Lutz Wegner

Programmierung in Pascal

Universität Kassel

```
program Summe;
var
 summe, i, wert: Integer;
begin
 summe := 0;
 i := 1;
 writeln('Bitte Werte eingeben, Abbruch mit Wert <= 0');
 write(i,'.ter Wert: '); readln(wert);
  while wert > 0 do
   begin
     summe := summe + wert;
     i := i + 1;
     write(i,'.ter Wert: '); readln(wert);
   end;
 writeln('Anzahl der Eingabewerte groesser Null: ', i-1);
 writeln('Summe der Werte ist: ', summe);
end.
```

SKRIPTEN DER PRAKTISCHEN INFORMATIK

Lutz Wegner

Universität Gh Kassel

FB Mathematik-Informatik

D-34109 Kassel

Folienvorlage zur Vorlesung im Wintersemester 1998/99

Inhalt

1 Einleitung

Programmieren bedeutet, eine Folge von Instruktionen — das *Programm* — so aufzuschreiben, daß sich damit eine Aufgabe auf einer Maschine — dem *Rechner* — ausführen läßt. Lernen zu programmieren heißt, die *Syntax* einer *Programmiersprache* — d.h. die Struktur und Form gültiger Programme — und ihre *Semantik* — d.h. ihre Wirkung — zu verstehen, um mit diesem Wissen korrekte, verständliche und effiziente Programme entwickeln zu können.

Seit Jahrzehnten wird darüber gestritten, ob Studenten, speziell wenn sie nicht Informatik im Hauptfach studieren, überhaupt lernen sollten, zu programmieren (vgl. die Debatte mit vielen Literaturstellen in [GH98]). Viele argumentieren, es genüge, mit dem Rechner umgehen zu können, etwa Textverarbeitung und Tabellenkalkulation zu beherrschen. Gern wird die Analogie zum Autofahren bemüht, bei dem man ja auch von *A* nach *B* kommt, ohne daß man nachweisen muß, daß man eine Kupplung zusammenbauen kann.

Die andere Seite behauptet, es gehe um die grundlegende Erfahrung, einer Maschine eindeutig und fehlerfrei in elementaren Schritten beizubringen, eine vorgegebene Aufgabe zu lösen. Dazu gehört die langwierige Korrektur des Programms, damit es vom wenig flexiblen Rechner — genauer dem Übersetzerprogramm im Rechner — überhaupt erst einmal akzeptiert wird. Zweitens der noch viel schwierigere Teil, durch Testen oder vorzugsweise formales Beweisen sicherzustellen, daß das Programm für alle möglichen Eingaben die korrekte Ausgabe liefert.

Donald Knuth, der wahrscheinlich am meisten respektierte Informatiker unserer Zeit, formulierte es 1974 in einem Artikel in *American Mathematical Monthly* so: “It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e. express it as an algorithm.“

Unter denjenigen, die das Erlernen einer Programmiersprache für wichtig erachten, bricht als nächstes sofort eine hitzige Debatte darüber aus, welches Programmierparadigma — prozedural, deklarativ, funktional, logisch oder objekt-orientiert — denn zu lehren und was heutzutage die zu bevorzugende Programmiersprache sei. Dogmatische Verfechter einer Richtung verweisen dabei auf den Einfluß, den die erste gelernte Sprache — sozusagen die Muttersprache — auf das Denken habe. Die gemäßigte Fraktion hält dies für nicht so kritisch, da Studenten im Laufe des Studiums sowieso eine zweite oder dritte Sprache lernen werden.

Eine radikale Splittergruppe, angeführt von Edsger Dijkstra, möchte Programmieren als Kurs in abstraktem mathematischen Denken sehen und die maschinelle Ausführung, und damit besonders die ad-hoc Methoden des Testens, verbieten.

Der Verfasser dieses Skripts ist der Meinung, daß das Erlernen einer Programmiersprache, bzw. der Methodik des Programmierens

- grundsätzlich nützlich ist
- in mehreren Schüben erfolgt, wobei die wirkliche Reife erst mit der zweiten oder dritten Sprache kommt[1]
- mit einer kleinen, einfachen, aber praktikablen Sprache anfangen sollte.

Erfüllt Pascal, die von Niklaus Wirth (ETH Zürich) 1968 entwickelte Programmiersprache, das letzte Kriterium?

---

1. Der Verfasser hat einmal nachgezählt und ist, dank seines fortgeschrittenen Alters, seit seinem ersten Fortran Kurs als Schüler 1968 in den USA bis heute auf 12 Sprachen gekommen, in denen er tatsächlich mindestens ein Programm selbst geschrieben, übersetzt (bzw. interpretiert) und zur Ausführung gebracht hat.

Bedingt! Pascal ist eine überschaubare, sauber formulierte Sprache für das prozedurale Programmierparadigma. In der Ausprägung Turbo Pascal der Fa. Borland, etwa als Version 5.5 oder später, hat es einige zusätzliche, praktische Vorteile:

- es läuft sogar unter MS DOS auf sehr mageren PCs, selbst auf einem 286er,
- es hat einen komfortablen Editor zum Eingeben und Korrigieren der Programme,
- es hat einen Debugger, der die Fehlersuche unterstützt,
- es bietet getrenntes Übersetzen einzelner Programmteile und Zubinden von Bibliotheksprogrammen, somit wird eine der Grundlagen zur Erstellung größerer Programmpakete vermittelt,
- es hat zusätzliche Funktionen zur Behandlung von Festplattendateien, wodurch selbst kleinere kommerzielle Anwendungen möglich sind,
- es hat mit Turbo Vision eine graphische Schnittstelle, so daß auch eine interaktive Bildschirmein-/ausgabe möglich wird.

Pascal ist aber nur bedingt erste Wahl, denn

- es ist nicht mehr sehr verbreitet, speziell seit C, C++ und Java in Mode gekommen sind
- es ist nicht wirklich objekt-orientiert, obwohl TURBO Pascal ab 6.0 Objektorientierung, auch Ereignisse und die damit mögliche Form der „asynchronen Ablaufsteuerung“, unterstützt
- es ist zwar unter einigen UNIX Systemen verfügbar (LINUX, Solaris und generell per GNU Compiler), ist aber keine typische UNIX-Sprache und hat eine schlechte Anbindung an die Bildschirmgestaltung.

Alternativ könnte man auf die von Wirth angebotenen Nachfolgesprachen Modula und Oberon wechseln, die aber ähnliche Probleme wie Pascal (Verbreitung, etc.) haben. Java ist z.Zt. sehr in der Diskussion, leidet aber unter der Verwandtschaft mit C (unschöne Syntax, etc.).

Eine wirklich interessante Alternative wäre John Ousterhout's Skriptsprache Tcl (Tool command language, ausgesprochen „Tickel"). Tcl kennt nur Zeichenketten als Typ, ist objektorientiert und unterstützt eine asynchrone Ablaufsteuerung. Die Sprache wurde direkt für die graphische Bildschirmgestaltung entwickelt und ist unter UNIX und Windows frei verfügbar. Sie bietet erstklassige Netzunterstützung und vieles mehr.

Tcl ist allerdings (momentan) noch nicht sehr verbreitet und gilt wegen der Herkunft als Macrosprache mit zunächst verwirrender textuellen Ersetzungsregeln als exotisch und für Anfänger nur bedingt geeignet.

Kurz und gut ...

... es bleibt bei Pascal, wobei wir auf unseren betagten PCs Turbo Pascal und bei der Erstellung dieses Skripts GNU Pascal unter UNIX verwenden wollen, allerdings jeweils nur mit den Sprachkonstrukten, die der erste Pascal Standard ISO 7185 [ISO83] aus dem Jahre 1983 vorsieht und einigen vorsichtigen Erweiterungen, soweit sie zwischen GNU und Turbo verträglich sind. Besonders wollen wir

- guten Programmierstil üben,
- Schritt für Schritt vorangehen, um auch absolute Anfänger nicht zu überfordern (Studenten mit Vorkenntnissen mögen Teile überschlagen)
- interessante, auch ein bißchen raffinierte Beispiele bringen
- ein wenig auch graphische Ausgaben mit einbeziehen, denn „das Auge ißt mit."

Einem alten Programmierbrauch folgend wollen wir die Einleitung mit einem ersten, kleinen Programm (abgespeichert in der Datei `hallo.pas`) abschließen, das traditionsgemäß den Text „Hallo Leute!" ausgibt[1].

1. im englischen Sprachraum ist „hallo world!" üblich.

```
program hallo;
begin
 Writeln('Hallo Leute!');
end.
```

Unter dem GNU Pascal Compiler `gpc` auf einem UNIX- oder LINUX-Rechner übersetzen wir das Pascal Quellprogramm `hallo.pas` mittels der Übersetzeroption `-o` in die Objektcodeausgabe `hallo` und lassen `hallo` laufen mit den folgenden Kommandos.

```
wegner@elsie(Sources)$ gpc hallo.pas -o hallo
wegner@elsie(Sources)$ ./hallo
Hallo Leute!
wegner@elsie(Sources)$
```

Übung 1–1

Treiben Sie einen Rechner auf, der Pascal kann, z. B. die in unserem Rechnerlabor. Geben Sie das oben aufgelistete Programm mit einem Editor ihrer Wahl ein und bringen Sie es zum Laufen! Lassen Sie sich nicht von Tippfehlern und komischen Fehlermeldungen entmutigen. Nerven Sie den Betreuer und fragen Sie Kommilitonen bei Problemen, bis Ihr Programm wirklich die gewünschte Ausgabe anzeigt.

Übung 1–2

Gehen Sie mit einem Web-Browser ihrer Wahl zur Seite

`http://www.cps.unizar.es/~jcampos/gpc_doc/team.html`

und studieren Sie, welche Pascal-Implementierung unter dem GNU-Projekt auf ihrer Anlage verfügbar ist.

2 Erste Schritte

Da wir keinerlei Wissen über Informationsverarbeitung voraussetzen, müssen wir zunächst ein gedankliches Modell der Verarbeitung von Daten in einem Rechner vermitteln.

2.1 Eingabe - Verarbeitung - Ausgabe

Schauen Sie sich das folgende tolle Programm `adam.pas` - auch ohne Kenntnisse von Pascal - an!

```
program AdamRiese;
var
 x, y, z: Integer;
begin
 Write('Eingeben x: '); Readln(x);
 Write('Eingeben y: '); Readln(y);
 z := x + y;
 Writeln('x plus y ist :', z);
 z := x * y;
 Writeln('x mal y ist :', z);
 Writeln('... und tschuess');
end.
```

Verfolgen Sie seine aufregende Abarbeitung!

```
$ gpc adam.pas -o adam
$ ./adam
Eingeben x: 17
Eingeben y: 4
x plus y ist : 21
x mal y ist : 68
```

```
... und tschuess
$
```

Es ist klar, daß hier wohl zwei ganze Zahlen eingelesen werden und deren Summe, bzw. Produkt berechnet und ausgegeben wird. Die einzulesenden Werte, im Beispiel die Zahlen 17 und 4, werden im Programm den Variablen x und y zugewiesen und können über diese beiden Bezeichner angesprochen werden. Für das Resultat haben wir eine Variable z eingeführt, obwohl es auch ohne sie ginge.

Im Rechner werden für x, y und z Speicherzellen mit uns unbekannten Adressen reserviert. In diese werden die momentan gültigen Werte von x, y und z geschrieben. Verändern wir einen Wert, z.B. indem wir z durch den Wert von x * y ersetzen, wird der alte numerische Wert (im Beispiel 21) für z durch den neuen Wert, hier 68, (unwiederbringlich) überschrieben.

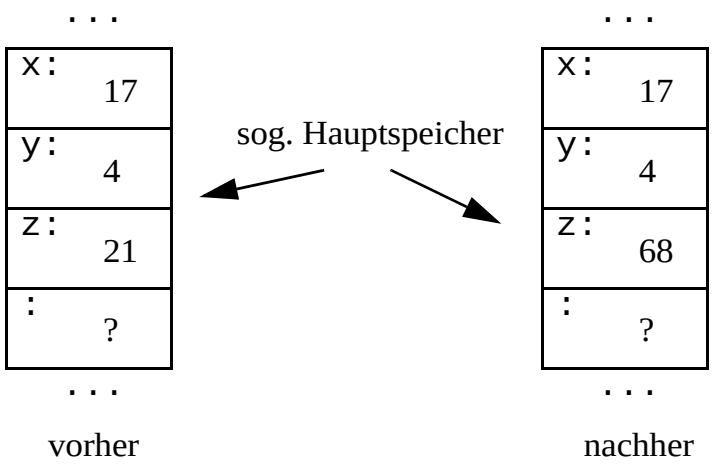


Abb. 2–1

Ferner sind alle Variablen, also hier x und y, bei Programmstart zunächst undefiniert und sie überleben das Programmende (und im übrigen auch einen Rechnerausfall) nicht, d.h. beim nächsten Lauf kann man nicht auf alte Werte zurückgreifen.

Die Zuordnung von Variablen zu speziellen Speicherzellen (Hauptspeicheradressen) ist dem Anwender des Programms unbekannt, kann in der Regel nicht beeinflußt werden und ist auch unwichtig. Sie wird vom **Übersetzer** (Compiler) in Verbindung mit Linker/Lader erledigt und variiert von Rechner zu Rechner (und ggf. von Programmstart zu Programmstart) je nach Betriebssystem, Rechnerausstattung, anderen geladenen Programmen, usw.

Die Tatsache, daß man sich nicht um die Speichervergabe und interne Darstellung selbst kümmern muß, im Gegensatz etwa zur Situation bei Verwendung einer Maschinensprache, ist einer der Vorteile einer **höheren Programmiersprache**, wie z.B. Pascal. Allerdings kann man auch bei Programmen in einer höheren Programmiersprache nicht auf Festlegungen bezüglich Form und Struktur verzichten. Sie werden durch die sog. **Syntax** der Programmiersprache sowie einige pragmatische weitere Anforderungen bestimmt.

2.2 Sprachelemente und Programmeingabe

Die Eingabe eines Programmtexts muß gewissen Regeln genügen. Dazu gehört, daß man in dem kleinen Beispielprogramm `hallo.pas`

```
program hallo;
begin
 Writeln('Hallo Leute!');
end.
```

einzelne **Sprachelemente** (engl. *tokens*) identifizieren kann — z.B. **`program`**, **`begin`**, `Writeln` oder Symbole wie `';'` — die durch andere Zeichen, z.B. Leerzeichen, Neue-Zeile, usw. voneinander getrennt sind.

Bei den Wörtern, also **`program`** oder `hallo`, unterscheiden wir

- reservierte Schlüsselwörter
- vordefinierte Bezeichner
- frei wählbare Bezeichner.

Zu den reservierten **Schlüsselwörtern** (engl. *keywords*) gehören etwa **`program`**, **`begin`**, **`end`**. Sie haben in Pascal eine fest vorgegebene

Bedeutung und dürfen nicht als Bezeichner für Variablen, also in dem Sinn von `x` und `y` oben, vom Programmierer frei verwendet werden. Wir heben sie in diesem Skript im Programmtext durch Fettdruck hervor. Der Anhang A enthält eine Liste aller Schlüsselwörter.

Viele Editoren erkennen automatisch solche Bezeichner und färben sie im Editorfenster besonders ein. Meist ist ihre Position im Programm auch genau vorgegeben, etwa **`begin`** und **`end`** als syntaktische Klammer um den Anweisungsteil eines Programms.

Vordefinierte Bezeichner, wie etwa `Writeln` oder `Integer`, nicht jedoch `hallo`, sind in Pascal mit einer speziellen Bedeutung vorbelegt, könnten aber ggf. vom Anwender umdefiniert werden. So ist `Writeln` der Bezeichner für die Standardausgaberoutine mit anschließendem Vorschub auf eine neue Zeile. Die wissentliche oder unbeabsichtigte Umdefinition wird man in der Regel vermeiden wollen und man tut gut daran, sich nach und nach diese Wörter, die auch **Standardbezeichner** (Standardnamen, vgl. Anhang B) heißen, zu merken. Im Skript hier werden sie ohne Fettdruck ausgegeben.

Frei wählbar sind die übrigen Bezeichner, wie z.B. `hallo`. Wie bei den Schlüsselwörtern und bei den vordefinierten Bezeichnern wird nicht zwischen Groß- und Kleinschreibung unterschieden. **Bezeichner** (auch **Namen** genannt) dürfen alle Buchstaben und Ziffern des Alphabets, mit Ausnahme der Umlaute und ‘ß’, sowie den Unterstrich (ASCII Code $5F) enthalten und müssen mit dem Unterstrich oder einem Buchstaben beginnen. Ungültige Namen wären demnach

`Klaus-Dieter`, `2good4you`, `34109Kassel`.

Gültig wären dagegen

`x1`, `x_1`, `_EnableDisplay`, `y2P_q17krz`

Den folgenden Symbolen kommt ferner in Pascal eine besondere Bedeutung zu:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { } $ #
```

z.T. haben Kombinationen davon Verwendung als Operatoren oder Trenner, z.B. `:=` als Zuweisungsoperator, `<>`, `<=`, `>=` als Vergleichsoperatoren, `..` für Teilbereiche, sowie `(.` und `.)`, bzw. `(*` und `*)` als Ersatzdarstellungen für `[ ]` bzw. `{ }`.

Bezeichner (Namen) sind wohlunterschieden von **Zeichenketten**, d.h. Folgen beliebiger Zeichen, die von einzelnen Apostrophen (Anführungszeichen) eingeschlossen werden. Die Zeichenkette `'hallo Leute!'` in `Writeln('Hallo Leute!')` wäre ein Beispiel. Statt von Zeichenketten spricht man auch von **Literalen**. Hier wird natürlich zwischen Groß- und Kleinschreibung unterschieden und Symbole, bis auf `'` und *Neue-Zeile* (ASCII 13), verlieren ihre Bedeutung. Wir gehen darauf später nochmals ein.

Die bisher diskutierten Regeln — bis auf die Schreibweise im Skript mit Fettdruck — sind Vorschriften der PASCAl-Syntax und notwendig, damit der Übersetzer unsere Programme als korrekt akzeptieren und ihnen eine eindeutige Bedeutung (Semantik) zuordnen kann. Obwohl hier jetzt manche Vorschrift sehr kleinlich erscheinen mag, sind die Regeln meist selbsterklärend, wenn man sich mögliche Mehrdeutigkeiten vor Augen hält.

Neben den strikten Syntaxregeln, die wir später auch durch eine **Grammatik** formalisieren wollen, gibt es auch Empfehlungen für das Aussehen der Programme. Da man die Tätigkeit des Aufschreibens eines Programms auch als **Kodieren** (engl. *coding*) bezeichnet, spricht man vom **Kodierstil**.

So ist der folgende Einzeiler

```
PROGRAM HALLO;BEGIN WRITELN('Hallo Leute!');END.
```

völlig gleichwertig in der Ausgabe mit dem oben gezeigten Programm. Trotzdem werden viele dieses Programm als weniger lesbar, oder weniger ästhetisch, als das vorher gezeigte empfinden. Der Kodierstil wird durch

Wahl der Bezeichner, Groß- und Kleinschreibung, Leerzeichen, Einrückung, Trennung in mehrere Zeilen, usw. geprägt und unterliegt gewissen Modeschwankungen. So war bis vor kurzem noch die Großschreibung von Schlüsselwörtern üblich, weil die Editoren und Programmierer keine Möglichkeiten hatten, diese am Bildschirm anders hervorzuheben:

```
PROGRAM hallo;
BEGIN
  WriteLn('Hallo Leute!');
END.
```

Durch moderne Farbmonitore, leistungsfähige Editoren und höhere Ansprüche in der Textgestaltung ist hier ein Wandel eingetreten. Ein anprechendes Aussehen der Programme läßt sich im übrigen durch Formatierprogramme (engl. *pretty print programs*) erreichen, oft gibt es in Unternehmen genaue Richtlinien für die Kodierung.

Neben dem Kodierstil spricht man noch vom **Programmierstil**. Letzterer ist wesentlich schwieriger zu definieren und befaßt sich mit dem Aufbau, der Logik, den eingebauten Sicherheiten, usw. eines Programms. Guter Programmierstil verlangt zunächst einen sauberen **algorithmischen Entwurf**, d.h. das „Kochrezept“, das die einzelnen Schritte und die verwendeten „Zutaten“, sprich Datenstrukturen, für die zu lösende Aufgabe festlegt und das unabhängig von der später verwendeten Programmiersprache ist, muß solide sein.

Obwohl man zur Erzielung eines guten Programmierstils Vorgaben machen kann, ist „gutes Programmieren“ weithin Erfahrungssache und zwischen Kunst, Handwerk und Wissenschaft angesiedelt.

Übung 2–1

Was geht in Sachen Trenner, Schlüsselworte, Literale und was geht nicht? Testen Sie die Faschingsversion des **Hallo Leute!** Programms!

```
programholla;be
gin;;;writeln    ('Heller
Wahn!')end.
```

2.3 Formale Beschreibung der Syntax

Die Festlegung der Syntax einer Programmiersprache im Sprachreport oder einem Handbuch kann mittels natürlicher Sprache oder **formalsprachlich**, z.B. durch eine **Grammatik**, erfolgen. De facto bestimmt der Übersetzer welche Programme als korrekt akzeptiert werden. Ahnliches gilt für die Semantik einer Programmiersprache, die als Funktion *S* betrachtet, jedem speziellen Programm *P* und jeder Eingabe *E* genau eine Ausgabe *A* zuordnet: $S(P, E) = A$.

Der Pascal-Report und in der Folge auch die Turbo Pascal Handbücher haben die Syntax von Pascal mittels einer graphischen Darstellung vorgenommen, die umgangssprachlich auch als Eisenbahndiagramme bezeichnet werden. So werden z.B *Ziffer* und *Ziffernfolge* wie folgt definiert:

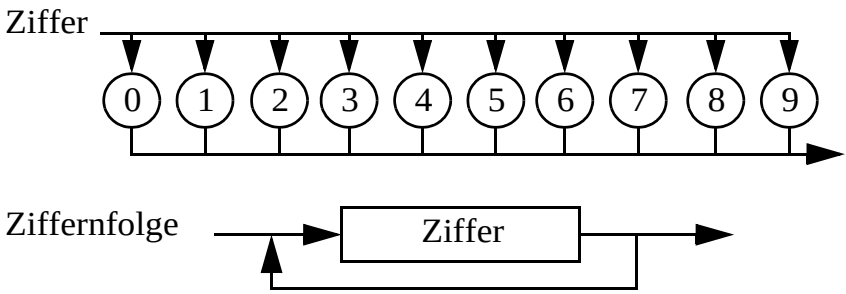


Abb. 2–2

Demnach ist eine Ziffer eines der Zeichen 0 ... 9, eine Ziffernfolge dann eine nichtleere Folge von Ziffern. Als Produktionsregeln einer Grammatik würden wir schreiben:

Ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Ziffernfolge ::= Ziffer | Ziffer Ziffernfolge

Die zweite Regel liest man als:

> eine Ziffernfolge ist eine Ziffer *oder* eine Ziffer gefolgt von einer Ziffernfolge.

Hierfür hat sich wiederum eine verkürzende formale Beschreibung eingebürgert, bei der die Wiederholungsoption durch [...] angedeutet wird:

Ziffernfolge ::= Ziffer [...]

Die anderen Möglichkeiten dieser sog. *erweiterten BNF-Beschreibung* ergeben sich aus der folgenden Tabelle.

syntaktischer Ausdruck	Schreibweise	Sprechweise
Alternative	H_1 | ... | H_n	H_1 oder ... oder H_n
Option	[*H*]	wahlweise *H*
Wiederholung	*H* [...]	*H* mindestens einmal
Wiederholung mit Trenner	*H* [*K* ...]	Liste von *H* mit Trenner *K*
syntaktische Klammerung	{ H }	
Prioritäten: Wiederholung vor Anreihung vor Alternative		

Tab. 2–1 *Erweiterte BNF Notation*

Zukünftig wollen wir diese erweiterte BNF (Backus-Naur-Form) verwenden und geben als weiteres Beispiel die Definition numerischer Werte in Pascal an, d.h. ganzer Zahlen (Integer) mit oder ohne Vorzeichen wie

```
5, -317, +2
```

und der Real-Zahlen (numerisch reele Zahlen) wie

```
3.17, -2.384E-3, 0.5e-20
```

Syntax

Zahl ::= [+ | -] vorzeichenlose-Zahl
vorzeichenlose-Zahl ::= vorzeichenloser-Integer
[. Ziffer [...]][[E | e] [+ | -] vorzeichenloser-Integer]
vorzeichenloser-Integer ::= Ziffer [...]

Ganze Zahlen tauchen in Zusammenhang mit Zählen und Numerieren auf, also für Einwohnerzahlen, die Waggons eines Zugs, Stockwerke,

Bankleitzahlen, Jahreszahlen. Beim Programmieren treten sie häufig als Indexwerte auf, etwa für die *i*. Zeile, das *j*. Element, usw.

Gültige Literaldarstellungen (d.h. Schreibweisen der Werte) in Programmen und bei der Eingabe sind

```
6, 0, -6, +7000000
```

ungültige dagegen

```
6,437,271, 3.700.000, -6.0
```

d.h. Dezimalkommata oder Dezimalpunkte, bzw. Abtrennungen von Tausenderstellen sind nicht zugelassen.

Intern ist die Darstellung einer ganzen Zahl immer genau und erfolgt meist als Binärzahl im Zweierkomplement mit 32 Bit je Wert. Daraus ergibt sich ein Wertebereich von -2147483648 bis 2147483647, das sind -2^{31} bis $2^{31} - 1$. Wir werden darauf im Zusammenhang mit Typdefinitionen nochmals eingehen.

Bei den Realwerten gibt es eine Gleitkommadarstellung (engl. *floating point*), auch Gleitpunktdarstellung oder Fließkommadarstellung genannt, bei der genau ein Punkt die Nachkommastellen vom ganzzahligen Teil abtrennt. Gültig sind demnach

```
0.0, 0.873, -74.1, 73.36789, 253.0
```

ungültig dagegen

```
0., .873, 73,36789
```

Die zweite Darstellung bezeichnet man als Exponentendarstellung, wobei der Buchstabe `E` oder `e` den Exponenten von der Mantisse trennt und als „zehn hoch“ zu lesen ist. Die Mantisse darf einen Dezimalpunkt enthalten, der Exponent nicht.

Gültig

```
0E0, 8.73E-1, -741e-1, 0.7336789E2, 2.53E2,
25.3E+01, 253E0, 2530E-1
```

ungültig z.B.

35.6e0.5, 35,6e2

Auf den Wertebereich und die Probleme mit Real-Zahlen gehen wir in den Anfängervorlesungen „Grundzüge der Informatik I und II“ ein. Hier sei nur darauf hingewiesen, daß die interne Darstellung nicht notwendigerweise genau ist. So ergibt z.B. dezimal 0.1 einen nichtabbrechenden Dualzahlenbruch, d.h. dieses Literal hat keine gleichwertige interne Speicherung.

Übung 2–2

In Turbo Pascal gibt es auch Hexadezimalzahldarstellungen, also Werte zur Basis 16. Entsprechend sind `A` ... `F`, bzw. `a` ... `f` die sog. Hexadezimalziffern für die Werte 10 bis 15. Hexadezimalzahlen sind immer vorzeichenlos und werden mit einem vorangestellten Dollarzeichen im Programm dargestellt. Somit ist

`$FF` die dezimale Zahl 255
`$11` die Dezimalzahl 17 ($1*16^1 + 1*16^0$).

Für die maschinennahe Programmierung ist die Hexadezimaldarstellung sehr nützlich, da eine Hexadezimalziffer gerade 4 bit entspricht (einer Tetrade = einem halben Byte). Bewaffnen Sie sich mit einer ASCII-Tabelle und überprüfen Sie, was das folgende Programm leistet! Na, klingelt’s?

Hinweis: `char(`*x*`)` wandelt den Wert *x*, hier jeweils eine Hexadezimalzahl, in ein „druckbares“ Zeichen um.

```
program klingel;
begin
 writeln(char($48), char($61), char($6C),
   char($6C), char($6F), char($07));
end.
```

2.4 Kommentare

Kommentare sind Anmerkungen in Programmen. Kommentare in Pascal werden durch geschweifte Klammern eingefaßt. Alternativ können statt { und } auch paarweise (* und *) verwendet werden. Der Compiler ignoriert Kommentare bei der Übersetzung, allerdings lassen sich darin auch Direktiven an den Compiler oder andere Vorübersetzer verstecken. In Turbo Pascal geschieht dies für Kommentare, die unmittelbar nach { bzw. (* ein Dollarzeichen ($) enthalten.

Kommentare werden wie Leerzeichen als Trenner interpretiert.

```
program OhneKommentar;

begin{jetzt geht's los}writeln('{Kein Kommentar!}');

 writeln((*Das is nix*)'')(*und jetzt (* (* das*)end.
```

Im Beispiel oben sollen die Kommentare natürlich den Leser verwirren. In der Regel setzt man Kommentare zur besseren Erläuterung eines Sachverhalts ein.

```
program Gruppenanzahl;
const
 GS = 20; {Gruppenstaerke}
var
 T,    {Teilnehmer}
 G :Integer; {Gruppen}
begin {Gruppenanzahl}
 write('Anzahl der Teilnehmer: '); readln(T);
 G := T div GS;
 {ganzzahlige Division: G Gruppen zu GS-vielen
Teilnehmern}
 if (T mod GS <> 0) then G := G + 1;
 {wenn welche uebrig, eine Gruppe mehr}
 writeln(T,' Teilnehmer ergeben ', G,' Gruppen');
end {Gruppenanzahl}.
```

Die Regeln für den Einsatz von Kommentaren lauten:

- eher mehr kommentieren als zu wenig
- als Faustregel - eine Zeile Kommentar je Zeile Code

- keine trivialen Kommentare
  ```
  x := y + z; {x ist Summe von y und z}
  ```
- faße dich kurz!
- ein Programm ohne Kommentare, wie clever es auch immer sei, ist wertlos
- ein verkorkstes Programm läßt sich auch mit den besten Kommentaren nicht retten.

Mehr dazu später, insbesondere Regeln zur Kommentierung von Funktionen und Prozeduren, zur Bennenung von Argumenten, zur Anreicherung von **`begin`** und **`end`**, für Vor- und Nachbedingungen, Schleifeninvarianten, usw.

3 Vereinbarungs- und Anweisungsteil

3.1 Allgemeine Programmstruktur

Ein Pascal-Programm besteht aus einem **Programmkopf** und einem **Programmblock**, wobei wir die Aufteilung eines Programms auf mehrere getrennte Einheiten (*units*) vorläufig außen vor lassen.

Programm ::= **program** Bezeichner [**(** Bezeichner [**,** ...] **)**] **;**
Block **.**
Block ::= Vereinbarungsteil Anweisungsteil
Vereinbarungsteil ::= [Markenvereinbarungen]
[Konstantenvereinbarungen]
[Typvereinbarungen]
[Variablenvereinbarungen]
[Prozedur-Funktionsvereinbarungen]
Anweisungsteil ::= **begin** Anweisung [**;** ...] **end**
Markenvereinbarungen ::= **label** Marke [**,** ...] **;**
Marke ::= vorzeichenloser-Integer
Konstantenvereinbarungen ::= **const** {Bezeichner =
Konstantendeklaration **;**} [...]
Typvereinbarungen ::= **type** {Bezeichner = Typ **;**} [...]
Variablenvereinbarungen ::= **var** {Bezeichner [**,** ...] **:**
Typ **;**}[...]
Prozedur-Funktionsvereinbarungen ::= ... *s. weiter unten* ...

Betrachten wir unser Beispielprogramm `adam.pas` aus Kapitel 2. Sein Programmkopf enthält neben dem Schlüsselwort **`pro-`**

gram und dem Programmbezeichner AdamRiese keine Programmparameter, die in Klammern nach dem Programmbezeichner auftreten dürfen.

```
program AdamRiese;
var
 x, y, z: Integer;
begin
 Write('Eingeben x: '); Readln(x);
 Write('Eingeben y: '); Readln(y);
 z := x + y;
 Writeln('x plus y ist :', z);
 z := x * y;
 Writeln('x mal y ist :', z);
 Writeln('... und tschuess');
end.
```

Programmparameter hätten input und output sein können, womit angedeutet wird, daß das Programm Eingaben erwartet und Ausgaben liefert.

```
program AdamRiese(input, output);
```

Diese Programmparameterlisten sind aber heute nicht mehr üblich.

Man beachte ferner, daß ein Pascal-Programm mit einem Punkt endet. Dieser steht hinter dem Anweisungsteil, der mit **begin** ... **end** geklammert ist. Auf Anweisungen, also den operativen Teil des Programms, wollen wir später eingehen.

Vor dem Anweisungsteil steht der Vereinbarungsteil, in unserem Beispiel nur eine Variablendeklaration für x, y und z. Die in diesem Teil vereinbarten Marken, Konstanten, Typen, Variablen und Prozeduren/Funktionen gelten im folgenden Anweisungsteil, d.h. sind dort bekannt und können referenziert (angesprochen, verwendet) werden. Man spricht deshalb auch von einem Block (= Vereinbarungen + Anweisungen), der einen **Gültigkeitsbereich** definiert. Pascal gehört somit zu den sog. block-orientierten höheren Programmiersprachen. Dahinter verbirgt sich ein Konzept, wonach Vereinbarungen von Variablen usw. Gültigkeit haben im zugehörigen Anweisungsteil und in allen enthaltenen (Unter-)Blöcken, jedoch nicht in umgebenden Blöcken. Wir kommen darauf später zu sprechen, wenn wir Prozeduren und Funktionen vereinbaren.

Der Vereinbarungsteil eines Pascal-Programms kann teilweise oder auch ganz fehlen, wie am Beispiel des `Hallo-Leute`-Programms gesehen. Treten Marken-, Konstanten-, Typ-, Variablen- und/oder Funktions-/Prozedurvereinbarungen auf, dann immer in der Reihenfolge: **label**, **cons**, **type**, **var**. Der Grund dafür ist, daß damit die Übersetzung vereinfacht wird[1]:

- Bezeichner aus Konstantenvereinbarungen können in Typ- und Variablenvereinbarungen auftauchen,
- Typvereinbarungen können in Variablenvereinbarungen eingehen und
- alle Arten von Vereinbarungen können in Prozeduren oder Funktionen auftauchen.

3.2 Marken (Label)

Eine Marke (engl. *label*) ist ein Sprungziel für eine `goto`-Anweisung.

```
program EdsgarVergibMir;
label 99;
var
 anf,  {Anfangsposition}
 len: Integer;{Länge}
 wort: String(100); {Eingabewort - in TP ohne
Laengenangabe}
begin {Beispiel fuer goto}
99: Write('Das Eingabewort lautet: '); Readln(wort);
 Write('Position Teilwortanfang ist: '); Readln(anf);
 Write('Laenge Teilwort ist: '); Readln(len);
 if anf < 1
 then begin writeln('Pos. zu klein - nochmal'); goto 99
end
 else if anf > Length(wort)
   then begin writeln('Pos. zu gross - nochmal'); goto 99
end
   else if anf + len - 1 > Length(wort)
```

1. Der Pascal-Übersetzer ist tendenziell ein sog. 1-Paß-Übersetzer, d.h. er braucht nur einmal den Programmtext von links nach rechts zu lesen, um die Syntax zu analysieren und daraus den Zielcode, bzw. eine Zwischendarstellung zu generieren.

```
      then begin writeln('Teilwort nicht in Wort -
nochmal');
     goto 99
      end
      else writeln(SubStr(wort, anf, len));
   {in TP verwende Copy() statt SubStr() }
 writeln('Warum nicht gleich so?')
end {Beispiel fuer goto}.
```

Das Programm oben (`edsgar.pas`) gibt ein Teilwort einer Eingabe ab der Zeichenposition `anf` in der Länge `len` aus. Es verwendet den ursprünglich in Pascal nicht vorgesehenen Datentyp `string` für Zeichenketten. Zeichen darin werden von 1 an durchnumeriert, die Länge des momentanen Werts kann mit der Funktion `Length()` ermittelt werden. `SubStr()`, bzw. `Copy()` liefern das gewünschte Teilwort.

Eine mögliche Ausgabe lautet:

```
wegner@elsie(Sources)$ ./edsgar
Das Eingabewort lautet: kurz
Position Teilwortanfang ist: 17
Laenge Teilwort ist: 3
Position zu gross - nochmal
Das Eingabewort lautet: lang
Position Teilwortanfang ist: 2
Laenge Teilwort ist: 17
Teilwort nicht in Wort - nochmal
Das Eingabewort lautet: laenger und baenger
Position Teilwortanfang ist: 3
Laenge Teilwort ist: 7
enger u
Warum nicht gleich so?
wegner@elsie(Sources)$
```

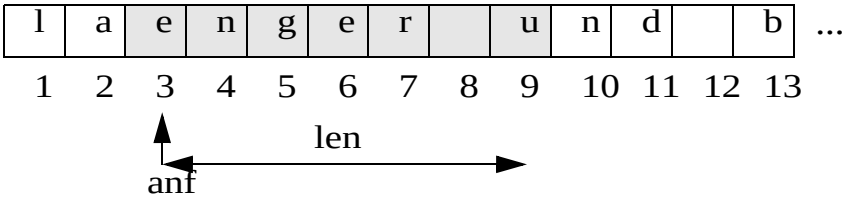


Abb. 3–1

Hier kommt es uns auf Marken (*label*) und ihren Gebrauch für Sprünge (`goto`-Anweisungen) an. Als Marken sind in Pascal ganze Zahlen vorgesehen (TP: max. vierstellig, also `0000` bis `9999`, TP auch Bezeichner als Marken). Im Beispiel vereinbaren wir eine solche Marke, nämlich `99`. Im Anweisungsteil tritt diese Marke dann vor einer Anweisung auf, getrennt durch Doppelpunkt (‘`:`’).

Damit wird eine Anweisung **markiert** und man kann durch `goto` zu dieser Stelle springen, d.h. der Kontrollablauf verzweigt von der Stelle der `goto`-Anweisung zu der durch die Marke angegebenen Anweisung und fährt dort fort.

In der sog. **strukturierten Programmierung** ist der Gebrauch von Marken und `goto`’s verpönt. Schaut man sich einen graphischen Ablaufplan eines solchen Programms an etwa den unseres Beispiels, erkennt

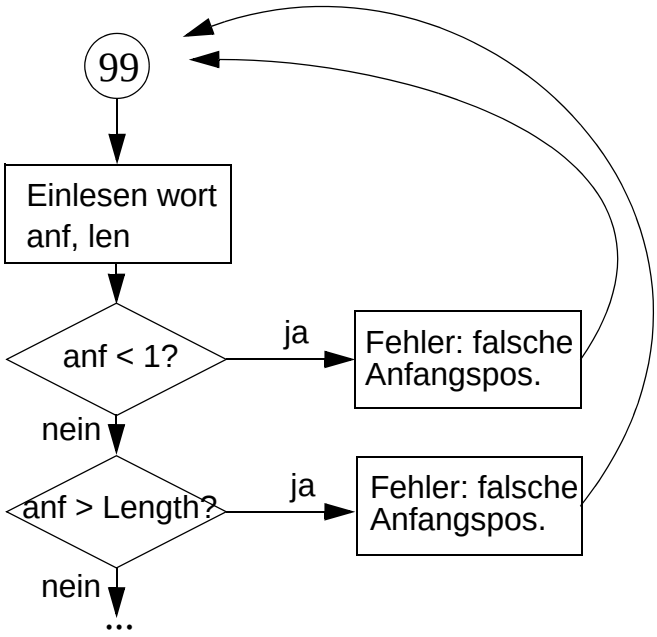


Abb. 3–2

man die etwas wirren Pfeile zum Sprungziel. Entsprechend heißen solche Programme „Spaghetti-Programme“, weil ihre unkontrollierten Sprünge ein undurchschaubares Gewirr von Kontrollflüssen erzeugen.

Seit Edsgar Dijkstras berühmten Artikel — „*Go To Statement Considered Harmful*“, Letter to the Editor, CACM 11 (1968) 147-148 — wird gern und heftig über Gefahren und Nutzen von `goto`’s gestritten. Die generelle Regel lautet: `goto`’s sind soweit wie möglich zu vermeiden, sie können nützlich sein in geschachtelten Strukturen, wenn diese in Fehlerfällen geordnet verlassen werden müssen.

Übung 3–1

Merken Sie sich die Aufgabe oben vor für eine Überarbeitung, wenn Sie genug über Schleifen und Boolsche Variablen wissen.

> **Anmerkung:** Wir weisen hier schon einmal vorsichtshalber darauf hin, daß eine Marke ihre Gültigkeit nur in dem Block hat, in dem sie vereinbart wurde. Wie wir später sehen werden, wenn wir Funktionen und Prozeduren behandelt haben, kann man nicht mittels `goto` zwischen verschiedenen Funktionen und Prozeduren hin- und herspringen. Auch ist es nicht möglich, in strukturierte Anweisungsteile hineinzuspringen, etwa in den Rumpf einer While-Schleife.

3.3 Konstantenvereinbarungen

Konstanten sind Werte, die innerhalb eines Programmlaufs unverändert bleiben. Ihnen kann man, wie Variablen, einen Bezeichner geben, unter dem die Werte angesprochen werden können.

```
const
  Normaljahr = 365;
  Schaltjahr = Normaljahr + 1;
  Seitengroesse = 4096;
  Segmentgroesse = Seitengroesse * 16;
  stern = '*';
  Leerzeichen = ' ';
  groesstezahl = maxint; {maxint ist Standardname}
  pi = 3.1415;
```

```
minuspi = -pi;{Reihenfolge!}
ja = true; {true: Standardnamen für Wahrheitswert}
nein = false;
Stopper = 'ZZZZZZZZ';
TextZeilen = 52;
ZeilenAufSeite = TextZeilen + 4;
```

Übung 3–2

Testen und erweitern Sie `konstant.pas`!

```
program Konstanten;
const
 pi    = 3.145; minuspi = -pi;
 Seitengroesse = 4096;
 HalbeSeite= Seitengroesse div 2;
 Segmentgroesse = Seitengroesse * 16;
 ... {weitere siehe oben} ...
type
 Seitentyp = array[0 .. Seitengroesse -1] of char;
begin {Beispiel fuer Konstanten}
 writeln(pi, ' ', minuspi, Leerzeichen, Stopper,
    Leerzeichen, Seitengroesse, Leerzeichen,
    Segmentgroesse, Leerzeichen, grosseZahl);
 writeln('... konstant veränderlich')
end {Beispiel fuer Konstanten}.
```

Übung 3–3

Einem Witz zufolge, dessen Bart dreimal um die Uni reicht, ist π im Einkauf 3 und im Verkauf 4. Versuchen Sie der Konstanten `pi` im Programm aus Übung einen dieser Werte zuzuweisen. Warum geht das nicht?

Als Regel sollte man sich angewöhnen:

\+ **Mache Konstanten konstant!**

Damit meint man, einen häufig auftretenden, konstanten Wert aus einem Programm herauszuziehen und ihn im const-Teil einmal zu vereinbaren. Liegen später Änderungen an, müssen diese nur an dieser einen Stelle

vorgenommen werden, z.B. wenn sich die Seitengröße verdoppelt oder wenn je nach Auslieferung unterschiedliche Werte einzusetzen sind.

3.4 Typvereinbarungen

Werden Variablen vereinbart, wie etwa `x`, `y` und `z` im Programm `adam.pas` unten,

```
program AdamRiese;
var
 x, y, z: Integer;
begin
 Write('Eingeben x: '); Readln(x);
 Write('Eingeben y: '); Readln(y);
 z := x + y;
 ...
end.
```

dann werden damit implizit und explizit eine Reihe von Festlegungen gemacht. Neben dem Namen der Variablen wird insbesondere ihr **Typ** festgelegt. Im Beispiel ist dies der Typ **Integer**, der in Pascal vordefiniert ist und für Variablen mit ganzzahligen Werten vorgesehen ist. Eine Typangabe bestimmt den **Wertebereich**, die zulässigen Operationen und deren Ergebnistyp. Implizit wird auch eine Speicherdarstellung festgelegt, die sich wiederum auf den Wertebereich auswirkt (Darstellung eines Integers in 2, 4 oder sogar 8 Bytes). Compiler und Linker/Lader errechnen zusätzlich noch einen Speicherplatz und belegen ihn ggf. mit einen Initialisierungswert. Im Ablauf des Programms nimmt der Speicherplatz dann den Wert auf, der momentan einer Variablen zugewiesen wurde.

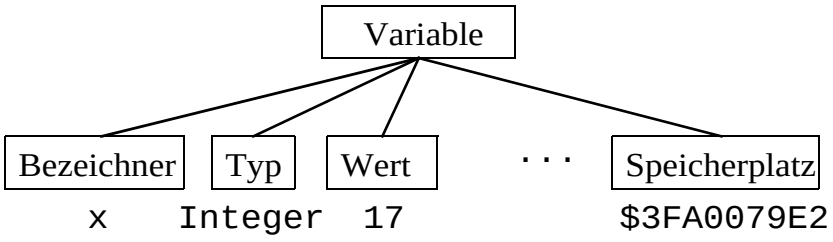


Abb. 3–3

Pascal sieht eine Reihe von vordefinierten einfachen Typen vor, daneben Konstruktoren für zusammengesetzte Typen. Zusätzlich können, aufbauend auf diesen vorgegebenen Typen, eigene Typen eingeführt werden. Dies geschieht mit der hier zu besprechenden Typvereinbarung.

Typvereinbarungen ::= **type** {Bezeichner **=** Typ **;**} [...]
Typ ::= einfacher-Typ | Zeigertyp | strukturierter-Typ |
String-Typ | Typ-Bezeichner
einfacher-Typ ::= ordinaler-Typbezeichner |
Real-Typbezeichner | Aufzählungstyp | Teilbereichstyp
Aufzählungstyp ::= **(** Bezeichner [**,** ...] **)**
Teilbereichstyp ::= Konstante .. Konstante

Zunächst sollen hier nur die einfachen Typen interessieren.

3.4.1 Ordinale Standardtypen

Darunter fallen die sog. ordinalen (aufzählbaren) Typen, für die als **Standardtypen** *Integer*, *Boolean* und *Char* definiert sind[1].

Ihre Wertebereiche sind offensichtlich

- die ganzen Zahlen (-32768 bis 32767)
- die Wahrheitswerte *true* und *false*
- die durch den Zeichensatz (ASCII Alphabet) definierten druckbaren Einzelzeichen.

Jedem Wert aus den Wertebereichen ist eine ganze Zahl, die sog. **Ordinalzahl**, zugewiesen. Sie kann mit der Standardfunktion `ord()` für jeden Wert ermittelt werden, wie im Programm unten gezeigt.

```
program Ordnung;
var
  x, y, z : Integer;
```

1. Häufig schreibt man *integer* und *char* statt *Integer* und *Char*, da Groß- und Kleinschreibung in Pascal bekanntlich Geschmacksache ist. Allerdings gibt es einen Trend, an der Großschreibung von *Boolean* festzuhalten, da dieser Bezeichner auf den Logiker *George Boole* (1815 - 1864) zurückgeht.

```
  ZeichenKleina, ZeichenGrossA, ZeichenNull : char;
begin
 x := 17;
 y := -17;
 ZeichenGrossA := 'A';
 ZeichenKleina := 'a';
 ZeichenNull := '0';
 writeln('x = ', x, ' ord(x) = ', ord(x));
 writeln('y = ', y, ' ord(y) = ', ord(y));
 writeln('ord(false) = ', ord(false));
 writeln('ord(A, a, Null) = ', ord(ZeichenGrossA),
   ord(ZeichenKleina), ord(ZeichenNull));
 Writeln('... Ordnung muss sein!');
end.
```

Sie weist dem kleinsten Wert, außer bei `Integer`, die Ordnungszahl 0 zu.

Übung 3–4

Finden Sie die hier gezeigte Ausgabe des Programms `ordnung.pas` in Ordnung?

```
./ordnung
x = 17 ord(x) = 17
y = -17 ord(y) = -17
ord(false) = 0
ord(A, a, Null) = 65 97 48
... Ordnung muss sein!
```

Übung 3–5

Besorgen Sie sich eine Tabelle des ASCII-Alphabets. Ist es eine 7-bit oder 8-bit Tabelle? Wo liegen die Großbuchstaben, wo die Kleinbuchstaben, die Umlaute, die Ziffern? Welche Zeichen heißen nichtdruckbar? Nennen Sie ein nützliches, „nichtdruckbares“ Zeichen? Na, klingelt’s oder sind Sie noch im Umbruch?

Für Argumente vom Typ `char` gibt es eine Umkehrfunktion zu *ord()*, die *chr()* heißt und für *chr(i)* den `char`-Wert liefert, dessen Ordnungszahl *i* ist. Speziell gilt

`chr(ord(`*c*`)) = `*c* ∀ `char`-Werte *c*, und
`ord(chr(`*i*`)) = `*i* ∀ ganzen Zahlen *i*, $0 \le i \le 255$

Über die Ordinalzahlen sind auch **Vorgänger** und **Nachfolger** eines Werts, mit Ausnahme für den ersten bzw. letzten Wert aus dem Wertebereich, definiert. Die entsprechenden Funktionen heißen *pred()* und *succ()* in Anlehnung an die englischen Begriffe *predecessor* und *successor*.

So gilt z.B. `false` < `true` und demnach

```
pred(true) = false und
succ(false) = true,
pred(false) und succ(true) undefiniert.
```

Schon hier sei ausdrücklich darauf hingewiesen, daß anders als bei dem ordinalen Typ `Integer`, die Funktionen `succ` und `pred` für Real-Typen (Gleitkommawerte, numerisch reele Zahlen) nicht definiert sind.

3.4.2 Der Standardtyp Integer

Dafür gibt es bei den ganzen Zahlen ein anderes, sehr lästiges Problem: die Größe des Wertebereichs.

Wie oben angedeutet, ist für die meisten Compiler, spez. auch für TP, beim Wertebereich von *Integer* die Spanne -32768 bis 32767 angegeben. Offensichtlich deutet dies auf eine interne Darstellung mit 16 bit (2^{15} = 32768) hin. Dem liegt aber eine veraltete Technik zugrunde, die in der PC-Welt einen Integer in ein sog. **Wort** speichert und ein Wort mit zwei Bytes assoziiert.

Heutzutage wird aber auch bei PCs mit 32-bit-Technologie gearbeitet, d.h. die Hauptspeicher sind zwar noch byteadressierbar, Zugriff und Verarbeitung, Registerlängen usw. sind aber an 32 bit ausgerichtet, dem traditionellen Großrechnerwort. Z.T. gibt es bereits einen Übergang auf 64 bit Speicherworte, spez. wenn Integer als Adressen gehandelt werden (2^{32} = $2^{10} * 2^{10} * 2^{10} * 2^{2}$ = 1024 * 1024 * 1024 * 4 = 4 Giga...).

Übung 3–6

Der Fahrweg für den Transrapid soll mindestens 6,9 Milliarden DM kosten. Paßt der Preis in DM in einen 32-Bit Integer?

Turbo Pascal 6.0 definiert demnach vier weitere ordinale Zahlentypen: *ShortInt*, *LongInt*, *Byte* und *Word*.

Typ	von	bis	Format
ShortInt	-128	127	8 Bit mit Vorzeichen
Integer	-32768	32767	16 Bit mit Vorzeichen
LongInt	-2147483648	2147483647	32 Bit mit Vorzeichen
Byte	0	255	8 Bit vorzeichenlos
Word	0	65535	16 Bitvorzeichenlos

Tab. 3–1 *Vordefinierte ganzzahlige Typen*

Der GNU Pascal Compiler dagegen definiert Integer vernünftigerweise als 32 bit Wert und kennt einen Standardtyp `LongInt` garnicht. Welche Größe die Integer haben, stellt man z.B. fest, indem man sich die vordefinierte Konstante `maxint` ausgeben läßt (vgl. Programm `ints.pas`).

```
program Ordnung;
type
 int = Integer;
var
 x, y : int;
begin
 x := 17;
 y := -17;
 writeln(x, y);
 writeln(maxint, -maxint);
end.
```

Die Ausgabe lautet:

```
$ ./ints
        17       -17
2147483647-2147483647
$
```

Übung 3–7

Interpretieren Sie die Ausgabe für `-maxint` oben! Stimmt Sie mit dem Wertebereich überein? Kann man überhaupt -2147483648 am Rechner eingeben?

Das Programm deutet einen Ausweg für dieses Problem an, das im übrigen nicht auf Pascal beschränkt ist, sondern genauso auch die Portabilität von C-Programmen plagt. Der hier vorgeschlagene Ausweg besteht darin, einen Datentyp `int` zu definieren (mittels Typvereinbarung) und diesem den in der jeweiligen Compilerversion gültigen 32-bit Integerbezeichner zuzuordnen. Im Beispiel oben ist dies `Integer`, da der GNU-Compiler darunter in UNIX einen 32 bit Integer versteht, auf einer älteren PC-Version würde man ggf. `LongInt` wählen. Im folgenden arbeitet man dann immer mit dem selbstdefiniertenTyp `int`.[1]

Dies zeigt eine Verwendung eines eigenen Typbezeichners mittels einer **type**-Vereinbarung. Nützlicher sind in der Regel die sog. **Ausschnitts-** und **Aufzählungstypen**.

3.4.3 Ausschnittstypen

Mit Ausschnitts oder Teilbereichstypen können wir neue, begrenzte Wertebereiche über ordinalen Typen (also nicht über `real`!) definieren, die z.T. zur Übersetzungszeit, z.T. zur Laufzeit automatisch auf Gültigkeit überprüft werden.

```
type
  Grossbuchstabe = 'A' .. 'Z';
  Wochentagstyp = 1 .. 7;
  Tag_im_Monat = 1 .. 31;
  Monat_im_Jahr = 1 .. 12;
  Jahreszahl = 0 .. 9999; {oder 0 .. 99 aus Sparsamkeit?}
  Index = 0 .. 99;
```

1. Die Verwendung des richtigen Standardbezeichners läßt sich zur Übersetzungszeit in Abhängigkeit definierter Bezeichner mittels ifdef-Direktiven in C-Manier angeben. Wir verzichten hier auf diese Methoden.

Übung 3–8

Wie kann man die Begriffe „zur Übersetzungs-zeit“ und „zur Laufzeit“ an den folgenden - fehlerhaften! - Beispielen (Programm `typen.pas`) erklären?

```
type
   ... wie oben ...
var
   Anfangstag, Endtag, Tag: Tag_im_Monat;
   Dauer: Integer;
...
begin
   Anfangstag := 0; Endtag := 0;{Initialisierung}
   write('Anfangstag (1 - 31) eingeben: ');
   readln(Anfangstag);
   write('Endtag (1 - 31) eingeben: '); readln(Endtag);
   Tag := Anfangstag; Dauer := 0;
   repeat
      Dauer := Dauer + 1; Tag := Tag + 1
      ...
   until Tag > Endtag;
   writeln('Projektdauer = ', Dauer);
   ...
end.
```

Übung 3–9

Nehmen wir an, der Compiler habe dafür gesorgt, daß die Eingabe für `Anfangstag` im Programm oben auf Gültigkeit, also Einhaltung der Bereichsgrenzen 1 bis 31, geprüft wird. Gibt jemand als `Anfangstag` z.B. die Zahl 45 ein, „macht das Programm den Abflug“, wie man salopp sagt. Ersetzt dies demnach eine selbstgeschriebene Eingabeprüfung der folgenden Art?

```
wiederhole
   lies Eingabezahl t vom Typ integer;
   wenn 1 <= t <= 31
      dann tag := t {tag eingeschränkter Typ}
      sonst ausgeben “falsche Eingabe <1 oder >31 -
nochmal“
bis Eingabe gültig.
```

Zu einem Ausschnittstyp gehört im übrigen immer automatisch ein sog. **Grundtyp**, der sich aus den Konstantenangaben ergibt. Für `Grossbuchstaben = 'A' .. 'Z'` also Grundtyp `Char`, für `Wochentag = 1 .. 7` demnach `Integer`. Der Grundtyp bestimmt, welche Operationen gültig sind, d.h. daß z.B. für eine Variable `t` vom Typ `Wochentag` die Addition definiert ist, usw.

3.4.4 Aufzählungstypen

Der zweite nützliche Typ sind die Aufzähltypen. Mit dieser Typvereinbarung werden eine Folge von Konstantenbezeichnern definiert, deren kleinster (erster) Wert die Ordinalzahl 0 hat.

```
type
  Wochentag = (Mo, Di, Mi, Don, Fr, Sa, So);{nicht Do!}
  Spielfarbe = (Karo, Herz, Pik, Kreuz);
  Pateien = (CDU, FDP, SPD);
  Farben = (rot, gruen, gelb, blau, schwarz);
  logo = (falsch, wahr);
```

Falsch wäre dagegen die Vereinbarung,

```
type
  Wochentag = ('Mo', 'Di', 'Mi', 'Do', 'Fr', 'Sa', 'So');
  TaginWoche = (1, 2, 3, 4, 5, 6, 7);
```

denn in beiden treten keine gültigen Bezeichner auf, sondern Literale.

Bereichstypen können über Aufzählungstypen (*enumerated types*) definiert werden:

```
type
  Wochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);
    {die zweibuchstabige Abkuerzung scheitert an Do!}
  Werktag = Mon .. Fre;
var
  Maloche : Werktag;
  Trumpf: Spielfarbe;
  ...
begin
  Trumpf := Herz;{Herz sticht immer}
  Maloche := Son;{Gibt Fehler bei vernünftigem Compiler}
```

```
  ...
end.
```

Für Aufzählungstypen, die ja ordinale Typen sind, ist *ord(), succ(), pred()* definiert. Genauso funktionieren Vergleiche, also z.B.

```
if Eintag < Fre then ...
```

wenn `Eintag` eine Variable vom oben definierten Typ `Wochentag`, bzw. `Werktag` ist. Werte eines Aufzählungstyps haben aber keine nach außen sichtbare Darstellung, können also nicht gelesen oder geschrieben werden.

Übung 3–10

Testen Sie das Programm `enum.pas`!

```
program Ausgezaehlt;
type
 Wochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);
 Werktag = Mon .. Fre;

var
 Tag, Heute : Wochentag;
 Malochentag : Werktag;

begin
 Tag := Mon;
 writeln('Montag hat Ordinalzahl ', ord(Tag));
 Malochentag := Son; {muss knallen - was sagt die
blinde GNU?}
 writeln('heute ist ', Tag); {muesste Fehler geben}
 readln('Tag eingeben (Mon, Die, ..., Son): ', Heute);
 {darf auch nicht gehen!}
 writeln('... und tschuess');
end.
```

Durch überlegten Einsatz von Aufzählungs- und Ausschnittstypen lassen sich Programme klarer gestalten als wenn man „zu Fuß“ eine Abbildung auf die ganzen Zahlen vornimmt.

+ Typbezeichner kann man mit dem Suffix ~typ enden lassen, z.B. Tag_im_Monat_Typ, Indextyp, Pateientyp, usw. Das macht klarer,

daß dieser Bezeichner ein Typbezeichner ist, klingt aber umständlich - Geschmacksache.

Übung 3–11

Kann man Ausschnittstypen auch „absteigend“ definieren, also z.B. `abi = 0 .. -9999`?

3.4.5 Der Standardtyp Boolean

Der Standardtyp Boolean hat Ähnlichkeit mit einem Aufzählungstyp mit den beiden Werten `true` und `false`. Allerdings können diese Namen z.B. ausgegeben werden (vgl. Programm `logo.pas`). Mit Variablen und Konstanten vom Typ Boolean und logischen Verknüpfungen (und, oder, Negation, ...) lassen sich Boolsche Ausdrücke bilden, etwa in der Art wie im Programm `wahrhaft.pas` unten.

```
program Wahrhaftig;
var
  Eingabefehler : Boolean;
  GTag, GMonat, GJahr : Integer;
  ...
begin
  ...
  repeat
    Eingabefehler := false;
    write('Geburtstag eingeben (1 - 31): ',
readln(GTag);
    Eingabefehler := (GTag < 1) or (GTag > 31);
    if not Eingabefehler
    then begin {Monat eingeben} ... end;
    ...
    if Eingabefehler
    then writeln('Eingabefehler - nochmal bitte!');
  until not Eingabefehler;
  ...
end {Eingabefehler Testen}.
```

Auf Ausdrücke gehen wir später noch gründlicher ein.

Übung 3–12

Ergänzen Sie das Programm oben um eine (zunächst einfache) Prüfung für die Eingabe von Monat und Jahr . Machen Sie später eine bessere Prüfung, spez. für 29. Februar! Ist der 29. Februar 2000 zulässig?

In der Programmiersprache C gibt es keinen Datentyp für Wahrheitswerte. Vielmehr wird die Null als Wert für `false` definiert und alle ganzzahligen Werte ungleich Null als `true`. Wie schon oben angedeutet, macht eine solche Umcodierung ein Programm nicht leserlicher. In Pascal wäre der Nichtgebrauch des Typs `Boolean` geradezu ein Frevel.

3.4.6 Der Standardtyp Char

Dieser Typ enthält Einzelzeichen in dem Umfang und der Ordnung wie sie durch das Basisalphabet, in der Regel ein 8-Bit ASCII Zeichensatz, vorgegeben ist. Die Funktionen `Ord()` und `Chr()` wurden bereits auf Seite 28 erwähnt.

Im Zusammenhang mit dem nachfolgenden Typ `String`, der nicht im ursprünglichen Pascal Standard vorgesehen war, sei erwähnt, daß eine Folge von mehr als einem Zeichen, also z.B. `'Hallo'` in `gruss := 'Hallo'`, kein Wert vom Typ `Char` sondern vom Typ `String` ist. Demnach sind `Char` und `String` wohlverschieden, wobei man `Char` als Sonderfall von `String` mit einem Zeichen auffassen kann und Zuweisungen in diese Richtung erlaubt (sog. automatische **Typerweiterung**), die Gegenrichtung aber verbieten sollte, auch wenn ein String zufällig oder per Vereinbarung die Länge 1 hat.

Übung 3–13

Was geht, was nicht?

```
program ZeichenSetzen;
var
 Zeichen : Char;
 ZKette : String(10);
begin
 ZKette := 'ABC';
```

```
    Zeichen := ZKette;{sollte nicht gehen, selbst wenn}
                 {Zeichenkette nur 1 Zeichen lang}
  end {ZeichenSetzen}.
```

Manche Programmierer setzen den Typ `Char` gleich einem **Byte**, bzw. einem vorzeichenlosen Wert zwischen 0 und 255. Das ist eine schlechte Gewohnheit, die etliche Gefahren birgt, und leider auch in C-Kreisen weitverbreitet ist:

- häufig sind sog. **Unicodes** im Gebrauch mit 16 und mehr Bit je Einzelzeichen
- genauso kann ein `Char` Wert auf eine vorzeichenbehaftete Zahl zwischen -128 und +127 abgebildet werden.

+ Ein defensiver Programmierer setzt keine interne Darstellungsform (Speicherzellengröße, Ausrichtung, Codierung) für einen Datentyp einer höheren Programmier-sprache voraus.

3.4.7 Der Typ String

Einer der nützlichsten Datentypen ist die Zeichenkette. Umso unverständlicher ist, daß N. Wirth ursprünglich für Pascal statt dem Typ `String` nur den zwar gleichwertigen, aber umständlichen Typ **`packed array[...] of char`** vorgesehen hatte. Inzwischen sehen fast alle Pascal-Übersetzer den Typ String mit einer optionalen Längenangabe vor.

```
type
  DosNamensTyp = String(12);
  AdressTyp = String(30);
  PLZ = String(5);
```

Intern werden Pascal-Strings meist so abgespeichert, daß der Zeichenkette ein „unsichtbares“ Byte vorangestellt wird, das die aktuelle Länge des Zeichenkettenwert speichert. Demnach ist die Länge auf 255 Bytes beschränkt. Diese Speicherform ist verschieden von C, wo das Ende eines Strings durch das ASCII-Zeichen NULL markiert ist (*nullterminated strings*). Wie oben angemerkt, wollen wir uns aber auf interne Speicherformen nicht verlassen.

Übung 3–14

Hat man längere Zeichenfolgen, muß man doch wieder auf die **`array[...] of char`** umsteigen. Manche kommerziellen Werkzeuge für Pascal bieten sog. *lange Strings* an, bei denen vorne zwei Bytes als Längenangabe vorgesehen sind. Wie groß dürfen deren Zeichenkettenwerte werden?

Turbo Pascal sieht für Strings die **Verkettungsoperation** vor und erlaubt als Operator das Pluszeichen (‘+’).

```
case Zielland do
Germany: Stadtzeile := 'D-'+Plznummer+Blank+Stadt;
USA:     Stadtzeile :=
Stadt+Blank+State+Blank+Postalnummer
...
end;
```

Daneben sind zahlreiche Funktionen für Zeichenketten definiert, darunter `Length()` zur Ermittlung der Länge, `Copy()` zur Teilstringermittlung (GNU: `SubStr`) und andere mehr.

Für Zeichenketten ist auch eine Ordnung definiert, die sog. **lexikographische Ordnung**. Sie besagt, daß eine Zeichenkette $A = a_1...a_n$ **vor** $B = b_1...b_m$ kommt, kurz $A < B$, genau dann wenn entweder

1. $a_i = b_i \; \forall \; i$, $1 \le i \le n$ und $m > n$, oder
2. $\exists \; j \le m$, n so daß $a_i = b_i \; \forall \; i$, $1 \le i \le j$-1 und $a_j < b_j$.

Im Fall (1) ist A ein Präfix (Anfangsstück) von B, im Fall (2) sind die Anfangsstücke ggf gleich, beim ersten unterschiedlichen Zeichen kommt aber das Zeichen an Position j in A im Basisalphabet vor dem entsprechenden Zeichen in B. Innerhalb dieser Ordnung ist die **leere Zeichenfolge** die kleinste mögliche.

Beliebtes Gesellschaftsspiel für verregnete Sonntagnachmittage sind auch Zeichenketten mit Anführungszeichen darin.

String ::= **'** String-Zeichen [...] **'**
String-Zeichen ::= alle-Zeichen-außer-Hochkomma | **''**

Man erreicht dies durch Verdopplung des Hochkommas.

Übung 3–15

Testen Sie `hamlet.pas` mit den folgenden Zuweisungen.

```
Frage := 'Wie geht's?';
ToBeOrNotToBe := 'Shakespeare''s ''''Hamlet''''';
SomethingRotten := 'Shakespeare''s "Hamlet"';
```

3.4.8 Der nicht-ordinale Typ Real

Über die Literale von numerisch-reelen Zahlen mittels Gleitpunkt- und Exponentialdarstellung haben wir bereits früher gesprochen. Für Typvereinbarungen sieht Standard Pascal den Standardnamen Real vor. Der Wertebereich dieses Typs ist offensichtlich nicht abzählbar, da sich theoretisch zwischen je zwei angegebene Werte ein weiterer schieben läßt. Praktisch hängt dies bei den numerisch reelen Zahlen aber von der internen Darstellung ab, die wiederum nicht festgelegt ist.

So bietet Turbo Pascal 6.0 Gleitkommazahlen in vier Geschmacksrichtungen an.

Typ	Bereich	Genauigkeit
Real	$2.9 * 10^{-39}$ bis $1.7 * 10^{38}$	11-12 Stellen
Single	$1.5 * 10^{-45}$ bis $3.4 * 10^{38}$	7-8 Stellen
Double	$5.0 * 10^{-324}$ bis $1.7 * 10^{308}$	15-16 Stellen
Extended	$3.4 * 10^{-493}$ 2 bis $1.1 * 10^{4932}$	19 - 20 Stellen

Tab. 3–2 *Turbo Pascal 6.0 Gleitkommatypen*

Der Hintergrund dieser Vielfalt waren die traditionellen Gleitkommadarstellungen (Mantisse und Exponent und Vorzeichen) in 32 bit und 64 bit. Diese sind aber z.T. veraltet. Seit einigen Jahren gibt es einen sog. IEEE Gleitkommastandard (siehe Vorlesung GdI II), der z.B. eine Darstellung mit mindestens 64 bit vorsieht. Zum zweiten ist heute praktisch jeder Rechner mit einer Hardwareeinheit für Gleitkommarechnungen (*floating*

point unit) ausgestattet, die dem IEEE Standard (vgl. Typ Double oben) genügt. Deshalb kennt auch der GNU Compiler nur den Typ `Real` und stellt diese intern mit der Genauigkeit von Double oben dar.

Gleitkommazahlen haben eine Reihe weiterer Besonderheiten, wie Überlauf, Unterlauf (Auslöschung), Zahlensprünge, Konvertierfehler, auf die wir hier nicht eingehen wollen.

Übung 3–16

Prüfen Sie, ob es eine Konstante maxreal oder ähnliches gibt! Wenn ja, welche Größe hat sie?

Wichtig sind aber zwei Einsichten:
Ein Integer kann immer zu einem Real erweitert werden. Die Zuweisung `x := 3` für `x` ein `Real` ist zulässig.

Ist `y` dagegen als `Integer` vereinbart, so liefert `y := 3.0` einen Laufzeitfehler (incompatible types in assignment).

Den für die kommerzielle DV sehr nützlichen Typ der Festkommazahl, z.B. mit genau zwei Nachkommastellen für Euro und Cent, gibt es in Pascal nicht. Intern wird dieser Wertebereich meist als BCD-Zahl mit je 4 Bit je Dezimalzahl gespeichert und auch so von FP-Einheiten unterstützt.

Einziger Ausweg in Pascal ist eine Integerrepräsentation, z.B. einen Typ Cents = Integer, bzw. Pfennige = Integer, zu vereinbaren und alle Berechnungen in Cents auszuführen. Bei der Ausgabe ist immer darauf zu achten, daß ein Dezimalkomma an der richtigen Stelle eingefügt wird.

3.4.9 Strukturierte, Zeiger- und Prozedurtypen

Die bisherigen Typen und Typvereinbarungen bezogen sich auf unstrukturierte (atomare) Typen. Daneben gibt es noch zwei strukturierte Typen: **Felder** (Arrays) und **Verbunde** (Records).

Erstere sind Aggregationen gleichen Typs, z.B. ein Vektor von ganzen Zahlen:

```
type
Niederschlag = Integer;
Regentyp = array[1..12] of Niederschlag;
```

Der zweite Typ ist eine Aggregation potentiell inhomogener Werte und ist die Programmiersprachenkonstruktion zur Aufnahme sog. **Datensätze**, die ggf. auf Platte gespeichert werden.

```
type
  Jahresdaten = record
     Jahr: integer;{4-stellige Angabe}
     Regen: Regentyp;{Monatsdurchschnitte}
     Sonnendauer: Sonnentyp;
     Temperatur: Temperaturtyp
  end {Jahresdaten};
```

Wir gehen auf diese Typen später nochmals im Detail ein.

Genauso wollen wir Zeiger auf Variablen, die als Werte Adressen aufnehmen, Mengentypen, Dateitypen und die Vereinbarung von Prozedur- und Funktionstypen später erst behandeln.

3.5 Variablenvereinbarungen

Mit den obigen Typvereinbarungen können wir neue Bezeichner für speziell konstruierte Wertebereiche vergeben. Mit Variablenvereinbarungen können wir Objekte über diesen Wertebereichen deklarieren, d.h. sie unter einem frei wählbaren Bezeichner verfügbar machen, um Werte zu speichern und abfragen zu können.

Die in einem Block vereinbarten Variablen sind in diesem Block und in allen darinliegenden Blöcken gültig. Variablen, die in dem `var`-Teil eines Programms (oder außen in einer Unit, hier bisher nicht behandelt) auftreten, heißen **globale Variable**. Eine Variable kann durch eine erneute (**lokale**) Vereinbarung in einem inneren Block verdeckt werden. Jeder Bezug auf diesen Bezeichner meint dann die lokale Variable. Die globale bleibt dabei weiterhin bestehen und ändert auch nicht ihren Wert.

Wir werden dies im Zusammenhang mit Prozeduren und Funktionen später kennenlernen.

Beispiel:

```
var
  i: Integer;
  j: Integer;
  x: Real;
  y: Real;
  z: Real;
  Ziffer: 0..9;
  Anfangsbuchstabe: Char;
  Richtung = (links, rechts, oben, unten);
```

kürzer:

```
var
  i, j: Integer;
  x, y, z: Real;
  ...
```

Man beachte, daß jede Programmvariable genau einmal vereinbart sein muß.

Turbo Pascal erlaubt ferner typisierte Konstanten. Sie werden in einer const-Vereinbarung nach der type-Vereinbarung aufgelistet. Da sie auch auf der linken Seite einer Zuweisung auftauchen dürfen, handelt es sich im Prinzip um **initialisierte Variablen**. Wir verzichten hier auf ihre Besprechung, weisen aber darauf hin, daß ein Pascal-Programmierer seine Variablen selbst initialisieren muß.

Übung 3–17

Was leistet das folgende Programm mit Namen `summe.pas`? Welche Variablen werden initialisiert. Warum wird die Variable `wert` nicht initialisiert?

```
program Summe;
var
 summe, i, wert: Integer;
begin
 summe := 0;
 i := 1;
 writeln('Bitte Werte eingeben, Abbruch mit Wert <=
0');
 write(i,'.ter Wert: '); readln(wert);
 while wert > 0 do
 begin
  summe := summe + wert;
  i := i + 1;
  write(i,'.ter Wert: '); readln(wert);
 end;
 writeln('Anzahl der Eingabewerte groesser Null: ', i-
1);
 writeln('Summe der Werte ist: ', summe);
end.
```

Übung 3–18

Geben Sie auch den Durchschnitt der eingelesenen Werte aus!

3.6 Prozedur- und Funktionenvereinbarungen

Die Vereinbarung von Prozeduren und Funktionen stellen wir zurück, bis wir die elementaren Ablaufstrukturen kennen.

3.7 Sprechende Bezeichner

\+ **Geeignete Bezeichner sind der halbe Kommentar.**

Die Kunst besteht darin, kurze, aber aussagekräftige Bezeichner zu wählen. Wie man diese schreibt, z.B. mit Groß- und Kleinbuchstaben (`DurchschnittsEinkommenProJahr`, `DEPJ`, `AbleseWert`), mit eingestreuten Unterstrichen (`Durchschnitts_einkommen_pro_jahr`, `depj`, `ablese_wert`), usw. ist Geschmackssache.

Zu lang gewählte Bezeichner machen ein Programm tendenziell unübersichtlich, weil häufig Zeilen umgebrochen werden müssen oder schlecht eingerückt sind. Ferner können Permutationen auftreten: `EinkommenProJahr`, `JahresEinkommen`, `EinkommenImJahr`, usw. Der Übersetzer findet zwar fehlende Vereinbarungen, bei langen Programmen kann in der Hektik aber auch einmal eine zweite Vereinbarung mit undurchsichtigen Folgen eingefügt werden!

Einer Konvention in FORTRAN folgend sind Bezeichner `i`, `j`, `k` für ganze Zahlen gebräuchlich, häufig für sog. Laufvariablen (Indexwerte). Bezeichner `x`, `y`, `z` deuten auf Real-Variablen. `S` steht häufig für Summe, `E` (error) oder `ENr`, `ENo` könne eine Fehlernummer sein.

Im Englischen kann man gut durch Weglassen von Vokalen kurze mnemonische Namen bilden, im Deutschen ist ein Prgrm mt Krzbz wngr lsbr, ggf vllg unvrstndl.

4 Zuweisungen und Ausdrücke

4.1 Zuweisungen und Zuweisungskompatibilität

Untersuchungen an Programmen verschiedener prozeduraler Sprachen zeigen, daß die weitaus häufigste Anweisung in einem Programm die Zuweisung ist, mit der eine Variable auf der linken Seite mit einem neuen Wert, der sich aus dem Ausdruck auf der rechten Seite ergibt, belegt wird.

Zuweisung ::= Variablenbezug `:=` Ausdruck |
Funktionsbezeichner `:=` Ausdruck

Der Zuweisungsoperator '`:=`' macht auch klar, daß es sich dabei nicht um eine Gleichung handelt, was z.B. bei

```
x := x + 1
```

auch sinnlos wäre.

Auf der linken Seite der Zuweisung wird in der Regel eine einfache Variable stehen. Daneben könnte auch ein Feldelement, z.B. `Sonnendauer[November] := 0`, oder eine Komponente eines Verbunds, z.B. `Beobachtungdaten.Jahr := 1998`, stehen.

Darauf, und auf die Möglichkeit, durch eine Zuweisung den Rückgabewert einer Funktion zu bestimmen, gehen wir später ein.

Auf der rechten Seite des Zuweisungsoperators steht ein Ausdruck, dessen Typ **zuweisungskompatibel** zum Typ der linken Seite sein muß. Zuweisungskompatibel heißt für eine Zuweisung, deren Zielobjekt den

Typ T_1 und deren Quellausdruck den Typ T_2 hat ($T_1 := T_2$), daß eine der folgenden Konstellationen gegeben sein muß:

- T_1 und T_2 sind gleich
- T_2 ist ein Teilbereich von T_1
- T_1 und T_2 sind Teilbereiche derselben Grundmenge und der Wert rechts liegt im Wertebereich von T_1
- T_1 ist *Real* und T_2 *Integer*
- T_1 ist *String* und T_2 *Char*.

Daneben gibt es Kompatibilitäten mit Prozedur-, Mengen- und Zeigertypen, die uns momentan nicht interessieren.

4.2 Ausdrücke, Operanden, Operatoren

Auf der rechten Seite einer Zuweisung können Ausdrücke der Art

```
(x * (y + 1) -2 <= trunc(sqrt(z))) or (z = 0) and not
ausnahme
```

stehen. Sie setzen sich zusammen aus **Operanden** und **Operatoren**. Operatoren können arithmetische, wie `*`, `+`, `-` , Vergleichsoperatoren wie `<=`, `=`, `>`, oder logische Operatoren wie **`or`**, **`and`**, **`not`** sein.Auch hier haben wir den Zwang zur Kompatibilität, denn Operanden müssen zueinander passen und können nur durch den passenden Operator miteinander verknüpft werden. Operatoren können dabei **unär** oder **binär** (dyadisch) sein, d.h. genau einen oder genau zwei Operanden verlangen. Ein unärer Operator ist z.B. **`not`**, die logische **Negation**. Binäre Operatoren, z.B. für Addition und Multiplikation, stehen in Pascal als **Infix-Operatoren** zwischen den Operanden (`a + b` statt z.B. `+ a b` oder `a b +`).

Die genaue Syntax läßt sich aus den Regeln unten ablesen.

Ausdruck ::= einfacher-Ausdruck
[{ < | <= | = | <> | >= | > | **`in`**} einfacher-Ausdruck]

einfacher-Ausdruck ::= [+ | -] Term
[{ + | - | **`or`** } Term] [...]

Term ::= Faktor [{ **`*`** | **`/`** | **`div`** | **`mod`** | **`and`** } Faktor] [...]

Faktor ::= vorzeichenlose-Konstante | Variable |
Mengenausdruck |
Funktionsbezeichner [**`(`** Ausdruck [**`,`** ...] **`)`**] |
`(` Ausdruck **`)`** | **`not`** Faktor

vorzeichenlose-Konstante ::= Konstantenbezeichner |
vorzeichenlose-Zahl | **`nil`** | Zeichenkettenliteral

Wie aus der Mathematik gewohnt, binden Operatoren unterschiedlich stark (Schulspruch: Punkt vor Strich). Die Prioritäten werden durch die Syntax oben definiert und in der folgenden Tabelle nochmals explizit aufgelistet.

Priorität	Operatoren
höchste	not
	* / div mod and
	+ - or
niedrigste	= <> > < >= <=

Tab. 4–1

Operatoren gleicher Priorität werden von links nach rechts abgearbeitet.

Die folgende Abbildung zeigt einen Ableitungsbaum für einen Ausdruck.

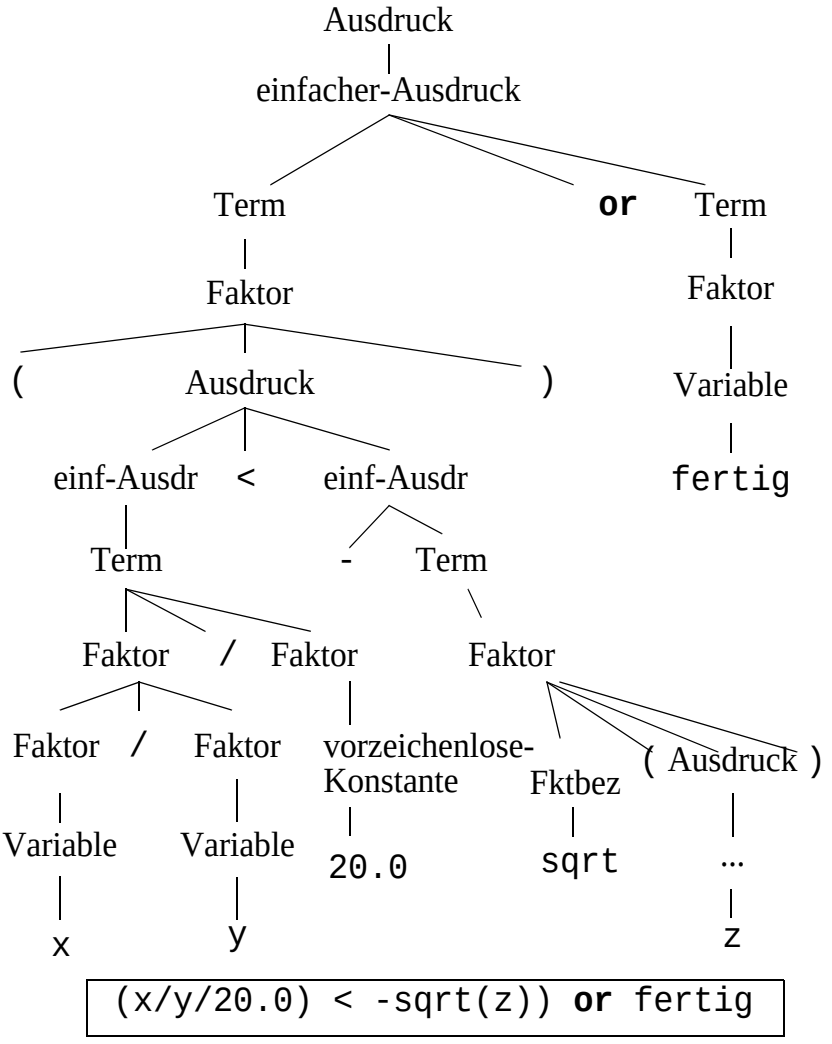


Abb. 4–1

Beispiele für Ausdrücke und ihre Werte sind:

```
1 + 2 + 10              13
1 + 2 * 10              21
(1 + 2) * 10            30
2 * 2 + 10 * 10         104
100.0 / (20.0 / 5.0)    25.0
100.0 / 20.0 / 5.0      1.0
(15 div 4 <= 10) or fertig and (a = 0)true
false and b             false
b or true               true
```

Bei mit **and** und **or** zusammengesetzten Ausdrücken schreibt der Pascal-Standard keine Abarbeitungsreihenfolge vor und gibt auch nicht an, ob abgebrochen wird, wenn das Ergebnis feststeht. Viele Implementierungen erlauben hierzu Angaben:

- **vollständige Berechung** wird zugesichert; wichtig bei sog. **Seiteneffekten** der Ausdrucksberechnung
- **Kurzschlußberechnung von links nach rechts** - sofortiger Abbruch zugesichert, wenn Ergebnis feststeht; wichtig wenn sonst bei Weiterberechnung ein Fehler aufträte.

Übung 4–1

Wieso ist eine Kurzschlußberechnung günstig im folgenden Fall?

```
var A: array[1..N] of Integer;
...
while (i <= N) and (A[i] <> 0) do
begin
   ...
   i := i + 1;
   ...
end;
```

Als Merkregel für die Bildung von Ausdrücken gilt, daß nie zwei arithmetische Operatoren zusammenstehen (also falsch: `a*-b`). Ferner hält man sich an den guten Rat der Boxer:
+ **wenn unsicher - klammere**.

Beliebt ist auch bei Anfängern, den Malpunkt wegzulassen.

Setzen Sie die folgenden mathematischen Formeln in gültige Pascal-Ausdrücke um!

$$\frac{\alpha + 2}{x + \frac{1}{2}} \qquad -\frac{xy}{-w + 2} \qquad \frac{b + \sqrt{b^2 - 4ac}}{2a} \qquad 0 < i \leq \phi$$

4.3 Typverträglichkeit und Ergebnistyp

Die Operatoren lassen sich nach arithmetischen, logischen, relationalen, String- und Mengenoperatoren unterscheiden.

Bei den binären arithmetischen Operatoren `+`, `-`, `*`, `/`, **`div`** und **`mod`** gilt:

- sind beide Operanden Integer, ist das Ergebnis auch vom Typ Integer, außer bei der „normalen“ Division, bei der die Operanden immer zu Real erweitert werden und das Ergebnis immer vom Typ Real ist.[1]
- ist einer der Operanden Integer, der andere Real, ist das Ergebnis Real.
- bei Real-Operanden kann unter Umständen `(x / y) * y` nicht wieder `x` ergeben
- **`div`** (ganzzahlige Division) und **`mod`** (Restbildung bei ganzzahliger Division) sind nur für Integer Operanden definiert.
- Division durch Null, sowohl für `/` als auch für **`div`** und **`mod`**, führt zu einem Laufzeitfehler (in der Regel Abbruch des Programms).
- Für Vorzeichen der Ergebnisse bei **`mod`** und **`div`** gilt:
  sind bei **`div`** beide Operanden positiv oder beide negativ, dann ist das Ergebnis positiv, sonst negativ; bei `i mod j` übernimmt in Turbo Pascal das Ergebnis das Vorzeichen von `i`, im Standard ist es immer positiv; im übrigen gilt `a = (a div b) * b + (a mod b)`

1. anders ausgedrückt: die „normale“ Division ist für Integer-Operanden nicht definiert.

- Standard Pascal kennt keinen Operator für Exponentiation, im erweiterten Standard gibt es den Operator '`**`', das Ergebnis ist immer ein Real (`2**4 = 1.600...e+01`)

Als Standardfunktionen, die Integer-Argumente verlangen und ein Integer-Resultat liefern, sind zu nennen:

abs	Absolutwert, auch für Real
sqr	Quadrat (square), auch für Real
succ	Nachfolger, generell für Ordinaltypen
pred	Vorgänger, ~

Beispiele für Integerausdrücke und deren Wert:

17 + 4	21
-2 - 8	-10
14 **div** 3	4
14 **mod** 3	2
5 **div** 6	0
-14 **div** 3	-4
abs(7)	7
abs(-7)	7
sqr(-7)	49
pred(7)	6
pred(-7)	-8

Beispiele für Ausdrücke mit Real-Operanden sind

2.1 + 1.4	3.5
2.1 - 1.4	0.7
2.1 * 1.4	2.94
2.1 / 1.4	1.5
abs(-6.4)	6.4
sqr(3.1)	9.61

Die Funktionen `succ` und `pred` sind bekanntlich für Real-Argumente nicht definiert.

Weitere Standardfunktionen sind

`ln(x)`	natürlicher Logarithmus von x, $x > 0.0$
`exp(x)`	e-Funktion (e^x)
`sqrt(x)`	Quadratwurzel (square root), $x \geq 0$
`sin(x)`	Sinusfunktion
`cos(x)`	...
`arctan(x)`	

Eine Reihe von Funktionen erlauben einen expliziten Übergang zwischen Real und Integer und werden deshalb als Transferfunktionen bezeichnet:

`trunc(x)`	Abschneiden von Dezimalstellen größte ganze Zahl $\leq x$ falls $x \geq 0$, kleinste ganze Zahl $\geq x$ falls $x < 0$
`round(x)`	Runden round(x) = trunc(x + 0.5) falls $x \geq 0$ round(x) = trunc(x - 0.5) falls $x < 0$

Die Funktionen `ord` und `chr` haben wir bereits behandelt.

Beispiele

trunc(4.3)	4
trunc(4.7)	4
trunc(-4.3)	-4
trunc(-4.7)	-4
round(4.3)	4
round(4.7)	5
round(4.5)	5
round(-4.3)	-4
round(-4.7)	-5

In logischen Ausdrücken werden die Wahrheitswerte wahr und falsch (darstellbar durch ihr Standardbezeichner `true` und `false`) mit **`and`**, **`or`**, **`not`** verknüpft. Zusätzlich definiert TP noch die Antivalenz **`xor`**

(exklusives Oder), sowie Links- und Rechtsverschieben um eine anzugebende Anzahl von Bits.

XOR	true	false
true	false	true
false	true	false

Tab. 4–2 *Exklusives Oder*

TP und der GNU-Compiler verstehen diese Operatoren auch als Bitoperatoren, d.h. sie sind bitweise auch auf Integer anzuwenden. So ergibt 8 xor 2 den Wert 10.

Die Verschiebefunktion `shl` und `shr` leiten sich entsprechend von Maschinenbefehlen (Register shift logical left, ~ right) her, bei denen Bitfolgen sehr effizient in Registern hin- und hergeschoben werden. Der Begriff „logisches“ Schieben kommt daher, daß bei dieser Form des „Shift“ Nullen nachgeschoben werden. Üblicherweise gibt es auf den Rechenanlagen noch „arithmetisches“ Schieben, bei dem ein negatives Vorzeichen erhalten bleibt.

In der Form `x shr 4` wird demnach eine Bitfolge, hier die Darstellung von `x`, 4 Positionen nach rechts geschoben. Ist `x` eine positive ganze Zahl, entspricht dies der Division von `x` durch 16. Da in DV-Aufgabenstellungen häufig mit Zweierpotenzen gerechnet werden muß, ist dies eine effiziente Manipulationsmethode, die aber vom Standard nicht abgedeckt wird und sehr maschinennah ist.

Funktionen, die beliebige Argumente nehmen und Wahrheitswerte abliefern heißen auch **Prädikate**. Ein Prädikat für Integerargumente ist `odd(x)`, das `true` liefert gdw. `x` ungerade ist.

Weitere Prädikate sind `eoln` und `eof`, die Zeilenende und Eingabeende signalisieren.

Hauptlieferant von Wahrheitswerten sind natürlich Vergleiche, die als Operanden Ausdrücke vom Typ Integer, Real, Boolean, Char, String,

Mengen, annehmen können. Die Vergleichsoperatoren (relationale Operatoren) sind:

=	ist gleich
<>	ist ungleich
<	ist kleiner als
>	ist größer als
<=	ist kleiner oder gleich (nicht größer als)
>=	ist größer oder gleich (nicht kleiner als)
in	Mengenzugehörigkeit

Für die bisher nicht behandelten Mengen sei hier im Vorgriff erläutert, daß *A* >= *B* testet, ob *A* Obermenge von *B* ist, analog <= die Untermengeneigenschaft feststellt. Für den relationalen Operator **in**, etwa *x* **in** *S*, muß der linke Operand *x* ein ordinaler Typ *T* sein und *S* eine Menge: *S*: **set of** *T*.

Stringausdrücke kennen neben einer Reihe von Stringfunktionen (`trim`, `substr`, `length`, ...) nur den Verkettungsoperator '+'. String-Vergleiche beruhen auf der oben genannten lexikographischen Ordnung. Auch auf die Verwendung von Werten vom Typ Char in der Rolle von String-Werten (aber nicht umgekehrt) wurde verwiesen.

Zuletzt wollen wir auf Mengenausdrücke eingehen und dabei diesen bisher überschlagenen Datentyp einführen.

4.4 Mengen

Pascal erlaubt als strukturierten Typ die Menge.

```
type
  Grossbuchstaben = set of 'A' .. 'Z';
var
  Grundmenge,
  Vokale,
  Konsonanten: Grossbuchstaben;
begin
  ...
  Vokale := ['A', 'E', 'I', 'O', 'U'];
  if 'Y' in Vokale then writeln('seltsam')
```

```
  ...
end.
```

Die Elemente müssen alle demselben Typ angehören und ihr Grundtyp muß ein Ordinaltyp sein. Die Kardinalität der Mengen ist begrenzt, in TP z.B. auf 256 Elemente.

Vieles zum Gebrauch von Mengen macht man sich leichter klar, wenn man berücksichtigt, daß Mengen als Bitvektoren abgespeichert werden, wobei ein Bitwert 0 andeutet „Element nicht vorhanden“ und 1 die Anwesenheit signalisiert.

Ausdrücke über Mengen kennen ‘+’ für die Vereinigung, ‘*’ für den Durchschnitt und ‘-’ für die Differenz. Mengenwertige Konstanten lassen sich angeben durch Auflistung der Elemente (mit Komma getrennt und in eckigen Klammern eingeschlossen) und durch Ausschnittsangabe des Grundtyps. Die leere Menge ist `[]`.

```
Grundmenge := ['A' .. 'Z'];
Vokale := ['A', 'E', 'I', 'O', 'U'];
Konsonanten := Grundmenge - Vokale;

if Konsonanten * Vokale <> []
then writeln('oberfaul')
...
```

Übung 4–2

Für die Verwaltung Ihres ausgedehnten Kellers von alten Bordeaux Weinen definieren Sie einen Aufzählungstyp `Jahrgang = 1800 .. 2055` und einen Mengentyp `Klasse` als **`set of`** `Jahrgang`. Vereinbaren Sie Mengenvariablen `Exzellent`, `Gut`, `Maessig` und weisen Sie den Variablen einige Elemente zu (1984 war ein sehr schwacher Jahrgang, 82 und 95 z.B. ausgezeichnet). Sprechen Sie eine Empfehlung für ein Jahr aus:

```
if Jahr in Exzellent
  then writeln('Kaufen!')
  else
    if Jahr in Gut
```

```
        then writeln('Pruefen!')
        else ...
```

Nicht möglich ist die Eingabe, Ausgabe und elementweise Verarbeitung von Mengen im Sinne von ($\forall x \in$S: ...). Recht nützlich sind Mengen über dem Grundtyp `Char`:

```
readln(ch);
if ch in ['a'..'z', '0'..'9'] then m := m + [ch];
```

Übung 4–3

Wenn ein Typ *b* Elemente enthält, z.B. *b* = 4 in

`Farbe = (Karo, Herz, Pik, Kreuz)` und eine Mengenvariable über diesem Typ definiert wird, wie groß ist dann der Wertebereich dieser Variablen? Wie groß wäre demnach der folgende Wertebereich?

```
MeineHand: set of Farbe
```

4.5 Typumwandlungen

Standard Pascal kennt eine begrenzte Anzahl von Transferfunktionen und einige implizite Typwandlungen, z.B. Integer erweitert auf Real oder Char auf String.

Turbo Pascal kennt auch explizite Wandlungen von Strings, wenn diese nur Ziffern oder Vorzeichen als Char-Elemente enthalten, in Integer oder Real-Werte (Prozedur `Val`) und die Umkehrung (Prozedur `Str`). Daneben gibt es in Turbo Pascal die sehr mächtige (und gefährliche) Form der expliziten Typumwandlung (*type casting, cast operation*).

Die allgemeine Form ist

Cast-Operation ::= Zieltypangabe (Quellwert)

Diese kann für maschinennahe Programmierung nützlich sein, etwa wenn man ein Feld von Bytes (z.B. eine Seite von 4KB) als ein Feld von 4-Byte Integer interpretieren will. Gewöhnlich sollte man solche Typumwand-

lungen aber meiden. Ganz schrecklich ist das folgende Programm (`cast.pas`) für TP.

```
program Cast; {Urheberschaft wuerde ich nie zugeben}
type
 S4Typ = String(3);{4 mit Längenbyte - au backe}
 Int = LongInt; {TP: hoffentlich in 4 Bytes}
var
 S : S4Typ;
 i : Int;
begin
 S := chr(0)+chr(0)+chr(65);{ASCII 65 ist 'A'}
 writeln(S, '****', Int(S)); {ziemlich grosser Integer}
end. {ganz uebler Hack!}
```

5 Anweisungen

Anweisungen lassen sich in einfache und strukturierte Anweisungen einteilen. Die wichtigste einfache Anweisung - die Zuweisung - haben wir bereits kennengelernt.

Vor jeder Anweisung kann, wie bereits oben angedeutet, eine Marke stehen (Standard: Zahl von 0 bis 9999, TP: Bezeichner), die laut Standard vorher vereinbart sein muß. Mittels einer goto-Anweisung, kann zu dieser so markierten Anweisung gesprungen werden.

Anweisung ::= [Marke :] { einfache-Anweisung |
strukturierte-Anweisung}

einfache-Anweisung ::= Zuweisung | Prozeduranweisung |
goto-Anweisung

goto-Anweisung ::= **goto** Marke

Prozeduranweisung ::= Prozedurbezeichner
[**(** Argumentliste **)**]

Zuweisungen haben wir im Kapitel 4 oben behandelt. Sprünge (goto-Anweisungen) und ihre Nachteile wurden in Abschnitt 3.2 angesprochen.

Prozeduranweisungen sind die Aufrufe von Prozeduren, wobei diese dann, sofern in der Prozedurvereinbarung so vorgesehen, aktuelle Parameter (Argumente) erhalten.

Beispiele wären

```
QSort(VektorA, i, j+1);
ggV(p, q, resultggV);
Strecke(InZeile, AusZeile, ZLaenge);
NeueZeile;
```

Auf die Details gehen wir ein, wenn wir Funktionen und Prozeduren erläutern.

Strukturierte Anweisungen umfassen mehrere einfache Anweisungen und erlauben die Konstruktion von Wiederholungen (Schleifen), von bedingten Anweisungen (if-then-else), usw.

strukturierte-Anweisung ::= Blockanweisung |
bedingte-Anweisung | Wiederholungsanweisung |
with-Anweisung

Blockanweisung ::= **begin** Anweisung [**;** ...] **end**

5.1 Der begin-end Block

Durch **begin** und **end** lassen sich Anweisungen klammern, die dann wie eine Anweisung in der angegebenen Reihenfolge ausgeführt werden. Diese Form der Zusammenfassung benötigt man z.B. in einer bedingten Anweisung, wenn im then- oder else-Teil mehr als eine Anweisung stehen soll.

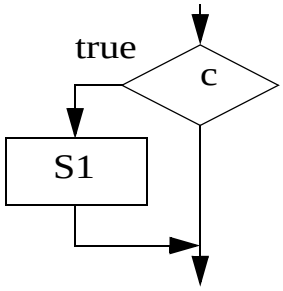
```
if a + b = 0
then begin writeln('Fehler: Division durch Null in
QSumme');
       writeln('Wert von a und b waren: ', a, b);
       AbraeumenInitialisierung(InitRecord, 317
{ECode});
       goto EAusgang
     end
else begin Q:= S div (a + b);
       writeln('Quersumme: ', Q)
     end;
```

5.2 Bedingte Anweisungen

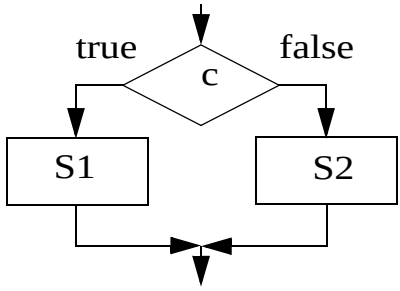
Pascal verfügt über zwei bedingte Anweisungen: die if-Anweisung und die case-Anweisung.

bedingte-Anweisung ::= **`if`** Ausdruck **`then`** Anweisung
[**`else`** Anweisung] |
`case` Ausdruck **`of`** [Konstante [**`,`** ...] **`:`**
Anweisung][**`;`** ...] **`end`**

Mit der if-Anweisung lassen sich demnach, je nach Ausgang des Tests, unterschiedliche Programmzweige durchlaufen, wobei der else-Teil auch ganz fehlen darf.



`if` c **`then`** S1



`if` c **`then`** S1 **`else`** S2

Abb. 5–1

Beispiel 5–1

```
if Wuerfel <> 6
then Brettposition := Brettposition + Wuerfel
{sonst einmal aussetzen}
```

Das folgende Programm **`veryodd.pas`** ist auch in den Quellen verfügbar.

```
program VeryOdd;
  {feststellen ob eine Zahl gerade ist
  (even or odd) ohne odd-Funktion -
  a biserl umstaendlich programmiert
  zwecks Didaktiktiktik ...}
```

```
type
   Int = Integer; {TP: LongInt}
var
   EZahl    : Int;
   ungerade : Boolean;
begin
   write('Zahl eingeben: '); readln(EZahl);
   if EZahl mod 2 = 0 {durch 2 teilbar ohne Rest}
   then ungerade := false
   else ungerade := true;
   write('Die Eingabe ', EZahl, 'ist ');
   if ungerade
      then writeln('ungerade')
      else writeln('gerade')
end {VeryOdd}.
```

Am Programmtext lassen sich auch die bewußt unterschiedlich gestalteten Einrückungsarten ablesen (Geschmackssache). Diese Einrückungen haben aber keinen Einfluß darauf, wie die Abarbeitung bei geschachtelten if-then-else erfolgt!

```
if c1
then  if c2
      then if c3 then s1 else s2
      else if c4 then s3 else s4
else s5
```

Die Abbildung unten zeigt den zugehörigen Ablaufplan.

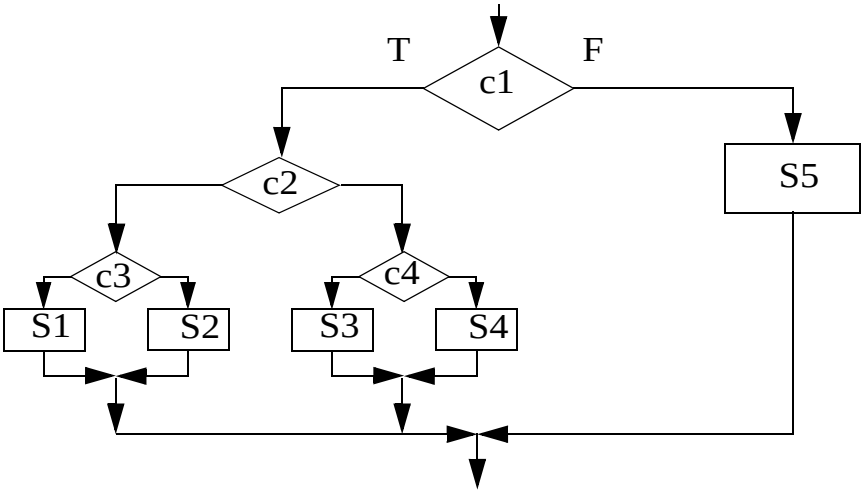


Abb. 5–2

Dieser ist noch recht einfach zu verstehen, weil jeder then-Teil einen zugehörigen else-Teil hat. Im Fall des sog. **hängenden else** (engl. *dangling else*) ist das aber nicht mehr der Fall.

Eine der beiden gezeigten Einrückungen suggestiert deshalb auch eine falsche Semantik!

```
if c1                {Einrückung (*)}
then if c2 then s1
else s2
if c1              {Einrückung (**)}
then if c2  then s1
            else s2
```

Hier gilt dieRegel: + ein else-Teil gehört immer zum letzten offenen then-Teil. Demnach sind beide obigen Programmteile äquivalent zu

```
if c1
then begin
       if c2
       then s1
       else s2
end
```

Wie man schon an diesem kleinen Beispiel sieht, werden Programme mit mehrfach gestuften bedingten Anweisungen leicht sehr unübersichtlich. Man geht dann ggf. lieber auf eine Art Fallunterscheidung mit einer case-Anweisung (siehe unten) über.

Zunächst aber nochmals ein Anwendungsbeispiel, bei dem wir uns bemühen, die Lösung durch fortschreitende Verfeinerung zu entwickeln. Die Aufgabe ist das sog. Knobelspiel, bei dem zwei Spieler je eine ganze Zahl ≥ 0 nennen. Gilt Zahl1 = Zahl2 dann ist das Spiel unentschieden. Im anderen Fall gilt: ist Zahl1 + Zahl2 gerade, dann gewinnt die kleinere Zahl, sonst die größere.

Grobentwurf

```
program Knobeln;
{Knobelspiel, vgl. Skript S. 72}
var
   {Vereinbarungen}
begin
   {Eingabe}
   if {Eingabe falsch}
      then {Meldung}
      else {Entscheidung}
end {Knobeln}.
```

1. Verfeinerung der Entscheidung

```
if Zahl1 = Zahl2
  then {unentschieden}
  else {ermittle Sieger}
```

2. Verfeinerung der Siegerermittlung

```
if Zahl1 + Zahl2 gerade
      then {kleinere Zahl siegt}
      else {größere Zahl siegt}
```

Das fertige Programm `knobel.pas` sieht dann wie folgt aus.

```
program Knobeln;
{Knobelspiel, vgl. Skript S. 72}
var   {Vereinbarungen}
```

```
   Zahl1, Zahl2:  Integer;
begin
   {Eingabe}
   write('Spieler 1 eine Zahl eingeben: ');
readln(Zahl1);
   write('Spieler 2 eine Zahl eingeben: ');
readln(Zahl2);
   if (Zahl1 < 0) or (Zahl2 < 0) {Eingabe falsch}
   then writeln('falsche Eingabe!')  {Meldung}
   else {Entscheidung}
      if Zahl1 = Zahl2
      then writeln('unentschieden')
      else {ermittle Sieger}
       if not odd(Zahl1 + Zahl2) {Summe gerade}
       then {kleinere Zahl siegt}
          if Zahl1 < Zahl2
          then writeln('1. Spieler siegt.')
          else writeln('2. Spieler siegt.')
       else {größere Zahl siegt}
          if Zahl1 > Zahl2
          then writeln('1. Spieler siegt.')
          else writeln('2. Spieler siegt.')
end {Knobeln}.
```

Ein weiteres Programm, das einen Geldautomaten simuliert, behandeln wir in den Übungen.

Die case-Anweisung entspricht einer Fallunterscheidung. Der nach dem Schlüsselwort **case** folgende *Ausdruck* - der sog. Selektor - wird ausgewertet und der resultierende Wert auf Übereinstimmung mit einer der *Fallmarken* (Konstanten zu den Alternativen) geprüft. Die Anweisung mit der passenden Konstanten wird abgearbeitet und danach wird die Programmausführung mit der ersten Anweisung, die dem Schlüsselwort **end** folgt, fortgesetzt.

```
case Wochentag of
  Mo, Di, Mi, Don, Fr: writeln('Werktag');
  Sa, So: writeln('Feiertag')
end
```

Für den Selektor wird ein Ordinaltyp gefordert, der sich auf Integer (TP: -32K bis +32K) abbilden läßt. Natürlich müssen die case-Konstanten vom

selben Typ wie der Selektor sein. Ferner darf eine Konstante als Marke nur in einer Anweisung auftauchen.

Gibt es für einen errechneten Selektorwert keine passende Fallmarke sieht der Pascal-Standard einen Fehler (Programmabbruch) vor. Der erweiterte Standard und Turbo Pascal dagegen erlauben eine Art Fehlerausgang, indem sie eine else-Alternative zulassen, wobei nach else auch mehrere Anweisungen stehen dürfen, also nicht extra mit begin-end geklammert werden müssen. Fehlt die else-Alternative und stimmt keine der Fallmarken mit dem Selektorwert überein, setzt TP die Abarbeitung nach **end** fort, d.h.. es wird kein Zweig der case-Anweisung ausgeführt und auch kein Fehler generiert.

```
case Punkte of
  0..4: begin writeln('nicht bestanden');
        writeln('Ruecksprache wegen muendl. Pruefung!')
        end;
  5, 6: writeln('ausreichend');
  7, 8: writeln('befriedigend');
  9: writeln('gut');
  10: writeln('sehr gut');
else
  writeln('Punktzahl ', Punkte,' groesser 10?');
  writeln('Auszeichnung oder Fehler!')
end
```

Semantisch ist demnach die case-Anweisung

```
case e of
k11, k12, ... : s1;
k21, k22, ... : s2;
...
kn1, kn2, ... : sn
end
```

äquivalent zu

```
if (e = k11) or (e = k12) or ...
then begin s1 end
else case e of
       k21, k22, ... : s2;
       ...
       kn1, kn2, ... : sn
     end
```

mit Abbruch über

```
if (e = kn1) or (e = kn2) or ...
then begin sn end
else {Fehlermeldung laut Standard - in TP: weiter}
```

Offensichtlich spielt die Reihenfolge der Alternativen in einer case-Anweisung keine Rolle. Ferner sei ausdrücklich darauf hingewiesen, daß in Pascal, anders als z.B. in C, **nur die passende** Alternative, nicht dagegen auch jede der danach folgenden Alternativen, abgearbeitet wird. Ein explizites **break** ist demnach nicht nötig und nicht vorgesehen.

5.3 Schleifen

Schleifen spielen, wie Zuweisungen und Abfragen, eine elementare Rolle in der algorithmischen Verarbeitung[1].

Wiederholungsanweisung ::= repeat-Schleife |
while-Schleife | for-Schleife

Wie man der Syntaxregel entnehmen kann, gibt es in Pascal davon drei Ausprägungen.

Die repeat-Schleife

Die Syntax der repeat-Schleife lautet:

repeat-Schleife ::= **repeat** Anweisung [**;** ...]
until Ausdruck

Die zwischen **repeat** und **until** stehenden Anweisungen werden solange ausgeführt, bis der Ausdruck wahr wird. Dieser muß ein Ergebnis vom Typ **Boolean** liefern.

Semantisch ist

```
repeat S until B
```

1. Die ersten programmierbaren Rechner von Zuse verwendeten gelochten 35mm Kinofilm als Programmspeicher. Schleifen wurden schlicht durch Zusammenkleben von Anfang und Ende realisiert!

äquivalent zu

```
begin
    S;
    if not B then repeat S until B
end
```

Damit ist klar, daß eine repeat-Schleife mindestens einmal durchlaufen wird.

Beispiel 5–2

Ulams Vermutung besagt, daß sich jede ganze Zahl $x > 0$ auf 1 reduzieren läßt durch Anwendung der folgenden Regel:

ist x gerade, setze $x := x/2$ sonst setze $x := 3x + 1$.

Beispielfolge: 12, 6, 3, 10, 5, 16, ..., 1

Das folgende Programm soll eine ganze Zahl x einlesen und die Folge der daraus erzeugten Zahlen gemäß obiger Regel ausgeben. Es ist eine repeat-Schleife zu verwenden.

```
program Ulam; {Ulams Vermutung, vgl. Skript S. 77}
var   {Vereinbarungen}
   x:  Integer;
begin
  write('Eine ganze Zahl > 1 eingeben: '); readln(x);
  if (x < 2) {Eingabe falsch}
  then writeln('falsche Eingabe!')  {Meldung}
  else {Schleife starten}
    begin
      repeat
        writeln(x);
        if odd(x)
        then x := 3 * x + 1
        else x := x div 2
        until x = 1;
        writeln(x, ' Sag ich doch - geht immer.');
    end {Schleife starten}
end {Ulam}.
```

Die while-Schleife

Die Syntax der while-Schleife lautet:

while-Schleife ::= **while** Ausdruck **do** Anweisung

Semantisch ist

```
while B do S
```

äquivalent zu

```
if B
then begin
         S;
         while B do S
end
```

Während also der Ausdruck hinter **until** in der repeat-Schleife eine Abbruchbedingung definiert, stellt der Ausdruck hinter **while** die Bedingung für eine Wiederholung dar.

Beispiel 5–3

```
program Therapie;
{Beschaeftigungstherapie: Summe der Zahlen von 1 bis N,
 vgl. Skript S. 78, danke Carl-Friedrich ...}
var   {Vereinbarungen}
   S, N:  Integer;
begin
   write('Eine ganze Zahl > 0 eingeben: '); readln(N);
   S := 0;   {Summe S initialisieren}
   while N > 0 do
   begin
      S := S + N;
      N := N - 1
   end;
   writeln('Summe = ', S);
end {Therapie}.
```

Die for-Schleife

Die Syntax der for-Schleife lautet

for-Schleife ::= **for** Variablen-Bezeichner **:=**
Ausdruck { **to** | **downto** } Ausdruck **do** Anweisung

Der Variablen-Bezeichner wird auch als Kontrollvariable bezeichnet, der Ausdruck vor `to`, bzw. `downto`, ist der Startwert, der danach der Endwerte. Kontrollvariable, Start- und Endwert müssen vom selben, ordinalen Typ sein.

Beispiel 5–4

Ein Sparbrief verzinst sich anfänglich mit 2%. Für die Laufzeit von 5 Jahren steigt der Zinssatz jährlich um 1%. Zinsen werden auf die Einlage und den bereits aufgelaufenen Zinsertrag gezahlt.

```
program Guthaben;
{Sparbrief mit wachsendem Zins}
var   {Vereinbarungen}
   Einlage, Guthaben, ZSatz, AnfSatz, Laufzeit:  Integer;
begin
   write('Einlage in vollen DM: '); readln(Einlage);
   Einlage := Einlage * 100;
   write('Anfaenglicher Zinssatz in %: ');
readln(AnfSatz);
   write('Laufzeit in Jahren: '); readln(Laufzeit);
   Guthaben := Einlage;{in Pfennigen}
   for ZSatz := AnfSatz to AnfSatz + Laufzeit - 1 do
   begin
      Guthaben := Guthaben + (Guthaben * Zsatz div 100);
      writeln('Zinssatz = ', ZSatz:2,
         ' Guthaben (volle DM) = ', Guthaben div 100);
   end;
end {Guthaben}.
```

Ein Lauf des Programms sähe wie folgt aus.

```
./guthaben
Einlage in vollen DM: 12000
Anfaenglicher Zinssatz in %: 2
Laufzeit in Jahren: 5
Zinssatz =  2 Guthaben (volle DM) =     12240
Zinssatz =  3 Guthaben (volle DM) =     12607
```

```
Zinssatz =  4 Guthaben (volle DM) =     13111
Zinssatz =  5 Guthaben (volle DM) =     13767
Zinssatz =  6 Guthaben (volle DM) =     14593
```

Ein weiteres Beispiel wäre eine „Einmal-Eins-Tabelle“, wobei wir nur die Dreiecksmatrix drucken.

```
program EinmalEins;
{das kleine Einmal-Eins}
var
   i, j: Integer;
begin
   write('1x1');
   for i := 9 downto 1 do
      write(i:3);
   writeln;
   for i := 1 to 10 do write(,___');
   writeln;
   for i := 1 to 9 do
   begin
      write(i:2,':');
      for j := 9 downto i do
        write(i * j: 3);
      writeln;
   end {for i};
end {EinmalEins}.
```

Die Ausgabe sieht - dank einiger Formatierangaben in den write-Anweisungen - wie folgt aus

```
./EinxEin
1x1  9  8  7  6  5  4  3  2  1
_____________________________
 1:  9  8  7  6  5  4  3  2  1
 2: 18 16 14 12 10  8  6  4
 3: 27 24 21 18 15 12  9
 4: 36 32 28 24 20 16
 5: 45 40 35 30 25
 6: 54 48 42 36
 7: 63 56 49
 8: 72 64
 9: 81
```

Semantisch ist die for-Schleife wie folgt definiert.

```
for kv := aw to ew do stmt
```

ist gleichwertig mit

```
kv := aw;
fin := ew;
if kv <= fin then stmt;
while kv < fin do
begin kv := succ(kv); stmt end
```

Zu beachten ist:

- Anfangs- und Endwert werden einmal (zu Anfang) ausgewertet.
- eventuell wird die Schleife garnicht durchlaufen (wenn ew < aw bzw. ew > aw für **downto**)
- die Kontrollvariable ist nicht veränderbar, demnach wäre es falsch zu schreiben

```
for i := n to m do
begin
   ...
   i := i - 1;
   ...
end
```

- der Wert der Kontrollvariablen ist nach Ende der Schleife undefiniert.

Andere als ordinale Typen (typisch Integer) sind für die Kontrollvariable nicht zulässig, obwohl manchmal eine Schrittweite von z.B. 0.1 ganz nützlich wäre.

Zuletzt gehört noch die with-Anweisung zu den strukturierten Anweisungen. Wir gehen darauf im Zusammenhang mit Record-Vereinbarungen weiter unten ein.

6 Records und Arrays

Bei der Behandlung von Typ- und Variablenvereinbarungen haben wir *Records* (Verbunde) und *Arrays* (Vektoren, Matrizen) ausgelassen. Dies wollen wir nun, zusammen mit der with-Anweisung, nachholen.

6.1 Der strukturierte Datentyp `record`

Ein Record (Verbund, Satz) erlaubt die Zusammenfassung inhomogener Komponenten, die „Feld“ heißen. Jede Komponente wird durch einen Namen identifiziert, der nur lokal zum Verbund definiert ist. Der Name und der beliebig zu wählende Typ des Felds werden in einer sog. Record-Vereinbarung, d.h. einer Typvereinbarung, festgelegt.

strukturierter-Typ ::= [**packed**] Array-Typ |
Mengen-Typ | Datei-Typ | Record-Typ

Record-Typ ::= **record** Feldliste **end**
Feldliste ::= [Bezeichner [**,** ...] **:** Typ **;**][...]
[**case** [Bezeichner **:**] Typbezeichner **of**
{Konstante [**,** ...] **: (** Feldliste **)** } [**;**...]]

Beispiel:

```
program SehrVerbunden;
{only for the record}
type
   Eigenschaften = set of (Garage, Keller, Balkon,
       GaesteWC, ZH, Aufzug);
   Wohntyp = (EinZApp, ZweiZKB, DreiZKB, VierZKB);
   Person = record
          Name,
          Vorname: string(40);
          SucheEine: Wohntyp;
          MussHaben: Eigenschaften;
          BGrenze: Integer
       end;
   Wohnung = record
    Wohnid: string(20);
    Groesse: integer;{Quadratmeter gerundet}
    IstEine: Wohntyp;
    Hat: Eigenschaften;
    Miete: Integer;{volle DM gerundet}
    NK: Integer;{volle DM gerundet}
   end;

var
   Sucher  : Person;
   Angebot : Wohnung;
begin
  {Werte für Sucher und Angebot einlesen}
  {...}
  if Sucher.SucheEine = Angebot.IstEine
  then {Wohnungstyp passt}
    if Sucher.Musshaben + Angebot.Hat = Angebot.Hat
    {A Teilmenge B gdw A vereinigt B = B}
    then
      if Angebot.NK + Angebot.Miete <= Sucher.BGrenze
      then writeln('Bingo: Angebot ', Angebot.Wohnid,
           ' passt!');
end {SehrVerbunden}.
```

Wie man an dem Beispiel sieht, geschieht der Zugriff auf Felder durch Pfadnamen, die sich aus dem Variablenbezeichner, aus dem folgenden Punkt und dem sich anschließenden Feldbezeichner zusammensetzen. Man bezeichnet sie auch als Selektoren. Records laßen sich nur kompo-

nentenweise verarbeiten, d.h. man muß einzelnen Komponenten Werte zuweisen oder Werte für sie einlesen. Ausnahme ist die Zuweisung eines Records an einen anderen, wenn diese typgleich sind.

```
var
  AlteBleibe, NeueBleibe: Wohnung;
...
begin
...
  {Mieterhöhung um 5%}
  NeueBleibe.Miete := round(NeueBleibe.Miete * 1.05)
...
  AlteBleibe := NeueBleibe;
...
end
```

Records dürfen im übrigen auch geschachtelt werden:

```
type
  Auftrag = record
     Kdnr : 0 .. 99999;
     Lieferanschrift : record
        Ort: string(40);
        Strasse: string(40);
        Nr: string(5)
     end;
     Rechnungsanschrift : record
        Ort: string(40);
        Strasse: string(40);
        Nr: string(5)
     end;
     Bestellung: array[1 .. 20] of
     record
        ArtNr: 10000 .. 99999;
        Menge: integer
     end
  end {Auftrag};
```

Entsprechend könnte die 1. Bestellung für eine (fünfstellige!) Artikelnummer 47110 für einen Auftrag `FolgeAuftrag` lauten:

```
FolgeAuftrag.Bestellung[1].ArtNr := 47110;
```

Häufig tritt der Fall ein, daß ein Objekt in verschiedenen „Spielarten“ auftritt, z.B. Mietwagen als normale Pkw, Wohnmobile, oder LkW. Pascal

sieht hierfür den sog. **Varianten- Record** vor, dessen varianter Teil mittels **`case`** eingeleitet wird.

```
type
  ArtTyp = (Pkw, Wohnmobil, Lkw);
  Mietwagentyp = record
    Marke: Typschluessel;
    Kennzeichen: String(12);
    Sitzplätze: Integer;
    case Art : ArtTyp of
    Pkw: (Kofferraum: Integer);{Groesse in Liter}
    Wohnmobil: (Schlafplaetze: Integer;
      Duschzelle: Boolean);
    Lkw: (Fuehrerschein: string(3);
      Ladevolumen: Integer; {in Kubikmeter);
      Ladegewicht: Integer; {in kg};
      Sonntagsfahrverbot: Boolean)
end {Mietwagentyp};
```

Zulässig ist **ein** varianter Teil, der den Abschluß der Record-Vereinbarung bildet (d.h. den hinteren Teil des Records belegt). Der Compiler reserviert in jedem Fall Platz für die größte Variante. Zur Laufzeit wird dieser Platz dann je nach gewähltem Fall mit den entsprechenden Werten gefüllt. Klar ist dabei, daß ein Zugriff auf Komponenten, für die momentan fallgerecht nichts gespeichert ist, z.B. das `Führerschein`-Feld für `Art = Pkw`, zu unvorhersagbaren Ergebnissen führt.

Übung 6–1

Definiert sei der Typ

```
Geotyp = (Viereck, Quadrat, Dreieck, Kreis).
```

Geben Sie einen Record-Typ `GeoOT` an, der im invarianten Teil z.B. Felder für Umfang und Inhalt des geometrischen Objekts hat und im varianten Teil, abhängig von `Geotyp`, die anderen relevanten Daten enthält, z.B. Länge und Breite für ein Viereck und Radius für einen Kreis. Die Längen- und Flächenangaben seien vom Typ `Int`.

Berechnen Sie für alle Arten von GeoObjekten Fläche und Umfang.

6.2 Die with-Anweisung

Beim Arbeiten mit Records stellt man schnell fest, daß der Zwang, Felder über den vollen Pfadnamen anzusprechen, umständlich ist. Mittels der with-Anweisung ist eine abkürzende Schreibweise möglich, indem man den Recordvariablenbezug, der für mehrere Anweisungen gleich ist, genau einmal herstellt („herausfaktorisiert“).

Im folgenden Beispiel geschieht dies für Sucher (Person) und Angebot (Wohnung).

```
with Sucher, Angebot do
  if SucheEine = IstEine
  then {Wohnungstyp passt}
     if Musshaben <= Hat
     {A Teilmenge von B}
     then if NK + Miete <= BGrenze
       then writeln('Bingo: Angebot ', Wohnid, '
passt!');
```

Die Syntax der with-Anweisung lautet:

with-Anweisung ::= **with** Record-Bezug [**,** ...] **do**
Anweisung

Record-Bezug ist dabei meist ein Recordvariablenbezeichner. Wie wir im Zusammenhang mit Zeigern später sehen werden, kann der Bezug auch über einen dereferenzierten Zeiger (Pointer) auf einen dynamisch angelegten Record erfolgen.

Bei mehrfach geschachtelten Records, z.B. Lieferanschrift in Auftrag, stellen in einer Liste von Variablenbezügen die links stehenden Variablen den Namensraum für die weiter rechts folgenden Variablenbezüge her. Damit ist semantisch

with v_1, v_2, ..., v_n **do** *S*

gleichwertig mit

```
with v1 do
    with v2 do
       ...
       with vn do S
```

Klar ist auch, daß bei gleichgewählten Feldbezeichnern eine eindeutige Auswahl nicht mehr gewährleistet ist. Für den oben definierten Typ Auftrag war dies für Liefer- und Rechnungsanschrift der Fall. Wenig befriedigend ist daher das Verhalten des GNU-Compilers im folgenden Fall:

```
var
   A1, A2  : Auftrag;
begin
   {Werte für Auftrag einlesen}
   {...}
   A1.Lieferanschrift.Ort := 'Niergendwo';
   A1.Rechnungsanschrift.Ort := 'Kassel';
   with A1,  Rechnungsanschrift, Lieferanschrift do
   if Ort = 'Kassel'
      then writeln('lokale Lieferung')
      else writeln('Lieferung nach ', Ort);
end {Auftrag}.
```

Welches `Ort`-Feld genommen wird, hängt von der Reihenfolge in der with-Anweisung ab (Liefer- vor Rechnungsanschrift oder umgekehrt), die Mehrdeutigkeit wird nicht erkannt.

6.3 Arrays

Gilt es, eine Reihe von Werten gleichen Typs im Programm zu halten und zu verarbeiten, z.B. die Umsatzzahlen der zehn Filialen der Fa. Reibach&Söhne, dann macht es wenig Sinn, dafür Variable `u1`, `u2`, ..., `u10` einzuführen. Vielmehr sehen alle Programmiersprachen dafür einen strukturierten Datentyp vor, der einem Vektor, bzw. mehrdimensionalen Matrizen, entspricht und in Pascal mit dem Schlüsselwort **`array`** deklariert wird.

```
type
  Umsaetze = array[1 .. 10] of Int; {Umsaetze der
Filialen}
  ...
var
  JahresUmsatz: Umsaetze;   {Arrayvariable}
```

Nach dem Schlüsselwort array folgt die Angabe der möglichen Indexwerte. Hier ist jeder ordinale Typ zugelassen (TP: nicht LongInt und Ausschnitte von LongInt). Zulässig wäre demnach

```
Haeufigkeit: array[Char] of Integer;
```

zum Zählen der Auftrittshäufigkeit eines Zeichens.

Oft gibt man Indexgrenzen explizit durch einen kleinsten und größten Indexwerte an. Im Beispiel oben etwa 1.. 10 für zehn mögliche Elemente (Komponenten).

Nach dem Schlüsselwort **of** folgt die Komponentenangabe, wobei alle Komponententypen zugelassen sind. Wie schon früher betont, definiert ein Array eine homogene Kollektion, d.h. alle Komponenten sind vom selben Typ.

Die Syntax einer Array-Vereinbarung lautet:

array-Typ ::= [**packed**] **array [** einfacher-Typ [**,** ...] **]**
of Typ

einfacher-Typ ::= Typbezeichner | **(** Bezeichner [**,** ...] **)** |
Konstante **..** Konstante

Die Verarbeitung der Array-Werte erfolgt - wie bei den Records - komponentenweise durch Angabe eines Indexwertes. Um etwa auf den Häufigkeitswert des Buchstabens `'e'` im obigen Array zuzugreifen, schreibt man

```
Haeufigkeit['e']
```

Iteriert man über alle Werte, bietet sich meist eine for-Schleife an, wie bei der Ermittlung des Durchschnittsumsatzes im Programm unten.

```
program Umsatz;
{Einfuehrung von arrays}
const
   MaxFiliale = 10;
type
   Int  = Integer;{TP: LongInt}
   Umsatz = array[1 .. MaxFiliale] of Int;

var
   JahresUmsatz: Umsatz ;
   i: Int;
   USumme,
```

```
 UDurchschnitt: Int;
 Next         : Int;

function rand: Int;
{selbstgestrickter Zufallszahlgenerator}
{liefert Pseudozufallszahl zw. 0 und 2**15-1}
{analog UNIX rand}
begin
  next := next * 1103515245 + 12345;
  rand := (next shr 16) and 32767
  {UNIX man page: return ((next >16) & 32767);}
end {rand};

begin
  next := 11;{initialisieren Zufallszahlgenerator}
  for i := 1 to MaxFiliale do
    JahresUmsatz[i] := rand;
    {initialisiert mit Zufallszahlen}
  {Durchschnittsumsatz ermitteln}
  USumme := 0;
  for i := 1 to MaxFiliale do
    USumme := USumme + JahresUmsatz[i];
  UDurchschnitt := USumme div MaxFiliale;
  writeln('Durchschnittsumsatz: ', UDurchschnitt);
  writeln('-------------------------------------');
  {Alle Umsätze ausgeben, 3 Sterne für alle Umsatze}
  {die mehr als 20% unter dem Durchschnitt liegen}
  for i := 1 to MaxFiliale do
  begin
    write('Filiale ', i:2,': ', JahresUmsatz[i]);
    if JahresUmsatz[i] < UDurchschnitt -
                  UDurchschnitt div 5
    then writeln(' ***')
    else writeln
  end {for i}
end {Umsatz}.
```

Die Ausgabe lautet:

```
Durchschnittsumsatz:      17802
-------------------------------------
Filiale  1:      21381
Filiale  2:      12504 ***
Filiale  3:      11371 ***
Filiale  4:       4268 ***
Filiale  5:      11672 ***
Filiale  6:      23702
Filiale  7:      27127
Filiale  8:      15270
Filiale  9:      24345
Filiale 10:      26386
```

Würde man für jede Filiale noch den Standort und die Anzahl der Mitarbeiter speichern, könnte dies durch Angabe eines Recordtyps für die Filiale wie folgt aussehen.

```
type
  Filialtyp = record
    Standort: string(30);
    UFiliale, Mitarbeiter: Integer;
  end;
  Filialen = array[1..MaxFiliale] of Filialtyp;
...
var
  F: Filialen;
  UM: Integer;
```

Die Ermittlung der Umsätze je Mitarbeiter verliefe dann ähnlich zu oben:

```
for i := 1 to MaxFiliale do
begin
  UM := F[i].UFiliale div F[i].Mitarbeiter
  ...
end
```

Alternativ wären drei Arrays für Standort, Umsatz und Mitarbeiteranzahl denkbar, aber weniger intuitiv.

Für den häufig auftretenden Fall der zwei- oder mehrdimensionalen Matrix ist statt

```
M: array[a .. b] of array[c .. d] of ...
```

die abkürzende Schreibweise

```
M: array[a .. b, c .. d, ...] of ...
```

zulässig. Entsprechend greift man auf Elemente dieses Arrays mit `M[i, j, ...]` statt mit `M[i][j]...` zu.

Ein Schachbrett läßt sich dann wie folgt vereinbaren.

```
Brett : array[1..8, 1..8] of
  record
    Farbe:(weiss, schwarz);
    Figur: (keine, SB, SL, SS, ST, SD, SK,
      WB, WL, WS, WT, WD, WK)
  end;
```

Man beachte, daß in dieser Vereinbarung keine expliziten Typnamen eingeführt werden. Dies ist in der Regel eine schlechte Taktik, da man oft später weitere Variablen, etwa eine Matrix `AltePosition`, vereinbaren will und dann nicht auf existierende Typen zurückgreifen kann. Auch ist so eine Zuweisung strukturierter Werte als ganzes nicht möglich.

Drei Anmerkungen sind zu Arrays generell zu machen.

- Pascal kennt keine dynamischen Arrays.

Damit drückt man aus, daß die Arraygrenzen in der Vereinbarung der Variablen durch Konstanten anzugeben sind. Der Übersetzer will also zur Übersetzungszeit wissen, wie groß der zu reservierende Bereich ist. Man spricht deshalb von statischen Arrays. Häufig wäre es geschickt, die Größe des Arrays erst zur Laufzeit (dynamisch) festzulegen, z.B. wenn eine Liste *T* von Klausurteilnehmern anzulegen ist. Bei dynamischen Arrays würde man zuerst die Anzahl der Teilnehmer `N` einlesen und anschließend das Feld *T* als **`array`**`[1..N]` **`of`** ... vereinbaren.

Dies geht in Pascal nicht und man muß die Grenzen so wählen, daß sie in jedem Fall passen.

- Pascal kennt keine assoziativen Arrays.

Nimmt man das Beispiel mit der Liste der Klausurteilnehmer und versucht man jetzt einen Teilnehmer `S = 'Klug'` zu finden, so muß man den Array durchlaufen und mit einem Test der Art

```
if T[i].Name = S
then writeln(T[i].Note)
...
```

prüfen, ob der i'te Eintrag der von „Klug" ist. Durch Sortieren der Einträge und andere geschickte Suchstrategien läßt sich dies verkürzen, generell operiert man aber mit Indexoperationen (sog. Ortsadressierung). Nicht möglich ist ein Zugriff der Art

```
T['Klug'].Note
```

der die Note mit dem Namen des Teilnehmers assoziiert. Einige moderne Programmiersprachen erlauben dies, machen allerdings im Hintergrund auch wieder Indexumrechnungen.

- Arraygrenzen sollten geprüft werden.

Einer der häufigsten Programmierfehler ist die Verletzung von Arraygrenzen.

Im folgenden Beispiel sucht ein Programmierer im Feld *F* mittels Laufvariable *j* ein Element *F*[*j*], das kleiner ist als *F*[*i*]. Die Schleife hält nur an, wenn es ein solches gibt.

```
const
  maxF = 20;
var
  F: array[1..maxF] of Int;
  H: i, j : Int;
  ...

begin
  ...
  H := F[i];
  j := i;
  repeat
    j:= j+ 1
  until F[j] < H;
  aux := F[j]; F[j] := F[i]; F[i] := aux;
```

```
  ...
end;
```

Sind zufällig alle Werte $F[j] \geq F[i]$ und enthält der erzeugte Code des Programms keine Überprüfung der Indexwerte für *F*, dann wird auf *F*[21], *F*[22], ... zugegriffen - die es nicht gibt -, bis ggf. erstmals eine referenzierte Speicherzelle einen Bitfolge enthält, deren numerische Interpretation größer als *F*[*i*] ist, sofern nicht vorher mit einer Speicherverletzung abgebrochen wurde.

Pascal überprüft - bei richtiger Compileroption - die strikte Einhaltung von Bereichsgrenzen zur Laufzeit. Diese Option sollte man unbedingt eingeschaltet lassen, bis das Programm absolut wasserdicht ausgetestet wurde. Danach kann man nochmals ohne die Option kompilieren und damit kleineren und schnelleren Code erzeugen, der auf die Bereichsgrenzenprüfung verzichtet.

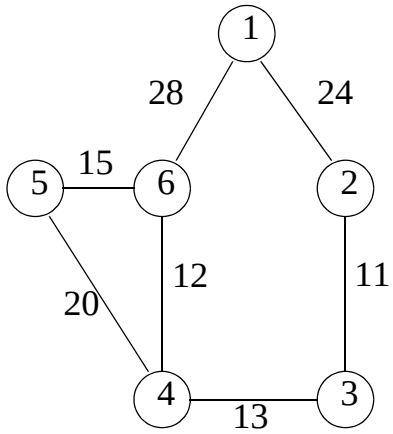
Wir schließen mit einem etwas größeren Beispiel: Floyd‘s Algorithmus zur Berechnung kürzester Pfade.

Beispiel 6–1

Hat man einen Graphen mit *n* Knoten, dessen Kantenbewertungen die Entfernung von einem Knoten zum anderen angeben, dann kann man diesen Graphen als $n \times n$ Matrix *M* darstellen, wobei $M[i, j] = M[j, i]$ die direkte Entfernung von *i* nach *j* (und umgekehrt) angibt. Gibt es keine solche direkte Kante, setzen wir den Wert auf Maxint/2. In der Hauptdiagonalen stehe immer eine Null.

Floyd‘s Algorithmus berechnet nun die Dist-Matrix, die anfänglich gleich der M Matrix ist, in der folgenden Schleife (das vollständige Programm steht unter `floyd.pas` zur Verfügung).

```
{Initialisieren Dist-Matrix}
for u := 1 to n do
  for v := 1 to n do
     Dist[u, v] := M[u, v];
{Dreifachschleife zur Berechnung}
for u := 1 to n do
  for v := 1 to n do
```



	1	2	3	4	5	6
1	0	24	.	.	.	28
2	24	0	11	.	.	.
3	.	11	0	13	.	.
4	.	.	13	0	20	12
5	.	.	.	20	0	15
6	28	.	.	12	15	0

```
for w := 1 to n do
  if dist[v, u] + dist[u, w] < dist[v, w]
  then dist[v, w] := dist[v, u] + dist[u, w];
```

Die Grundidee ist, den kürzesten Pfad von *v* nach *w* dadurch zu bestimmen, daß man alle Zwischenstationen *u* betrachtet. Ist der Weg über *u* kürzer als der bisher beste Wert, wird dieser als neuer Wert in `dist[v, w]` gespeichert.

Für den Graphen oben sieht die Anfangs- und Endbelegung wie folgt aus:

```
 0 24  .  .  . 28
24  0 11  .  .  .
 . 11  0 13  .  .
 .  . 13  0 20 12
 .  .  . 20  0 15
28  .  . 12 15  0
------------------
 0 24 35 40 43 28
24  0 11 24 44 36
35 11  0 13 33 25
40 24 13  0 20 12
43 44 33 20  0 15
28 36 25 12 15  0
```

Der genaue Beweis, daß dieses Verfahren auf die optimale Lösung hin konvergiert, ist nicht so einfach und findet sich z.B. in Aho/Ullman, Foundations of Computer Science, Computer Science Press, 1992, S. 499 - 501, dem auch das Beispiel entnommen ist.

Als weiterer Hinweis sei genannt, daß das Verfahren n^3 Vergleiche (Schritte) benötigt. Andere Verfahren, z.B. das von *Dijkstra* zur Bestimmung kürzester Pfade, benötigen weniger Schritte. Darauf wird in anderen Veranstaltungen, z.B. *Algorithmen und Datenstrukturen*, eingegangen.

Damit benden wir die Behandlung von Records und Arrays.

7 Ein- und Ausgabe

Programmiersprachen tun sich traditionell schwer mit der Ein- und Ausgabe. Schon für die einfache Tastaturein- und Drukkerausgabe gibt es so viele Besonderheiten zu beachten, daß Sprachdefinitionen diesen Teil gerne auslassen und auf gesonderte „Standardpakete“ (Bibliotheksroutinen) verweisen. Durch moderne Interaktionsformen (z.B. graphische Schnittstellen) ist die Vielfalt eher noch gestiegen.

In Pascal ist im Sprachreport die Ein- und Ausgabe relativ vernünftig geregelt. Dies betrifft die Ein- und Ausgabeanweisungen `read`, `readln`, `write` und `writeln`, die einen linearen Textdatenstrom von der Tastatur lesen, bzw. an den Bildschirm oder einen Drucker liefern.

Diese Befehle werden um die Bearbeitung von Dateien, speziell Plattendateien, in Turbo Pascal (und bei vielen anderen Compilern) erweitert. Dazu gehört das Setzen von Lese-/Schreibzeigern (`seek`), das Öffnen und Schließen der Dateien und andere Übergänge zum Betriebssystem. Wir können hierauf nur begrenzt eingehen.

Drittens gehört zur Ein- und Ausgabe heute der Umgang mit Pfeiltasten, Maus, Bildschirmzeiger (Cursor), usw. Da Pascal eigentlich keine objekt-orientierte Sprache ist und den asynchronen, reaktiven Programmierstil nicht unterstützt, wollen wir darauf auch weniger eingehen. Allerdings sei darauf hingewiesen, daß sich z.B. mit Turbo Pascal‘s Turbo Vision sehr praktische, interaktive Benutzerschnittstellen programmieren lassen.

7.1 Einfache Eingabe

Eine durchaus praktikable Grundidee der Ein- und Ausgabe ist, sich vorzustellen, der Rechner sei mit einem Eingabestrom von Zeichen verbunden und gebe analog eine Folge von Einzelzeichen aus. Dabei kann ein einmal gelesenes Zeichen nicht nochmals gelesen werden und ein einmal ausgegebenes Zeichen nicht wieder eingesammelt werden.

Deshalb sieht Pascal in der einfachsten Form nur Lesebefehle für Einzelzeichen des Typs `char` aus der Standardtextdatei *input* vor. Praktikabler ist es aber, die Eingabe als Folge von Zeilen mit einer Reihe von Werten unterschiedlichen Typs auf jeder Zeile zu behandeln. Die Werte werden von links nach rechts den Variablen der Eingabeanweisung typgerecht zugewiesen.

Eingabeanweisung ::= `read(` Variable [`,` ...] `)` | `readln` |
`readln(` Variable [`,` ...] `)`

Semantisch ist demnach `read(`v_1, v_2, ..., v_n`)` äquivalent zu

`read(`v_1`);`
`read(`v_2`);`
...
`read(`v_n`)`

und bewirkt Wertezuweisungen

v_1 := ...;
v_2 := ...,
...
v_n := ...;

Dabei gilt für integer und real: führende Leerzeichen, Tabulatorzeichen und Zeilenende werden ignoriert (überlesen).

Bei `char`: nächstes Zeichen wird genommen, Zeilenende wird zu Leerzeichen (TP: wird zu chr(13), Newline).

Bei `string`: Rest der Zeile wird genommen.

Beispiel:

```
var
  r : real;
  c1, c2, c3: char;
  i : integer;
begin
  read(r, c1, c2, c3, i);
...
```

Mit der Eingabe

```
640.0XYZ104
```

bewirkt dies die Zuweisungen

```
r := 640.0;
c1 := 'X';
c2 := 'Y';
c3 := 'Z';
i := 104
```

Um mit dem Zeilenende und mit dem Eingabeende (Dateiende) vernünftig umgehen zu können, gibt es zwei Standardfunktionen, die ein Boolsches Resultat liefern: `eoln` und `eof`.

Steht der Dateizeiger **nach** einem Lesebefehl `read(c)` für ein Zeichen `c` auf dem Zeilenendezeichen oder ist das Dateiende erreicht, liefert `eoln` den Wert `true`. Wurde das Zeilenende gelesen, d.h. erfolgte erneut ein `read(c)`, ist danach `eoln` wieder `false`, sofern nicht danach sofort erneut eine leere Zeile oder das Dateiende kommt.

Turbo Pascal liefert im übrigen auch ein Zeichen für das Lesen des Dateiendes zurück, nämlich chr(26), das Ctrl-Z Steuerzeichen. Allerdings sollte man nicht unbedingt annehmen, daß das Dateiende durch ein gespeichertes Zeichen signalisiert wird. In UNIX wird z.B. das Dateiende durch Ctrl-D signalisiert, das aber nicht gespeichert wird.

Restliche, ungelesene Werte einer Zeile kann man durch `readln` ohne weitere Angabe von Argumenten überspringen. Ein `readln` mit

Argumenten überspringt den Rest der Zeile nach dem Lesen des Werts für das letzte Argument.

Beispiel 7–1

Bei Vorliegen der Eingabezeilen

```
1 2 3
4 5 6
```

würden die beiden Leseanweisungen

```
readln(a, b); readln(c, d)
```

die Zuweisungen `a := 1; b := 2; c := 4; d := 5` bewirken.

7.2 Einfache Ausgabe

Die Syntax der Ausgabeanweisung lautet:

Ausgabeanweisung ::= `write(` Ausgabeparameter [`,` ...] `)` |
`writeln` | `writeln(` Ausgabeparameter [`,` ...] `)`

Ausgabeparameter ::= Ausdruck [`:` Feldlänge
[`:` Dezimalstellen]]

Dabei müssen *Feldlänge* und *Dezimalstellen* Integer-Ausdrücke sein.

Beispiel:

```
write(' Bruttobetrag: ', Brutto + MWSt : 8 : 2);
```

Ausgabewerte vom Typ Integer, Char, Boolean, Real und String werden im Standardformat (implementationsabhängig) dargestellt, jedoch immer

- char: mit einer Druckstelle
- Boolean: `False`, bzw. `True`
- real: in Exponentendarstellung wenn keine `:m:n` Angabe

Bei Angabe der Feldlänge `m` erfolgt die Ausgabe in `m` Druckstellen mit ggf. führenden Leerstellen. Ist der Ausgabewert breiter, erfolgt die Aus-

gabe mit der erforderlichen Anzahl an Druckstellen, d.h. es wird nicht abgeschnitten.

Realwerte mit der Angabe : *m* : *n* werden in Gleitkommadarstellung mit *n* Dezimalstellen ausgegeben.

`Writeln` ohne Argumente erzeugt nur einen Zeilenvorschub, `writeln` mit Argumenten gibt analog zu `write` die Argumente aus und produziert dann einen Zeilenvorschub.

Recht nützlich ist in Turbo Pascal die `ClrScr` (ClearScreen) Prozedur, die den Bildschirm mit Leerzeichen in der für den Bildschirmhintergrund gesetzten Farbe „säubert“ und den Cursor in die linke, obere Ecke setzt.

```
program Ausgabe;
{Ausgabeformate testen}
type
   Int  = Integer;
var
   i, j, k    : Int;
   r      : Real;
   s      : string(100);
   c1, c2, c3 : char;
begin
   i := 17;
   j := 4711;
   k := maxint;
   writeln(i, j, k);
   writeln(i:10, j: 10, k: 10);
   writeln(i: 3, j: 3, k: 3);
   {wahnsinnig aufregend}
   writeln(27+3,'=',30,' is',27+3=30);
   s := "Hallo Leute!";
   writeln(s, '**', s:5);
   r := -1234567890.0987654321;
   writeln(r, r: 12, r:15:3);
end {Ausgabe}.
```

Die Ausgabe lautet:

```
$ ./ausgabe
        17      47112147483647
        17      47112147483647
 1747112147483647
        30=        30 is  True
Hallo Leute!**Hallo
-1.2345679e+09-1.23457e+09-1234567890.099
```

7.3 Grundlagen der Dateiverarbeitung

Pascal hat noch einen weiteren strukturierten Datentyp, den wir unterschlagen haben: Dateien (engl. *files*). Dateien erlauben es, Werte über die Laufzeit eines Programms hinaus auf Datenträgern (Platten, Disketten, CD-ROM, Bändern) zu speichern.

Man vereinbart eine Dateivariable mit der folgenden Syntax.

Dateivereinbarung ::= **`file of`** Satztyp

Eine so vereinbarte Datei ist immer eine Folge von Sätzen fester Länge, die von 0 an durchnumeriert sind. Je nachdem, ob das Speichermedium wahlfreien (direkten) oder nur sequentiellen Zugriff zuläßt, kann man einen Lesezeiger auf jeden dieser Sätze setzen und dessen Wert einlesen oder (über-)-schreiben, bzw. im sequentiellen Fall den Lesezeiger von einem Satz zum nächsten oder wieder ganz an den Anfang setzen.

Da jeder Typ als Satztyp zugelassen ist, spricht man von *typisierten Dateien*. Häufigster Typ in einer Dateiverarbeitung ist der Record (Satz, Verbund). Die bisher in `read` und `write` zugrundegelegten Dateien sind dagegen sog. *Textdateien*.

Speziell mit den in Turbo Pascal zur Verfügung stehenden Dateibefehlen, dem UNIT-Konzept und einer schlichten, textorientierten Fenster- und Menügestaltung kann man sehr praktische Anwendungen, etwa eine kleine Buchausleihe oder eine Mitgliederverwaltung, schreiben.

Wir wollen dies vereinfacht an einem größeren Beispiel verdeutlichen. Die Anwendung ist die klassische Aufgabe der Erstellung von Lieferscheinen.

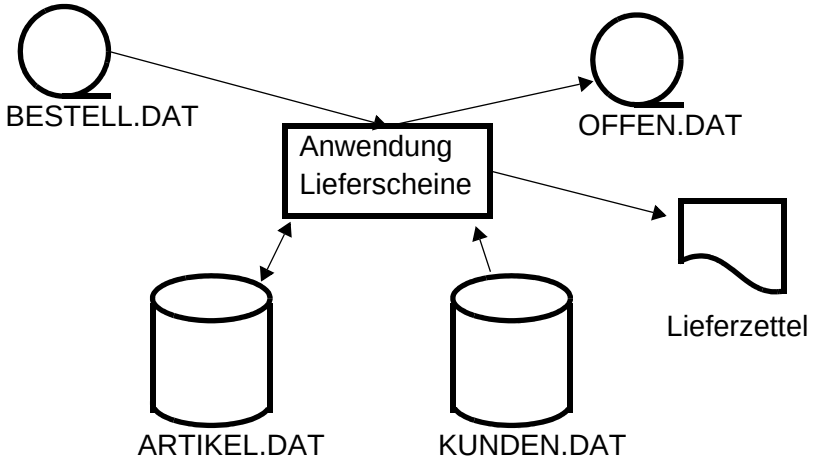


Abb. 7–1

Dabei werden Bestellungen von Kunden (BESTELL.DAT) nacheinander abgearbeitet und mit dem Artikelbestand (ARTIKEL.DAT) abgeglichen. Für abwickelbare Bestellungen werden Lieferzettel geschrieben und es wird der Bestand korrigiert. Sonst wird die Bestellung in eine Datei „offene Posten“ geschrieben zur späteren Neuvorlage.

Neben dem eigentlichen Programm (`lscheine.pas`) brauchen wir eine Reihe von Hilfsprogrammen zum Erzeugen der Artikel- und Kundendateien und deren Auflistung:
`martikel.pas`, `mkunden.pas`, `lartikel.pas`, `lkunden.pas`. Diese sind natürlich wenig komfortabel, erlauben aber einen Einstieg in die Dateiverarbeitung.

Die gemeinsamen Typvereinbarungen für Bestellungen, Kunden und Artikel ziehen wir in eine getrennte UNIT, hier namens `DataDict`, zusammen, die jedem der später übersetzten Programme durch die Direktive

```
USES DataDict;
```

zugebunden wird. Dabei wird nicht nur Schreibarbeit gespart, es wird auch klargestellt, daß isolierte Änderungen der Satztypen unzulässig sind, da auf die Daten von mehreren Programmen aus zugegriffen wird. Auf eine weitere Erläuterung des UNIT-Konzepts muß hier verzichtet werden.

```
UNIT DataDict;
{gemeinsame Satzdefinitionen: „Data Dictionary“}
INTERFACE

TYPE
  BestellType = RECORD
                  KdNr,                 {wer bestellt}
                  ArtNr,                {was}
                  Menge: Integer;       {wieviel}
                  Datum: string[10]     {wann}
  END;
  KundenType = RECORD
                 KdNr: Integer;
                 Name,
                 Ort: String[31];
  END;
  ArtikelType = RECORD
                  ArtNr,
                  Bestand: Integer;
                  LagerOrt: string[7]
  END;

IMPLEMENTATION
BEGIN
END {DataDict}.
```

Im folgenden Programm wird nun sequentiell eine Artikeldatei erstellt. Artikel bestehen dabei nur aus Artikelnummer, Bestand und LagerOrt.

Wesentlich für uns sind die Befehle

- `Assign`(Dateivariable, Dateinamensstring)
- `Rewrite`(Dateivariable), bzw. Reset(Dateivariable)
- `Append`(Dateivariable)
- `Close`(Dateivariable)

Mit `Assign()` wird die Verbindung von Dateivariable im Programm zum Dateisystem des Rechners hergestellt. Für den Dateinamensstring gelten die Konventionenen des Betriebssystems (absolute oder relative Pfade, Laufwerksbezeichner, Längenbeschränkungen für Namen, usw.).

Mit `Rewrite` wird eine neue Datei angelegt oder eine existierende geleert. Der Dateizeiger wird auf den Anfang gesetzt. Bei `Append` wird er in einer existirenden Datei ans Ende der Datei gesetzt (engl. append: anfügen). `Close()` schließt die Datei, ggf. wird der Datenträger freigegeben und es ist sichergestellt, daß alle Datensätze aus dem Speicher (Puffer) des Rechners auch physisch auf den Datenträger geschrieben wurden.

Zuletzt sei noch auf den Schreibbefehl für die Datei verwiesen:

```
write(f, Artikel);
```

Für typisierte Dateien nennt der Lese-/Schreibbefehl als erstes Argument die Dateivariable.

```
PROGRAM Artikeldatei;
{Sequentielles Anlegen einer Artikeldatei}
USES DataDict;
VAR
  f: FILE OF ArtikelType;
  Artikel: ArtikelType;
  Fertig: Boolean;
  ch: char;
BEGIN
  Assign(f, 'ARTIKEL.DAT');
  Write('Neue Datei? (Y/N): ');
  Readln(Ch);
  IF (Ch = 'y') OR (Ch = 'Y') THEN Rewrite(f) ELSE
Reset(f);
  writeln('BITTE ARTIKELDATEN NACHEINANDER EINGEBEN -
             ENDE MIT ARTNR = 0');
  Fertig := False;
  WHILE NOT Fertig DO
  WITH Artikel DO
  BEGIN
    write('Artikelnummer: '); readln(ArtNr);
    IF ArtNr <= 0 THEN Fertig := true
    ELSE BEGIN
      write('Bestand: '); readln(Bestand);
```

```
        write('Lagerort: '); readln(LagerOrt);
        write(f, Artikel)
      END {else - ArtNr > 0}
    END {while};
    Close(f);
END {Anlegen Artikeldatei}.
```

Wesentlich ist jetzt noch die Methode, wie direkt auf eine typisierte Datei zugegriffen wird. Hierzu muß man den Dateizeiger mit `Seek(`Dateivariable, *i*`)` setzen, wobei $0 \le i <$ Anzahl-Sätze-in-Datei. Er steht dann „am Anfang" des *i*-ten Satzes, den man normal mit `read(`Datzeivariable, Empfängervariable`)` liest, bzw. analog schreibt. Danach steht der Zeiger „am Ende" des Satzes, bzw. am Anfang des nächsten Satzes. Abfragen auf *eof* sind natürlich weiterhin möglich (siehe unten).

Wir geben hier noch die Programme zum Auflisten der Bestellungen und für die Lieferscheine an. Letzteres enthält den wahlfreien Zugriff auf ARTIKEL.DAT zur Bestandskorrektur.

```
PROGRAM Bestellliste;
{Sequentielles Auflisten einer Bestelldatei}
USES DataDict;
VAR
    f: FILE OF BestellType;
    Bestell: BestellType;
    NeuerKunde: Boolean;
    LetzteKdNr: Integer;
BEGIN Assign(f, 'BESTELL.DAT');
  Reset(f);
  writeln;
  writeln('AUFLISTUNG BESTELLUNGEN/OFFENE POSTEN');
  IF NOT Eof(f) THEN read(f, Bestell) ELSE Exit;
  REPEAT
    WITH Bestell DO BEGIN
      writeln('KUNDE: ', KdNr,' BESTELLUNG VOM: ',
             Datum);
      writeln('ARTIKEL NR.              MENGE');
      writeln('-----------------------------');
      writeln(ArtNr: 10, Menge: 20);
      LetzteKdNr := KDNr;
      NeuerKunde := False;
      WHILE NOT Eof(f) AND NOT NeuerKunde DO BEGIN
        read(f, Bestell);
        IF LetzteKdNr = KdNr
```

```
          THEN writeln(ArtNr : 10, Menge: 20)
          ELSE BEGIN
            writeln('----------------------------');
            NeuerKunde := True
          END
        END {while};
      END {with}
    UNTIL Eof(f);
    writeln('-----------------------------');
    writeln('ENDE DER BESTELLUNGEN');
    Close(F)
END {Auflisten Bestelldatei}.
```

Eine mögliche Ausgabe könnte lauten.

```
AUFLISTUNG BESTELLUNGEN/OFFENE POSTEN
KUNDE:       4711 BESTELLUNG VOM: 12.12.1998
ARTIKEL NR.              MENGE
------------------------------
     100                  10
       200                  20
------------------------------
KUNDE:       4712 BESTELLUNG VOM: 13.12.1998
ARTIKEL NR.              MENGE
------------------------------
       200                  40
       300                  60
------------------------------
ENDE DER BESTELLUNGEN
```

Das zugehörige Bestellscheine-Programm `lscheine.pas` verwendet Prozeduren, um den Zugriff auf die Dateien besser vom restlichen Programmtext zu trennen. Prozeduren und Funktionen sind das Thema des nächsten Kapitels. Wir geben den Programmtext zu `lscheine.pas` in der Borland Turbo Pascal 6.0 Version hier an, wollen ihn aber nicht besprechen.

```
PROGRAM Lieferscheine;
{Sequentielles Lesen der Bestelldatei}
{Suchen Kundenname und Ort zu Kundennummer}
{Suchen Bestand der Artikel, Bestand berichtigen}
{Lieferschein ausgeben, wenn Bestand ausreichend}
USES DataDict;
VAR
  fBestell: FILE OF BestellType;
```

```
  fKunden: FILE OF KundenType;
  fArtikel: FILE OF ArtikelType;
  fOffen: FILE OF BestellType;
  Kunde: KundenType;
  Bestell: BestellType;
  Artikel: ArtikelType;
  LetzteKdNr: Integer;  {Gruppenwechsel}
  ch: char;
PROCEDURE PickKunde(KN: Integer; VAR KRec: KundenType);
  BEGIN
    Seek(FKunden, 0);
    REPEAT
      read(FKunden, KRec);      {sequentiell}
    UNTIL Eof(FKunden) OR (KRec.KdNr = KN);
    IF KRec.KdNr <> KN THEN WITH KRec DO BEGIN
      Name := '???'; Ort := '???'; KdNr := 0;
    END;
  END;

  PROCEDURE PickArtikel(AN: Integer;
    VAR ARec: ArtikelType);
  BEGIN
    Seek(FArtikel, 0);
    REPEAT
      read(FArtikel, ARec);     {sequentiell}
    UNTIL Eof(FArtikel) OR (ARec.ArtNr = AN);
    IF ARec.ArtNr <> AN THEN WITH ARec DO BEGIN
      Bestand := 0; LagerOrt := '???';
    END;
  END;

BEGIN
  Assign(fBestell, 'BESTELL.DAT'); Reset(FBestell);
    {muss existieren}
  Assign(fKunden, 'KUNDEN.DAT'); Reset(fKunden); {ditto}
  Assign(fArtikel, 'ARTIKEL.DAT'); Reset(fArtikel);
{ditto}
  Assign(fOffen, 'OFFEN.DAT');
  Write('OFFEN.DAT: Datei offene Posten - Neu? (Y/N): ');
  Readln(Ch);
  IF (Ch = 'y') OR (Ch = 'Y') THEN Rewrite(fOffen)
    ELSE Reset(fOffen);
  writeln;
  writeln('LISTE DER LIEFERSCHEINE');
  IF NOT Eof(FBestell)
  THEN read(FBestell, Bestell) ELSE Exit;
```

```
    REPEAT
      WITH Bestell DO BEGIN

        PickKunde(KdNr, Kunde);
             {liefert Kundenname oder ???}
        writeln('KUNDE ', KdNr:5,' ',Kunde.Name: 31,
           ' ',Kunde.Ort: 31);
        writeln('BESTELLUNG VOM: ', Datum);
        LetzteKdNr := KDNr;
        writeln('ARTIKEL NR.              MENGE');
        writeln('------------------------------');
        REPEAT
          write(ArtNr: 10, Menge: 20);
          PickArtikel(ArtNr, Artikel);
          IF Artikel.Bestand >= Menge THEN BEGIN
            writeln(' Status ok');
            Artikel.Bestand := Artikel.Bestand - Menge;
            Seek(FArtikel, FilePos(FArtikel) - 1);
            write(FArtikel, Artikel);
          END
          ELSE BEGIN
            writeln(' wird nachgeliefert!');
            write(FOffen, Bestell);
          END;
         IF NOT Eof(FBestell) THEN read(fBestell, Bestell)
          ELSE LetzteKdNr := 0
        UNTIL LetzteKdNr <> KdNr;
        writeln('------------------------------');
        writeln;
      END {with Bestell}
    UNTIL Eof(fBestell);
    writeln('ENDE DER LISTE');
    Close(FBestell); Close(FKunden); Close(FArtikel);
    Close(FOffen);
  END {Lieferscheine}.
```

Die restlichen Programme finden sich im Verzeichnis `Kap7`.

Der GNU-Compiler lehnt sich in seiner Dateibehandlung an die Vorschläge für Extended Pascal ISO 10206 an. Statt `Assign` muß dort mit `bind(f, b)` die Verbindung von Dateivariable *f* (die mit dem Schlüsselwort **`bindable`** vereinbart sein muß) zum externen Dateinamen *b* hergestellt. Letzterer wird aber nicht direkt angegeben, vielmehr ist *b* vom

Typ `BindingType`. Für `BindingType` ist eine Komponente `name` definiert, der man den Dateinamen zuweisen kann.

Insgesamt überzeugt auch dieses Konzept wenig, wie überhaupt die unterschiedlichen Vorstellungen über Dateien (was z. B. zu tun ist, wenn eine Datei geöffnet wird, die noch garnicht existiert: Fehler oder als leere Datei anlegen?) in Pascal und C sehr störend sind. Sie werden verstärkt durch unterschiedliche Auffassungen in den Betriebssystemen, z.B. DOS/Windows versus UNIX.

Dies schlägt sich bis auf die Interprozeßkommunikation durch und erschwert die Verknüpfung von Programmen untereinander.

8 Prozeduren und Funktionen

Sind in einem Programm *P* wiederkehrende Aufgaben zu erledigen, kann man den Programmtext hierfür in einem (Unter-)-Programm *Q* zusammenfassen. An den Stellen, an denen die Leistungen von *Q* gebraucht werden, verzweigt man in das Unterprogramm *Q* und arbeitet es ab. Danach kehrt man an die alte Verzweigungsstelle in *P* zurück.

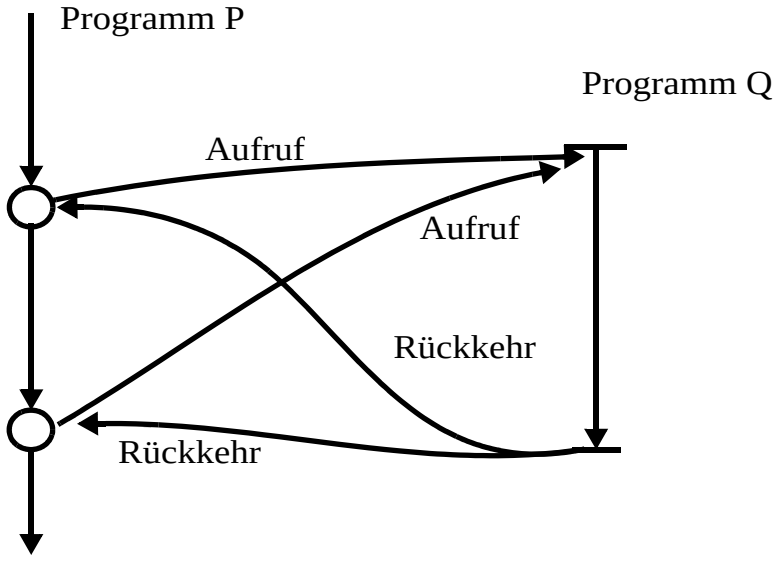
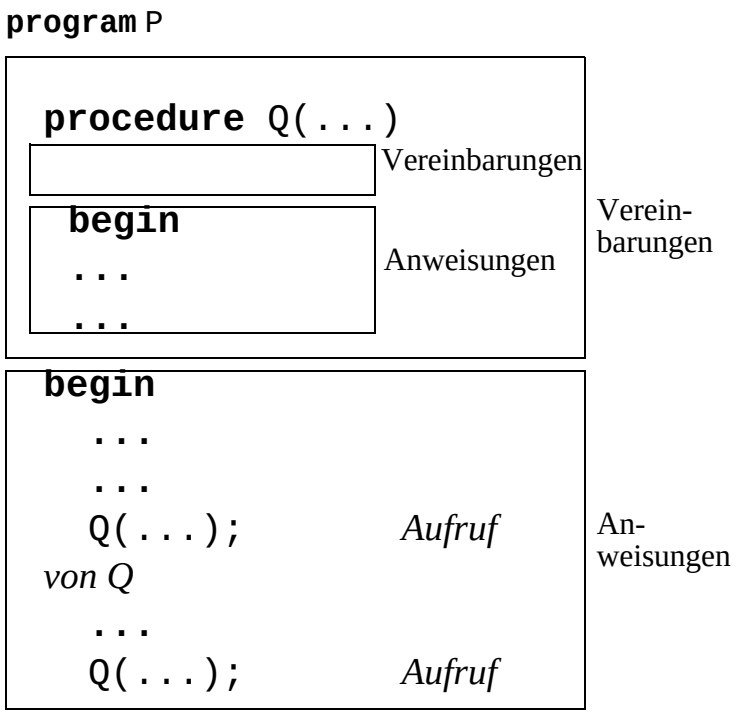


Abb. 8–1

Dies ist die uralte Idee der Unterroutine. Wird *Q* in *P* untergebracht, z.B. am Ende von *P*, muß man zusätzlich aufpassen, daß man nicht in *Q* „hineinläuft“, ohne es zu wollen. Wird in *Q* eine weitere Unterroutine *R* benötigt, wird die Verwaltung der Rücksprungadressen schon schwieriger.

Will man *Q* auch in anderen Programmen einsetzen, muß man *Q* von *P*‘s Programmkontext (den in *P* global vereinbarten Variablen) unabhängig machen.

Man erkennt daran, daß ein geordnetes Konzept, das z.B. über das einfache GOSUB in BASCIC hinausgeht, erforderlich wird. Pascal bietet hierfür *Prozeduren* und *Funktionen* an.



Beispiel 8–1

P sei das Programm `Max4`: „Größte von 4 Zahlen bestimmen“. Q sei die Funktion `Max2`, die das größere von zwei Argumenten zurückliefert.

```
program Max4;
{P = Max4: bestimmen der groessten von 4 Zahlen}
var
   a, b, c, d, x, y, z :  Integer;

{hier jetzt Q = Max2 = Maximum von 2 Zahlen}
function Max2(Zahl1, Zahl2 :  Integer): Integer;
```

```
begin
   if Zahl1 > Zahl2
   then Max2 := Zahl1
   else Max2 := Zahl2
end {Max2};

begin
   write(”Vier Zahlen eingeben: ”);
   readln(a, b, c, d);
   x := Max2(a, b); {Maximum von a, b}
   y := Max2(c, d); {Maximum von c, d}
   z := Max2(x, y);  {Maximum von x, y}
   writeln(”Maximum ist ”, z);
end {Max4}.
```

Insgesamt ist dieses Programm noch etwas umständlich. Die Hilfsvariablen x, y, z sind entbehrlich und wir schreiben:

```
readln(a, b, c, d);
writeln(”Maximum ist ”, Max2(Max2(a, b), Max2(c, d)));
```

Die Vorteile dieser Technik sind:

- das Programm wird kürzer
- es wird weniger Speicherplatz bei der Ausführung benötigt
- das Programm wird leichter überprüfbar
- die Strukturierung wird besser
- Arbeitsteilung wird möglich
- die Wiederverwendung wird erhöht.

Nachteile:

- keine

... na ja, ggf. verwirrend für Anfänger und bei schlechtem Gebrauch Laufzeitverlängerung und undurchsichtige Seiteneffekte.

8.1 Unterschied Prozedur und Funktion

Prozeduren und Funktionen sind beides Unterroutinen. Funktionen liefern aber als Ergebnis ihrer Abarbeitung einen Wert zurück. Deshalb darf überall dort, wo ein Ausdruck eines bestimmten Typs T stehen darf, auch ein Funktionsaufruf einer Funktion, die einen Wert vom Typ T zurückliefert, stehen.

Beispiel 8–2 Funktionsaufrufe

```
i := length(”Hallo Leute!”);
r := sqrt(x + y);
if IstPrimzahl(a) then ...
writeln(HundertSterne);
writeln(”Resultat: ”, min(abs(v1) + trunc(r2),
       100 * q));
sk := Skalarprodukt(V1, V2);
```

Zurückgegebene Werte müssen dabei im Standard Pascal einfache Werte sein, Arrays und Records sind nicht möglich.

> + Anfänger verwechseln häufig „zurückliefern“ mit „ausgeben“. Eine Funktion druckt nichts (sofern nicht in der Funktion untypischerweise `writeln` auftaucht), sondern produziert einen Wert (eine Zahl, einen String, einen Wahrheitswert) den sie dem aufrufenden Programm an die Stelle übergibt, an der der Funktionsaufruf erfolgte.

Welcher Wert zurückgeliefert wird, bestimmt die Funktion. In Pascal geschieht dies durch Nennung des Funktionsnamens auf der linken Seite einer Zuweisung. Man vergleiche hierzu die Funktion `Max2(Zahl1, Zahl2)`:

```
if Zahl1 > Zahl2
then Max2 := Zahl1
else Max2 := Zahl2
```

Meist ist diese Zuweisung die letzte Anweisung der Funktion, mit der sie dann verlassen wird. Andere Programmiersprachen, speziell C, sehen für den Rücksprung mit Bestimmung des Rückgaberesultats den return()-Befehl vor:

```
if Zahl1 > Zahl2
then return(Zahl1)
else return(Zahl2)
```

Diese Form wird auch vom GNU-Pascalübersetzer akzeptiert.

Verwendet man die Form aus dem Pascal-Standard mit dem Funktionsbezeichner auf der linken Seite, muß man etwas aufpassen, daß man die Funktion nicht aus Versehen auch rechts hinschreibt, wie in dem folgenden (falschen!) Beispiel:

```
program PSigma;
{Funktion mit Vektorargument}
const
   Feldmax =  100;
type
   Elementtyp =  Integer;
   AFeld      = array[1..Feldmax] of Elementtyp;
var
   V : AFeld;
   j : Integer;{Laufvariable}

  function Sigma(F : AFeld; von, bis: Integer):
    Elementtyp;
    {Liefert Summe alle Werte F[von], ..., F[bis]}
  var i :  Integer;
  begin
    Sigma := 0;
    for i := von to bis do
      Sigma := Sigma + F[i]{falsch !!!}
  end {Sigma};
begin
  for j := 1 to 20 do V[j] := j;
    writeln(”Summe 1 bis 10 ist ”, Sigma(V, 1, 10));
end {sigma}.
```

Die Fehlermeldung des Compilers lautet:

```
gpc -o sigma sigma.pas
sigma.pas: In function `Sigma‘:
sigma.pas:19: too few arguments to function `Sigma‘

Compilation exited abnormally with code 1 at Mon Dec 14
17:12:13
```

Daran erkennt man, daß der Compiler das Auftauchen des Bezeichners `Sigma` auf der rechten Seite der Zuweisung als einen (rekursiven) Aufruf der Funktion auffaßt, dem es aber an der richtigen Anzahl von Argumenten mangelt.

Man muß deshalb innerhalb von `Sigma` eine lokale Variable `S` vereinbaren vom Typ `Elementtyp` und den Rumpf der Funktion wie folgt ersetzen.

```
S := 0;
for i := von to bis do
  S := S + F[i];
Sigma := S
```

Als weiteres Beispiel vereinbaren wir eine Funktion `Skalarprodukt`, der wir zwei Vektoren und deren Dimension n übergeben. Den Elementtyp haben wir diesmal auf Real gesetzt.

```
function Skalarprodukt(A, B : AFeld; n: Integer):
  Elementtyp;
var i : Integer;
   sk : Elementtyp;
begin
   sk := 0.0;
   for i := 1 to n do
      sk := sk + A[i] * B[i];
   Skalarprodukt := sk
end {Skalarprodukt};

begin {Hauptprogramm}
  ...
  writeln(”Skalarprodukt [1, 2, ..., n] * [1, 1, ..., 1]
    fuer n = 20 ist ”, Skalarprodukt(V1, V2, 20) );
end {skalar}.
```

Pascal erlaubt als Rückgabetyp nur elementare Werte. Entsprechend ist auch in Turbo Pascal die folgende Funktion nicht möglich.

```
function Vektorsumme(A, B: AFeld; n:Integer): AFeld;
var i : Integer;
   C  :  AFeld;
begin
   for i := 1 to n do
      C[i] := A[i] + B[i];
```

```
    Vektorsumme := C
end {Vektorsumme};
```

Es überrascht wenig, daß der GNU-Compiler diese Konstruktion zuläßt.

8.2 Werte- versus Referenzübergabe

Möchte man ein strukturiertes Ergebnis zurückgeben, muß man sich über den „Wertetransfer“ zwischen Programm und Funktion unterhalten, d.h. die *Parameterübergabe*.

Für das Beispiel mit der Vektorsumme wäre man geneigt, eine Prozedur zu definieren, die den Ergebnisvektor als 3. Parameter erhält:

```
procedure Vektorsumme(A, B, C: AFeld; n: integer);
```

Bei einem Testlauf wird man aber schnell feststellen, daß intern in der Prozedur zwar C richtig aufbereitet wird, das Ergenis aber nicht nach draußen gelangt, d.h. der für C übergebene Vektor hat die ursprünglichen Inhalte.

Der Grund ist, daß A, B und C als **Werteparameter** übergeben werden, d.h. ihre Werte gehen in die Funktion hinein, dort vorgenommene Änderungen schlagen aber nicht nach außen durch. Dies wird sichergestellt, indem die Prozedur (oder Funktion) eine **Kopie der Werte** erhält und in der Prozedur mit dieser Kopie gearbeitet wird.

Will man Ergebnisse zurückgeben und sind dies mehr als **ein** elementarer Wert oder ist es ein strukturierter Wert, dann muß man einen **Variablenparameter** vereinbaren. Dazu stellt man das Schlüsselwort **var** dem Parameter in der Funktions- oder Prozedurvereinbarung voraus. Im Beispiel sähe die Vereinbarung demnach so aus:

```
procedure Vektorsumme(A, B: AFeld; var C: AFeld; n:
integer);
var i: Integer;

begin
  for i := 1 to n do
    C[i] := A[i] + B[i]
end {Vektorsumme};
```

Der Aufruf lautet dann einfach

```
Vektorsumme(V1, V2, V3, xyz);
```

wobei `V1`, `V2` und `V3` Arrays vom Typ `AFeld` und `xyz` ein geeigneter Integer in den Arraygrenzen ist.

Die Übergabe eines Arguments an einen Parameter, der mit **var** vereinbart wurde, wirkt wie die Ersetzung des Parameters durch die Argumentvariable. Jede Wertänderung an diesem Parameter schlägt auf die Variable außen durch. Dies wird erreicht, indem der Compiler die *Adresse der Argumentvariable* als Parameter übergibt und diese für alle Zugriffe auf den Parameter verwendet.

Man spricht in diesem Fall auch von „call-by-reference“ statt „call-by-value“. Andere Programmiersprachen, z.B. Ada, verwenden die klareren Schlüsselworte **in**, **out** und **inout** zur Kennzeichnung der Parameter.

Als weiteres Beispiel betrachten wir ein kleines Programm, das drei Integervariablen a, b, c so umsortiert, daß $a \leq b \leq c$. Intern verwenden wir dazu eine Prozedur `switch(x, y)` die sicherstellt, daß nach dem Aufruf $x \leq y$ gilt.

```
program Sort3;
{sort of example for variable parameter}
var
   a, b, c      :Integer;
procedure Switch(var x, y :  Integer);
var
   aux :  Integer;
begin
   if x > y
      then begin aux := x; x := y; y := aux end
end {Switch};

begin
   readln(a, b, c);
   writeln(a:5, b:5, c:5);
   switch(a, b); {jetzt a <= b}
   switch(b, c); {jetzt c Maximum}
   switch(a, b); {jetzt a <= b <= c}
   writeln(a:5, b:5, c:5)
end {Sort3}.
```

Die Prozedur `switch(x, y)` können wir auf alle Argumente loslassen, die Integervariablen sind, z.B. auf Arrayelemente `A[i]`, wie im folgenden Beispiel gezeigt.

```
program Bubblesort;
{classical n**2 sort, works by bubbling up the maximum}
const
   maxFeld =  1000;
type
   Int =  Integer;
   AFeld = array[1..maxFeld] of Integer;
var
   i, j, n : Int;
   next   : Int;
   M   : AFeld;

procedure Switch(var x, y :  Integer);
{... wie oben ...}

function rand: Int;{selbstgestrickter
Zufallszahlgenerator}
{liefert Pseudozufallszahl 0 - 2**15-1 analog UNIX rand}
{... siehe kapitel 6 Umsatz ...}
begin
  next := 1;
  write(”Wieviele Zahlen sortieren
       (1<n< ”,maxFeld:5,”)? ”);
  readln(n);
  {Werte erzeugen}
  for i := 1 to n do M[i] := rand;
  {jetzt sortieren, indem jeweils das
  Maximum ans Ende kommt}
  for i := n downto 2 do
    for j := 2 to i do
      switch(M[j-1], M[j]);
  {Ausgabe anzeigen}
  for i := 1 to n do
    write(M[i]:6);
  writeln(” basta!”);
end {Bubblesort}.
```

Dagegen wäre ein Aufruf `switch(a + 1, b - c)` nicht zulässig, da als Argument (aktueller Parameter) bei Referenzübergabe immer eine Variable übergeben werden muß. Man macht sich das leicht dadurch klar,

daß ein Ausdruck keine Adresse hat und eine Zuweisung innerhalb von `switch(var x, y: integer)` an Parameter `y`, z.B. `y := aux`, bei Übergabe des Ausdrucks `b - c` gleichwertig wäre mit

```
b - c := a + 1
```

d.h. links vom Zuweisungszeichen stünde ein Ausdruck!

Übung 8–1

Wir verwenden statt `switch(var x, y: integer)` jetzt

```
procedure swap(var A: AFeld; i, j: integer);
var aux: Integer;
begin
  if A[i] > A[j]
  then begin aux := A[i]; A[i] := A[j]; A[j] := aux end
end {swap};
```

Wie muß der Aufruf von `swap` in Bubblesort lauten?

Die Übergabe eines Feldes oder einer großen Matrix mit **var**-Parameter ist im übrigen so viel effizienter (es wird das Kopieren vieler Elemente gespart), daß man Arrays und große Records generell mit *call-per-reference* übergeben sollte. Das setzt natürlich ein gewisses Vertrauen in die Prozeduren und Funktionen voraus, da die übergebenen Felder nicht vor Veränderung geschützt sind.

Als Beispiel betrachte man die Ermittlung des Medians[1] für eine Folge von *n* Zahlen, *n* ungerade.

Wie man sieht, lösen wir die Aufgabe „auf die bequeme Methode“, indem wir unseren Sortieralgorithmus von oben recyclen, hier `bsort()` genannt. In einem sortieren Array ist natürlich der Median das Element auf der mittleren Position.

1. Der Median einer Folge von Zahlen ist der Wert, für den es genausoviele größere wie kleinere Werte gibt. Im allgemeinen verlangt man eine ungerade Anzahl von Werten, ggf. erlaubt man einen mehr auf einer Seite, z.B. bei den kleineren Werten.

```
procedure bsort(var A : AFeld; unten, oben: Integer);
begin
   {A sortieren, indem Elemente vertauscht werden}
   for i := n downto 2 do
      for j := 2 to i do
   switch(A[j-1], A[j]);
end; { bsort }

function Median(var A : AFeld; m: Integer): Integer;
{median berechnen auf die faule Tour: A sortieren und
 mittleres Element nehmen. Nicht effizient, aber easy}
var
   B : AFeld;
   i :  Integer;
begin
   for i := 1 to m do B[i] := A[i];
   bsort(B, 1, m);
   Median := B[m div 2 + 1]
end {Median};

begin
  next := 1;
  write(”Wieviele Werte f. Median
       (1<n< ”, maxFeld:5,”)? ”);
  readln(n);
  {Werte erzeugen}
  for i := 1 to n do M[i] := rand;
  Medianwert :=  Median(M, n);
  writeln(”Medianwert = ”, Medianwert);
  {Werte anzeigen}
  for i := 1 to n do
    write(M[i]:6);
end {Median ermitteln}.
```

Wie man auch sieht, werden die Felder in `Median()` und `bsort()` per **var** übergeben. Innerhalb von `Median()` wird aber eine Kopie des Felds in der benötigten Größe erstellt. Damit wird verhindert, daß das übergebene Orginalfeld umsortiert wird, was in vielen Fällen unerwünscht wäre, etwa wenn es sich um Werte in zeitlicher Reihenfolge handeln würde.

Statt „zu Fuß“ umzukopieren, könnten wir auch in `Median()` eine Werteübergabe verlangen, d.h. das Schlüsselwort **`var`** weglassen. Dann würde der Compiler das Kopieren für uns einbauen.

Generell gelten die folgenden Programmierregeln:

+ Funktionen verändern ihre Argumente nicht!

+ Felder sollte man mittels Variablenparameter übergeben.

Wie gesehen, sind als Parameter auch indizierte Variablen (Elemente von Arrays) zugelassen. Hierzu ist anzumerken, daß die Indexwerte zum Zeitpunkt des Aufrufs ausgewertet werden. Man vergleiche hierzu das folgende Programm, das die Werte 20 und 40 ausgibt.

```
program TrickyIndex;
var
   i : integer;
   X : array[1..4] of integer;

procedure tricky(var a, b :  integer);
begin
   a := a + 1;
   writeln(b)
end; { tricky }

begin {Indexauswertung}
   X[1] := 10; X[2] := 20;
   X[3] := 30; X[4] := 40;
   i := 2;
   tricky(i, X[i]);
   tricky(i, X[i+1]);
end {TrickyIndex}.
```

Wir schließen die einführende Behandlung von Funktionen und Prozeduren mit den Syntaxregeln für die Vereinbarungen ab. Weitergehende Konzepte, wie Funktionsparameter und Rekursion, folgen im nächsten Kapitel.

8.3 Prozedur- und Funktionsvereinbarungen

Prozedur- und Funktionsvereinbarungen stehen im **var** Vereinbarungsteil eines Programms oder einer anderen Prozedur oder Funktion.

Prozedurvereinbarung ::= Prozedurkopf **;** Prozedurrumpf

Prozedurkopf ::= **procedure** Prozedurbezeichner
[formale-Parameterliste]

Prozedurrumpf ::= Block

Block ::= [Vereinbarungsteil] Anweisungsteil
... wie gehabt ...

formale-Parameterliste ::= **(** formaler-P-abschnitt [**;**...] **)**

formaler-P-abschnitt ::= Werte-P-abschnitt |
Variablen-P-abschnitt | Prozedur-P-abschnitt |

Funktions-P-abschnitt

Werte-P-abschnitt ::= Parametergruppe

Variablen-P-abschnitt ::= **var** Parametergruppe

Prozedur-P-abschnitt ::= **procedure** Bezeichnerliste

Funktions-P-abschnitt ::= **function** Parametergruppe

Parametergruppe ::= Bezeichnerliste : Typbezeichner

Funktionsvereinbarung ::= Fkt-Kopf **;** Fkt-Rumpf

Fkt-Kopf ::= **function** Funktionsbezeichner
[formale-Parameterliste] **:** Resultatstyp

Resultatstyp ::= Typbezeichner

Fkt-Rumpf ::= Block

Wichtig ist zunächst nur, daß formale Parameter mit Semikolon getrennt werden, typgleiche aber mit Komma zusammengefaßt werden können. Aktuelle Parameter werden immer mit Komma getrennt.

Beispiel 8–3

```
procedure P(x: integer; y: integer);

procedure P1(x, y: integer); {äquivalent zu P}

procedure Q(var x, y: integer);

procedure R(var x: integer; y: integer);

procedure S(x: integer; var y: integer);

function T(x, y, z: Real): Boolean;

function U(x: Farbe): Trump;
       {Farbe und Trumpf sind Aufzählungstypen}

function Zufall: Real;
  {wie rand in den Beispielen oben,
  aber mit Real Ergebnis}


procedure Tabulate(von, bis, step: Real;
  function f: Real);

procedure Sort(var F: Feldtyp; n: Integer;
  procedure Method);
```

Die letzten zwei Vereinbarungen in den Beispielen oben zeigen Parameter, die vom Typ Funktion, bzw. Prozedur, sind. Darauf wollen wir im nächsten Kapitel eingehen.

9 Weitergehende Prozedurkonzepte

9.1 Funktions- und Prozedurparameter

In den Beispielen für Prozedur- und Funktionsvereinbarungen haben wir zwei Vereinbarungen angegeben, die Prozeduren, bzw. Funktionen als Argumente verlangen. Die Idee ist, zur Laufzeit parametrisiert die richtige Methode zum Einsatz zu bringen.

Im Programm `tabulate.pas` unten haben wir dies verwandt, indem wir Funktionswerte auflisten. Welche Funktion aufgelistet wird, bestimmen wir beim Aufruf (hier: Sinus und Quadratwurzel).

```
program Listen;
{Funktionsparameter ueben}
var
   unten, oben, Schritt: Real;

 procedure Tabulate(von, bis, step : Real;
    function f(x : Real) : Real);
 var
    x : Real;
    i :  integer;
 begin
    x := von;
    while x <= bis do
    begin
       writeln(x:10:3, f(x):10:3);
       x := x + step;
    end {while};
 end; {Tabulate}
```

```
 function MySin(x :  Real): Real;
 begin
    MySin := Sin(x)
 end; { MySin }

function MySqrt(x :  Real): Real;
 begin
    MySqrt := Sqrt(x)
 end;

begin {Listen}
   Tabulate(0.0, 1.0, 0.01, MySin);
   Tabulate(1.0, 50.0, 2.0, MySqrt)
end {Listen}.
```

Bei genauer Durchsicht wird auffallen, daß wir in der Funktionsparameterangabe **`function`** `f(x: Real): Real` den Parameter von `f` mitangegeben haben, er im Kapitel 8 im Beispiel aber weggelassen wurde.

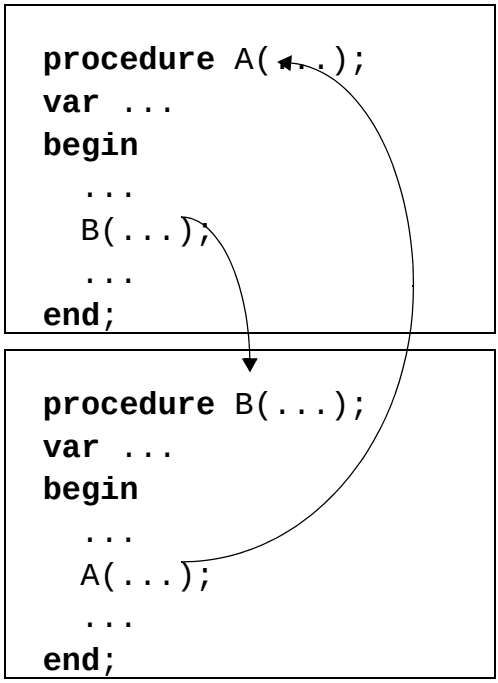
Der Hintergrund ist, daß im ursprünglichen Pascal-Standard bei Funktions- und Prozedurparametern die Argumentanzahl und deren Typ nicht angegeben wurde, nur der Resultatstyp.

Damit ist es aber für den Übersetzer sehr schwierig, die korrekte Verwendung der parametrisierten Funktion, im Beispiel `f`, zu überprüfen. Aus der Zeile `writeln(x: 10: 3, f(x): 10: 3)` könnte er erkennen, daß `f` genau ein Argument vom Typ `Real` verlangt. Stünde woanders `f(x, y)` müßte er erkennen, daß dies ein Widerspruch zum Aufruf `f(x)` ist. Die gesammelte Erkenntnis muß dann ins Hauptprogramm übernommen werden und mit dem Aufruf, hier `Tabulate(a, b, c, MySin)`, dahingehend abgeglichen werden, daß `MySin` genau die richtige Anzahl und die passenden Typen von Argumenten hat.

Dies läßt sich viel leichter und genauer machen, wenn schon in der formalen Parameterangabe einer Funktion oder Prozedur Anzahl und Typ der Parameter bekanntgegeben wird. Deshalb unterstützt (und verlangt) der erweiterte Standard diese Form.

9.2 Die forward-Direktive

Für Prozeduren und Funktionen gilt, wie für alle anderen Bezeichner auch, daß sie deklariert sein müssen, bevor man sie einsetzen kann. Dies kann aber im Falle eines wechselseitigen (zirkulären) Aufrufs in der abgebildeten Art nicht gewährleistet werden. Als Ausweg bietet Pascal die *forward-Direktive* an. Die im Programmtext zuerst deklarierte Funktion oder Prozedur wird ohne Rumpf, aber stattdessen mit dem Schlüsselwort **`forward`** vereinbart. Aus den genannten Parametern des Kopfs kann auf die korrekte Aufrufform geschlossen werden. Weiter unten wird dann die Vereinbarung wiederholt, diesmal mit Rumpf, aber ohne Parameterliste im Kopf [1].



Das folgende Programm zeigt eine solche Konstruktion.

1. der GNU-Compiler stört sich nicht an der Wiederholung der Parameter

```
program Vorwitzig;
var
   k :  Integer;

procedure Runter(var i :  Integer); forward;

 procedure Rauf(var i : Integer);
 begin
    write(i: 3);
    if i > 0 then begin i := i * 2; Runter(i) end
 end; { Rauf }

 procedure Runter(var i : Integer);{TP: ohne
Parameterliste!}
 begin
   write(i: 3);
   i := i div 3;
   Rauf(i)
 end { Runter };

begin {JoJo}
   write("JoJo Startwert eingeben: ");
   readln(k);
   Rauf(k);
   writeln(k)
end {Vorwitzig}.
```

Ein Ablauf des Programm sieht wie folgt aus:

```
./vorwitzig
JoJo Startwert eingeben: 12
 12 24  8 16  5 10  3  6  2  4  1  2  0         0
```

Wechselseitige Aufrufe der obigen Art bezeichnet man auch als *Korouti-nenkonzept*, d.h. zwei oder mehr Prozeduren „spielen sich gegenseitig den Ball zu“, etwa in einer Simulation, bei der eine Prozedur ein Ereignis produziert und die andere es konsumiert.

Gleichzeitig haben wir es mit einer *indirekten Rekursion* zu tun, d.h. *A* ruft sich selbst über den Umweg *B* auf. Direkte Rekursion, d.h. der Aufruf von *A* in *A* selbst, wird unten gezeigt.

Die forward-Direktive ist auch ein Beispiel für die getrennte Deklaration und Definition von Prozeduren und Funktionen. Dies ist Grundlage

des Modulkonzepts, bei dem Prozeduren und Funktionen in einem *Interface-Teil* nach außen mit Bezeichnern und Parameterlisten bekanntgemacht werden, ihre inneren Strukturen aber im sog. *Implementierungs-Teil* versteckt werden.

```
unit Stapel
  interface
  type
    Stacktype = record ...;
  procedure CreateStack(var S: Stacktype);
  function IsEmpty(var S: Stacktype): Boolean;
  implementation
  type
    ...;
  procedure CreateStack(var S: Stacktype);
  var ...
  begin
    ...
  end;
  function IsEmpty(var S: Stacktype): Boolean;
  begin
    IsEmpty := (S.Top = 0)
  end;
```

Turbo Pascal unterstützen dies durch ein einfaches, aber wirkungsvolles Unit-Konzept, bei dem der Schnittstellenteil mit dem Schlüsselwort **`interface`** und der eigentliche Implementierungsteil mit **`implementation`** eingeleitet wird. Will man das Modul in anderen Modulen oder im Hauptprogramm verwenden, muß man es mit der Uses-Direktive einbinden, im Beispiel unten durch **`Uses`** `Stapel`.

Die Idee hierzu geht bis in die Sechziger Jahre zurück, als man unter dem Begriff *abstrakte Datentypen* die Forderung erhob, den Gebrauch einer Methode von ihrer Realisierung zu trennen und letztere intern zu verstecken.

Dies führt in der Konsequenz zur Funktionsbibliothek, bei der eine ganze Klasse von Routinen angeboten wird, mit der sich gewisse Aufgaben leichter lösen lassen, etwa die Datenbank- oder Fensteroberflächen-

programmierung. Die Routinen liegen dann bei kommerziellen Paketen häufig in bereits übersetzter (und damit „geheimer“) Form vor, ihr Gebrauch ergibt sich aus der Schnittstellenbekanntmachung. Das fertige ausführbare Programm entsteht durch das Linken (Binden) aller Teile.

Auf das Unit-Konzept von Turbo Pascal wollen wir nicht eingehen. Es sollte aber erwähnt werden, daß die forward-Direktive für im Interface genannte Prozeduren und Funktionen nicht benötigt wird und auch nicht zulässig ist.

9.3 Gültigkeitsbereiche

Prozeduren und Funktionen etablieren neue Gültigkeitsbereiche. Wie in den Beispielen oben zu sehen war, können wir dadurch problemlos lokale Variablen, z.B. mehrfach die Variable *i*, einführen, obwohl ein Bezeichner gleichen Namens auch im Hauptprogramm existiert.

Dies ist möglich, weil alle deklarierten Bezeichner einer Funktion oder Prozedur, sowie alle Parameter als **lokal** zu dieser Funktion oder Prozedur betrachtet werden. Damit sind sie nur in diesem Rumpf sowie in allen innenliegenden Prozeduren und Funktionen sichtbar.

Lokale Bezeichner überdecken gleichnamige globale Bezeichner und machen letztere unsichtbar, d.h. jede Referenz bezieht sich auf den lokalen Vertreter.

Man beachte auch, daß im Beispiel unten in `ProcC` die Zuweisung `c := a + b` zulässig wäre, genauso wie `b := a + 1` in `ProcB`, natürlich auch Zuweisungen an die globalen Variablen, also a := c + b in ProcC, usw. Allerdings wären dies Verwendungen und sogar Veränderungen von globalen Variablen. Dies gilt zu Recht als schlechter Programmierstil, weil derartige Nebeneffekte tückisch sind. Vielmehr sollte man alle verwendeten Variablen in der Parameterliste aufführen. Sind es recht viele, kann man sie ggf. in einem Record zusammenfassen und nur diesen Record, bzw. einen Zeiger auf den Record, übergeben.

```
program ProgA;
var a, ...
```

global zu ProcB und ProcC

```
  procedure ProcB;
  var b, ...
```

lokal zu ProcB, global zu ProcC

```
      procedure ProcC;
      var c, ...
      begin
        c:= a + b
      end;
  begin
    b := 2 * a
    ProcC;
    ...
  end;

begin {ProgA}
  ...
  a := ...
  ProcB;
  ...
end {ProgA}.
```

lokal zu ProcC

Funktionen mit Nebeneffekten können auch zu Programmresultaten führen, die je nach Compiler und je nach Optionseinstellung variieren.

```
program Seiteneffekt;
{Fkt oder Prozedur mit aktivem Gebrauch globaler
Variable}
var
   g: Integer;

 function E: Integer;
 begin
    g := g + 1;  {globale Variable - pfui Teufel}
    E := g
 end; {E}

 begin {Seiteneffekt}
  g := 1;
```

```
    write(g); writeln(E);
    writeln(g, E)
end {Seiteneffekt}.
```

Wie man an der Ausgabe unten sieht, hat im zweiten Schreibbefehl g bereits den erhöhten Wert, den es in E als Seiteneffekt bekommt. Häufig variieren Ergebnisse in Additionen und Multiplikationen, weil der Compiler bei kommutativen Operationen die Freiheit hat, die Operanden zu vertauschen, um die Codeerzeugung zu optimieren.

```
./effekt
          1         2
          3         3
```

Aus diesen und anderen Gründen sollte man auf die Verwendung globaler Variablen, speziell bei aktivem (veränderndem) Gebrauch, verzichten.

9.4 Prozeduren als Strukturierungsmittel

Mit Prozeduren und Funktionen lassen sich mehrfach verwendete Programmteile herausfaktorisieren. Dadurch werden Programme kürzer.

Es lohnt sich aber auch, Teile in Prozeduren und Funktionen zusammenzufassen, wenn diese nur einmal verwandt werden. Programme werden viel übersichtlicher, wenn einzelne Aufgaben und ihre komplexen Details in Unterroutinen verpackt werden. Als Faustregel gilt, daß eine einzelne Routine nie mehr als 1 - 2 DIN A4 Seiten Code enthalten sollte.

Ferner lohnt sich häufig der Aufwand, die Routinen besonders sauber und allgemein einsetzbar (Fachausdruck: generisch) zu programmieren und zu dokumentieren. Derartige Bibliotheken von Routinen lassen sich später für ähnliche Aufgaben wiederverwenden.

9.5 Rekursion

Rekursive Definitionen sind häufig besonders elegant, man denke etwa an die Definition der Fibonacci-Zahlen:

$$\text{fib}(n) \text{ ist 1 für } n \leq 2, \text{ sonst } \text{fib}(n) = \text{fib}(n\text{-}1) + \text{fib}(n\text{-}2)$$

Die ersten sieben Fibonacci-Zahlen sind demnach 1, 1, 2, 3, 5, 8, 13. Ein englischer Programmiererspruch lautet deshalb: „to iterate is human, to recurse divine.“[1]

Für mathematisch ungeübte Menschen hat die Rekursion aber etwas unheimliches, weil sie paradox wirkt: etwas wird erklärt, indem man auf den Gegenstand der Erklärung selbst zurückgreift.

Für den Rechner ist die Rekursion völlig undramatisch: der Aufruf einer Prozedur bedeutet die Abspeicherung einer Rücksprungadresse auf einem Stapel und die Verzweigung zu der Startadresse der Routine. Da für den Rechner alle Adressen gleichwertig sind, ist die Verzweigung zur eigenen Startadresse völlig normal.

Wichtig ist nur, daß der Vorgang des rekursiven Aufrufs irgendwann einmal abbricht, wie dies an der Realisierung der Funktion `fibonacci` abzulesen ist (vgl. Programm `fibo.pas` in Kap9).

```
function fibonacci(n: Integer): Integer;
begin
   if n <= 2 then
      fibonacci := 1
   else
      fibonacci := fibonacci(n-1) + fibonacci(n-2)
end; {fibonacci}
```

Allerdings ist die Berechnung unnötig aufwendig, wie man erkennt, wenn man die Aufrufe mitprotokolliert.

1. im Deutschen heißt es weniger prosaisch: „rekursiv geht meistens schief.“

```
$ ./fibo2
Die wievielte Fibonaccizahl? 7
Aufruf mit:   7
Aufruf mit:   6
Aufruf mit:   5
Aufruf mit:   4
Aufruf mit:   3
Aufruf mit:   2
Aufruf mit:   1
Aufruf mit:   2
Aufruf mit:   3
Aufruf mit:   2
Aufruf mit:   1
Aufruf mit:   4
Aufruf mit:   3
Aufruf mit:   2
Aufruf mit:   1
Aufruf mit:   2
Aufruf mit:   5
Aufruf mit:   4
Aufruf mit:   3
Aufruf mit:   2
Aufruf mit:   1
Aufruf mit:   2
Aufruf mit:   3
Aufruf mit:   2
Aufruf mit:   1
        13
```

Offensichtlich werden viele Werte mehrfach (wie oft?) berechnet. Eine viel einfachere iterative Lösung sieht wie folgt aus:

```
program Fibonaccizahlen;
{Iterative Lösung}
var
   m, i,
   fn_1, fn_2, f : Integer;
begin {Fibonaccizahlen}
  write(”Die wievielte Fibonaccizahl? ”); readln(m);
  fn_1 := 1; fn_2 := 1;
  if m <= 2 then f := 1
  else
    for i := 3 to m do
    begin
     f := fn_1 + fn_2;
```

```
      fn_2 := fn_1;
      fn_1 := f
     end;
   writeln(f);
 end {Fibonaccizahlen}.
```

Noch deutlicher wird dies bei der Fakultätsfunktion *n*!, deren rekursive Definition fakt(*n*) = *n* * fakt(*n*-1) mit fakt(1) = 1 sich trivial iterativ aufschreiben läßt:

```
f := 1; for i := 2 to n do f := f * i.
```

An der rekursiven Funktion läßt sich ablesen, daß es sich um eine sog. *lineare Rekursion* handelt, d.h. innerhalb von `fakt` wird `fakt` nur einmal aufgerufen.

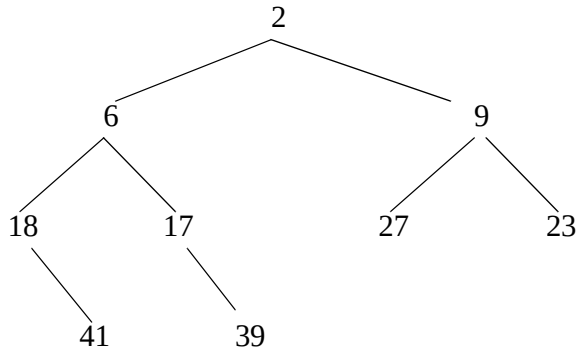
```
function fakt(n: Integer): Integer;
begin
  if n <= 1 then fakt := 1
  else fakt := n * fakt(n-1)
end;
```

Der Nutzwert einer solchen Rekursion ist entsprechend gering. Im Falle von `fakt` ist allerdings mit der Rekursion kein unnötiger Aufwand verbunden, denn `fakt` wird insgesamt nur *n*-mal aufgerufen.

Wichtiger sind nichtlineare Rekursionen, etwa wenn eine Menge *M* von ganzen Zahlen rekursiv wie folgt definiert ist:

1. $2 \in M$
2. Ist $x \in M$, dann auch $3x \in M$ und $2x + 5 \in M$
3. keine anderen Elemente sind in *M*.

Die Berechnung der Elemente von *M*, die kleiner 50 sind, kann man baumartig wie folgt vornehmen.



Die Verwaltung der noch nicht abgearbeiteten Werte „von Hand“ ist hier aufwendig. In diesem Fall ist die Rekursion die ideale Implementierung.

```
program Rekursivmenge;
{Rekursiven Aufruf ueben}
var
   grenze : integer;

procedure M(x, limit : integer );
   begin
      writeln(x);
      if 3*x < limit then M(3*x, limit);
      if 2*x + 5 < limit then M(2*x + 5, limit)
   end {M};

begin {Menge}
   write(”Werte bis zu welcher Grenze? ”);
   readln(grenze);
   M(2, grenze);
 end {Menge}.
```

Durch Programme mit rekursiven Prozeduren läßt sich auch ein wichtiges Paradigma der Algorithmenentwicklung verwirklichen: Teile und herrsche (divide and conquer).

Ein Beispiele dafür wären die Maximumssuche in einer Topographie: Teile den Bereich in zwei oder mehr Teilbereiche und bestimme (rekursiv) die lokalen Maxima; eines dieser gemeldeten Maxima ist dann auch

das globale Maximum. Ein anderes Beispiel wäre Sortieren durch Teilen (Quicksort und Varianten) oder Sortieren durch Verschmelzen (Teile Folge auf, sortiere Teilfolgen, verschmelze diese).

Wir beenden hier die Behandlung rekursiver Prozeduren mit dem wohl bekanntesten Beispiel, den *Türmen von Hanoi*.

Einer Sage zufolge sind im laotischen Dschungel Mönche damit beschäftigt, 50 goldene Scheiben unterschiedlichen Durchmessers von einer Stange auf die benachbarte unter Zuhilfenahme einer weiteren Stange umzuschichten. Dabei darf nie eine größere Scheibe auf einer kleineren zu liegen kommen. Wenn die Mönche diese Aufgabe beendet haben, wird das Ende der Welt erreicht sein.

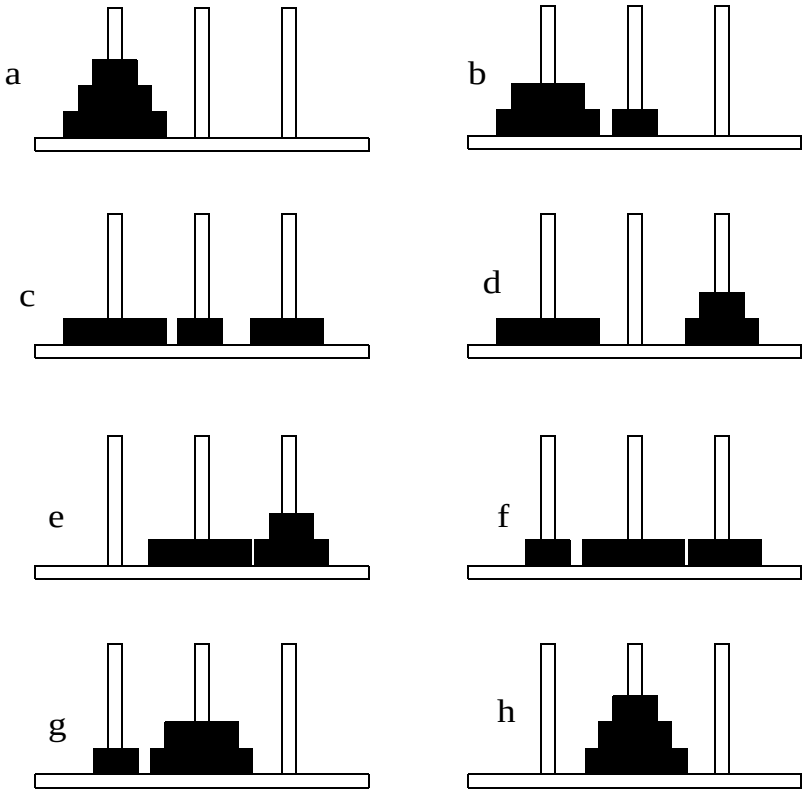


Abb. 9–1

Wenn wir die Stangen mit den Ziffern 1, 2 und 3 bezeichnen, dann besteht die Lösung für *n* Scheiben darin, *n*-1 Scheiben von 1 nach 3 zu transportieren, die größte Scheibe von 1 nach 2 zu bewegen und dann wieder die *n*-1 Scheiben von 3 nach 2 zu schichten. Damit hat sich das Problem für *n* Scheiben reduziert zu einem für *n*-1 Scheiben. Parametrisiert man noch die Stangen, hier durch die Parameter `von`, `nach`, `ueber`, wird die Lösung offensichtlich.

Im folgenden Programm wird die Scheibenbewegung durch eine Druckzeile dargestellt. Schöner sind Lösungen, bei denen die Scheiben graphisch umgeschichtet werden. Sofern Sie eine solche programmieren wollen, achten Sie darauf, daß die Scheibe zuerst am alten Ort gelöscht wird und dann am neuen abgelegt wird, sonst erscheint die Lösung visuell „unglaubwürdig“. Besonders elegant sind Lösungen, bei denen man die Bewegung der Scheibe sieht.

```
program TuermevonHanoi;
{Klassische Rekursion}
var
   anzahl : integer;{Anzahl der Scheiben}

procedure Scheibe(von, nach:  Integer);
begin
   writeln(„Scheibe von „, von:2,“ nach „, nach:2);
end {Scheibe};

procedure Schichten(n, von, nach ,ueber: integer );
begin
  if n > 0
  then begin
   Schichten(n-1, von, ueber, nach);
   Scheibe(von, nach);
   Schichten(n-1, ueber, nach, von);
   end
end {Schichten};

begin {Tuerme von Hanoi}
   write(”Wieviele Scheiben? ”);
   readln(anzahl);
   Schichten(anzahl, 1, 2, 3);
   writeln(”Fertig”);
 end {Tuerme von Hanoi}.
```

Zwei Hinweise und eine Überlegung seien angefügt.

- Wo sich die Sage abspielt, ist umstritten. Ottmann/Widmayer siedeln sie in Hinterindien an (Türme von Brahma). Generell ist die Aufgabe als *Towers of Hanoi* bekannt.
- Es gibt auch eine iterative Lösung: man denkt sich die Stangen kreisförmig angeordnet und bewegt immer die zuletzt nicht angefaßte Scheibe im Uhrzeigersinn (oder generell immer gegen den Uhrzeigersinn) auf die nächste zulässige Position. Abhängig von der Anzahl der Scheiben und der Drehrichtung kann allerdings der Endstapel auf Stange 3 statt 2 landen, ein (wie?) behebbarer Schönheitsfehler.
- Wenn jeder Scheibentransport 1 Sekunde dauert und die Mönche vor tausend Jahren angefangen haben, wann ist mit dem Ende der Welt zu rechnen?

10 Zeiger und dynamische Strukturen

Pascal-Programme sind Programme für sog. *Universalrechner* (im Gegensatz zu Spezialrechnern, etwa einem Schachcomputer, einem Verkehrsleitrechner, usw.), mit denen sich im Prinzip jedes algorithmisch lösbare Problem berechnen läßt. Alle heute verbreiteten Rechner werden ferner als sog. *von-Neumann-Rechner* bezeichnet, bei denen Programm(e) und Daten gleichberechtigt (und nicht unterscheidbar) im Hauptspeicher abgelegt werden. Von-Neumann-Rechner heißen deshalb auch im Englischen *stored program computers*.

Zur Verarbeitung wird eine Instruktion in den Prozessor geladen, diese fordert die notwendigen Daten (z.B. zwei ganze Zahlen, die addiert werden sollen) an, verknüpft diese und legt das Ergebnis wieder in den Hauptspeicher ab, bzw. hält es als Zwischenergebnis zunächst im Prozessor.

Die Unterscheidung zwischen Programm und Daten ist dabei häufig willkürlich. Ein keineswegs exotisches Beispiel für ein Programm in der Rolle von Daten ist die Übersetzung eines Programms, d.h. das Übersetzerprogramm liest als Eingabedaten ein anderes Programm, den Quellcode, und liefert als Ausgabe wieder ein Programm, nämlich den Zielcode in Maschinensprache.

10.1 Adressen als Werte

Eine ähnliche Situation hat man bei der Unterscheidung von Werten und Adressen, etwa wenn der Wert 17 einer Variablen *i* im Hauptspeicher unter der Adresse #342A77F abgelegt wird. Andere Instruktionen eines Maschinenprogramms betrachten die Adresse als Wert und speichern sie z.B. in einer Variablen *p*.

Abb. 10–1

Wir sprechen dann von *indirekter Adressierung*: um an *i* heranzukommen, müssen wir den Inhalt von *p* als Adresse interpretieren, zu dieser Adresse gehen und den dort gespeicherten Wert holen. Diesen Vorgang nennt man *Dereferenzieren*.

Ein vielleicht anschaulicheres Beispiel wäre ein kleines Schließfach, in dem sich ein Schlüssel und die Nummer eines anderen Schließfachs befindet. Wer den Inhalt des letzteren Fachs will, muß über das kleine Fach mit einem Indirektionsschritt gehen. Ein Vorteil einer solchen Regelung wäre z.B., daß die Größe und der Ort des wirklichen Fachs variieren können, je nach Anforderung und Verfügbarkeit.

Pascal unterstützt dieses Konzept mittels des Datentyps *Zeiger* (engl. *pointer*). Als Beispiel wird eine Variable *p* als Zeiger auf Integer-Variablen vereinbart.

```
var
  i: Integer;
  p: ↑integer;
```

In Turbo Pascal und im GNU Pascal kann man jetzt schreiben

```
...
i := 17;
p := @i;
```

```
p^ := p^ + 1;
writeln("Der Wert von i ist: ", i);
{schreibt den Wert i = 18}
...
```

Dabei ist @ der Adressoperator, in TP gleichwertig mit dem Funktionsaufruf `p := Addr(i)`, der *p* die Adresse von *i* als Wert zuweist. Mit *p^* dereferenziert man *p*, d.h. man sagt „nimm nicht den Wert von *p* selbst, sondern den Wert, auf den *p* zeigt."

Da im Beispiel *p* momentan auf *i* zeigt, wird dessen Wert, hier 17, genommen, um Eins erhöht und wieder *i* zugewiesen.

10.2 Das Prinzip des typisierten Zeigers

Wichtig ist, daß der Zeiger *p* „weiß", auf welchen Typ von Objekt er zeigt. Zunächst enthält *p* nämlich nur eine Byteadresse. Erst mit der Typinformation, in unserem Beispiel mit der Vereinbarung „*p* ist Zeiger auf Integer", kann der Übersetzer den richtigen Code generieren, der dafür sorgt, daß eine typabhängige Anzahl von Bytes ab der gelieferten Adresse in einer typabhängigen Art als Wert interpretiert werden, z.B. die nächsten 4 Bytes als Binärwert im Zweierkomplement für einen Integer. Man nennt solche Zeiger typisiert und Standard Pascal erlaubt nur solche[1].

Turbo Pascal kennt auch einen generischen Zeigertyp, der mit dem Standardbezeichner `pointer` vereinbart wird, also z.B. mit

```
var
  i, j : Integer;
  myptr : pointer;
```

Aber selbst dann bewirken die Anweisungen

```
myptr := @i;
j := myptr^;
```

1. Standard Pascal verlangt sogar generell, daß alle Typprüfungen zur Übersetzungszeit möglich sind. Man sagt daher, Pascal gehöre zur Klasse der starktypisierten Sprachen (strongly typed programming languages).

einen Übersetzerfehler in der zweiten Zeile, da der Übersetzer nicht weiß, wie er den typlosen Pointer `myptr` dereferenzieren soll. Dies könnte man zwar wieder reparieren, indem man in TP durch eine Cast-Operation (explizite Typanpassung) die Dereferenzierung steuert, über diese Art der Programmierung geben wir aber unten unsere unfreundliche Meinung ab.

10.3 Kopieren versus gemeinsame Referenz

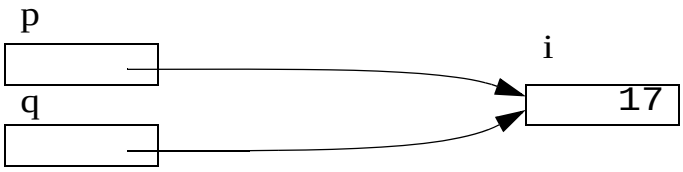
Vereinbart man eine weitere Zeigervariable, z.B. *q*, und eine weitere Integer-Variable, z.B. *j*,

```
p, q : ^Integer;
i, j : Integer;
```

dann kann man durch

```
p := @i;
q := p;
```

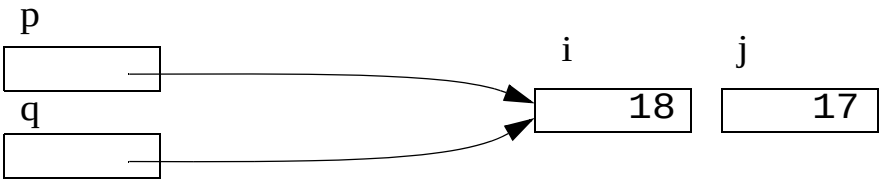
mit *p* **und** *q* auf *i* zeigen.



Schreibt man jetzt

```
i := 17;
j := i;
i := i + 1;
```

dann hat *i* den Wert 18, *j* als alte Kopie von *i* weiterhin den Wert 17, und die Zeigervariablen *p* und *q* zeigen beide auf den aktuellen Wert von *i*, also auf 18.



Genauso könnten wir sowohl mit `p^ := 19` als auch mit `q^ := 19` den Wert von *i* auf 19 ändern. Man erkennt daran, daß es qualitativ etwas anderes ist, ob ein Wert über zwei oder mehr (Alias-)Namen angesprochen werden kann, oder ob nur eine oder mehrere Kopien des Werts einer Variablen existieren.

Das Beispiel zeigt auch die Gefahren von Zeigern. Zeigervariablen bleiben an ein Objekt gebunden, solange nicht ausdrücklich die Adresse eines anderen Objekts zugewiesen wird. Damit zeigt der Zeiger immer auf die aktuellen Werte des Objekts, was meist erwünscht ist.

Wird allerdings an einer Stelle der Wert des Objekts geändert, im Beispiel oben die Variable *i*, gibt es keinen „Weckmechanismus“ oder etwas ähnliches, der *p* und *q* „benachrichtigt“, daß sich bei ihrem Zielobjekt etwas getan hat. Verschwindet gar das Objekt, auf das *p* und *q* gezeigt haben, weisen die Zeiger „ins Leere“, man springt im Englischen von einem *dangling pointer*.

10.4 Zeiger ins Nichts und die Konstante `nil`

Eine der häufigsten Ursachen von schwerwiegenden Programmabstürzen sind solche Zeiger, die ins Nirwana zeigen, z.B. als Folge einer fehlenden Initialisierung. Dabei ist zu beachten: eine vergessene Initialisierung einer einfachen Integer-Variablen, z.B. *i*, liefert bei eine Referenz, z.B. in der Zuweisung

```
k := i
```

an *k* einen beliebigen, unvorhersagbaren Wert ab, der aber im Wertebereich von *i* und *k* liegt. In der Regel produziert das Programm Unsinn, wird aber selten sofort abstürzen.

Anders bei der Referenz

```
k := p^;
```

wenn die Zeigervariable *p* nicht initialisiert wurde. Liegt die in *p* zufällig gespeicherte Adresse in einem zugänglichen Bereich des Hauptspeichers, werden die dort gespeicherten Werte, also in der Regel irgendwelcher

Unsinn, geholt - siehe oben. Liegt die Adresse aber in einem geschützten Bereich, wird meist eine Speicherverletzung signalisiert und das Programm bricht ab.

Steht `p^` auf der linken Seite einer Zuweisung, kann man noch „lustigere“ Effekte produzieren. Jetzt wird ein Wert in eine zufällige Speicheradresse gespeichert. Meist zerschießt man sich damit sein Programm. Auf Rechnern mit schlecht implementiertem Speicherschutz (typisch PCs mit einem Betriebssystem aus Redmond) kann man auch schon mal ins Allerheiligste schreiben und den Rechner so lahmlegen, daß nur noch der Griff zum Stecker hilft.

Recht häufig passiert auch die Referenz einer Zeigervariablen, die mit der Zeigerkonstanten **`nil`** initialisiert wurde. Die Konstante **`nil`** steht für einen nullwertigen Zeiger, d.h. einen Zeiger, der auf nichts zeigt. Es ist die einzige Konstanten-zuweisung, die an eine Pointervariable, z.B. einen Zeiger *p*, mittels

```
p := nil;
```

möglich ist. Ob ein Zeiger den Wert **`nil`** hat, kann man abfragen:

```
if p = nil then {p zeigt auf nichts} ... else ...
```

Je nach Programmzusammenhang sollte man diesen Test einer Dereferenzierung einer Zeigervariablen vorausschicken, denn wenn diese auf **`nil`** steht, löst der Zugriff in jedem Fall einen Programmabbruch aus.

Zuletzt noch eine recht subtile Form, bei der ein Zeiger *p* ebenfalls ins Nirwana zeigt:

```
program Zeiger;
type
  Ptype = ^Integer;
var
  i: Integer;
  p: Ptype;

  function Wahnsinn: Ptype;
  var lokalesi: Integer;
  begin
    lokalesi := 17;
```

```
    Wahnsinn := @lokalesi;
  end;

begin
  i := 17;
  p := @i;
  p^ := p^ + 1;
  writeln("Der Wert von i ist ", i:3);
  p := Wahnsinn;
  writeln("Der wahnsinnige Wert ist ", p^ : 3);
end.
```

Das Problem hier ist, daß *p* auf eine Variable zeigt, die nur solange existiert, wie die Funktion `Wahnsinn` existiert, da sie lokal zur Funktion vereinbart ist. Lokale Variablen einer Funktion oder Prozedur kommen auf einen Stapel und werden zum Überschreiben freigegeben, wenn das Programm aus der Funktion zurückgekehrt.

Das Programm oben läuft trotzdem und wird in der Regel die Werte 18 und 17 liefern, weil *p* noch auf den nicht wieder überschriebenen Bereich des Stapels zeigt. In einem längeren Programm, speziell einem mit vielen Aufrufen, wird *p* aber einen unsinnigen Wert liefern oder das Programm „macht den Abflug", weil dieser Stapelbereich als nicht gültig erkannt wird.

Es ist der Alptraum eines jeden Programmierers, Programmfehler in Programmen finden zu müssen, die einen liberalen Umgang mit Pointern betreiben und womöglich Konstruktionen der obigen Art enthalten.

10.5 Dynamische Strukturen

Standard Pascal läßt den Adreßoperator @ (bzw. `Addr()`) nicht zu. Die Methode, eine Zeigervariable auf ein Objekt zeigen zu lassen, ist das Objekt mit `new(p)` anzulegen, wobei p ein typisierter Zeiger ist, der dann auf das Objekt zeigt. Aus der Typisierung weiß der Compiler, welche Art von Objekt erzeugt werden soll.

```
type
  IntPtr = ^Integer;
  RealPtr = ^Real;
  KundenPtr = ^Kundentyp;{Zeiger auf Kundenrecords}
```

```
  Kundentyp = record
    KdNr: Integer;{Kundennummer als Int?}
    Name : String(30);{Kundenname}
    EndVerbraucher : Boolean;
         {true gdw Endverbraucher, sonst Haendler}
    next: KundenPtr{Zeiger auf naechsten Kunden}
  end {Kundentyp};
  ...
var
  p : IntPtr;
  r : RealPtr;
  KP, EndP : KundenPtr;
  ...
begin
  ...
  new(p);
  pˆ := 17;
  new(r);
  rˆ := 3.14;
  new(KP);
  with KPˆ do
  begin
    KdNr := 4711;
    Name := “Lieschen Mueller“;
    EndVerbraucher := true;
    new(next);
  end;
  EndP := KPˆ.next;
  EndPˆ.KdNr := 0;
  EndPˆ.Name := ““;
  EndPˆ.next := nil;
  ...
end.
```

Die oben geschaffenen Objekte lassen sich wie folgt bildlich darstellen.

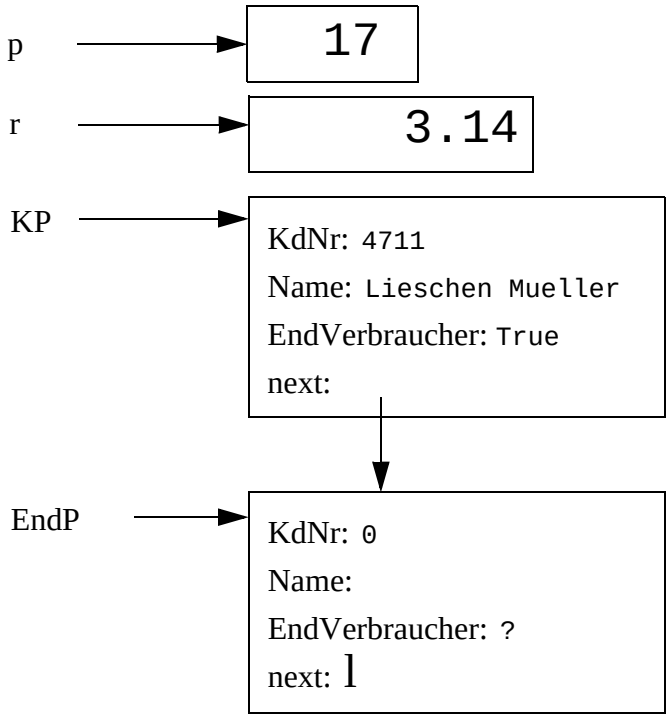


Abb. 10–2

Mit `new(p)` angelegte Strukturen lassen sich mit `dispose(p)` auch wieder aus dem Speicher löschen. So angelegte Objekte nennt man daher auch *dynamische Objekte*. Ihr Speicherplatz ist nicht der normale Bereich für Variablen und Konstanten, auch nicht der Stapelbereich für lokale Variablen und Parameter von Prozeduren und Funktionen, sondern der sog. *Heap*. Darunter versteht man im Zusammenhang mit Speicherverwaltung einen zusammenhängenden Hauptspeicherbereich aus dem Speicherplatz für Objekte variabler Größe vergeben wird. Werden diese gelöscht, wird der Platz wieder freigegeben und ggf. mit benachbarten freien Stücken verschmolzen, um zu größeren freien Bereichen zu kommen. Die Buchführung über freie und belegte Fragmente übernimmt die sog. *Heapverwaltung*. Meist kommen in den Heap auch die Strings ohne explizite Längenangabe, da diese je nach zugewiesenem Wert variable Speicherplatzanforderungen haben.

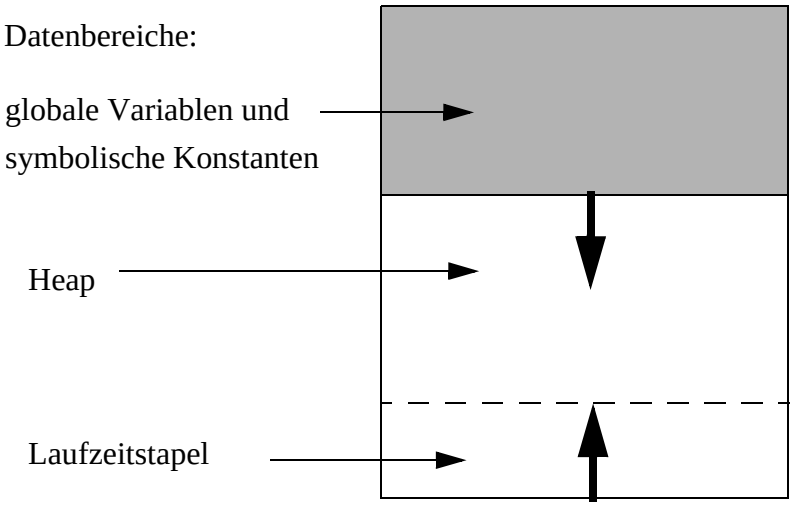


Abb. 10–3

Wie die Abb. 10–3 zeigt, läßt man den Heap gegen den Laufzeitstapel wachsen, bis diese sich treffen und damit den Speicherplatz aufgebraucht haben. Damit ist man flexibel, da der genaue Bedarf sich (dynamisch) erst durch den speziellen Programmablauf ergibt.

Turbo Pascal erlaubt zusätzlich, Speicherbereiche gewünschter Größe im Heap mittels `GetMem()` anzufordern und diese Bereiche durch `Release()` wieder freizugeben. Der Pascal Standard verbietet diesen aus der Programmiersprache C (`malloc` und `free`) bekannten Gebrauch.

10.6 Einschränkungen in Standard Pascal

Um die schlimmsten Folgen einer zu maschinennahen Programmierung zu vermeiden, schränkt der Pascal Standard den Gebrauch von Zeigern wie folgt ein:

- Der Adreßoperator ist nicht zugelassen.
- Nur über den Systemaufruf `new(p)` wird ein neues Objekt erzeugt, auf das `p` zeigt. Der Typ des Objekts ergibt sich aus dem

Zeigertyp. Eigenhändige Speicheranforderungen beliebiger Größe mittels `GetMem()` sind nicht möglich.

- Mit `new(p)` angelegte Objekte können mit `dispose(p)` (und nur mit `dispose()`, nicht mit `FreeMem()`) wieder aus dem Speicher gelöscht werden.
- Jeder Zeiger muß typisiert sein und kann nur auf solche Objekte zeigen, für die er vereinbart wurde; der generische (auf alle Objekte passende) Zeigertyp `pointer` ist verboten.
- Man kann Zeigerwerte vergleichen auf Gleichheit und Ungleichheit, nicht auf `<`, `<=`, usw. Ferner kann man einen Zeigerwert mit **`nil`** vergleichen, andere Konstanten sind nicht zulässig und können einer Pointervariablen auch nicht explizit zugewiesen werden.
- Es ist keine Arithmetik mit Zeigern erlaubt, d.h. die Zuweisung `p := p + 4`, mit der *p* um 4 Bytes (oder 4 * *x* Bytes, wobei *x* = Speicherplatzbedarf des Wertetyps auf den *p* zeigen darf) weitergesetzt werden soll, ist verboten[1].

Generell kann man vor trickreicher Pointerprogrammierung nur warnen. Besonders gefährlich sind Programme, die Annahmen über Größe und Darstellung von Maschinenadressen machen, denn die Computerrevolution hat alle „schlauen“ Konstruktionen altaussehen lassen; man denke an:

- die unselige Intel 8088 Segmentierung mit 64 KB
- die diversen MS DOS Grenzen und HiMem Klimmzüge
- die Motorola 68000 und IBM /360 Grenzen mit 24 bit Adressen
- die heutigen Forderung nach 48 oder 64 bit Adressen, usw.

10.7 Anwendungen

Als Beispiel einer sinnvollen Anwendung dynamischer Objekte wollen wir auf die oben angedeutete Kette von Kunden-Records zurückkommen. Die Kunden (Records vom Typ `Kunde`) werden in einer *Warteschlange*

1. Es überrascht wenig, daß der GNU Compiler `gpc` dies schluckt!

verwaltet. Zur Veranschaulichung denke man an Patienten in einem Wartezimmer.

Eine Warteschlange ist dabei definiert als eine Reihung von Kunden. Neue Kunden reihen sich am **Ende der Schlange** ein, der am **Kopf der Schlange** stehende wird als Nächster bedient und dazu aus der Schlange genommen.

Eine solche Schlange ließe sich in einem Feld

```
S: Array[1..n] of Kunde
```

implementieren. Dabei müßte *n* so gewählt werden, daß einerseits alle denkbaren Warteschlangengrößen behandelt werden können, andererseits das Feld zwecks Platzersparnis nicht zu groß ist. Soll der Kopf immer in S[1] zu finden sein, muß man die Kunden verschieben, andernfalls „wandert“ die Schlange rückwärts kreisförmig im Feld.

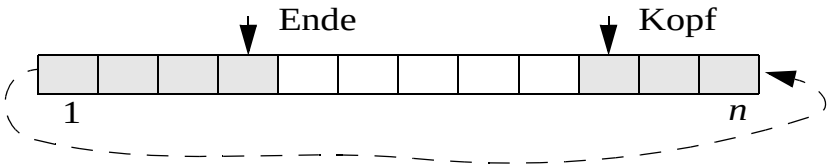


Abb. 10–4

Alternativ realisiert man die Schlange als *verkettete Liste*. Wie schon im Beispiel oben angedeutet - und auch im 1. Entwurf unten zu sehen - wird dazu im Verbund `Kunde` eine Komponente `next` vorgesehen, die vom Typ „Zeiger auf Kunde“ ist. Das Ende der Kette markieren wir mit einem **`nil`**-Wert. Zusätzlich vereinbaren wir zwei globale Zeiger, `Kopf` und `Ende`, die auf den vordersten, bzw. letzten Kunden zeigen.

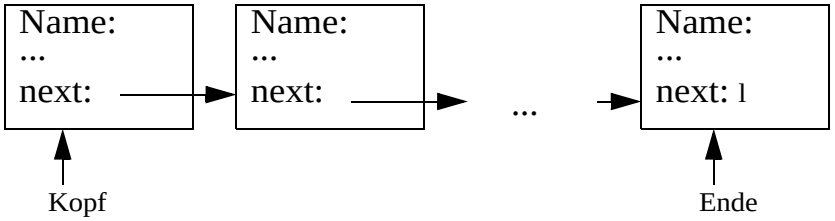


Abb. 10–5

An notwendiger Funktionalität brauchen wir sicherlich Anmelde- und Abmelderoutinen, wobei die Gestaltung der Parameter Geschmacksache ist. Einen Vorschlag haben wir unten gemacht. Zugleich haben wir Prioritäten vorgesehen, wobei 0 neutral (Standardvorgabe) wäre, positive Werte höhere Prioritäten ausdrücken, negative Werte geringere Ansprüche signalisieren. Damit lassen sich neben der üblichen „first-come-first-serve“ Warteschlange auch gewisse andere Strategien realisieren, z.B. die in Krankenhäusern übliche „sikkest first“.

```
program Warten;
type
  KPtr = ^Kunde;
  Kunde = record
    Name : String(30);{Name des “Kunden“}
    Prio : Integer;{Prioritaet: 0 = neutral, + hoch}
    next: KPtr  {naechster Kunde in Schlange}
  end {Kunde};
  SPtr = ^Schlange;{Zeiger auf Warteschlangen}
  Schlange = record {Warteschlange}
    Kopf,
    Ende : KPtr;
  end {Schlange};
var
  Server1: SPtr;
  DerNaechste: KPtr;

procedure Anmelden(K: KPtr; S : SPtr);
begin
  {K^ in Warteschlange S^ einreihen};
end {Anmelden};

function Abmelden(S : SPtr; var K : KPtr);
begin
  {den am Kopf der Warteschlange S^ stehenden Kundenaus
der
  Schlange entfernen und Zeiger K auf den Kunden liefern}
end {Abmelden};

begin
  writeln(“Willkommen in der Praxis!“);
end.
```

Bei der Realisierung der Routinen stellt man schnell fest, daß die leere Warteschlange einen lästigen Spezialfall darstellt, der zusätzliche Abfragen und Vorkehrungen verlangt. Eine gängige und gute Praxis ist, einen fiktiven Kunden einzuführen und ihn am Kopf (oder Ende) der Schlange zu halten.

```
procedure Erzeugen(var S: SPtr);
begin
  new(S); {Schlangenobjekt erzeugt}
  new(S^.Kopf); {Dummyobjekt erzeugt; Kopf zeigt darauf}
  S^.Ende := S^.Kopf;{Ende auch}
  with S^.Kopf^ do {fuellen mit fiktiven Werten}
  begin
    Name := ""; Prio := maxint; next := nil
  end {with};
end {Erzeugen};
```

Per Definition ist die Schlange leer, wenn Kopf und Ende auf den selben (fiktiven) Kunden zeigen.

Dies verwenden wir, wenn wir die Anzahl der wartenden Kunden zählen.

```
function Anzahl(S: SPtr): Integer;
var
  P : KPtr;   {zeigt auf Kunden}
  n : Integer;{Zaehlt Kunden}
begin
  n := 0;
  P := S^.Kopf;
  {Schlange leer wenn Kopf und Ende auf fiktivem Kunden}
  while P <> S^.Ende do
  begin
    P := P^.next; n := n + 1
  end;
  Anzahl := n
end {Anzahl};
```

Jetzt wollen wir neue Kunden *K* am Ende der Warteschlange *S* einreihen. Wir haben `Anmelden()` so definiert, daß ein Kundenrekord schon existiert und er nur in die Schlange integriert werden muß.

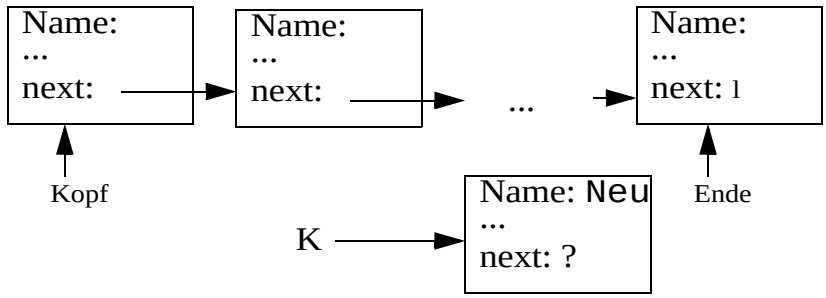


Abb. 10–6 *Situation vor Einfügen von Kunde K*

Dazu muß die letzte `next`-Komponente auf den neuen Kunden zeigen, dieser bekommt einen `next`-Wert **`nil`** und der Ende-Zeiger wird auf das neue letzte Element der Kette gesetzt.

```
procedure Anmelden(K: KPtr; S : SPtr);
begin
  {K^ in Warteschlange S^ einreihen};
  if K = nil then writeln("Fehler - Kundenzeiger ist
nil")
  else
  begin
    S^.Ende^.next := K; K^.next := nil; S^.Ende := K;
  end
end {Anmelden};
```

Die Situation läßt sich gut an den Bildern ablesen.

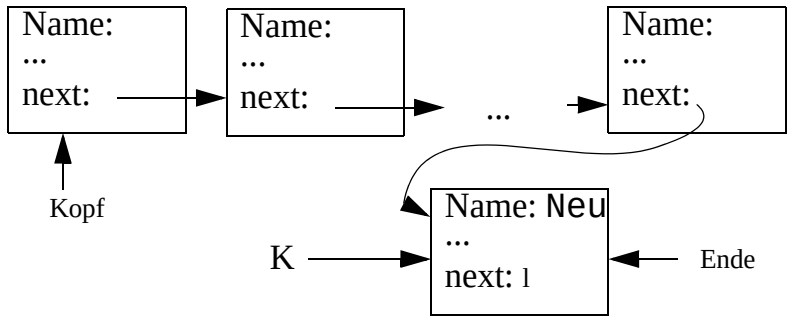


Abb. 10–7 *Situation nach Einfügen von Kunde K*

Auch die Abmeldung geht schnell. Man muß nur bedenken, daß der wirkliche „erste“ Kunde hinter dem fiktiven Kunden am Kopf steht. Aufpassen muß man, falls nur noch ein echter Kunde da ist. Dann steht der Ende-Zeiger auf ihm und muß auf Kopf vorgerückt werden, wodurch wir wieder eine leere Schlange haben.

```
procedure Abmelden(S : SPtr; var K: Kptr);
begin
  {den am Kopf der Warteschlange stehenden Kunden aus der
  Schlange entfernen und Zeiger auf den Kunden liefern}
  K := nil;
  if S = nil
  then writeln(“Fehler - Schlangenzeiger ist nil.“)
  else if S^.Kopf = S^.Ende
     then writeln(“Fehler - Schlange leer“)
     else
     begin
        K := S^.Kopf^.next; {Kunde nach Dummy-Kopf}
        if K = S^.Ende
        then S^.Ende := S^.Kopf; {letzter echter Kunde}
        S^.Kopf^.next := S^.Kopf^.next^.next;{***}
        {funktioniert auch, wenn nur noch 1 Kunde da}
     end;
end {Abmelden};
```

Die mit {***} markierte Zeile ist natürlich sehr kompliziert und für Programmieranfänger schwer zu durchschauen. Auch Programmierprofis unterlaufen bei solchen Listenmanipulationen häufig Fehler. Das folgende Bild versucht, den Sachverhalt zu veranschaulichen.

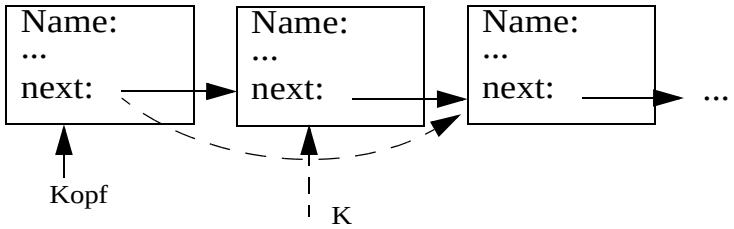


Abb. 10–8

Abgemeldete Kunden werden im Hauptprogramm mit `dispose()` auch aus dem Hauptspeicher geräumt. Ist der übergebene Zeiger allerdings **`nil`**, führt dies zum Crash. Tückisch sind auch andere Zeiger, die noch

auf das zu löschende Objekt zeigen. Ihre Dereferenzierung produziert meist auch einen Fehler.

Offensichtlich muß auch das Anlegen und Löschen dynamischer Objekte sehr sorgfältig verwaltet werden. Ein Löschversuch für ein nicht mehr existentes Objekt führt in der Regel zum Programmabbruch. Ein vergessenes Löschen eines nicht mehr referenzierbaren Objekts führt zu einer schleichenden Abnahme an freiem Speicher; man spricht von einem *Speicherleck* (engl. *storage leak*). Programmiersprachen, die automatisch nach nicht mehr referenzierten Objekten suchen und diese einer Müllsammlung (engl. *garbage collection*) zuführen, sind da besser gerüstet, benötigen dafür aber extra Laufzeit.

Im folgenden zeigen wir noch den Rest des Hauptprogramms mit geeigneten weiteren Vereinbarungen. Erzeugt wird damit eine kleine Testumgebung. Ein Probelauf von `warten.pas` folgt weiter unten.

```
begin {Hauptprogramm}
  writeln(“Willkommen in der Praxis!“);
  Erzeugen(Server1);{Anlegen einer leeren Warteschlange}
  repeat
    writeln(“Moegliche Aktion sind“);
    writeln(“(1) Anmelden (2) Bedienen
           (3) Programmende.“);
    write(“Bitte waehlen: “); readln(Wahl);
    case Wahl of
    1 : begin
      Status := An;
      new(DerNaechste);
      with DerNaechste^ do
      begin
        write(“Name eingeben: “); read(Name);
        write(“Proritaet eingeben (Default 0): “);
        readln(Prio);
        next := nil; {zur Sicherheit}
      end;
      Anmelden(DerNaechste, Server1);
    end {case 1 - Anmeldung};
    2 : begin
      Status := Ab;
      Abmelden(Server1, DerNaechste);
      if DerNaechste <> nil
```

```
        then
        begin
          with DerNaechste^ do
            writeln(“Kunde “, Name, “ mit Prioritaet “,
              Prio: 3, “ ist bedient.“);
          dispose(DerNaechste);
        end
      end {Abmeldung - Bedienung - Loeschen};
      3 : Status := HatFertig;
      else Status := Ungueltig {kein Standard Pascal}
      end {case};
      writeln(“Anzahl Kunden ist: “, Anzahl(Server1));
  until Status = HatFertig;
  writeln(“Programm wird beendet“);
end.
```

Ein Ablauf mit zwei Anmeldungen und zwei Bedienvorgängen sähe wie folgt aus.

```
$ ./warten
Willkommen in der Praxis!
Moegliche Aktion sind
(1) Anmelden, (2) Bedienen, (3) Programmende.
Bitte waehlen: 1
Name eingeben: Studentin
Proritaet eingeben (Default 0): 100
Anzahl Kunden ist:          1
Moegliche Aktion sind
(1) Anmelden, (2) Bedienen, (3) Programmende.
Bitte waehlen: 1
Name eingeben: Dozent
Proritaet eingeben (Default 0): 0
Anzahl Kunden ist:          2
Moegliche Aktion sind
(1) Anmelden, (2) Bedienen, (3) Programmende.
Bitte waehlen: 2
Kunde Studentin mit Prioritaet 100 ist bedient.
Anzahl Kunden ist:          1
Moegliche Aktion sind
(1) Anmelden, (2) Bedienen, (3) Programmende.
Bitte waehlen: 2
Kunde Dozent mit Prioritaet   0 ist bedient.
Anzahl Kunden ist:          0
Moegliche Aktion sind
(1) Anmelden, (2) Bedienen, (3) Programmende.
```

```
Bitte waehlen: 2
Fehler - Schlange leer
Anzahl Kunden ist:          0
Moegliche Aktion sind
(1) Anmelden, (2) Bedienen, (3) Programmende.
Bitte waehlen: 3
Anzahl Kunden ist:          0
Programm wird beendet
$
```

Sofern auch Sie nach diesem Kraftakt bedient sind, kann ihr Dozent das gut verstehen. Wir beenden deshalb hier die Behandlung der Programmiersprache Pascal und verweisen auf den Epilog, der abschließende Bemerkungen zum Programmieren enthält.

Epilog

Zum Ende dieser Vorlesung gibt es eine gute und eine schlechte Nachricht.

Die gute Nachricht ist, daß der Leser mit dem bisherigen Stoff eine solide Ausbildung in der Programmierung mit Pascal bekommen hat, die sogar die meist in Anfängerkursen übergangenen Pointer und Dateitypen enthalten hat. Vertieft man die Hinweise auf die Turbo Pascal Units, kann man sich auch an größere Projekte wagen. Beruhigen kann man auch alle diejenigen, die manches nicht ganz verstanden haben. Ein leichtes Unsicherheitsgefühl ist normal und verschwindet mit der zweiten und dritten Programmiersprache oder etwas Übung.

Die schlechte Nachricht ist, daß der bisher vermittelte Stoff nur die Spitze des Eisbergs in der Programmierung ausmacht!

Wer Interesse an der Programmierung bekommen hat, wer nebenher damit Geld verdienen oder sonst systematisch in die Informatik einsteigen möchte, sollte seine Programmierkenntnisse Zug um Zug ergänzen:

- Eine oder zwei Grundlagenveranstaltungen (typisch eine software- und eine hardwareorientierte) können nicht schaden. Dort lernt man z.B., was die Zweierkomplementdarstellung einer ganzen Zahl ist und wie ein Mikroprozessor funktioniert.
- Begriffe wie Stapel, Schlange, Suchbaum, Sortierverfahren, Verschmelzen, Mischen, usw. sind hier gefallen, wurden aber nicht systematisch erläutert. Hierzu sollte man die klassische Veranstaltung *Algorithmen und Datenstrukturen* besuchen; erfahrungsge-

mäß festigt man dabei nebenher nochmals seine Programmiersprachenkenntnisse.

- Anwendungsprogramme sind meist in Betriebssysteme eingebettet. Hier hilft der Besuch einer BS-Vorlesung, wobei meist auch eine kleine Shellsprache (die Sprache des Kommandointerpreters, in UNIX z.B. die Korn- oder bash-Shell) vermittelt wird, mit der man mit wenig Aufwand durch die Verknüpfung existierender Kommandos komplexe Aufgaben erledigen kann.
- Ansprechende Anwendungsprogramme haben eine graphische Benutzerschnittstelle. Mit Turbo Pascal und diversen Werkzeugen kann man Menüs und Eingabemasken generieren. Daneben lohnt aber auch das Erlernen einer zweiten oder dritten Sprache mit Stärken bei graphischen oder Internet-Anwendungen, z.B. Tcl/Tk, Java, Perl, ...
- Viele Einschränkungen bei Programmierkonstrukten werden erst dann richtig klar, wenn man selbst versucht, einen Übersetzer zu schreiben. In einer Compilerbau-Vorlesung wird dies gezeigt. Dabei kann man heutzutage auf frei verfügbare Werkzeuge, z.B. `lex` und `yacc` für die Syntaxanalyse, zurückgreifen.
- Größere Dateiverwaltungsaufgaben, die über die Verwaltung der eigenen Briefmarkensammlung hinausgehen, wird man nicht in Pascal „from scratch“ neu schreiben, sondern sich entweder auf im Netz verfügbare Software stützen oder das Programm an eine kommerzielle Datenbank binden.

Ansatzweise fielen immer wieder Begriffe wie „gutes Programmieren“, „guter/schlechter Programmierstil“, „sichere Programme“, „erfolgreiches Softwareprojekt“, usw. Dahinter verstecken sich zwei ausgewiesene Theorien:

- die sog. **Programmiermethodik**, die man auch als *programming-in-the-small* bezeichnet, und
- das **Software Engineering**, auch als *programming-in-the-large* bekannt.

In einer Vorlesung oder einem Seminar über Programmiermethodik lernt man, Programme formal korrekt zu beweisen. Man lernt Regeln für die Strukturierung von Programmen, gutes und schlechtes Codieren einschließlich sinnvoller Kommentare. Auch Effizienzbetrachtungen spielen eine Rolle, sowie der Einsatz objektorientierter Programmiersprachen mit Klassenbibliotheken. Dieses Wissen hilft einem Programmierer, professionell Programme zu konstruieren, die mehrere tausende Zeilen Code umfassen.

Großprojekte umfassen aber viele Millionen Zeilen von Code, wurden von tausenden von Systementwicklern über viele Jahre hinweg geschrieben, benötigen Pflege und enthalten in der Regel eine konstante Anzahl (je tausend Zeilen Code) von bekannten und unbekannten Fehlern . Wie man damit umgeht, wie man testet, was „Reinraumtechniken“ leisten, usw. lernt man, wenn man sich mit ingenieurmäßiger Softwareerzeugung beschäftigt. Speziell erfährt man dort, daß ab einer gewissen Größe und Komplexität der Programme der Einsatz von mehr Personal in einem in Schwierigkeiten steckenden Projekt keinen Zeitgewinn bringt, sondern im Gegenteil, den Programmfortschritt nur behindert.

Auch im kleinen sollte man ökonomisch denken, deshalb ein letzter Rat. Wenn der Schwager für seine Doktorarbeit in Medizin eine Regressionsanalyse braucht, die Daten als Tortengraphik oder Histogramm visualisieren will und sie gerne sicher auf Platte geschrieben hätte, besorgt man ihm lieber für wenig Geld (oder umsonst als freeware aus dem Internet) eine Tabellenkalkulation, statt das Rad neu zu erfinden.

Andererseits macht Programmieren Spaß und manchmal kann man mit einem kleinen Programm, das man in einer Stunde schreibt, echtes Geld sparen. Ein Beispiel ist die Berechnung der Restschuld einer Hypothek für verschiedene Zinssätze und anfängliche Tilgungsraten. Anders als bei geschlossenen Zinsformeln kann man Sonderfälle, wie z.B die Effekte jährlicher Sonderzahlungen usw., leicht einbauen. Turbo Pascal, ruhig auch in einer ältlichen Versionen ab 4.0, das klaglos auch auf einem müden 286er PC läuft, ist dabei wegen der klaren Syntax und Semantik nicht das schlechteste Werkzeug.

In diesem Sinne, many happy codings und immer daran denken ... ECHTE Programmierer hassen koffeinfreien Kaffee!

Anhang A

Schlüsselwörter, Operatoren

The keywords are taken from the following standards:

ISO 7185 Standard Pascal (SP)
ISO 10206 Extended Pascal (EP)
ANSI draft Object Pascal (OP) (not yet implemented)
Borland Pascal 7.0 (BP)
Pascal-SC (PXSC, Pascal eXtensions for Scientific Calculations)

Keyword	Pascal standard	Remarks
Absolute	Borland	overload variables
Abstract	Object	not implemented
All	GNU	EP „export foo = all“ extension
And	ISO Standard	
And_then	ISO Extended	short-circuit Boolean AND operator
Array	ISO Standard	
Asm	Borland, GNU	GNU-style assembler
Begin	ISO Standard	
Bindable	ISO Extended	external binding of files, etc.
Case	ISO Standard	
Class	Object	not implemented
Const	ISO Standard	
Constructor	Object, Borland	only BP version implemented
Destructor	Object, Borland	only BP version implemented

Div	ISO Standard	
Do	ISO Standard	
Downto	ISO Standard	
Else	ISO Standard	
End	ISO Standard	
Export	ISO Extended	Module Interface export
File	ISO Standard	
For	ISO Standard	
Function	ISO Standard	
Goto	ISO Standard	
If	ISO Standard	
Import	ISO Extended	Module Interface import
Implementation	ISO Extended, Borland	Module (EP) or Unit (BP) Impl. part
Inherited	Object, Borland	only BP version implemented
In	ISO Standard	
Inline	Borland, GNU	only GNU inline functions implem.
Interface	ISO Extended, Borland	Module (EP) or Unit (BP) Int. part
Is	Object	not implemented
Label	ISO Standard	
Mod	ISO Standard	
Module	ISO Extended, PXSC	PXSC version only partially implem.
Nil	ISO Standard	
Not	ISO Standard	
Object	Borland	BP 7.0 style class definition
Of	ISO Standard	
Only	ISO Extended	import specification
Operator	PXSC	operator definition
Or	ISO Standard	
Or_else	ISO Extended OR operator	short-circuit Boolean
Otherwise	ISO Extended	default case label
Packed	ISO Standard	does not yet pack
Pow	ISO Extended	exponentiation op. (integer expon.)
Procedure	ISO Standard	
Program	ISO Standard	
Property	Object	not implemented
Protected	ISO Extended	read-only formal parameters

```
Qualified          ISO Extended     import specification
Record             ISO Standard
Repeat             ISO Standard
Restricted         ISO Extended     type specification
Set                ISO Standard
Shl                Borland          left bit-shift
                                    operator
Shr                Borland          right bit-shift
                                    operator
Then               ISO Standard
To                 ISO Standard
Type               ISO Standard
Unit               Borland          Borland (or UCSD)
                                    style Modules
Until              ISO Standard
Uses               Borland          Borland (or UCSD)
                                    style import
Value              ISO Extended     variable initializer
Var                ISO Standard
View               Object           not implemented
Virtual            Borland, Object  only Borland version
                                    implemented
While              ISO Standard
With               ISO Standard
Xor                Borland          Boolean/bitwise
                                    exclusive OR op.
```

Operators

GNU Pascal operators, ordered by precedence.

The PXSC operators '`+<`', '`-<`', etc. are not implemented into GNU Pascal but may be defined by the user. If you do so and meet the PXSC requirements, please let us know. The other real operators do not meet PXSC requirements.

The Object Pascal operator '`IS`' is not implemented.

```
:=
< = > IN <> >= <=
+ - OR +< -< +> ->
* / DIV MOD AND SHL SHR XOR *< /< *> />
POW ** IS
NOT
```

Anhang B

Standardbezeichner

Redefineable built-in identifiers

The following identifiers are built in into GNU Pascal but may be redefined according to any supported Pascal standard.

```
Maxint
False
True
Input
Output

Rewrite
Reset
Put
Get
Write
Read
Writeln
Readln
Page
New
Dispose
Abs
Sqr
Sin
Cos
Exp
Ln
Sqrt
Arctan
Trunc
```

```
Round
Pack
Unpack
Ord
Chr
Succ
Pred
Odd
Eof
Eoln
```

Directives

```
Asmname     Specify case-sensitive external name
            for Function
C           External name for Function shall be lowercase
C_language  same as C
Forward
External    External Name of Function Has First
            Letter Uppercase
Extern      same as External
```

Extended Pascal required module interfaces

```
Standardoutput
Standardinput
```

Object Pascal directive (not implemented)

```
Override
```

Extended Pascal required words

```
Maxchar
Maxreal
Minreal
Epsreal
```

Extended Pascal required Procedures and Functions

```
Gettimestamp
Date
Time
Halt
Extend
Seekwrite
Seekread
```

```
Seekupdate
Empty
Update
Position
Lastposition
Re
Im
Cmplx
Card
Arg
```

Extended Pascal external binding

```
Bind
Unbind
Binding
```

Extended Pascal complex type functions

```
Polar
```

Extended Pascal String functions

```
Readstr     Read from a string rather than a file
Writestr    Write to a string rather than a file
Length
Index       Search in a string
Substr      also 'MyStr [ 1..5 ]'
Trim
Eq          lexical string comparision
Lt
Gt
Ne
Le
Ge
```

Extended pascal required string schema type generator

```
String
```

Object pascal (not implemented)

```
Copy
Null
Root
Textwritable
Self
```

Borland Pascal

```
GetMem      Allocate memory with given size in bytes
FreeMem     Free memory allocated with GetMem
Inc         Increment
Dec         Decrement
```

More exotic fruits and birds (GPC extensions)

```
Static      C-sense storage class specifications
__const__
__external__
__inline__
__static__
__volatile__
__byte__    C-style type size modifiers
__short__
__long__
__longlong__
__unsigned__

Asm         Inline assembly (GNU style)
Alignof
Break C-style
Continue
Return
Sizeof
Max         for enumeral and real types
Min

Conjugate
Mark
Release

Default     like ‘otherwise’ in a ‘case’ statement
Others

Close
Definesize
```

Standard Pascal data types

```
Integer
Real
Boolean
```

```
Char
Text
```

Extended Pascal complex type

```
Complex
```

GPC extensions: void type (two spellings)

```
__void__
Void
```

GPC extension: C compatibility string type

```
__cstring__
```

Extended Pascal: TimeStamp and BindingType

```
Timestamp
Bindingtype
```

Literatur

[1] Judith Gal-Ezer and David Harel, *What (Else) Should CS Educators Know?* Comm. ACM 41:9 (Sept. 1998), 77-84.

[2] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*, Third Edition, Computer Science Press, New York, 1990.

[3] A. Ertl, R. Machholz und A. Schallmaier, *Turbo Pascal 6.0 Turbo Vision*, 2. Aufl., Addison-Wesley, Bonn, 1991.

[4] K. Horn, *PC-Nutzung mit TURBO-PASCAL*, B.G. Teubner, Stuttgart, 19989.

[5] T. Ottmann und P. Widmayer, *Programmieren mit PASCAL*, Teubner Studienskripten, Stuttgart, 1986. (6. Auflage 1995)

[6] T. Ottmann, M. Schrapp und P. Widmayer, *PASCAL in 100 Beispielen*, B.G. Teubner, Stuttgart, 1983.

[7] G.M. Schneider, S.W. Weingart and D.M. Perlman, *An Introduction to Programming and Problem Solving with PASCAL*, John Wiley & Sons, New York, 1978.

[8] R. Marty, *Methodik der Programmierung in Pascal*, Springer-Lehrbuch, Springer, Berlin, 4. Aufl. 1994.

[9] K. Jensen, Kathleen, A.B. Mickel und N. Wirth, *PASCAL User Manual and Report - Revised for the ISO PASCAL Standard*, Springer, Berlin, 4. Aufl. 1991. XVI,

[10] E. Kaier, *TURBO PASCAL-Wegweiser: Für Version 6.0*, m. Diskette (5 1/4 Zoll), Vieweg, 5. Aufl. 1991.

[11] Borland, *Turbo Pascal 6.0 Programmier-, Referenz- und Benutzerhandbuch*, Borland GmbH, München, 1. Auflage 1990

[12] -, ISO 7185: 1983, *Programming Languages - PASCAL*, *„Standard Pascal“*, verfügbar über ftp://ftp.digital.com/pub/DEC/Pascal/.

[13] -, ISO 10206:1990, *Extended Pascal*, verfügbar über ftp://ftp.digital.com/pub/DEC/Pascal/.

[14] H. Ledgard, P. Nagin, and J. Hueras, *Pascal with Style*, Hayden Book Comp., Rochelle Park, NJ, 1979

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z