit ∀A



Klar wegen: Modifiziere J so, dass der Algorithmus J anstelle der Ausgabe 0 in eine unendliche Schleife geht; so erhält man J'.

Was macht J' bei Eingabe von J'?



Fall 1: Bei Eingabe von J' hält J' an.
Dann liefert J' bei Eingabe von J' den Wert 1.
Nach Definition von J' hält dann J' bei Eingabe J' *nicht* an.
Widerspruch!

Fall 2: Bei Eingabe von J' hält J' *nicht* an.
Dann läuft J' bei Eingabe von J' in eine unendliche Schleife.
Nach Definition von J' hält dann J' bei Eingabe J' an.
Widerspruch!

Beide möglichen Fälle führen also auf einen Widerspruch.
Folglich muss die Annahme, dass es den Algorithmus H gibt, falsch gewesen sein. Es gilt daher der

Satz 1.1:

Es gibt keinen Algorithmus H, der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A mit den Eingabedaten w nach endlich vielen Schritten anhält.

Man nennt die Suche nach einem solchen Algorithmus H auch das Halteproblem und sagt kurz:

Das Halteproblem ist algorithmisch nicht lösbar.

Historischer Hinweis:
Dieses Resultat kennt man seit 1931. In der hier vorgestellten Form wurde es 1936 unabhängig voneinander von Church, Kleene und Turing bewiesen.

Aus diesem Resultat lässt sich eine Fülle von weiteren Problemen ableiten, die alle algorithmisch nicht lösbar sind, z.B.
- entscheide, ob Programme für *alle* Eingaben nicht anhalten,
- entscheide, ob zwei beliebige Programme dasselbe berechnen,
- entscheide, ob eine kontextfreie Grammatik alle Wörter erzeugt

usw.
Weitere Aussagen hierzu in der Vorlesung "Theorie I" (Satz von Rice) und in Büchern.

1.3. Korrektheit von Algorithmen

Wie lässt sich beweisen, dass ein Algorithmus (= ein Programm) genau das realisiert, was es realisieren soll?

Dazu muss man zweierlei kennen:
- die Spezifikation, was zu realisieren ist,
- die Ermittlung dessen, was ein Programm realisiert.

Zu realisieren ist eine Zuordnung von Eingabedaten zu Ausgabedaten oder die Erfüllung von Anforderungen.

Ein Programm realisiert eine Abbildung von der Menge der Eingabedaten in die Menge der Ausgabedaten. (Dies kann implementierungsabhängig sein!)

Spezifikation:
f: E $\rightarrow$ A

oder

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

```
x, z: Integer;
   Get(x);
  z:= x + x;
   Put(z);
```

Realisierung
f": 9 $\rightarrow$ 9 mit
f"(n) = 2n für alle n$\in$ 9

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

Beispiel 1.2:
Spezifikation: Die Funktion f: 9 $\rightarrow$ 9 mit f(n) = 2n für alle n$\in$ 9 soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;
Get(x);
for i in 1..x loop x := x+1; end loop;
Put(x);
```

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 vom Programm realisiert.

Beispiel 1.3:
Spezifikation: Die Funktion $f: \mathbb{9} \rightarrow \mathbb{9}$ mit $f(n) = 2n$ für alle $n \in \mathbb{9}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;
Get(x);
Put(x, Base=>2);
Put("0");
```

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 1.4:
Spezifikation: Die Funktion $f: \mathbb{9} \rightarrow \mathbb{9}$ mit $f(n) = 2n$ für alle $n \in \mathbb{9}$ soll realisiert werden.

Folgendes Programmstück wird vorgeschlagen:

```
x: Integer;  Negativ: Boolean;
Get(x);
Negativ := x<0;
if Negativ then x:=-x; end if;
for i in 1..x loop x:=x+1; endloop;
if Negativ then x:=-x; end if;
Put(x);
```

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?

Es gibt mehrere Methoden, die Bedeutung eines Programms (also die realisierte Abbildung) zu ermitteln. Wir wollen hier nur eine betrachten, nämlich die prädikatenlogische Methode; sie wird in der Literatur *"axiomatische Semantik"* genannt.

Man geht davon aus, dass zu jedem Zeitpunkt des Programmablaufs gewisse Bedingungen erfüllt sind; diese beschreibt man als prädikatenlogische Formeln; man nennt sie ***Zusicherungen*** oder ***assertions*** und schreibt sie in geschweifte Klammern.

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 Modell für die Interpretation prädikatenlogischer Formeln müssen wir hier die ganzen Zahlen betrachten. Hinzu kommt der Fall "undefiniert", den wir durch das Zeichen $\perp$ 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

$$a \in 9 \wedge x = a$$

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

$$x = a + 2$$

Damit erhalten wir folgendes Programm mit zusätzlichen Zusicherungen zwischen je zwei aufeinander folgenden Anweisungen:


```
x: Integer;
{x = ⊥}
Get(x);
{a ∈ 9 ∧ x = a}
x := x+2;
{x = a + 2}
Put(x);
```

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

Die realisierte Abbildung dieses Programms lässt sich nun schrittweise an jeder Anweisung beweisen (bitte selbst durchführen).

Dies sieht so aus, als ob der Mensch als Beweiser benötigt wird. Dies ist aber nur teilweise richtig.

Denn bis auf die while-Schleife (und die Rekursion) kann man alle anderen Anweisungen weitgehend automatisch nachprüfen lassen.

Es gelten nämlich folgende sechs Regeln. 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.

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 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.

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

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

5

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

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

Beispiel 1.5: Was macht folgendes Programmstück?

```
x, y, z: Natural;
Get (x, 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);
```

Realisiert dieses Programmstück die Multiplikation?
Füge Zusicherungen ein und führe einen Beweis.

```
x, y, z: Natural;
{x=⊥ ∧ y=⊥ ∧ z=⊥}
Get (x, y);
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ b≥0 ∧ y=b ∧ z=⊥}
z:=0;
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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)
```

Wir betrachten nun nur die while-Schleife.


```
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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;
Put(z)
```

then-Zweig betrachten

```
{a∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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;
Put(z)
```

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∈ 9 ∧ a≥0 ∧ x=a ∧ b∈ 9 ∧ 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;
Get (x, 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);
```

Also wird das Produkt zweier nicht-negativer Zahlen berechnet.

Beispiel 1.6: *Plateau-Problem*

Gegeben: $n \geq 1$ und ein sortiertes Feld natürlicher Zahlen

a: array (1..n) of Natural;

Gesucht ist die maximale Anzahl gleicher Zahlen, also

Max $\{d \geq 1 \mid \exists\, i$ mit $a(i) = a(i+1) = ... = a(i+d-1)\}$.

Lösungsvorschlag:

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

```
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;
```

" Ergebnis ist p. "

Der Beweis erfolgt in der Vorlesung an der Tafel. Hierzu definiere man das Prädikat m(j,p) als "p ist die maximale Anzahl gleicher Zahlen im Teil-array a(1..j)". Dann zeige man, dass m(j,p) eine Schleifeninvariante der while-Schleife ist. Der Rest ist einfach.