Grundvorlesung Informatik

Universität Stuttgart, Studienjahr 2002/03

~~0. Vorbemerkungen (14.10.02)~~

1. Grundlagen der Programmierung (17.10.-19.12.02)

2. Interaktionen (9.1. - 13.2.03)

3. Grundlegende Verfahren (28.4. - 25.7.03)

Klausur (Orientierungsprüfung): ca. 4.-7. August 2003

Hochschullehrer: Volker Claus, Fakultät 5 "I, E und I"
Institut für Formale Methoden der Informatik (FMI)

Üblicherweise motiviert man zu Beginn den kommenden Stoff, das Informatikstudium, die beruflichen Tätigkeiten usw.

Statt dessen beginnen wir sofort mit den Inhalten, wobei nur die Voraussetzung wiederholt sei, *dass Informatik keine Geheimwissenschaft schwer durchschaubarer Computer und ihrer konkreten Softwaresysteme ist.* Es geht um das grundsätzliche Verständnis des Rohstoffs "Information" und um Konzepte, nach denen dieser dargestellt, aufbereitet, abgelegt, verarbeitet, implementiert, wiederverwendet, eingesetzt, zugeschnitten oder anderweitig manipuliert werden kann. Dies erfordert eine theoretische Durchdringung (in der Vorlesung und in den Übungen) und zugleich den praktischen Umgang (Programmierkurs und -übungen sowie z.T. in den Übungen).

Teil 1 der Grundvorlesung

1. Grundlagen der Programmierung

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

Gliederung des Kapitels 1.1

1.1 Algorithmen und Sprachen

1.1.1 Darstellung von Algorithmen
1.1.2 Grundlegende Datenbereiche
1.1.3 Realisierte Abbildung
1.1.4 (Künstliche) Sprachen
1.1.5 Grammatiken
1.1.6 BNF, Syntaxdiagramme
1.1.7 Sprachen zur Beschreibung von Sprachen
1.1.8 Übungsaufgaben

1.1.1 Darstellung von Algorithmen

Umgangssprachliche grobe Festlegung 1.1.1.1:
Ein Verfahren, das prinzipiell von einer mechanisch arbeitenden Maschine durchgeführt werden kann, nennen wir einen Algorithmus.

Etwas präzisere Festlegung 1.1.1.2:
Ein Algorithmus ist ein exakt beschriebenes Verfahren einschließlich der genauen Festlegung der Eingabe und der Ausgabe, der Zwischenspeicherung von Daten usw. Das Verfahren muss so genau ausformuliert sein, dass jede(r) den Algorithmus nachvollziehen und einem Computer übertragen kann, ohne Rücksprache mit den Verfassern zu nehmen.

Beispiel 1.1.1.3: Addiere 1 zu einer dezimal dargestellten Zahl.

Umgangssprachliche Formulierung: Eine Zahl sei als eine Folge von Ziffern aus der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} gegeben. Falls die letzte Ziffer nicht die 9 war, so ersetze sie durch die nächste größere Ziffer und beende das Verfahren, anderenfalls ersetze sie durch 0 und wiederhole das Verfahren für die zweitletzte Ziffer usw.

Hier wird erwartet, dass der, der das Verfahren ausführt, ein wenig mitdenkt oder ein Vorwissen besitzt. So muss man wissen, dass die Ziffern in der angegebenen Reihenfolge $0 < 1 < 2 < ... < 8 < 9$ angeordnet sind, man muss wissen, was das "usw." bedeutet, und man muss wissen, was man zu tun hat, wenn es keine zweitletzte Ziffer gibt.

Als Mensch macht man sich einen Algorithmus an konkreten Beispielen klar. So liefert der obige Algorithmus "+ 1":

2 + 1 = 3, 44 + 1 = 45, 103 + 1 = 104,
0 + 1 = 1, 19 + 1 = 20, 199 + 1 = 200.

Nicht klar ausformuliert wurden die Fälle, in denen die Folge nur aus Neunen besteht, wie 9 + 1 = 10, 99 + 1 = 100.

Weiterhin entspricht die Aussage, dass eine Zahl eine Folge von Ziffern sei, nicht der üblichen Anschauung, da man i.A. (außer im Falle der Null selbst) keine führenden Nullen zulässt. Das Verfahren liefert zwar 0068 + 1 = 0069, aber in der Regel schreibt man nicht 0068, sondern 68, da man nur eindeutige Darstellungen für Zahlen verwenden möchte.

Das Verfahren beschränkt die Addition der 1 auf natürliche Zahlen (einschl. der Null). Man kann 1 aber auch zu einer ganzen, einer rationalen, einer reellen oder einer komplexen Zahl addieren. Diese Erweiterung wird durch das angegebene Verfahren nicht erfasst, sondern muss durch einen neuen Algorithmus beschrieben werden.

Wie und wo man die Zahlen aufschreibt, bleibt offen. Wir denken sicher sogleich an Papier und Bleistift, doch kämen andere Völker möglicherweise nicht auf diese Realisierung. (Beachten Sie, dass die Dezimaldarstellung bzw. allgemein die Darstellung in einem Stellenwertsystem zu einer Basis keineswegs naheliegend ist. Die Römer haben beispielsweise mit einem völlig anderen System gearbeitet.)

An diesem Beispiel erkennt man einige zentrale Sprachelemente, um Algorithmen zu beschreiben.

- Algorithmen bestehen aus einfachen Handlungen, sog. "elementaren Anweisungen". Im Beispiel sind dies: "ersetze Ziffer durch nächst größere Ziffer" oder "beende das Verfahren".
- Einzelne Handlungen können nacheinander ausgeführt werden. Im Beispiel: "ersetze Ziffer durch nächst größere Ziffer" und danach "beende das Verfahren".
- Die nächste Handlung kann von einer aktuellen Bedingung (Alternative oder Fallunterscheidung) abhängen. Im Beispiel: "Falls die Ziffer nicht die 9 war, dann ..." .

- Handlungen können wiederholt werden. Im Beispiel wird dies durch das "usw." beschrieben, welches besagt, man solle die Ersetzungen solange vornehmen, bis eine von 9 verschiedene Ziffer erreicht wird.
- Durch den Algorithmus werden irgendwelche Gebilde manipuliert. Deren anfängliche Darstellung ist anzugeben. In unserem Beispiel sind dies natürliche Zahlen und deren Darstellung als Ziffernfolgen.
- Die Gebilde, die das Ergebnis des Algorithmus sind, ergeben sich durch den Algorithmus selbst. Man sollte sie aber möglichst zuvor beschreiben können. In unserem Beispiel sind dies ebenfalls Dezimaldarstellungen.

Beispiel 1.1.1.4:
Addieren zweier dezimal dargestellter Zahlen.

Für diese Aufgabe gibt es eine einfache Formulierung eines Algorithmus: Wenn a und b zwei dezimal dargestellte Zahlen (also dargestellt als Ziffernfolgen) sind, so addiere b-mal 1 zu a.

Wie man 1 zu einer Zahl addiert, wissen wir ja bereits.

Hinweis: "addiere b-mal 1 zu a" kann missverstanden werden. Gemeint ist, dass man 1 zu a addiert, dann zum Ergebnis 1 addiert, danach zu dem neuen Ergebnis 1 addiert usw.

An diesem Beispiel erkennt man zwei weitere zentrale Sprachelemente zur Beschreibung von Algorithmen.

- Wiederhole eine Handlung b-mal. Die Anzahl der Wiederholungen ist hier also vor der ersten Ausführung bekannt und hängt nicht von einer aktuellen Bedingung ab.
- Man darf Algorithmen, die bereits anderweitig beschrieben wurden, in anderen Algorithmen verwenden. In unserem Beispiel darf man den "Algorithmus Addiere 1" für die Addition zweier Zahlen benutzen.

Weiterhin muss man sich Gedanken darüber machen, wo Zwischenergebnisse abgelegt und wie sie weiter verwendet werden.

Ein anderes, effizienteres Additionsverfahren lernten wir alle in der Grundschule.

Aufgabe:
Beschreiben Sie diesen Additionsalgorithmus möglichst präzise.

Einige Beispiele aus dem Alltag:

- Kochrezepte.
- Bastelanleitungen.
- Ermittlung der Abiturnote aus den Leistungen der Oberstufe.
- Benutzungsalgorithmus des Übungssystems.
- Ablauf der Gesetzgebungsverfahren.
- Ermittlung kürzester Verbindungen zwischen zwei Orten.
- Feststellen von Rechtschreibfehlern in einem Text.
- In der Industrie eingesetzte Produktionsvorgänge.
- Abläufe in Verwaltungen.
- Berechnung der Reisekostenerstattung durch eine Verwaltung.
- Erstellung einer Häufigkeitsstatistik aller Wörter, die Goethe in seinem Gesamtwerk verwendet hat (das Gleiche für Noten und Musiker, Farben und Maler, ...)

Festlegungen 1.1.1.5: (A1) bis (A9)
Diese und weitere Untersuchungen führten zu folgenden Vorgaben und Sprachelementen für die Beschreibung von Algorithmen und den von ihnen manipulierten Daten:

(A1) Ein Algorithmus ist eine Folge von Anweisungen. Eine Anweisung besteht aus elementaren Anweisungen, die nach den folgenden Regeln zu Anweisungen zusammengefügt werden können. Der Algorithmus erhält einen Bezeichner (einen Namen, vgl. auch (A9)).

(A2) Ein Algorithmus arbeitet auf Daten. Daten werden in "Behältern", genannt Variablen, abgelegt. Variablen sind zu Beginn des Algorithmus aufzulisten einschließlich der Angabe, welche Daten in die Variable gelegt werden dürfen und welche nicht (dies nennt man "Deklaration" oder "Vereinbarung" der Variablen). Diese durch "," oder ";" getrennte Auflistung beginnt mit dem Wort var.

(A3) Elementare Anweisungen sind von der Form:

skip	*Bedeutung:* Tue nichts.
X := α	"Wertzuweisung". α ist ein Ausdruck. *Bedeutung:* Rechne den Ausdruck α aus und lege den erhaltenen Wert in der Variablen X ab.
read (X)	Leseanweisung. *Bedeutung:* Lies den nächsten Wert ein und lege ihn in der Variablen X ab.
write (α)	Schreibanweisung. *Bedeutung:* Drucke den Wert, den der Ausdruck α besitzt, aus.
halt	*Bedeutung:* Beende den Algorithmus.
$F(X_1, ..., X_n)$	*Bedeutung:* Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.

(A4) Ausdrücke sind entweder übliche arithmetische Ausdrücke (aufgebaut aus Zahlen, Variablen, Klammern und Operatoren wie +, -, *, /, div, mod) oder logische Ausdrücke (aufgebaut aus den Wahrheitswerten true und false, Variablen, Klammern, Vergleichen und Operatoren wie and, or, not usw.) oder Zeichenausdrücke (aufgebaut aus den Zeichen eines Alphabets, Variablen und Operatoren wie append, empty, remove usw.). Logische Ausdrücke nennt man auch Boolesche Ausdrücke. Wir setzen voraus, dass jede(r) weiß, wie man Ausdrücke auswertet.

(A5) Jede elementare Anweisung ist auch eine Anweisung.

(A6) Hintereinanderausführung oder Sequenz: Wenn D und E Anweisungen sind, dann ist auch D;E eine Anweisung.
Bedeutung: Führe erst D und danach E aus.

(A7) Alternative oder Fallunterscheidung:

Wenn C und D Anweisungen und β ein Boolescher Ausdruck sind, dann ist auch

if β then C else D fi

eine Anweisung.

Bedeutung: Wenn zu dem Zeitpunkt, zu dem man auf diese Anweisung stößt, der Boolesche Ausdruck β den Wert true besitzt, so führe die Anweisung C aus, anderenfalls die Anweisung D.

Spezialfall: Falls D die Anweisung skip ist, so schreibt man kurz if β then C fi (einseitige Alternative).

Hinweis: fi ist wie die "Klammer Zu" zum Symbol if. Die Klammerpaare "(" und ")" oder "[" und "]" entstehen auseinander ebenfalls durch Spiegelung. Auch das in (A8) auftretende "od" bildet mit "do" eine solche Klammerung.

(A8a) while-Schleife:

Wenn C eine Anweisung und β ein Boolescher Ausdruck sind, dann ist while β do C od eine Anweisung.
Bedeutung: Solange der Boolesche Ausdruck β den Wert true ergibt, wiederhole die Anweisung C. Hierbei wird der Ausdruck β stets *vor* der Ausführung von C ausgewertet. Wenn also β zu dem Zeitpunkt, zu dem man auf die while-Schleife stößt, false ist, wird C überhaupt nicht ausgeführt.

(A8b) repeat-Schleife:

Wenn C eine Anweisung und β ein Boolescher Ausdruck sind, dann ist repeat C until β eine Anweisung.
Bedeutung: Solange der Boolesche Ausdruck β den Wert false ergibt, wiederhole die Anweisung C. Hierbei wird der Ausdruck β erst *nach* der Ausführung von C ausgewertet. C wird also stets mindestens einmal ausgeführt.

(A8c) for-Schleife oder Zählschleife:

Wenn C eine Anweisung, i eine Variable, in die ganze Zahlen gelegt werden dürfen, und a und e zwei ganze Zahlen sind, dann ist auch

for i := a to e do C od

eine Anweisung.

Bedeutung: Setze i auf den Wert a. Falls i nicht größer als e ist, führe C aus. Erhöhe nun i um 1 und wiederhole diesen Vorgang, bis i größer als e ist; anschließend führe die Anweisung, die auf die for-Schleife folgt, aus.

Präzisere Festlegung: Die for-Schleife besitzt genau die gleiche Bedeutung wie folgende Anweisung

i := a; while i ≤ e do C; i := i+1 od

Man verbietet, dass die Variable i durch C verändert werden darf. (Insbesondere gibt es in C keine Wertzuweisung der Form i := ... .)

(A8d) Allgemeine for-Schleife:

Wenn C eine Anweisung, i eine Variable, in die ganze Zahlen gelegt werden dürfen, und α, δ und ω arithmetische Ausdrücke, die eine ganze Zahl als Ergebnis liefern, sind, dann ist auch

for i := α by δ to ω do C od

eine Anweisung.

Bedeutung: Werte den Ausdruck α aus. Setze i auf diesen Wert. Nun wiederhole folgendes bis zum Abbruch: [Werte den Ausdruck ω aus. Falls i größer als dieser Wert ist, brich die for-schleife ab. Anderenfalls führe C aus. Werte danach den Ausdruck δ aus und addiere diesen Wert zu i hinzu.]

Die for-Schleife besitzt also genau die gleiche Bedeutung wie folgende Anweisung

i := α; while i ≤ ω do C; i := i+δ od

Anmerkungen zu for-Schleifen (A8c) und (A8d):

Anmerkung 1: In der Praxis verwendet man meist nur die Zählschleife (A8c), allerdings ergänzt um das "Herunterzählen":

for i := a downto e do C od

Hierbei wird am Ende der Schleife nicht i um 1 erhöht, sondern um 1 erniedrigt. Die Bedeutung lautet für "downto" also:

i := a; while i ≥ e do C; i := i-1 od

Anmerkung 2: Statt nur ganze Zahlen zuzulassen, erweitert man die Zählschleife (A8c) auf beliebige, total angeordnete Mengen, also zum Beispiel: for i := 'F' to 'K' do C od (bzw. ... downto ...)
Hier wird die Anweisung C für alle Buchstaben F, G, H, I, J und K durchgeführt. i muss in diesem Fall eine Variable sein, die Buchstaben (anstelle von Zahlen) als Werte annehmen kann.

Anmerkung 3: Falls jedoch in der Praxis die allgemeinere Form der for-Schleife zugelassen wird, so verbietet man in der Regel, dass die Variable i sowie die Ausdrücke α, δ und ω in der Anweisung C verändert werden dürfen.

Das heißt, man legt die Bedeutung der allgemeinen for-Schleife dann nicht wie in (A8d), sondern durch folgende Anweisung fest:

i := α; D := δ; E := ω;
while i ≤ E do C; i := i+D od

wobei D und E zwei neue Variable sind, die sonst nirgends im Algorithmus auftauchen.

Machen Sie sich an Beispielen klar, dass dies eine andere Bedeutung als die in (A8d) ist.

Achten Sie in der Praxis also genau auf die festgelegte Bedeutung!

(A9) Ergänzende Vorschriften

Aus Gründen der Übersichtlichkeit und Lesbarkeit macht man meist weitere Vorschriften, z.B.:

Die Deklarationen müssen stets am Anfang des Algorithmus angegeben werden.

Die auf die Deklarationen folgende Anweisung wird in begin ... end eingeklammert. (halt kann dann entfallen.)

Kommentare trennt man bis zum Zeilenende durch das Zeichen % vom Algorithmus ab.

Jeder vorkommende Bezeichner muss vorher im in einer Deklaration vereinbart worden sein.

Der Algorithmus beginnt stets mit dem Wort program, danach folgt der Name des Algorithmus, danach das Wort is, dann die Deklarationen und schließlich die in begin und end eingeschlossene Anweisung.

Einen nach den Vorschriften (A1) bis (A9) aufgeschriebenen Algorithmus bezeichnen wir als **Programm**.

Unsere Programme sind also folgendermaßen aufgebaut:

```
program <Name des Algorithmus> is
var <Deklarationen>;
begin <Anweisung> end
```

Später werden wir dies erweitern. Insbesondere werden wir den Deklarationsteil ausgestalten und wir werden Parameter hinzufügen, um mehr Flexibilität zu erreichen.

Programme in der Praxis sind prinzipiell auch nach diesem Muster aufgebaut.

Hinweise:

Wir werden (A1) bis (A9) später um weitere Regelungen ergänzen müssen. Auf diese Weise erhalten wir dann eine "richtige Programmiersprache".

Zunächst ist die Deklaration von Variablen zu präzisieren, insbesondere wenn die Variablen nicht Zahlen, sondern komplexere Bereiche wie Vektoren, Matrizen oder Graphen als Werte besitzen.

Sodann ist die Wiederverwendung von bereits ausformulierten Programmen festzulegen. Hierfür werden wir Prozeduren, Funktionen, Moduln und Objekte einführen.

Schließlich ist das Zusammenspiel von Programmen zu regeln (Dialoge und andere Interaktionen).

Beispiel 1.1.1.6: Stelle fest, ob eine natürliche Zahl a eine Quadratzahl ist. Falls ja, drucke 1 aus, sonst drucke 0 aus.

Verfahren: Prüfe für alle Zahlen von 0 bis a, ob deren Quadrat gleich a ist. Falls es eine solche Zahl gibt, dann ist eine Quadratzahl, anderenfalls nicht. Übertragen in unsere Sprache:

```
program quadratzahl1 is
var x, i, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                % Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i:=0 to x do         % prüfe für i von 0 bis a, ob i² = a ist
           if i*i = x then ergebnis := 1 fi od;
       write (ergebnis)
end
```


Überprüfe, ob die Regeln (A1) bis (A9) eingehalten wurden:

```
program quadratzahl1 is
var x, i, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                % Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i:=0 to x do         % prüfe für i von 0 bis a, ob i² = a ist
           if i*i = x then ergebnis := 1 fi od;
       write (ergebnis)
end
```

Also: korrekt gebildetes Programm

Fortsetzung Beispiel 1.1.1.6:
Rechnet man das Programm für eine Zahl, z.B. für a = 33 durch, so quadriert man alle Zahlen von 0 bis 33, wobei man jedes Mal feststellt, dass i^2 ungleich 33 ist. Man hätte bereits bei i=6 aufhören können, da ab dann $i^2 > 33$ ist. Dies führt zu folgendem "effizienter" arbeitenden Programm:

```
program quadratzahl2 is
var x, i, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                % Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;  i := 0;
       while i*i ≤ x do         % prüfe nur für i von 0 bis wurzel(a)
           if i*i = x then ergebnis := 1 fi; i := i+1 od;
       write (ergebnis)
end
```


Fortsetzung Beispiel 1.1.1.6:
Das Programm quadratzahl2 führt nicht mehr a, sondern nur noch 2·wurzel(a) Multiplikationen durch. Frage: Kann man die lästigen Multiplikationen sparen?

Ja, das geht. Beachte dass die Differenz zwischen zwei Quadratzahlen immer eine ungerade Zahl ist und dass man die n-te Quadratzahl erhält, indem man die ungeraden Zahlen zwischen 1 und 2n-1 aufsummiert.

1 = 1 4 = 1+3 9 = 1+3+5 16 = 1+3+5+7
25 = 1+3+5+7+9 usw.

Wir notieren daher die nächste ungerade Zahl in der Variablen u (erster Wert ist 1) und das aktuelle Quadrat in der Variablen q (deren erster Wert ist 0). Die nächste Quadratzahl ermitteln wir dann durch die Anweisung q := q+u; u := u+2 .

Fortsetzung Beispiel 1.1.1.6:
Dies führt auf das Programm

```
program quadratzahl3 is
var x, u, q, ergebnis: Variablen für natürliche Zahlen;
begin  read (x);                    % Es wird a eingelesen und in x abgelegt.
       q := 0;  u := 1; ergebnis := 0;
       while q ≤ x do               % prüfe für i von 0 bis a, ob i² = a ist
            if q = x then ergebnis := 1 fi; q := q+u; u := u+2  od;
       write (ergebnis)
end
```

Hier werden keine Multiplikationen mehr benötigt, sondern nur noch 2·wurzel(a) Additionen. Dieses Programm wird daher wesentlich schneller arbeiten als quadratzahl1.

Frage: Wie prüft man nach, was ein Programm macht?

Einfache Antwort: Man vollzieht es schrittweise nach, wobei man die Veränderungen aller Variablen notiert. Ein solches Schema nennt man ein Ablaufprotokoll. Das Schema hierfür lautet (man schreibe die Eingabe und Ausgabe gesondert auf):

Schritt	Aktion	<Var. 1>	<Var. 2>	<Var. 3>	<Var. 4>	···

Definition 1.1.1.7: Es sei ein Programm mit seinen aktuellen Eingabedaten gegeben. Bilde eine zweidimensionale Tabelle, die für jede im Programm vorkommende Variable eine Spalte, zwei Spalten für die fortlaufende (Zeilen-) Nummerierung und für die aktuelle Aktion (dies ist in der Regel eine Anweisung oder die Auswertung eines Ausdrucks) sowie eventuelle weitere Spalten für Hilfsinformationen besitzt.

Trage in die erste Zeile die Anfangssituation ein, also die erste Aktion des Programms und die Werte der Variablen nach Durchführung dieser Aktion. Trage in die jeweils nächste Zeile mit der Nummer k die im k-ten Schritt durchgeführte Aktion und die Werte der Variablen nach Durchführung dieser Aktion ein, solange bis halt oder end erreicht wurden. Ein- und Ausgabe notiere man gesondert. Die so entstandene Tabelle heißt Ablaufprotokoll des Programms für die gegebenen Eingabedaten.

Fortsetzung Beispiel 1.1.1.6

```
program quadratzahl3 is
var x, u, q, erg: Variablen für natürliche Zahlen;
begin  read (x);
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
            if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write (erg)
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
1	read(x)	14	⊥	⊥	⊥	
2	q := 0	14	⊥	0	⊥	
3	u := 1	14	1	0	⊥	
4	erg := 0	14	1	0	0	
5	q ≤ x	14	1	0	0	true
6	q = x	14	1	0	0	false
7	q := q+u	14	1	1	0	
8	u := u+2	14	3	1	0	
9	q ≤ x	14	3	1	0	true


```
program quadratzahl3 is
var x, u, q, erg: Variablen für natürliche Zahlen;
begin  read (x);
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
            if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write (erg)
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
10	q = x	14	3	1	0	false
11	q := q+u	14	3	4	0	
12	u := u+2	14	5	4	0	
13	q ≤ x	14	5	4	0	true
14	q = x	14	5	4	0	false
15	q := q+u	14	5	9	0	
16	u := u+2	14	7	9	0	
17	q ≤ x	14	7	9	0	true
18	q = x	14	7	9	0	false


```
program quadratzahl3 is
var x, u, q, erg: Variablen für natürliche Zahlen;
begin  read (x);
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
            if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write (erg)
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
19	q := q+u	14	7	16	0	
20	u := u+2	14	9	16	0	
21	q ≤ x	14	9	16	0	false
22	write(erg)	14	9	16	0	false
23	"end"	14	9	16	0	

Ausgabe ist 0, d.h., die Eingabe ist keine Quadratzahl.

Vergleich: Wie wird dieser Algorithmus in Ada 95 formuliert?

```
program quadratzahl3 is
var x, u, q, ergebnis:
  Variablen für natürliche Zahlen;
begin  read (x);
   q := 0;  u := 1; ergebnis := 0;
   while q ≤ x do
      if q = x then
          ergebnis := 1 fi;
      q := q+u; u := u+2
   od;
   write (ergebnis)
end
```

```
PROCEDURE  quadratzahl3  IS
x, u, q, ergebnis: INTEGER;
        -- INTEGER sind ganze Zahlen
BEGIN  GET (x);
   q := 0;  u := 1; ergebnis := 0;
   WHILE q ≤ x LOOP
      IF q = x THEN
          ergebnis := 1; END IF;
      q := q+u; u := u+2;
   END LOOP;
   PUT (ergebnis);
END
```


Beispiel 1.1.1.8:
Wir greifen nun noch einmal das Beispiel 1.1.1.3 auf.
Erinnerung: Addiere 1 zu einer dezimal dargestellten Zahl.

Umgangssprachliche Formulierung: Eine Zahl sei als eine Folge von Ziffern aus der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} gegeben. Falls die letzte Ziffer nicht die 9 war, so ersetze sie durch die nächste größere Ziffer und beende das Verfahren, anderenfalls ersetze sie durch 0 und wiederhole das Verfahren für die zweitletzte Ziffer usw.

Aufgabe: Formulierung dieses Algorithmus mit Hilfe unserer Sprachelemente.

Problem: Wie beschreibt man eine Ziffernfolge?
Vorschlag: mit indizierten Variablen.

Betrachte irgendeine Folge von Ziffern
aus der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}:

5	2	6	8	3	9
↕	↕	↕	↕	↕	↕
X_1	X_2	X_3	X_4	X_5	X_6

"Indizierte Variable"

Vorläufig schreiben wir bei der Deklaration:
var X_1, ..., X_n: Variablen für ...
und in Anweisungen X_i, wenn wir die i-te dieser Variablen verwenden wollen. (In Abschnitt 1.1.2 präzisieren wir dies.)

Formulierung des oben umgangssprachlich formulierten Algorithmus mit Hilfe unserer Sprachelemente:


```
program  add1_versuch1  is
var  i, n: Variablen für natürliche Zahlen;
     x1, ..., xn: Variablen für Ziffern;
begin  read(n);
    for i:=1 to n do read(xi) od;
    for i:=n downto 1 do
        if xi ≠ 9 then xi := nächsteziffer(xi)
        else xi := 0 fi
    od;
    for i:=1 to n do write(xi) od
end
```

Vorsicht: falsches Programm!

Hier treten alle Unklarheiten, die wir schon in Beispiel 1.1.1.3 besprochen hatten, deutlich auf. Zugleich befindet sich in der zweiten Zählschleife ein gravierender Fehler. Zum Beispiel liefert die Eingabe 1492 die Ausgabe 2503.

Klärung: Was bedeutet x_i := nächsteziffer(x_i) ? Antwort:
Diese Anweisung steht als Abkürzung für

```
if xi = 0 then xi := 1
else if xi = 1 then xi := 2
else if xi = 2 then xi := 3
else if xi = 3 then xi := 4
else if xi = 4 then xi := 5
else if xi = 5 then xi := 6
else if xi = 6 then xi := 7
else if xi = 7 then xi := 8
else if xi = 8 then xi := 9 fi  fi  fi  fi  fi  fi  fi  fi  fi
```


Beseitigung des gravierenden Fehlers: Die Anweisung
x_i := nächsteziffer(x_i) darf nur genau einmal ausgeführt werden.

```
for i:=n downto 1 do
      if xi ≠ 9 then xi := nächsteziffer(xi)
      else xi := 0 fi
od;
```

müsste ersetzt werden durch:

```
for i:=n downto 1 do
      if xi ≠ 9 then xi := nächsteziffer(xi); "for-Schleife abbrechen"
      else xi := 0 fi
od;
```

Wir müssten also eine neue elementare Handlung, nennen wir sie "exit" (englisch "Ausgang"), einführen mit der *Bedeutung:* Verlasse die aktuelle Schleife, d.h., setze die Ausführung des Programms mit der Anweisung fort, die unmittelbar auf diese Schleife folgt.

In Ada 95 gibt es genau diese Anweisung EXIT. Da die Algorithmen aber hierdurch meist schwerer zu lesen sind bzw. hierdurch leicht Fehler entstehen, wollen wir diesen Weg zunächst nicht beschreiten.

Statt dessen können wir eine Boolesche Variable "fertig" einführen, die anfangs false ist und die auf true gesetzt wird, sobald x_i := nächsteziffer(x_i) ausgeführt wurde. Wenn fertig den Wert true hat, darf kein x_i mehr verändert werden. Wir bauen diesen Gedanken in das Programm ein.

Die Ergänzungen sind farbig gekennzeichnet:

```
program  add1_versuch2  is
var  i, n: Variablen für natürliche Zahlen;
     fertig: Variable für Boolesche Werte;
     x1, ..., xn: Variablen für Ziffern;
begin  read(n); fertig := false;
    for i:=1 to n do read(xi) od;
    for i:=n downto 1 do
        if not fertig then
            if xi ≠ 9 then xi := nächsteziffer(xi); fertig := true
            else xi := 0 fi
        fi
    od;
    for i:=1 to n do write(xi) od
end
```

Vorsicht: immer noch falsches Programm!

Was ist nun noch falsch?

Besteht die Eingabe nur aus Neunen, so werden nur Nullen ausgegeben. In diesem Fall muss also eine '1' als erstes ausgegeben werden.

Diesen Fall erkennen wir daran, dass nach Durchlaufen der zweiten for-Schleife die Variable fertig immer noch den Wert false besitzt. In diesem Fall geben wir also zuerst eine '1' aus.

Den Fall, dass eine Zahl n < 1 anfangs eingegeben wird, müssen wir noch berücksichtigen. In diesem Fall wird nichts eingelesen und nichts ausgegeben, so dass auch die Ziffer 1 nur im Fall n ≥ 1 ausgedruckt werden darf.


```
program  add1_versuch3  is
var  i, n: Variablen für natürliche Zahlen;
     fertig: Variable für Boolesche Werte;
     x1, ..., xn: Variablen für Ziffern;
begin  read(n); fertig := false;
    for i:=1 to n do read(xi) od;
    for i:=n downto 1 do
        if not fertig then
            if xi ≠ 9 then xi := nächsteziffer(xi); fertig := true
            else xi := 0 fi
        fi
    od;
    if (not fertig) and (n ≥ 1) then write(1) fi;
    for i:=1 to n do write(xi) od
end
```

Vorsicht: schlechter Programmierstil!

Warum ist dies ein "schlechter Programmierstil" gewesen?

Wichtig ist, dass sich im Algorithmus die Struktur des Problems und eine angemessene Lösung widerspiegelt und dass das Programm möglichst keine Anteile enthält, die auf Grund der Darstellung (also der vorgegebenen Sprachelemente) eingefügt werden müssen, die aber mit dem Problem nichts zu tun haben.

Wenn das Programm das Lösungsverfahren nicht klar zum Ausdruck bringt, sondern es verschleiert, und wenn das Programm nicht "gut lesbar" und daher auch nicht "wartbar" (d.h., es lassen sich Fehler nur schwer finden und das Programm kann kaum an ähnliche Situationen angepasst werden), so liegt ein schlechter Programmierstil vor.

Unser Programm add1_versuch3 mag zwar korrekt arbeiten, aber es gibt die Lösung nicht angemessen wieder.

Auffälligste Abweichung: Unser Programm durchläuft in jedem Fall alle Variablen, obwohl die Lösungsmethode zu Ende ist, sobald ein $x_i \neq 9$ erreicht ist.

Grund für diese Abweichung: Wir haben eine Zählschleife verwendet, die aber die Lösungsmethode nicht angemessen widerspiegelt. Vielmehr besagt die Lösungsmethode: Solange man (von hinten beginnend) eine 9 vorfindet, ersetze sie durch eine 0. Die danach erreichte Ziffer wird durch ihre nächst größere Ziffer ersetzt bzw., wenn man am Anfang angelangt war, wird die Ziffer 1 vorangestellt.

Wir müssen statt der for- also eine while-Schleife verwenden.

Diese lautet:

i := n;
while $x_i = 9$ do $x_i := 0$; i := i-1 od;

Nun sind wir bei einem $x_i \neq 9$ angelangt und können sie durch ihre nächst größere Ziffer ersetzen.

Fehlerhaft ist noch, dass im Falle einer Folge aus Neunen die Variable i den Wert 0 erhält und dann die nicht vorhandene Variable x_0 in der while-Bedingung auftritt. Wir müssten also noch ein "i > 0" in die while-Bedingung aufnehmen:

i := n;
while $(x_i = 9)$ and (i > 0) do $x_i := 0$; i := i-1 od;

Dies ist aber wenig hilfreich, denn wenn i = 0 geworden ist, dann steht im Booleschen Ausdruck $(x_i = 9)$ and (i > 0) immer noch die nicht vorhandene Variable x_0.

Ada 95 behilft sich hier folgendermaßen: Es wird neben dem symmetrischen Operator and ein weiterer Operator and then eingeführt. α and then β bedeutet: Werte erst α aus. Falls der Wert false ist, so ist der ganze Ausdruck false; anderenfalls werte β aus und dessen Wert ist dann der Wert des gesamten Ausdrucks. In Ada 95 könnte man also die Bedingung formulieren (dort schreibt man x(i) an Stelle von x_i):

i := n;
WHILE (i > 0) AND THEN (x(i) = 9) LOOP
x(i) := 0; i := i-1 END LOOP;

Wir müssen aber nicht so vorgehen. Vielmehr können wir von hinten beginnend maximal nur bis zur zweiten Ziffer prüfen und anschließend den kritischen Fall abfangen:

i := n;
while $(x_i = 9)$ and (i > 1) do $x_i := 0$; i := i-1 od;
if (i = 1) and $(x_1 = 9)$ then $x_1 := 0$; write (1)
else x_i := nächsteziffer(x_i) fi;
gib x_1 bis x_n aus.

Dies führt zu folgendem Programm (wir fangen noch den Fall n < 1 zu Beginn ab):


```
program add1_version1 is
var i, n: Variablen für natürliche Zahlen;
    x1, ..., xn: Variablen für Ziffern;
begin read(n);
 if n > 0 then
    for i:=1 to n do read(xi) od;
    i := n;
    while (xi = 9) and (i > 1) do xi := 0; i := i-1 od;
    if (i = 1) and (x1 = 9) then x1 := 0; write (1)
        else xi := nächsteziffer(xi) fi;
    for i:=1 to n do write(xi) od
 fi
end
```


Eine andere Variante besteht darin, mit einem "Stopper" zu arbeiten. Hierzu erweitert man die Daten um ein Element, mit dem garantiert wird, dass die while-Schleife stets korrekt beendet wird.

Im Falle der Addition einer 1 fügen wir die Variable x_0 hinzu und setzen sie auf 0. Dann ist garantiert, dass die while-Schleife spätestens für i = 0 abgebrochen wird.

Wurde nach der while-Schleife x_0 verändert, dann lag eine Folge aus Neunen vor und die Anzahl der Ziffern erhöht sich um eine Ziffer, anderenfalls bleibt die Anzahl gleich. Alle Abfragen bzgl. "i > 1" oder "i = 1" entfallen jetzt.

Mit dem Stopper x_0 erhalten wir folgendes Programm:


```
program add1_version2 is
var i, n: Variablen für natürliche Zahlen;
    x0, x1, ..., xn: Variablen für Ziffern;
begin read(n);
 if n > 0 then
    for i:=1 to n do read(xi) od;
    x0 := 0; i := n;
    while xi = 9 do xi := 0; i := i-1 od;
    xi := nächsteziffer(xi);
    if x0 ≠ 0 then write(x0) fi;
    for i:=1 to n do write(xi) od
 fi
end
```

Dieses Programm add1_version2 ist (nach einiger Erfahrung mit Programmen unter Verwendung unserer Sprachelemente) leicht lesbar und folgt zugleich der üblichen Vorstellung, dass man zu jeder Zahl eine führende Null hinzufügen kann, die verändert werden muss, sofern ein Überlauf bei der Addition entsteht.

Unbefriedigend ist noch, dass die Datenbereiche und die Deklaration der Variablen umgangssprachliche Elemente enthält. Wir müssen uns daher als nächstes der genauen Festlegung der Wertebereiche zuwenden.