KAPITEL 3

Berechnen von Variablen

Im vorgehenden Kapitel wurde die Eingabe der Daten beschrieben. Da dies im Allgemeinen eine schweißtreibende Angelegenheit ist, sollten nur so genannte *Rohdaten* eingegeben werden und nicht solche Daten, die aus diesen Rohdaten berechnet werden können. Denn hierfür, nämlich zum Rechnen, wurde der Computer seinerzeit ja erfunden.

Für die Berechnung neuer Variablen gibt es verschiedene Anweisungen, die im Folgenden erläutert werden. Mit der gängigsten Anweisung, der compute-Anweisung, soll dabei begonnen werden.

3.1 Die compute-Anweisung

Die Datei gewicht.txt enthält Daten von übergewichtigen Probanden, die sich einer Diätkur unterzogen haben. Dabei verteilen sich die Probanden auf zwei verschiedene Gruppen: Die eine Gruppe machte nur Diät, die andere schloss sich zusätzlich noch einem Verein Gleichgesinnter an, um die Diät zu unterstützen. Die Datei ist nach folgendem Spaltenplan aufgebaut.

Spalten	Variablen
1	Gruppe (1 = Diät, 2 = Diät + Verein)
3–5	Körpergröße in cm
7–11	Körpergewicht in kg am Anfang der Kur (1 Dezimalstelle)
13–17	Körpergewicht am Ende der Kur (1 Dezimalstelle)

Die Zeilen der Datei seien im Folgenden aufgelistet.

```
1 175  85,4  82,3
1 164  71,7  68,9
1 183  88,1  86,3
1 180  84,8  81,2
1 159  67,0  64,4
1 187  99,6  93,2
1 165  80,9  74,3
1 177  88,7  84,3
1 162  69,5  67,4
2 185 105,6  94,3
2 156  80,1  72,5
```

```
2 161  88,0  80,9
2 172  98,1  92,3
2 180 107,8  95,6
2 164  78,6  72,8
2 170  90,7  82,3
2 166  89,0  81,0
2 158  77,0  71,5
2 190 110,9  99,8
2 170  88,4  80,9
```

Mit den Kenntnissen von Kap. 2 kann man dazu unter Verwendung passender Variablennamen das folgende Einleseprogramm schreiben (Datei gewicht.sps):

```
data list file='c:\syntax\gewicht.txt'/
  gruppe 1 gr 3-5 gew0 7-11 (1) gew1 13-17 (1).
variable labels gruppe 'Gruppe'/
  gr 'Koerpergroesse'/
  gew0 'Koerpergewicht am Anfang der Kur'/
  gew1 'Koerpergewicht am Ende der Kur'.
value labels gruppe 1 'Diaet' 2 'Diaet und Verein'.
execute.
```

3.1.1 Arithmetische Ausdrücke

Ein Maß für die Gewichtsabnahme ist offensichtlich die Differenz zwischen Anfangs- und Endgewicht. Für diese Differenz soll die Variable *gewab* gebildet werden:

```
compute gewab = gew0 - gew1.
execute.
```

Dem Befehlsnamen *compute* folgt der Name der Zielvariablen (der neuen Variablen), ein Gleichheitszeichen und ein arithmetischer Ausdruck. Im gegebenen einfachen Beispiel besteht dieser arithmetische Ausdruck (auch: numerischer Ausdruck, mathematische Formel) aus zwei Variablennamen, die mit dem arithmetischen Operator - verbunden sind.

Im Allgemeinen setzen sich arithmetische Ausdrücke zusammen aus Zahlen, Namen von bestehenden Variablen, arithmetischen Operatoren, mathematischen Funktionen und Klammern.

Mit der compute-Anweisung können allerdings nicht nur numerische Variablen berechnet werden, es gibt auch Funktionen zur Bearbeitung von Text- und Datumsvariablen.

Insgesamt gibt es fünf arithmetische Operatoren:

Priorität	Operator	Bedeutung
1	**	Exponentiation
2	*	Multiplikation
2	/	Division
3	+	Addition
3	–	Subtraktion

Die Prioritätenregel besagt, dass Multiplikation und Division beim gemeinsamen Auftreten mit den Operatoren für Addition und Subtraktion zuerst ausgeführt werden. Wird eine andere Reihenfolge gewünscht, sind entsprechend runde Klammern zu setzen.

Als Maß für die Gewichtsreduktion einfach die Gewichtsdifferenz heranzuziehen, berücksichtigt nicht das Anfangsgewicht. So ist z. B. eine Gewichtsabnahme um 5 kg bei einem Menschen mit 85 kg Körpergewicht gravierender als bei einem mit 110 kg Körpergewicht. Es bietet sich also an, die Gewichtsdifferenz auf den Ausgangswert zu beziehen und die *prozentuale* Gewichtsabnahme nach der Formel

$$pgewab = \frac{gew0 - gew1}{gew0} \cdot 100$$

zu berechnen. In SPSS formuliert lautet dies

```
compute pgewab = (gew0-gew1)/gew0 * 100.
execute.
```

Die Klammersetzung um die Differenz herum ist wegen der Prioritätenregel notwendig.

Die Angabe des Körpergewichts ist allerdings ohne die gleichzeitige Betrachtung der Körpergröße nicht aussagekräftig. So hat der Pariser Chirurg Paul Broca einen Index entwickelt, nach dem das Normalgewicht eines Menschen nach der Faustformel

$$\text{Normalgewicht in kg} = \text{Körpergröße in cm} - 100$$

ermittelt wird. Das auf das Normalgewicht prozentuierte tatsächliche Körpergewicht wird Broca-Index genannt:

$$\text{Broca-Index} = \frac{\text{Körpergewicht}}{\text{Normalgewicht}} * 100$$

Normalgewichtige Personen haben demnach einen Broca-Index um 100 Prozent. Werte über 100 signalisieren Übergewicht, Werte unter 100 Untergewicht.

Setzt man die beiden Anweisungen zur Berechnung des Broca-Indexes in entsprechende compute-Anweisungen um, so ergibt sich

```
compute broca0 = gew0/(gr-100)*100.
compute broca1 = gew1/(gr-100)*100.
execute.
```

Ein moderneres Maß zur Beurteilung von Normal- bzw. Übergewicht ist der Body-Mass-Index (BMI):

$$\text{BMI} = \frac{\text{Körpergewicht in kg}}{(\text{Körpergröße in m})^2}$$

Unter Beachtung der Prioritätenregel (Exponentiation vor Division) lautet das in SPSS

```
compute bmi0 = gew0/(gr/100)**2.
compute bmi1 = gew1/(gr/100)**2.
execute.
```

Die BMI-Werte liegen gewöhnlich zwischen 20 und 30. Werte unter 25 gelten als normalgewichtig, Werte ab 25 bis unter 30 als übergewichtig und Werte ab 30 als adipös.

Das passende Maß für den Kurerfolg ist dann wohl die prozentuale Abnahme des BMI

```
compute pbmiab = (bmi0-bmi1)/bmi0*100.
execute.
```

Wollen Sie die beiden Probandengruppen hinsichtlich des Kurerfolgs mit dem t-Test nach Student vergleichen, können Sie das mit folgender Anweisung erreichen:

```
t-test groups=gruppe(1,2)/variables=pbmiab.
```

Der Gebrauch dieser Syntax bringt allerdings gegenüber der Menüwahl

Analysieren
Mittelwerte vergleichen
T-Test bei unabhängigen Stichproben…

keinen Vorteil.

Der t-Test ergibt, dass sich die beiden Gruppenmittelwerte 4,47 (Diät) und 8,80 (Diät und Verein) höchst signifikant voneinander unterscheiden ($p < 0{,}001$).

Es sei noch einmal ausdrücklich auf die Notwendigkeit der Klammersetzung hingewiesen, wenn die Prioritätenregelung der Operatoren außer Kraft gesetzt werden soll. Im Zweifelsfall können Sie auch Klammern setzen, wenn es die Prioritätenregeln eigentlich überflüssig machen würden. Da erfahrungsgemäß trotzdem immer wieder Fehler auch von fortgeschrittenen SPSS-Programmierern gemacht werden, empfiehlt es sich dringend, vor der Weiterrechnung mit den neuen Variablen diese im Daten-Fenster auf Stimmigkeit zu überprüfen bzw. zumindest einen Wert per Hand nachzurechnen.

Wem die Betrachtung im Daten-Fenster zu unübersichtlich ist, dem sei empfohlen, die berechneten Werte zusammen mit den Werten der Ursprungsvariablen mit Hilfe der list-Anweisung im Viewer auszugeben und gegebenenfalls auszudrucken.

Wir wollen annehmen, Sie hätten bei der Berechnung des Broca-Indexes die Klammerung im Nenner weggelassen und daher die compute-Anweisung folgendermaßen formuliert:

```
compute broca0=gew0/gr-100*100.
execute.
```

Sie formulieren die list-Anweisung

```
list gr,gew0,broca0.
```

und erhalten für die ersten Fälle den folgenden Ausdruck:

```
GR   GEW0   BROCA0

175   85.4 -9999.51
164   71.7 -9999.56
183   88.1 -9999.52
180   84.8 -9999.53
159   67.0 -9999.58
```

Aufgrund der Prioritätenregel hat der Computer zunächst die beiden Terme gew/gr und $100 * 100$ berechnet und die Ergebnisse dann subtrahiert. Sie bemerken den Fehler und formulieren richtig

```
compute broca0 = gew/(gr-100)*100.
execute.
```

Die Auflistung der ersten Fälle ergibt nun korrekt

```
GR   GEW0   BROCA0

175   85.4   113.87
164   71.7   112.03
183   88.1   106.14
180   84.8   106.00
159   67.0   113.56
```

Arithmetische Ausdrücke können nur dann berechnet werden, wenn keine der eingehenden Variablen einen fehlenden Wert hat. In diesem Fall wird dann der Wert der Zielvariablen auf „fehlend" gesetzt. Ausnahmen hiervon bilden einige statistische Funktionen.

3.1.2 Funktionen

In SPSS stehen zahlreiche Funktionen zur Verfügung. Wollen Sie etwa die Werte einer Variablen logarithmieren und hierfür den dekadischen Logarithmus verwenden, so setzen Sie die Funktion lg10 ein:

```
compute x = lg10(x).
execute.
```

In diesem Fall werden die Werte der Variablen x selbst verändert. Ein Funktionsaufruf besteht also aus dem Funktionsnamen und dahinter, in runden Klammern eingeschlossen, die Argumentenliste. Die Funktion lg10 hat dabei *ein* Argument.

Interessiert Sie bei der Differenz zweier Variablen x und y nur der Absolutbetrag, so verwenden Sie die Funktion *abs*:

```
compute diff=abs(x-y).
execute.
```

Selbstverständlich können Funktionsaufrufe auch in komplexere arithmetische Ausdrücke eingebaut werden.

Die in SPSS zur Verfügung stehenden Funktionen werden im Folgenden nach Kategorien geordnet aufgeführt.

Arithmetische Funktionen

Funktion	Bedeutung
$abs(x)$	absoluter Betrag
$rnd(x)$	gerundeter Wert
$trunc(x)$	abgeschnittener Wert
$mod(x, y)$	Divisionsrest von x durch y
$sqrt(x)$	Quadratwurzel
$exp(x)$	Exponentialfunktion
$lg10(x)$	dekadischer Logarithmus
$ln(x)$	natürlicher Logarithmus
$arsin(x)$	Arcus sinus
$artan(x)$	Arcus tangens
$sin(x)$	Sinus
$cos(x)$	Cosinus

Mit Ausnahme der Funktion *mod* mit den beiden (ganzzahligen) Argumenten haben alle arithmetischen Funktion ein Argument. Bei den Winkelfunktionen sind die Argumente im Bogenmaß anzugeben.

Statistische Funktionen

Die Anzahl der Argumente bei den statistischen Funktionen ist beliebig.

Funktion	Bedeutung	Voreinstellung für *n*
sum.n(*x*, *x*, ...)	Summe der Werte der Argumente	1
mean.n(*x*, *x*, ...)	Mittelwert	1
sd.n(*x*, *x*, ...)	Standardabweichung	2
var.n(*x*, *x*, ...)	Varianz	2
cfvar.n(*x*, *x*, ...)	Variationskoeffizient	2
min.n(*x*, *x*, ...)	Minimum	1
max.n(*x*, *x*, ...)	Maximum	1

Mit dem optionalen Zusatz *.n* kann angegeben werden, wie viele Argumente mindestens einen gültigen Wert haben müssen, damit der Funktionswert berechnet wird. Dieser Zusatz am Funktionsnamen kann auch weggelassen werden; es gilt dann die Voreinstellung.

Die gebräuchlichsten statistischen Funktionen dürften *sum* und *mean* sein. Dazu sei ein typischer Anwendungsfall gezeigt.

In Abschnitt 2.1.1 wurden die 22 Items eines Fragebogens vorgestellt, der sich mit Gefühlslagen bei Arbeit und Beruf beschäftigt. Die Items waren mit einem Skalenwert von 1 (völlig unzutreffend) bis 7 (völlig zutreffend) codiert. Ein Einleseprogramm für die betreffende Datei beruf.txt ist in Abschnitt 2.3 wiedergegeben.

Folgt man diesem Programm, wurden die Items auf den Variablen *gl*1 bis *gl*22 gespeichert.

Aus diesen Items sollen nun drei Skalenwerte gebildet werden. Dabei umfasst die Skala „Emotionale Erschöpfung" die Items 1, 2, 3, 6, 8, 13, 14, 16 und 20, die Skala „Reduzierte Leistungsfähigkeit" die Items 4, 7, 9, 12, 17, 18, 19 und 21 und die Skala „Dehumanisierung" die Items 5, 10, 11, 15 und 22.

Betrachten wir etwa die Items zur Skala „Emotionale Erschöpfung", so stellen wir fest, dass alle Items gleich gepolt sind und hohe Itemwerte in Richtung emotionaler Erschöpfung zielen. Daher kann der Skalenwert als Summe der betreffenden Variablenwerte gebildet werden:

```
compute ee=gl1+gl2+gl3+gl6+gl8+gl13+gl14+gl16+gl20.
execute.
```

Falls es unter den Daten keine fehlenden Werte gibt, ist gegen diese Art der Summenbildung nichts einzuwenden, und die Skalenwerte bewegen sich im Bereich von 9 bis 63. Gibt es aber fehlende Werte und hat auch nur eines der in die Skalenbildung eingehenden Items einen solchen fehlenden Wert, so kann die Summe nicht gebildet werden. Dann ist es besser, die Funktion *sum* zu benutzen:

```
compute ee=sum(gl1,gl2,gl3,gl6,gl8,gl13,gl14,gl16,gl20).
execute.
```

In diesem Fall wird die Summe berechnet, wenn auch nur ein Item (voreingestellter Zusatz .1) einen gültigen Wert hat.

Hier erhebt sich nun aber der berechtigte Einwand, dass die Skalenwerte über die einzelnen Probanden nicht mehr recht vergleichbar sind. Wenn ein Proband bei den insgesamt neun Items z. B. drei fehlende Werte hat, ist der maximal erreichbare Skalenwert von $9 \cdot 7 = 63$ nicht mehr möglich, sondern allenfalls ein Wert von $6 \cdot 7 = 42$.

Hier ist es nun wiederum vorteilhafter, nicht die Funktion *sum*, sondern die Funktion *mean* zu benutzen:

```
compute ee = mean(gl1,gl2,gl3,gl6,gl8,gl13,gl14,gl16,gl20).
execute.
```

Bei dieser Art der Skalenbildung wird die Itemsumme auf die Anzahl gültiger Werte relativiert, so dass vergleichbare Werte entstehen. Der Skalenwert weist dann Nachkommastellen auf und bewegt sich im Bereich zwischen 1 und 7.

Möchten Sie auch in diesem Fall wieder den Wertebereich von 9 bis 63 erreichen, können Sie den berechneten Mittelwert in der Weise „nachbearbeiten", dass Sie ihn wieder mit der Anzahl der Items (hier: 9) multiplizieren und das Ergebnis mit Hilfe der Funktion *rnd* runden:

```
compute ee=sum(gl1,gl2,gl3,gl6,gl8,gl13,gl14,gl16,gl20).
compute ee=rnd(ee*9).
execute.
```

Es sei aber der Hinweis gegeben, dass auch bei diesem Verfahren die Vergleichbarkeit nicht völlig gegeben ist, und zwar besonders dann nicht, wenn z. B. ein Proband ein im Sinne der Itemanalyse „leichtes" Item nicht beantwortet hat, also ein Item, das häufig mit der höchsten Codierung (7 = völlig zutreffend) beantwortet wurde. Ihm ist dann im Vergleich mit den anderen Probanden sozusagen die Chance genommen worden, seinen Skalenwert entsprechend zu erhöhen.

Datumsfunktionen

In Abschnitt 2.1.5 wurde die Eingabe von Datumsvariablen beschrieben; außerdem wurden schon einige Datumsfunktionen vorgestellt.

Zum Verständnis der Datumsfunktionen in SPSS muss man wissen, dass jeder Datumsangabe in SPSS ein so genannter interner Wert zugeordnet wird, und zwar ist dies die Anzahl der Sekunden, die seit dem 14. Oktober 1582 00:00 Uhr verstrichen sind. Die Datumsfunktionen unterteilen sich dann in solche, die aus Teilen von Datumsangaben den internen Wert berechnen, und in solche, die aus dem internen Wert wieder Teile von

Datumsangaben herausfiltern. Im Folgenden seien die wichtigsten Datumsfunktionen zusammengestellt.

Eine Sonderrolle spielt die Funktion *yrmoda*, die den internen Wert nicht in Sekunden, sondern in Tagen berechnet.

Funktion	Bedeutung
date.dmy(*tag,monat,jahr*)	interner Wert zu drei getrennten Datumsvariablen
date.moyr(*monat,jahr*)	interner Wert zum 1. des betreffenden Monats
yrmoda(*jahr,monat,tag*)	interner Wert als Anzahl der Tage seit dem 15. Oktober 1582 zu drei getrennten Datumsvariablen
xdate.mday(*dat*)	gibt aus dem internen Datumswert den Tag zurück
xdate.month(*dat*)	gibt den Monat zurück
xdate.year(*dat*)	gibt das Jahr zurück
xdate.wkday(*dat*)	gibt die Nummer des Wochentags zurück (1 = Sonntag, …, 7 = Samstag)
xdate.tday(*dat*)	interner Wert in Tagen

Einige der Datumsfunktionen wurden bereits in Abschnitt 2.1.5 erläutert. Wir greifen noch einmal auf die dort vorgestellte Datei zurück, die zu den beiden numerischen Variablen Geschlecht und Unfallart die drei Datumsvariablen Geburtsdatum, Untersuchungsdatum und Operationsdatum enthält. Das folgende SPSS-Programm regelt das Einlesen der Variablen sowie die erweiterte Variablenbeschreibung und berechnet aus den Datumsvariablen das Alter bei Operation in Jahren und die Wartezeit zwischen Untersuchung und Operation in Tagen (Datei datfunk.sps):

```
data list file='c:\syntax\datum2.txt'/geschl 1 uart 3
  gebdat 5-14 (date) udat 16-25 (date) opdat 27-36 (date).
variable labels geschl 'Geschlecht'/
  uart 'Unfallart'/
  gebdat 'Geburtsdatum'/
  udat 'Untersuchungsdatum'/
  opdat 'Operationsdatum'.
value labels geschl 0 'maennlich' 1 'weiblich'/
  uart 1 'Haus' 2 'Verkehr' 3 'Arbeit' 4 'Sport'.
```

```
compute alter=rnd((xdate.tday(opdat)-xdate.tday(gebdat))/365.25).
compute wzeit=xdate.tday(opdat)-xdate.tday(udat).
variable labels alter 'Alter bei Operation'/
  wzeit 'Wartezeit zwischen Untersuchung und Operation'.
execute.
```

Bei der Berechnung des Alters bestimmt die Differenz zwischen den beiden Funktionsaufrufen von *xdate.tday* zunächst das Alter in Tagen. Dieses wird durch 365.25 geteilt, um das Alter in Jahren zu erhalten. Die Rundungsfunktion *rnd* sorgt schließlich für eine ganzzahlige Jahresangabe. Beachten Sie bitte dabei, dass innerhalb der Programmsyntax als Dezimaltrennzeichen der Punkt zu verwenden ist.

Zeitfunktionen

In Abschnitt 2.1.6 wurde die Eingabe von Zeitvariablen behandelt. Der interne Wert bei Zeitvariablen ist die seit 00:00 Uhr verstrichene Zeit in Sekunden. Die gebräuchlichsten Zeitfunktionen sind in der folgenden Tabelle zusammengestellt.

Funktion	Bedeutung
time.hms(*h,m,s*)	interner Zeitwert (Sekunden) zu drei getrennten Variablen (Stunde, Minute, Sekunden)
ctime.minutes(*zeit*)	gibt den internen Zeitwert in Minuten zurück
xdate.time(*zeit*)	gibt den internen Zeitwert in Sekunden zurück
xdate.hour(*zeit*)	gibt vom internen Zeitwert die Stunde wieder
xdate.minute(*zeit*)	gibt vom internen Zeitwert die Minute wieder
xdate.second(*zeit*)	gibt vom internen Zeitwert die Sekunde wieder

Als Beispiel sei die Zeitangabe 12:57:45 betrachtet. Die verschiedenen Zeitfunktionen liefern hierfür die folgenden Werte.

Funktion	Wert
ctime.minutes	777,75
xdate.time	46665
xdate.hour	12
xdate.minute	57
xdate.second	45

Funktionen für fehlende Werte

Als sehr nützlich erweisen sich zuweilen die Funktionen, die Abfragen nach fehlenden Werten erlauben. Dabei wird unterschieden zwischen systemdefinierten und benutzerdefinierten fehlenden Werten.

Systemdefinierte fehlende Werte sind solche, die durch Leerstellen in den Eingabedaten kenntlich gemacht wurden. Benutzerdefinierte fehlende Werte haben zunächst codierte Werte (z. B. die Ziffer 9 bei einspaltigen Angaben), die dann aber mittels einer missing-values-Anweisung zu „fehlend" deklariert wurden.

missing(***var***)***:*** Das Ergebnis ist 1, falls die betreffende Variable einen fehlenden Wert hat (gleich ob system- oder benutzerdefiniert), sonst 0.

sysmis(***var***)***:*** Das Ergebnis ist 1, falls die betreffende Variable einen systemdefinierten fehlenden Wert hat, sonst 0.

nmiss(***var1,var2,…***)***:*** Anzahl der fehlenden Werte in den Variablen

nvalid(***var1,var2,…***)***:*** Anzahl der gültigen Werte in den Variablen

value(***var***)***:*** Aufhebung der Benutzerdefinition eines fehlenden Wertes

Zur Funktion *value* sei des besseren Verständnisses wegen ein Beispiel gegeben. Betrachten Sie die Datei leuko.sav, die als einzige Variable (*leuko*) von 185 Patienten den Leukozytenwert (in Tausend) beinhaltet. Vier Werte, die über 50 000 liegen, werden als Ausreißerwerte eingestuft und sollen daher als „fehlend" deklariert werden:

```
missing values leuko (50000 thru highest).
execute.
```

In einer der späteren Berechnungen, z. B. einer Medianberechnung, sollen die als fehlend deklarierten Werte aber doch einbezogen werden. Dies ist möglich mit Hilfe der value-Funktion:

```
compute leuko2=value(leuko).
frequencies variables=leuko2/format=notable/statistics=median.
```

Funktion über Fälle hinweg

Mit der Funktion *lag*(*var,n*) können Sie auf den betreffenden Variablenwert des n-ten Falls vor dem aktuellen Fall zugreifen. Wird die Funktion in der Variante *lag*(*var*) benutzt, wird der betreffende Variablenwert des vorhergehenden Falls ($n = 1$) geliefert.

Ein Beispiel soll dies erläutern. Nehmen Sie an, Sie führen Buch über Ihren Benzinverbrauch und notieren bei jeder Tankfüllung den Kilometerstand und die getankte Benzinmenge in Litern. Die Daten speichern Sie in der Datei tank.txt:

```
16.12.2000 20580 60,3
23.12.2000 21250 57,4
 4. 1.2001 21874 56,6
17. 1.2001 22476 56,3
28. 1.2001 22954 45,4
12. 2.2001 23450 48,6
27. 2.2001 24020 57,0
14. 3.2001 24611 56,7
```

Sie möchten jeweils die seit dem letzten Tanken auf einhundert Kilometer verbrauchte Benzinmenge berechnen. Dazu benötigen Sie die jeweilige Differenz zum vorgehenden Kilometerstand, der im folgenden SPSS-Programm mit Hilfe der lag-Funktion bestimmt wird (Datei tank.sps):

```
data list file='c:\syntax\tank.txt'/
  datum 1-10 (date) kmstand 12-16 liter 18-21(1).
compute kmdiff=kmstand-lag(kmstand).
compute liter100=liter*100/kmdiff.
list datum,liter,kmdiff,liter100.
```

Die folgenden Ergebnisse werden ausgegeben:

```
      DATUM KMSTAND   KMDIFF LITER LITER100

16.12.2000  20580       ,    60,3      ,
23.12.2000  21250    670,00  57,4     8,57
04.01.2001  21874    624,00  56,6     9,07
17.01.2001  22476    602,00  56,3     9,35
28.01.2001  22954    478,00  45,4     9,50
12.02.2001  23450    496,00  48,6     9,80
27.02.2001  24020    570,00  57,0    10,00
14.03.2001  24611    591,00  56,7     9,59
```

In den ersten n Fällen (hier: $n = 1$) erzeugt die lag-Funktion einen fehlenden Wert.

Funktionen zur Erzeugung von Zufallszahlen

Insgesamt gibt es 24 Funktionen zur Erzeugung von Zufallszahlen, u. a. zu den 20 verfügbaren statistischen Verteilungsfunktionen (siehe diese). Informationen über alle Funktionen liefert die Menüwahl

Hilfe
Themen
Funktionen

Hier seien nur die beiden gebräuchlichsten Funktionen vorgestellt.

***rv.normal*(*x,s*):** Normalverteilte Zufallszahlen mit Mittelwert x und Standardabweichung s.

***rv.uniform*(*a,b*):** Gleichverteilte Zufallszahlen im Bereich von a bis b.

Der Einsatz von Zufallsfunktionen spielt bei SPSS-Auswertungen eigentlich keine Rolle. Sie können sie aber dazu benutzen, zu Testzwecken oder ähnlichem künstliche Datenmengen zu erzeugen. Das folgende SPSS-Programm (Datei simdat.sps) simuliert eine Datenmenge mit insgesamt 1000 Fällen und den Variablen *geschl* (Geschlecht), *gr* (Körpergröße) und *gew* (Körpergewicht):

```
input program.
loop #i=1 to 500.
compute geschl=1.
compute gr=rnd(rv.normal(175,7)).
compute gew=gr-100+rnd(rv.normal(0,10)).
end case.
end loop.
loop #i=1 to 500.
compute geschl=2.
compute gr=rnd(rv.normal(165,7)).
compute gew=gr-100+rnd(rv.normal(0,10)).
end case.
end loop.
variable labels geschl 'Geschlecht'/
  gr 'Koerpergroesse'/
  gew 'Koerpergewicht'.
value labels geschl 1 'maennlich' 2 'weiblich'.
end file.
end input program.
execute.
```

Es ist dies ein SPSS-Programm, das keine Datendatei vorfindet, sondern eine solche selbst erzeugt. In diesem Fall sind die dazu benötigten Anweisungen in die Anweisungsstruktur

```
input program.
.....
end input program.
```

einzuschließen (siehe Abschnitt 3.7).

Mit Hilfe der Schleifenstruktur (siehe Abschnitt 3.8)

```
loop #i=1 to 500.
.....
end loop.
```

werden 500 Fälle mit den Variablen *geschl*, *gr* und *gew* erzeugt. Die Anweisung *end case* signalisiert, dass nach der Bestimmung von jeweils drei Variablenwerten die Berechnungen für einen Fall abgeschlossen sind. Das Zeichen # zu Beginn des Variablennamens *#i* bedeutet, dass es eine so genannte Arbeitsvariable ist, die nicht in die Datendatei übernommen wird.

Bei der Simulation der Körpergröße wird die Zufallsfunktion *rv.normal* benutzt, wobei bei den Männern normalverteilte Zufallszahlen mit dem Mittelwert 175 und der Standardabweichung 7 und bei den Frauen solche mit dem Mittelwert 165 und gleicher Standardabweichung erzeugt werden. Was das Köpergewicht anbelangt, wird zunächst

durch *gr* – 100 das Normalgewicht berechnet und dann eine normalverteilte Zufallsgröße hinzugefügt, die wegen des angegebenen Mittelwertes 0 positiv oder negativ sein kann.

Die Anweisung *end file* markiert das Ende der Erstellung des Datenfiles. Laden Sie das Programm simdat.sps in den Syntax-Editor, und starten Sie das Programm. Sie erhalten plausible Daten, die zudem mit einem Korrelationskoeffizienten von 0,641 recht wirklichkeitsnah korrelieren.

Statistische Verteilungsfunktionen

Es gibt Funktionen zu den folgenden statistischen Verteilungen, wobei jeweils die entsprechenden Flächen unter der Verteilungskurve (also im Prinzip die Wahrscheinlichkeiten) berechnet werden: Beta, Cauchy, Chi-Quadrat, Exponential, F, Gamma, Laplace, Logistic, Lognormal, Normal, Pareto, t, Uniform, Weibull (kontinuierlich) sowie Bernoulli, Binomial, Geometric, Hypergeometric, Negativ-Binomial und Poisson (diskret). Zu den kontinuierlichen Funktionen gibt es entsprechende Umkehrfunktionen.

Innerhalb der gängigen SPSS-Auswertungen spielen diese Verteilungsfunktionen keine Rolle, da die Berechnung der Wahrscheinlichkeiten (p-Werte) stets Bestandteil der jeweiligen Prozeduren ist. Nützlich sind die von SPSS angebotenen Verteilungsfunktionen dann, wenn Sie eine Prüfgröße „mal eben" von Hand berechnet haben und dazu den p-Wert bestimmen möchten. Dies könnte z. B. beim t-Test nach Student der Fall sein, wenn Sie aus den bereits gegebenen Mittelwerten, Standardabweichungen und Fallzahlen die Prüfgröße t und die Anzahl der Freiheitsgrade df ermittelt haben.

Die vier gängigsten Verteilungsfunktionen und ihre Umkehrfunktionen werden in der folgenden Tabelle vorgestellt.

Funktion	Bedeutung
cdfnorm(z)	Fläche unter der Standardnormalverteilungskurve von $-\infty$ bis z
cdf.t(t,df)	Fläche unter der t-Verteilungskurve von $-\infty$ bis t bei df Freiheitsgraden
cdf.f(f,df1,df2)	Fläche unter der F-Verteilungskurve von $-\infty$ bis F bei $df1$, $df2$ Freiheitsgraden
cdf.chisq(chi2,df)	Fläche unter der Chi-Quadrat-Verteilungskurve von $-\infty$ bis Chi-Quadrat bei df Freiheitsgraden

Funktion	Bedeutung
idf.normal(p,0,1)	Umkehrfunktion zu *cdfnorm*
	(z-Wert zur vorgegebenen Fläche)
idf.t(p,df)	Umkehrfunktion zu *cdf.t*
	(t-Wert zur vorgegebenen Fläche bei *df* Freiheitsgraden)
idf.f(p,df1,df2)	Umkehrfunktion zu *cdf.f*
	(F-Wert zur vorgegebenen Fläche bei *df*1, *df*2 Freiheitsgraden
idf.chisq	Umkehrfunktion zu *cdf.chisq*
	(Chi-Quadrat-Wert zur vorgegebenen Fläche bei
	df Freiheitsgraden)

Haben Sie etwa bei einem t-Test die Prüfgröße $t = 2{,}674$ berechnet und ist die Anzahl der Freiheitsgrade $df = 18$, so berechnet sich wegen der Symmetrie der t-Verteilung der zugehörige p-Wert mit

```
compute p=2*(1-cdf.t(2.674,18)).
execute.
```

Die compute-Anweisung wird allerdings nur dann ausgeführt, wenn eine (beliebige) Datendatei geladen ist. In diesem Fall wird nach Ausführung der compute-Anweisung der Datendatei die Variable *p* hinzugefügt, die durchgängig den gleichen Wert (hier: $p = 0{,}02$) hat.

Eleganter ist es, wie schon bei der Beschreibung der Zufallsfunktionen gezeigt, ohne Laden einer Datendatei ein entsprechendes Eingabeprogramm zu formulieren (siehe Abschnitt 3.7):

```
input program.
compute p=2*(1-cdf.t(2.674,18)).
formats p(f5.3).
end case.
end file.
end input program.
list p.
```

Das Programm enthält die formats-Anweisung, mit der die Anzahl der Dezimalstellen des p-Wertes auf drei gestellt wird.

Im Folgenden sind vier solcher Eingabeprogramme (Dateien normal.sps, t.sps, f.sps, chiqu.sps) aufgeführt. Diese ermitteln zur Standardnormalverteilung, t-Verteilung, F-Verteilung und Chi-Quadrat-Verteilung zu vorgegebenen Prüfgrößen z, t, F und Chi-Quadrat und zugehörigen Freiheitsgraden den p-Wert. Dieser hat in allen Fällen den Wert 0,050.

```
comment Standardnormalverteilung.
input program.
compute p=2*(1-cdfnorm(1.96)).
formats p(f5.3).
end case.
end file.
end input program.
list p.

comment t-Verteilung.
input program.
compute p=2*(1-cdf.t(1.984,100)).
formats p(f5.3).
end case.
end file.
end input program.
list p.

comment F-Verteilung.
input program.
compute p=1-cdf.f(1.93,10,100).
formats p(f5.3).
end case.
end file.
end input program.
list p.

comment Chi-Quadrat-Verteilung.
input program.
compute p=1-cdf.chisq(3.841,1).
formats p(f5.3).
end case.
end file.
end input program.
list p.
```

Was die Umkehrfunktionen betrifft, so soll jeweils ein Programmbeispiel zur t- bzw. F-Verteilung gegeben werden. So gehen z. B. bei der Bestimmung von Konfidenzintervallen (gewöhnlich 95-%- oder 99-%-Konfidenzintervalle) zugehörige t- bzw. F-Werte in die Formeln ein. Die beiden folgenden Programme (Dateien tinvers.sps, finvers.sps) ermitteln diese Werte bei vorgebenen Freiheitsgraden.

```
comment Bestimmung eines t-Wertes (bei 100 Freiheitsgraden).
input program.
compute proz=95.
compute p=(proz+100)/200.
```

```
compute t=idf.t(p,100).
formats t(f5.3).
end case.
end file.
end input program.
list t.

comment Bestimmung eines F-Wertes (bei 10,100 Freiheitsgraden).
input program.
compute proz=95.
compute p=proz/100.
compute f=idf.f(p,10,100).
formats f(f5.3).
end case.
end file.
end input program.
list f.
```

Sie können die Verteilungsfunktionen auch benutzen, um komplette t- oder F-Tabellen usw. auszudrucken. Das folgende Programmbeispiel (Datei ttab.sps) erstellt eine t-Tabelle für Freiheitsgrade von 1 bis 500 und Signifikanzniveaus $p = 0{,}05$, 0,01 und 0,001.

```
comment t-Tabelle.
input program.
loop df=1 to 500.
compute t05=idf.t(0.975,df).
compute t01=idf.t(0.995,df).
compute t001=idf.t(0.9995,df).
formats t05,t01,t001 (f7.3).
end case.
end loop.
end file.
end input program.
list df,t05,t01,t001.
```

Logische Funktionen

Unter diese Rubrik fallen zwei nützliche Hilfsfunktionen, die die Werte 1 und 0 annehmen können. Sie können auch als logische Ausdrücke (siehe Abschnitt 3.2) Verwendung finden.

***any*(*var,x1,x2,...*):** Der Funktionswert ist 1, falls die betreffende Variable einen der Werte x1, x2, ... annimmt, sonst 0.

***range*(*var,a,b*):** Der Funktionswert ist 1, falls die betreffende Variable einen Wert zwischen den Grenzen a und b hat, sonst 0.

In einer Befragung von Studierenden wurde der Beruf des Vaters notiert, was in eine Variable *beruf* mit etwa 30 verschiedenen Codierungen mündete. Die Codierungen 3 = Arzt, 4 = Zahnarzt, 12 = Heilpraktiker, 23 = Psychotherapeut, 27 = Psychologe sollten dabei in einer Variablen „helfender Beruf" zusammengefasst werden. Mit Hilfe der any-Funktion kann dies folgendermaßen formuliert werden:

```
compute helfber=any(beruf,3,4,12,23,27).
variable labels helfber 'Helfender Beruf?'.
value labels helfber 1 'ja' 0 'nein'.
execute.
```

Das Gleiche leistet allerdings auch die recode-Anweisung (siehe Abschnitt 3.4).

Funktionen für Zeichenketten

Zeichenketten werden auf so genannten Textvariablen gespeichert (siehe Abschnitt 2.1.4). Werden Zeichenketten innerhalb der Syntax angegeben, sind sie in Apostrophe einzuschließen.

Einige der bisher vorgestellten Funktionen gibt es auch für Zeichenketten: *any, lag, max, min, range*. Ein Aufruf der Funktion *any* mit der Textvariablen *gebort* könnte z. B. so aussehen:

```
any(gebort,'Hamburg','Bremen','Kiel','Bremerhaven','Oldenburg')
```

Die speziell für Zeichenketten vorgesehenen Funktionen sind im Folgenden aufgeführt. Die mit *var* bezeichneten Argumente können nicht nur numerische Variablen oder Stringvariablen sein; an ihre Stelle können auch entsprechende numerische Konstanten bzw. Zeichenketten treten. Im Anschluss werden einige Beispiele gezeigt.

***concat*(*var*1, *var*2, ...):** Die Argumente werden ohne Zwischenräume zu einer weiteren Stringvariablen verkettet.

***index*(*var*1, *var*2):** Das Ergebnis ist ein numerischer Wert; es ist die erste Position, in der die Zeichenkette *var*2 in der Zeichenkette *var1* vorkommt. Falls *var*2 nicht vorkommt, wird das Ergebnis auf 0 gesetzt.

***length*(*var*):** Länge der Zeichenkette *var* (numerischer Wert)

***lower*(*var*):** Umwandlung aller Großbuchstaben der Variablen *var* in Kleinbuchstaben; die übrigen Zeichen bleiben unverändert.

***lpad*(*var*1, *var*2, *var*3):** Das Ergebnis ist eine Stringvariable, die so viele Zeichen hat, wie durch den numerischen Wert *var*2 bestimmt wird. Die Stringvariable *var*1 wird dabei von links her mit dem Zeichen aufgefüllt, das in der Stringvariablen *var*3 enthalten ist.

***ltrim*(*var*1, *var*2):** Beginnt die Stringvariable *var*1 mit dem in der Stringvariablen *var*2 enthaltenen Zeichen, so werden diese Zeichen gelöscht.

***number*(*var*, FORTRAN-Format):** Konvertiert, falls möglich, eine Zeichenkette in eine Zahl im angegebenen FORTRAN-Format.

***rindex*(*var*1, *var*2):** Das Ergebnis ist ein numerischer Wert; es ist die letzte Position, in der die Zeichenkette *var*2 in der Zeichenkette *var*1 vorkommt.

***rpad*(*var*1, *var*2, *var*3):** Das Ergebnis ist eine Stringvariable, die so viele Zeichen hat, wie durch den numerischen Wert *var*2 bestimmt wird. Die Stringvariable *var*1 wird dabei von rechts her mit dem Zeichen aufgefüllt, das in der Stringvariablen *var*3 enthalten ist.

***rtrim*(*var*1, *var*2):** Endet die Stringvariable *var*1 mit dem in der Stringvariablen *var*2 enthaltenen Zeichen, so werden diese Zeichen gelöscht.

***string*(*var*1, FORTRAN-Format):** Umwandlung einer numerischen Variablen im angegebenen Format in eine Stringvariable.

***substr*(*var*1, *var*2, *var*3):** Das Ergebnis ist der Teil der Stringvariablen *var*1, der ab Position *var*2 beginnt und *var*3 Zeichen lang ist.

***upcase*(*var*):** Umwandlung aller Kleinbuchstaben der Variablen *var* in Großbuchstaben; die übrigen Zeichen bleiben unverändert.

Einige Beispiele sollen den Gebrauch und den Sinn einiger Stringfunktionen näher erläutern.

Beispiel zur Funktion ltrim:

In der Datei string1.txt seien auf einer Stringvariablen die folgenden Berufe gespeichert:

```
  Maschinenbauingenieur
Gastronom
Steuerberater
    Schlosser
Schulleiter
Arzt
Manager
Dreher
  Dipl.-Ing.
Dipl.-Ing.
     Beamter
Kripo-Beamter
Dipl.-Handelslehrer
 Landwirtschaftsmeister
Arzt (Internist)
Arzt
```

Aufgrund nachlässiger Eingabe haben sich teilweise am Anfang Leerstellen eingeschlichen. Das folgende Programm (string1.sps) erstellt zu dieser Stringvariablen eine weitere Stringvariable, bei der diese Leerstellen gelöscht sind.

```
data list file='c:\syntax\string1.txt'
  /beruf 1-30 (a).
string beruf2 (a30).
compute beruf2=ltrim(beruf,' ').
list beruf,beruf2.
```

Die string-Anweisung definiert die Variable *beruf2* als Stringvariable mit 30 Zeichen. Das wird durch den Zusatz a30 kenntlich gemacht, wobei der Buchstabe a als Abkürzung für „alphanumerisch" steht.

Die list-Anweisung (siehe Abschnitt 5.1) dient zur Auflistung der Werte der beiden Stringvariablen:

```
BERUF                          BERUF2

  Maschinenbauingenieur        Maschinenbauingenieur
Gastronom                      Gastronom
Steuerberater                  Steuerberater
    Schlosser                  Schlosser
Schulleiter                    Schulleiter
Arzt                           Arzt
Manager                        Manager
Dreher                         Dreher
  Dipl.-Ing.                   Dipl.-Ing.
Dipl.-Ing.                     Dipl.-Ing.
     Beamter                   Beamter
Kripo-Beamter                  Kripo-Beamter
Dipl.-Handelslehrer            Dipl.-Handelslehrer
 Landwirtschaftsmeister        Landwirtschaftsmeister
Arzt (Internist)               Arzt (Internist)
Arzt                           Arzt
```

Natürlich braucht keine zweite Stringvariable erzeugt zu werden; die bestehende Variable kann auch selbst verändert werden:

```
data list file='c:\syntax\string1.txt'
  /beruf 1-30 (a).
compute beruf=ltrim(beruf,' ').
execute.
```

Beispiel zu den Funktionen index und substr:

In der Datei string2.txt sind die folgenden Berufe gespeichert:

```
Dreher
Arzt (Internist)
```

```
Arzt (HNO)
Jurist
Arzt (Orthopaede)
Steuerberater
Meister (Kommunik.-Techn.)
Arzt (Internist)
Arzt (Allgem.)
Rentner
Schulleiter (Ges.-Schule)
Elektrotechniker
Arzt (Urologe)
Kaufmann (Auto-)
Lehrer (Berufsschule)
```

Die in Klammern gesetzten Erläuterungen sollen im Sinne einer stringenteren Berufsbezeichnung entfernt werden. Betrachten Sie dazu das folgende Programm (string2.sps):

```
data list file='c:\syntax\string2.txt'
  /beruf 1-30 (a).
compute pos=index(beruf,'(').
if pos=0 pos=31.
string beruf2 (a30).
compute beruf2=substr(beruf,1,pos-1).
list beruf,beruf2.
```

Mit Hilfe der Funktion index wird die Position der geöffneten Klammer festgestellt, mit der Funktion substr werden dann die Zeichen vor dieser Position in der Stringvariablen *beruf*2 gespeichert. Die Auflistung ergibt:

```
BERUF                          BERUF2

Dreher                         Dreher
Arzt (Internist)               Arzt
Arzt (HNO)                     Arzt
Jurist                         Jurist
Arzt (Orthopaede)              Arzt
Steuerberater                  Steuerberater
Meister (Kommunik.-Techn.)     Meister
Arzt (Internist)               Arzt
Arzt (Allgem.)                 Arzt
Rentner                        Rentner
Schulleiter (Ges.-Schule)      Schulleiter
Elektrotechniker               Elektrotechniker
Arzt (Urologe)                 Arzt
Kaufmann (Auto-)               Kaufmann
Lehrer (Berufsschule)          Lehrer
```

Auch hier kann eine Änderung der Variablen *beruf* selbst erfolgen:

```
data list file='c:\syntax\string2.txt'
  /beruf 1-30 (a).
compute pos=index(beruf,'(').
if pos=0 pos=31.
compute beruf=substr(beruf,1,pos-1).
execute.
```

Beispiel zu den Funktionen substr, lower und concat:

In der Datei string3.txt seien die folgenden Berufe gespeichert:

```
MASCHINENBAUINGENIEUR
GASTRONOM
STEUERBERATER
SCHLOSSER
SCHULLEITER
ARZT
MANAGER
DREHER
BEAMTER
LANDWIRTSCHAFTSMEISTER
ARZT
JURIST
```

Die Eingabe entstammt offensichtlich noch der Lochkarten-Zeit, als Groß- und Kleinschreibung noch nicht möglich war. Die Berufsbezeichnungen sollen nun in diese übliche Schreibweise überführt werden (Programm string3.sps):

```
data list file='c:\syntax\string3.txt'
  /beruf 1-30 (a).
string beruf1 (a1)/beruf2(a29)/beruf3(a30).
compute beruf1=substr(beruf,1,1).
compute beruf2=substr(beruf,2,29).
compute beruf2=lower(beruf2).
compute beruf3=concat(beruf1,beruf2).
list beruf,beruf3.
```

Das erste Zeichen wird in die Stringvariable *beruf*1, die restlichen Zeichen werden in die Stringvariable *beruf*2 überführt. Auf die Variable *beruf*2 wird dann die lower-Funktion angewandt, welche die Großbuchstaben in Kleinbuchstaben umwandelt. Schließlich werden die Variablen *beruf*1 und *beruf*2 zur Stringvariablen *beruf*3 verkettet:

```
BERUF                          BERUF3

MASCHINENBAUINGENIEUR          Maschinenbauingenieur
GASTRONOM                      Gastronom
STEUERBERATER                  Steuerberater
SCHLOSSER                      Schlosser
SCHULLEITER                    Schulleiter
ARZT                           Arzt
MANAGER                        Manager
DREHER                         Dreher
BEAMTER                        Beamter
LANDWIRTSCHAFTSMEISTER         Landwirtschaftsmeister
ARZT                           Arzt
JURIST                         Jurist
```

Das Programm kann wie folgt kürzer gefasst werden:

```
data list file='d:\syntax\string3.txt'
  /beruf 1-30 (a).
compute beruf=concat(substr(beruf,1,1),lower(substr(beruf,2,29))).
execute.
```

Wer gerne einmal mit Zeichenketten-Funktionen üben möchte, dem sei die Datei string.sav empfohlen, die Berufsangaben der Väter und Mütter von 376 Studierenden enthält.

3.2 Die if-Anweisung

Die if-Anweisung ermöglicht es, die Berechnung einer Variablen von einer Bedingung abhängig zu machen. Die allgemeine Form ist:

```
if Bedingung Zielvariable = Ausdruck.
```

Beim Ausdruck handelt es sich wie bei der compute-Anweisung im Regelfall um einen arithmetischen Ausdruck. Dabei ist die Bedingung als *logischer Ausdruck* zu formulieren. Bevor näher auf die Regeln zur Bildung der logischen Ausdrücke eingegangen wird, seien einige typische Anwendungsbeispiele für if-Anweisungen gegeben.

Der BMI (Body Mass Index, siehe Abschnitt 3.1.1) soll nur für Personen über 18 Jahre berechnet werden:

```
if alter > 18 bmi=gew/(gr/100)**2.
execute.
```

Für die Personen bis 18 Jahre wird der Wert der Variablen *bmi* auf „fehlend" gesetzt. In diesem Beispiel ist „*alter* > 18" die Bedingung (der logische Ausdruck). Falls Sie diesen

aus Gründen der besseren Übersichtlichkeit deutlicher absetzen möchten, können Sie ihn in runde Klammern einschließen:

```
if (alter > 18) bmi=gew/(gr/100)**2.
execute.
```

Sie möchten die fehlenden Werte der Variablen *chol* durch den (vorher berechneten) Mittelwert der gültigen Fälle ersetzen:

```
if missing(chol) = 1 chol=184.5.
execute.
```

Alle Werte der Variablen *tgl*, die größer als 500 sind, sollen auf 500 gesetzt werden:

```
if tgl > 500 tgl = 500.
execute.
```

In den beiden letzten Beispielen wird nicht eine neue Variable erzeugt, sondern eine bestehende Variable verändert.

Die bisher gezeigten Bedingungen sind logische Ausdrücke der einfachsten Bauart:

```
alter > 18
missing(chol) = 1
tgl > 500
```

Diese einfachen logischen Ausdrücke nennt man „Vergleiche“ oder „Relationen“. Sie haben den Aufbau

```
Ausdruck Vergleichsoperator Ausdruck
```

In dem meisten Fällen wird eine Variable mit einer Konstanten verglichen. Insgesamt gibt es sechs verschiedene Vergleichsoperatoren:

Vergleichsoperator	alternative Darstellung	Bedeutung
=	eq	gleich
<>	ne	ungleich
<	lt	kleiner (less than)
<=	le	kleiner oder gleich (less than or equal to)
>	gt	größer (greater than)
>=	ge	größer oder gleich (greater than or equal to)

Jeder Vergleich und im Allgemeinen jeder logische Ausdruck hat nur zwei mögliche Ergebnisse: die Wahrheitswerte *true* und *false*. Genau in dem Fall, wo der logische Ausdruck den Wahrheitswert *true* hat und somit die Bedingung erfüllt ist, wird die betreffende Variablenberechnung durchgeführt.

Zwei (oder auch mehr) Vergleiche können durch die logischen Operatoren *and* bzw. *or* verbunden werden. In einer medizinischen Studie sollten Patienten mit Cholesterin bis 200 und Tryglizeriden bis 150 mit Patienten mit Cholesterin ab 260 und Tryglizeriden ab 200 hinsichtlich ihres systolischen Blutdrucks verglichen werden. Hier ist mit Hilfe zweier if-Anweisungen zunächst eine Gruppenvariable zu konstruieren, die dann bei der t-test-Anweisung Verwendung findet:

```
if chol <= 200 and tgl <= 150 gruppe=1.
if chol >= 260 and tgl >= 200 gruppe=2.
t-test groups=gruppe(1,2)/variables=rrs.
```

Ein logischer Ausdruck (eine Bedingung), der aus einer and-Verbindung zweier Vergleiche besteht, ist genau dann „wahr" (true), wenn beide Vergleiche wahr sind.

Hingegen ist ein logischer Ausdruck, der aus einer or-Verbindung zweier Vergleiche besteht, in all den Fällen wahr, bei denen mindestens einer der beiden Vergleiche wahr ist. Soll im gegebenen Beispiel die zweite Gruppe aus Patienten bestehen, deren Cholesterin-*oder* Triglyzeridwerte erhöht sind, ist folgendermaßen zu formulieren:

```
if chol <= 200 and tgl <= 150 gruppe=1.
if chol >= 260 or tgl >= 200 gruppe=2.
t-test groups=gruppe(1,2)/variables=rrs.
```

Die Wahrheitswerte von and-Verbindungen (Konjunktionen) und or-Verbindungen (Disjunktionen) sind in der folgenden Tabelle zusammengestellt. Der Tabelle ist auch die Behandlung fehlender Werte zu entnehmen.

Verbindung	Ergebnis
true and true	true
true and false	false
false and false	false
true and missing	missing
false and missing	false
missing and missing	missing

Verbindung	Ergebnis
true or true	true
true or false	true
false or false	false
true or missing	true
false or missing	missing
missing or missing	missing

Besonders aufpassen muss man, wenn in logischen Ausdrucken sowohl and- als auch or-Verbindungen auftreten. Hier ist dann zu beachten, dass zuerst die and- und dann die or-Verbindungen ausgewertet werden. Beispiele dieser Art werden in Kap. 4 gezeigt.

Als dritten logischen Operator gibt es den Operator *not*. Dieser kehrt den Wahrheitswert des dahinter in Klammern eingeschlossenen logischen Ausdrucks um.

3.3 Die count-Anweisung

In Abschnitt 2.1.7 wurde ein aus 48 Items bestehender Fragebogen vorgestellt, der das Verhalten, Fühlen und Handeln der Befragten erforscht. Die folgenden Items gehören dabei zu einer psychologischen Skala, die man „Extraversion" nennt.

1. *Haben Sie oft Lust, etwas Aufregendes zu erleben?*
3. *Sind Sie im Allgemeinen ohne Sorgen?*
5. *Nehmen Sie sich Zeit, um erst einmal über die Lage nachzudenken, bevor Sie etwas tun?*
7. *Handeln und reden Sie gewöhnlich schnell, ohne zwischendurch lange nachzudenken?*
9. *Lassen Sie sich leicht herausfordern?*
11. *Folgen Sie oft Ihren spontanen Einfällen?*
13. *Lesen Sie im Allgemeinen lieber als sich mit anderen Menschen zu treffen?*
15. *Gehen Sie gern viel aus?*
17. *Haben Sie lieber wenige, dafür aber besonders gute Freunde?*
19. *Wenn man Sie anschreit, schreien Sie dann zurück?*
21. *Können Sie im Allgemeinen in einer fröhlichen Gesellschaft richtig mitmachen und sich gut amüsieren?*
23. *Halten andere Leute Sie für sehr lebhaft?*
25. *Halten Sie sich in Gegenwart anderer Menschen meistens zurück?*

27. *Wenn Sie über irgend etwas genau Bescheid wissen möchten, sehen Sie dann lieber in einem Buch nach als andere zu fragen?*
29. *Haben Sie Arbeiten gern, die konzentrierte Aufmerksamkeit erfordern?*
31. *Sind Sie ungern unter Leuten, die sich gegenseitig durch den Kakao ziehen?*
33. *Unternehmen Sie gern etwas, bei dem Sie schnell handeln müssen?*
35. *Sind Ihre Bewegungen langsam und bedächtig?*
37. *Sprechen Sie so gern mit anderen Menschen, dass Sie keine Gelegenheit auslassen, sich mit einem Fremden zu unterhalten?*
39. *Wären Sie sehr unglücklich, wenn Sie nicht meistens mit vielen anderen zusammensein könnten?*
41. *Könnten Sie von sich behaupten, einigermaßen selbstbewusst zu sein?*
43. *Fällt es Ihnen schwer, auf einer lebhaften Gesellschaft wirklich aus sich heraus zu gehen?*
45. *Gelingt es Ihnen leicht, eine langweilige Party in Schwung zu bringen?*
47. *Spielen Sie anderen gern kleine Streiche?*

Die Antworten auf diese Fragen seien auf den Variablen *v*1 bis *v*48 mit 1 = ja und 2 = nein codiert.

Im Sinne von Extraversion sind die Items 1, 3, 7, 9, 11, 15, 19, 21, 23, 33, 37, 39, 41, 45 und 47 mit „ja“ und die Items 5, 13, 17, 25, 27, 29, 31, 35 und 43 mit „nein“ zu beantworten. Die Anzahl der „Richtigantworten“ ist der Skalenwert. In SPSS kann dies mit Hilfe der count-Anweisung formuliert werden:

```
count extra=v1,v3,v7,v9,v11,v15,v19,v21,v23,v33,v37,v39,v41,v45,v47 (1)
            v5,v13,v17,v25,v27,v29,v31,v35,v43 (2).
execute.
```

Es wird ausgezählt, bei wie vielen Variablen der in Klammern angegebene Wert zutrifft. Anstelle eines einzelnen Wertes wie im gegebenen Beispiel kann eine Werteliste angegeben werden. Dabei sind die Schlüsselwörter *lowest, highest, thru, missing, sysmis* erlaubt.

3.4 Die recode-Anweisung

Die ursprünglich erfassten Daten können in SPSS „umcodiert" werden. Dies geschieht vor allem in den beiden folgenden Situationen:

- Bei kategorialen (nominal- oder ordinalskalierten) Variablen erweist sich die ursprünglich gewählte Codierung als zu fein. Hier werden dann Codierungen zusammengefasst.
- Bei intervallskalierten Variablen werden Klassen gebildet.

In einer zahnmedizinischen Untersuchung an Kindern wurde nach der Putzhäufigkeit der Zähne (Variable *putz*) gefragt, die folgendermaßen codiert wurde:

1 = 1 mal pro Woche
2 = 2-3 mal pro Woche
3 = 4–6 mal pro Woche
4 = 1 mal pro Tag
5 = oefter

Da die ersten drei Kategorien recht selten auftraten, sollen diese der Kategorie „1 mal pro Tag" zugeschlagen werden. Dies kann mit folgender recode-Anweisung erreicht werden:

```
recode putz (1,2,3,4=1) (5=2) into putz2.
value labels putz2 1 'bis 1 mal pro Tag' 2 'oefter'.
execute.
```

Dem Anweisungsnamen *recode* folgt der betreffende Variablenname, gefolgt von in Klammern eingeschlossenen Konstrukten der Form

Liste alter Werte = neuer Wert

Dem Schlüsselwort *into* folgt der Name einer neuen Variablen; diese nimmt dann die umcodierten Werte auf. Der Name *putz2* dieser neuen Variablen erscheint im gegebenen Beispiel etwas einfallslos, er soll aber darauf hinweisen, dass es sich um die Umcodierung der Variablen *putz* in zwei Kategorien handelt.

Die Einführung einer neuen Variablen ist nicht zwingend notwendig. Sie hätten auch formulieren können

```
recode putz (1,2,3,4=1) (5=2).
value labels putz 1 'bis 1 mal pro Tag' 2 'oefter'.
execute.
```

In diesem Fall werden die Werte der ursprünglichen Variablen *putz* selbst verändert. Diese Vorgehensweise ist in vielen Fällen nicht zu empfehlen, da hierdurch die alte Codierung, die man vielleicht doch noch einmal braucht, verloren geht.

Die value-labels-Anweisung im Anschluss an die recode-Anweisung ist natürlich wahlfrei. Gerade bei solchen Umcodierungen ist aber die Vergabe von Labels sehr zu empfehlen, da man sonst leicht den Überblick verliert.

Als weiteres Beispiel sei eine Variable namens *schule* betrachtet, die den Schulabschluss mit folgender (ungeschickter) Codierung wiedergibt:

1 = Gymnasium
2 = Realschule
3 = Hochschule
4 = Sonderschule
5 = Fachhochschule
6 = Hauptschule

Ungeschickt ist diese Codierung deshalb, weil man mit der folgenden Codierung eine ordinalskalierte Variable erstellen könnte:

1 = Sonderschule
2 = Hauptschule
3 = Realschule
4 = Gymnasium
5 = Fachhochschule
6 = Hochschule

Dies kann aus der ersten Codierung mit folgender recode-Anweisung erreicht werden:

```
recode schule (4=1) (6=2) (2=3) (1=4) (5=5) (3=6).
execute.
```

In diesem Fall erscheint die Bildung einer neuen Variablen entbehrlich. Mit der Recodierung ist diesmal kein Informationsverlust verbunden, und die alte Codierung hat sich als untauglich erwiesen.

Das Konstrukt „(5=5)“ ist nicht entbehrlich, da sonst die alte Codierung 5 als „fehlend“ deklariert würde.

In Abschnitt 3.3 wurden 48 Variablen *v*1 bis *v*48 vorgestellt, die mit 1 = ja und 2 = nein codiert sind und von denen 24 Variablen zu einer Skala „Extraversion“ gehören. Einige Variablen (*v*5, *v*13, *v*17, *v*25, *v*27, *v*29, *v*31, *v*35, *v*43) sind im Sinne einer „Richtigantwort“ anders gepolt. Wollen Sie daher mit diesen Variablen z. B. eine Reliabilitätsanalyse ausführen, so ist vorher bei diesen Variablen die Codierung zu vertauschen:

```
recode v5,v13,v17,v25,v27,v29,v31,v35,v43 (1=2) (2=1).
value labels v1 to v48 1 'Richtigantwort' 2 'Falschantwort'.
execute.
```

Anstelle einer einzelnen Variablen kann also auch eine Variablenliste angegeben werden. Die angegebenen Variablen werden dann in gleicher Weise recodiert. Bei der Angabe der beiden Konstrukte

(1=2) (2=1)

zucken erfahrene Programmierer etwas zusammen. In SPSS funktioniert dies aber, da es bedeutet, dass die *alte* Codierung 1 in 2 und die *alte* Codierung 2 in 1 umgewandelt wird.

Bei der eingangs erwähnten zahnmedizinischen Studie wurde auch die Anzahl der erkrankten bzw. fehlenden Zähne erfasst (Variable *dmft*). Dabei wurde festgestellt, dass sich bis zu 19 Zähne als erkrankt bzw. fehlend erwiesen.

Für eine nachfolgende statistische Prozedur (logistische Regression) soll die Variable dichotomiert werden:

0 = kein erkrankter bzw. fehlender Zahn

1 = mindestens ein erkrankter bzw. fehlender Zahn

Bei einer Recodierung müssten also alle Werte von 1 bis 19 einzeln aufgeführt werden. Um dies zu vermeiden, wurde das Schlüsselwort *thru* geschaffen:

```
recode dmft (0=0) (1 thru 19=1) into dmft2.
value labels dmft2 0 'gesund' 1 'erkrankt'.
execute.
```

Bei dieser Formulierung der recode-Anweisung musste schon einmal eine Häufigkeitsauszählung der Variablen *dmft* vorgenommen worden sein, sonst wäre der Maximalwert nicht bekannt. Zahnmedizinische Grundkenntnisse vorausgesetzt, hätten Sie auch formulieren können

```
recode dmft (0=0) (1 thru 32=1) into dmft2.
```

Eleganter und Denkarbeit bei solch offenen Klassen abnehmend ist der Gebrauch des Schlüsselwortes *highest*:

```
recode dmft (0=0) (1 thru highest=1) into dmft2.
execute.
```

Allerdings ist hier zu beachten, dass gegebenenfalls der benutzerdefinierte fehlende Wert mit einbezogen wird, falls er in den angegebenen Bereich fällt. Dies kann durch folgenden Trick verhindert werden, der die beiden Schlüsselwörter *missing* und *sysmis* benutzt:

```
recode dmft (missing=sysmis) (0=0) (1 thru highest=1) into dmft2.
execute.
```

Das Schlüsselwort *missing* ist nur in der Liste der *alten* Werte erlaubt und bezeichnet sowohl den benutzerdefinierten als auch den systemdefinierten fehlenden Wert. Das Schlüsselwort *sysmis* bezeichnet den systemdefinierten fehlenden Wert. Mit Hilfe des Konstrukts

(missing = sysmis)

werden also vor den anderen Recodierungen alle benutzerdefinierten fehlenden Werte in systemdefinierte fehlende Werte umgewandelt.

Eine weitere, in diesem Fall aber untaugliche Variante, benutzt das Schlüsselwort *else*:

```
recode dmft (0=0) (else=1) into dmft2.
execute.
```

Alle Werte außer der 0 werden also zu 1 codiert; dies schließt aber sowohl den benutzerdefinierten als auch den systemdefinierten fehlenden Wert mit ein.

Manchmal möchte man nur wenige Codierungen abändern und die anderen belassen. So sei die Variable *fam* (Familienstand) folgendermaßen codiert:

1 = ledig

2 = verheiratet

3 = verwitwet

4 = geschieden

5 = getrennt lebend

Die Getrenntlebenden sollen zu den Geschiedenen geschlagen, die sonstigen Codierungen aber belassen werden. Hierzu benutzt man das Schlüsselwort *copy*:

```
recode fam (5=4) (else=copy) into fam4.
execute.
```

Bei Klasseneinteilungen intervallskalierter Variablen, z. B. dem in Jahren angegebenen Alter (Variable *alter*), verfährt man folgendermaßen:

```
recode alter (lowest thru 30=1) (31 thru 50=2) (51 thru highest=3)
       into alterk.
variable labels alterk 'Altersklasse'.
value labels 1 'bis 30 Jahre' 2 '31–50 Jahre' 3 'ueber 50 Jahre'.
execute.
```

Das Schlüsselwort *lowest* begrenzt die untere offene Klasse nach unten, das bereits bekannte Schlüsselwort *highest* die obere offene Klasse nach oben. Sie können dafür auch die Abkürzungen *lo* bzw. *hi* verwenden.

In Abschnitt 3.1.1 wurde der Body Mass Index (BMI) vorgestellt. Anhand dieses Indexes bezeichnet man Personen mit einem BMI bis 25 als normalgewichtig, darüber bis einschließlich 30 als übergewichtig und über 30 als adipös. Ist der BMI als Variable *bmi* gegeben, kann man diese Klassenbildung in SPSS folgendermaßen umsetzen:

```
recode bmi (lowest thru 25=1) (25 thru 30=2) (30 thru highest=3)
           into bmi3.
value labels bmi3 1 'normalgewichtig' 2 'uebergewichtig' 3 'adipoes'.
execute.
```

Bei dieser Art von Recodierung gilt die Regel, dass jeder Wert nur einmal umcodiert wird. So fällt der Wert 25 in die erste Klasse, die Umcodierangabe im zweiten Konstrukt

```
25 thru 30=2
```

hat sozusagen das Nachsehen.

Diese Beispiele sollten genügen, um Sie mit der Technik des Recodierens vertraut zu machen. Es ist ein häufig benutztes Verfahren und mit Hilfe der Befehlssyntax wohl bequemer zu lösen als über entsprechende Dialogboxen.

3.5 Die do-repeat-Anweisung

Sind für mehrere Variablen Berechnungen nach der gleichen Formel auszuführen, kann der Schreibaufwand durch Benutzung der do-repeat-Anweisung erheblich reduziert werden.

Als Beispiel betrachten wir die Datei gewab.txt, die neben einer fortlaufenden Fallnummerierung zunächst das Körpergewicht und die Körpergröße von 12 übergewichtigen Probanden enthält. Diese Probanden unterzogen sich einer Abmagerungskur; die Gewichtsabnahmen an zehn Versuchstagen gegenüber dem Anfangsgewicht wurden entsprechend festgehalten:

```
 1 100,9 173  0,5  1,2  2,6  3,6  4,6  6,0  7,3  8,4  9,4 10,9
 2  99,8 168  0,8  1,8  2,8  3,7  4,3  5,6  6,4  7,3  8,0  8,7
 3  99,5 172  1,2  2,5  3,3  4,7  5,7  6,7  8,1  9,0  9,5 10,6
 4 110,7 180  0,9  1,5  2,9  3,0  4,9  6,0  7,1  7,2  9,0  9,9
 5 109,3 178  1,0  1,9  2,0  3,5  4,2  5,7  6,0  7,0  7,8  9,4
 6 100,3 172  1,5  2,5  4,0  5,1  6,0  6,5  6,6  7,6  8,9 10,0
 7  99,5 175  0,3  1,9  3,5  4,0  5,1  7,4  7,6  8,9  9,9 11,5
 8  98,7 167  1,2  1,9  3,1  4,5  5,6  7,0  8,9  9,0 10,8 10,9
 9 109,6 176  1,8  2,9  3,2  3,9  4,9  5,4  6,5  7,8  8,4  8,9
10 102,7 180  0,5  0,9  1,8  2,9  3,9  4,5  5,6  6,5  6,6  7,3
11 105,6 174  1,0  1,9  2,5  3,8  4,5  5,6  6,9  7,6  8,4  9,8
12  99,8 170  2,0  2,9  3,8  4,9  6,7  7,6  8,4  9,0 10,9 11,9
```

Ein Einleseprogramm hierzu kann folgendermaßen aussehen (Datei gewab.sps):

```
data list file='c:\syntax\gewab.txt'/
  nr 1-2 gewicht 4-8 (1) groesse 10-12 a1 to a10 13-62 (1).
execute.
```

Zum Ausgangswert und zu den zehn Folgezeitpunkten soll der in Abschnitt 3.1.1 erläuterte Broca-Index berechnet werden. Dies gelingt mit den folgenden Anweisungen:

```
compute broca0=gewicht/(groesse-100)*100.
compute broca1=(gewicht-a1)/(groesse-100)*100.
compute broca2=(gewicht-a2)/(groesse-100)*100.
compute broca3=(gewicht-a3)/(groesse-100)*100.
compute broca4=(gewicht-a4)/(groesse-100)*100.
compute broca5=(gewicht-a5)/(groesse-100)*100.
compute broca6=(gewicht-a6)/(groesse-100)*100.
compute broca7=(gewicht-a7)/(groesse-100)*100.
compute broca8=(gewicht-a8)/(groesse-100)*100.
compute broca9=(gewicht-a9)/(groesse-100)*100.
compute broca10=(gewicht-a10)/(groesse-100)*100.
execute.
```

Es fällt auf, dass zumindest die compute-Anweisungen für die Variablen *broca*1 bis *broca*10 von gleicher Bauart sind. Sie variieren nur in der jeweiligen Ergebnisvariablen und den Variablen für die Gewichtsabnahme. Dies ist ein typischer Fall für den Gebrauch einer do-repeat-Anweisung, in der Platzhalter für die Variablen *broca*1 bis *broca*10 sowie *a*1 bis *a*10 definiert werden, die dann in eine gemeinsame Formel eingehen:

```
compute broca0=gewicht/(groesse-100)*100.
do repeat b=broca1 to broca10/a=a1 to a10.
compute b=(gewicht-a)/(groesse-100)*100.
end repeat.
execute.
```

Die do-repeat-Anweisung definiert die Platzhalter *b* und *a*, welche in der nachfolgenden compute-Anweisung Verwendung finden. Diese „Schleife" wird durch die end-repeat-Anweisung abgeschlossen.

Die durch *do repeat* und *end repeat* definierte Schleife wird insgesamt zehnmal durchlaufen, wobei im ersten Schritt für *b* und *a* die Variablen *broca*1 bzw. *a*1, im zweiten Schritt die Variablen *broca*2 und *a*2 eingesetzt werden usw. Die Namen der Platzhalter können innerhalb der Namensregeln frei gewählt werden; sie sind nur innerhalb der jeweiligen Schleife definiert.

Die Platzhalter können für Variablen- oder Wertelisten oder auch für gemischte Listen aus Variablen und Werten stehen.

Bei genauerem Überlegen kann auch die Berechnung von *broca*0 in die Schleife mit einbezogen werden:

```
do repeat b=broca0 to broca10/a=0,a1 to a10.
compute b=(gewicht-a)/(groesse-100)*100.
end repeat.
execute.
```

Ein zweites Beispiel beschäftigt sich mit einem Problem, wie es sich bei Fragen mit Mehrfachantworten ergeben kann. Bei einer Touristenbefragung in Kenia wurde u. a. die Frage gestellt „Welche Vor- bzw. Nachteile bringt Ihrer Meinung nach der Tourismus für Kenia?“ . Dabei waren die wie folgt codierten neun Antwortmöglichkeiten vorgegeben:

1 = Devisenzufluss

2 = Verteuerung der Lebensmittel

3 = Umweltbelastungen

4 = Arbeitsplätze

5 = Ausbau der Infrastruktur

6 = Landflucht

7 = Völkerverständigung

8 = Zerstörung von Kulturen

9 = Erhaltung von Kulturen

Die befragten Touristen gaben maximal sechs Antworten, die unter der angegebenen Codierung zusammen mit einer fortlaufenden Fallnummerierung und der Angabe des Geschlechts der Befragten in der Datei kenia.txt gespeichert sind. Deren erste Zeilen seien im Folgenden aufgelistet:

```
 1 1 13469
 2 1 136789
 3 1 78
 4 1 3468
 5 1 1378
 6 1 1348
 7 2 48
 8 2 13467
 9 1 34678
10 2 147
```

Mit den maximal sechs Mehrfachantworten wurde linksbündig ein Bereich von sechs Spalten gefüllt, denen sechs Variablen zugeordnet werden. So hat der erste Proband die Antworten 1, 3, 4, 6 und 9 gegeben, der dritte Proband die Antworten 7 und 8 usw. Man nennt dies die kategoriale Methode bei der Analyse von Mehrfachantworten. Das Einleseprogramm sieht wie folgt aus (Datei kenia.sps):

```
data list file='c:\syntax\kenia.txt'/nr 1-3 g 5 vn1 to vn6 7-12.
variable labels g 'Geschlecht'.
value labels g 1 'maennlich' 2 'weiblich'/
  vn1 to vn6 1 'Devisenzufluss' 2 'Verteuerung Lebensmittel'
             3 'Umweltbelastungen' 4 'Arbeitsplaetze'
             5 'Ausbau Infrastruktur' 6 'Landflucht'
             7 'Voelkerverstaendigung' 8 'Zerstoerung Kulturen'
             9 'Erhaltung Kulturen'.
execute.
```

Der Nachteil dieser Methode ist, dass es keine Variable gibt, die z. B. angibt, ob ein Befragter „Völkerverständigung" als Antwort gegeben hat. Die betreffende Information ist gegebenenfalls in einer der Variablen *vn*1 bis *vn*6 versteckt.

Eine solche Variable mit der Codierung 1 = ja und 2 = nein wird durch die folgenden Anweisungen erzeugt:

```
compute voelk=2.
if vn1=7 or vn2=7 or vn3=7 or vn4=7 or vn5=7 or vn6=7 voelk=1.
variable labels voelk 'Voelkerverstaendigung'.
value labels voelk 1 'ja' 2 'nein'.
execute.
```

Alternativ, wenn auch hier keinen Schreibvorteil bringend, können Sie die Variable *voelk* mit Hilfe der do-repeat-Anweisung kreieren:

```
compute voelk=2.
do repeat vn=vn1 to vn6.
if vn=7 voelk=1.
end repeat.
execute.
```

Wollen Sie alle neun Antwortmöglichkeiten als dichotome Variable formulieren, so verfahren Sie wie folgt:

```
do repeat v=v1 to v9/k=1,2,3,4,5,6,7,8,9.
compute v=2.
if vn1=k or vn2=k or vn3=k or vn4=k or vn5=k or vn6=k v=1.
end repeat.
variable labels v1 'Devisenzufluss'/
  v2 'Verteuerung Lebensmittel'/
  v3 'Umweltbelastungen'/
  v4 'Arbeitsplaetze'/
  v5 'Ausbau Infrastruktur'/
  v6 'Landflucht'/
  v7 'Voelkerverstaendigung'/
```

```
  v8 'Zerstoerung Kulturen'/
  v9 'Erhaltung Kulturen'.
value labels v1 to v9 1 'ja' 2 'nein'.
execute.
```

Die do-repeat-Anweisung ist über Dialogboxen nicht umzusetzen.

3.6 Die do-if-Struktur

Mit der if-Anweisung (siehe Abschnitt 3.2) kann nur die Berechnung *einer* Variablen von einer Bedingung abhängig gemacht werden. Die Struktur *do if — end if* hingegen erlaubt es, *mehrere* Variablenberechnungen unter die gleiche Bedingung zu stellen. Mit der Variante *else if* kann angegeben werden, was bei Nichterfüllung der Bedingung gerechnet werden soll.

Wir wollen annehmen, dass Sie mit den folgenden compute-Anweisungen drei Quotienten bilden wollen:

```
compute q1=x/a.
compute q2=y/a.
compute q3=z/a.
execute.
```

Weiterhin wollen wir annehmen, dass die Variable *a* den Wert 0 haben kann. In diesen Fällen können die Quotienten nicht berechnet werden, und es erscheint jeweils eine entsprechende Warnung und der Hinweis darauf, dass der Ergebnisvariablen der systemdefinierte fehlende Wert zugewiesen wird. Um dies zu vermeiden, kann man die Berechnung der Quotienten jeweils unter eine entsprechende Bedingung stellen:

```
if a > 0 compute q1=x/a.
if a > 0 compute q2=y/a.
if a > 0 compute q3=z/a.
execute.
```

Eleganter und kürzer geht es in solchen Fällen mit der durch *do if* und *end if* begrenzten Struktur:

```
do if a > 0.
compute q1=x/a.
compute q2=y/a.
compute q3=z/a.
end if.
execute.
```

Die do-if-Anweisung mit der allgemeinen Form

do if Bedingung.

markiert den Beginn der Anweisungsfolge, die unter die betreffende Bedingung (siehe Abschnitt 3.2) gestellt werden soll. Die end-if-Anweisung, die keine weiteren Spezifikationen hat, markiert das Ende der Anweisungsfolge.

Ist die Bedingung in der do-if-Anweisung erfüllt (genauer: hat der betreffende logische Ausdruck den Wert *true*), werden die nachfolgenden Anweisungen ausgeführt; im anderen Fall wird den Ergebnisvariablen der systemdefinierte fehlende Wert zugewiesen.

Nun könnte es sein, dass es eine „Ersatzvariable" (im Folgenden *b* genannt) gibt, die den Wert 0 nicht annimmt und die an die Stelle der Variablen *a* treten kann, falls der Wert dieser Variablen 0 ist. In diesem Fall kann dann eine Alternative formuliert werden:

```
do if a > 0.
compute q1=x/a.
compute q2=y/a.
compute q3=z/a.
else.
compute q1=x/b.
compute q2=y/b.
compute q3=z/b.
end if.
execute.
```

Diese Struktur hat also die folgende allgemeine Form:

```
do if Bedingung.
  Anweisungen
else.
  Anweisungen
end if.
```

Typisch ist auch folgende Anwendung. Die Patienten einer klinischen Studie sollen bezüglich eines bestimmten Parameters (Variable *ligg*) in „normal" und „erhöht" eingeteilt werden. Der Grenzwert sei aber je nach Geschlecht unterschiedlich, nämlich 380 bei den Männern (Variable *geschl* = 1) und 300 bei den Frauen. Das ist folgendermaßen zu formulieren:

```
do if geschl=1.
recode ligg (lowest thru 380=1) (380 thru highest=2) into ligg2.
else.
recode ligg (lowest thru 300=1) (300 thru highest=2) into ligg2.
end if.
value labels ligg2 1 'normal' 2 'erhoeht'.
execute.
```

Das funktioniert deshalb, weil es zu „männlich" (*geschl* = 1) nur die Alternative „weiblich" gibt. Fehlt die Angabe des Geschlechts (hat die Variable *geschl* einen fehlenden Wert), so wird keine der beiden recode-Anweisungen ausgeführt, und der Variablen *ligg2* wird der systemdefinierte fehlende Wert zugewiesen.

Als letztes Beispiel zu dieser Struktur sei ein Beispiel zur Ereignisdatenanalyse (früher: Überlebensanalyse) betrachtet. Alle innerhalb des Jahres 1999 hinzugekommenen Arbeitslosen des Bezirkes eines Arbeitsamtes wurden bis zum Ende des Jahres 2000 daraufhin beobachtet, ob sie während dieser Zeit wieder eine Arbeitsstelle gefunden hatten.

Die Datei arblos.sav enthält hierzu drei Variablen:

Variable	Bedeutung
gruppe	Altersgruppe 1 = bis 25 Jahre 2 = 26 - 54 Jahre 3 = ab 55 Jahre
datrein	Datum des Beginns der Arbeitslosigkeit im Format mmm jjjj
datraus	Datum des Endes der Arbeitslosigkeit im gleichen Format

Blieb die Arbeitslosigkeit bis zum Ende des Jahres 2000 bestehen, hat die Variable *datraus* einen fehlenden Wert.

In der Ereignisdatenanalyse werden vor allem zwei Parameter relevant: die Dauer bis zum Eintreten des Ereignisses und eine so genannte Statusvariable. Ist das Ereignis im Beobachtungszeitraum nicht eingetreten („zensierte" Fälle), ist dies mit Hilfe der Statusvariablen zu markieren. In diesem Fall wird das Datum des Eintretens des Ereignisses durch das Datum des Endes des Beobachtungszeitraums (hier: 31.12. 2000) ersetzt.

Bei Kenntnis der benutzten Datumsfunktionen (siehe Abschnitt 3.1.2) sollten Sie folgende Anweisungsfolge verstehen (Datei arblos.sps):

```
do if missing(datraus)=0.
compute zeit=rnd((xdate.tday(datraus)-xdate.tday(datrein))/30).
compute status=1.
else.
compute zeit=rnd((yrmoda(2000,12,31)-xdate.tday(datrein))/30).
compute status=0.
end if.
value labels status 1 'Ereignis eingetreten' 0 'zensiert'.
execute.
```

Eine Ereignisdatenanalyse (in SPSS noch Überlebensanalyse genannt) mit einer Aufschlüsselung nach den drei Altersgruppen können Sie dann mit folgendem Befehl berechnen:

```
survival
  table=zeit by gruppe(1 3)
  /interval=thru 24 by 1
  /status=status(1)
  /print=table
  /plots (survival)=zeit by gruppe
  /compare=zeit by gruppe
  /calculate pairwise.
```

Falls es zur Bedingung der do-if-Anweisung mehrere Alternativen gibt, kann die Struktur mit Hilfe der else-if-Anweisung erweitert werden:

```
do if Bedingung.
  Anweisungen
else if Bedingung.
  Anweisungen
else if Bedingung.
  Anweisungen

......

else.
  Anweisungen
end if.
```

Als Beispiel zur Anwendung einer solchen erweiterten Struktur betrachten wir eine epidemiologische Studie über die Zahngesundheit von Kindern. In der Datei kinder.sav ist von insgesamt 6437 zwölfjährigen Kindern die Zugehörigkeit zum Bundesland und die Anzahl der fehlenden und erkrankten Zähne (dmft-Wert) angegeben. Die folgende Tabelle zeigt die nach den Bundesländern aufgeschlüsselten Fallzahlen und die jeweiligen Einwohnerzahlen sowie entsprechende Prozentuierungen.

Code	Land	Fallzahl	dto. %	Einwohnerzahl	dto. %
1	Baden-Württemberg	134	2,1	10,5	24,9
2	Berlin	1163	18,1	3,4	8,1
3	Bremen	56	0,9	0,7	1,7
4	Hessen	1277	19,8	6,1	14,5
5	Mecklenburg-Vorpommern	1013	15,7	1,8	4,3
6	Niedersachsen	272	4,2	7,9	18,7
7	Rheinland-Pfalz	592	9,2	4,0	9,5
8	Sachsen-Anhalt	586	9,1	2,6	6,2
9	Schleswig-Holstein	634	9,8	2,8	6,6
10	Thüringen	710	11,0	2,4	5,7

In Baden-Württemberg wurden also nur 2,1 % aller Fälle untersucht, obwohl 24,9 % der Einwohner aus den teilnehmenden Ländern aus diesem Land stammen. Aus Berlin hingegen stammen 18,1 % der untersuchten Kinder, obwohl dieses Land einwohnermäßig nur zu 8,1 % beteiligt ist. Diese Ungleichgewichtung ist dann störend, wenn Berechnungen wie z. B. des Medians nicht für die einzelnen Bundesländer, sondern für das Gesamtkollektiv vorgenommen werden. Hier bietet es sich an, die einzelnen Fälle nach ihrer Landeszugehörigkeit zu gewichten.

Man bildet dazu eine Gewichtungsvariable (siehe Abschnitt 7.2.1), und zwar als Verhältnis von Soll- zu Istzustand:

$$\text{Gewichtungsfaktor} = \frac{\text{Soll-Prozent}}{\text{Ist-Prozent}}$$

Für das Land Hessen z. B. ergibt sich damit

$$\text{Gewichtungsfaktor} = \frac{14{,}5}{19{,}8} = 0{,}73$$

Die Bildung der Gewichtungsvariable kann mit Hilfe der erweiterten do-if-Struktur erfolgen (Datei kinder.sps):

```
do if land=1.
compute gewicht=24.9/2.1.
else if land=2.
compute gewicht=8.1/18.1.
else if land=3.
compute gewicht=1.7/0.9.
else if land=4.
compute gewicht=14.5/19.8.
else if land=5.
```

```
compute gewicht=4.3/15.7.
else if land=6.
compute gewicht=18.7/4.2.
else if land=7.
compute gewicht=9.5/9.2.
else if land=8.
compute gewicht=6.2/9.1.
else if land=9.
compute gewicht=6.6/9.8.
else.
compute gewicht=5.7/11.
end if.
execute.
```

Die Berechnung z. B. des Medians unter dem Einfluss der Gewichtung kann dann mit folgenden Anweisungen vorgenommen werden:

```
weight by gewicht.
formats dmft (f8.1).
frequencies dmft
  /format=notable
  /grouped=dmft
  /statistics=median.
```

Aufgrund des grouped-Unterbefehls wird der Median für gruppierte Daten berechnet. Die formats-Anweisung fügt dem dmft-Wert eine Nachkommastelle an, was eigentlich sinnlos ist. Es hat aber die Wirkung, dass der Median nicht mit zwei, sondern mit drei Nachkommastellen ausgegeben wird. Als Wert des Medians ergibt sich 0,748. Die Fallzahl sollte im Prinzip bei dieser Art von Gewichtung gleich bleiben, aus Rundungsgründen hat sie sich aber leicht zu 6444 (von ursprünglich 6437) verändert. Wird keine Gewichtung vorgenommen, ist der Median 0,736.

Weitere Beispiele von do-if-Strukturen sind den Beispielen in Abschnitt 3.9 zu entnehmen.

3.7 Eingabeprogramme

Das Einlesen oder Erzeugen von Daten wird bei SPSS durch ein Eingabeprogramm vorgenommen. Im einfachsten Fall ist dies eine einzelne data-list-Anweisung (siehe Abschnitt 2.1.2).

Ein Eingabeprogramm beginnt mit der Anweisung *input program* und endet mit der Anweisung *end input program*:

```
input program.

.....

end input program.
```

Im Fall einer einzelnen data-list-Anweisung ist die Angabe dieses „Rahmens“ entbehrlich, aber auch nicht schädlich:

```
input program.
data list file='c:\syntax\beruf.txt'
  /nr 1-3 gl1 to gl10 5-14 gl11 to gl20 16-25 gl21,gl22 27-28
  geschl 30 alter 32-33 fam 35 abitur 37 bdauer 39-40.
end input program.
execute.
```

Dieses Programmbeispiel ist Abschnitt 2.1.2 entnommen. Sinnvoll wird die explizite Formulierung eines Eingabeprogramms z. B. dann, wenn Sie unter Benutzung der fortlaufenden Fallnummerierung (Variable *nr*) nur die ersten fünfzig Fälle einlesen möchten (Datei eingabe.sps):

```
input program.
data list file='c:\syntax\beruf.txt'
  /nr 1-3 gl1 to gl10 5-14 gl11 to gl20 16-25 gl21,gl22 27-28
  geschl 30 alter 32-33 fam 35 abitur 37 bdauer 39-40.
do if nr > 50.
end file.
end if.
end input program.
execute.
```

Dieses Einleseprogramm enthält innerhalb einer mit *do if* und *end if* gebildeten Struktur die Anweisung *end file*, welche die Erstellung der Datendatei beendet. Diese end-file-Anweisung ist innerhalb eines Eingabeprogramms, das nicht nur aus einer data-list-Anweisung besteht, obligatorisch, um das Ende der Erstellung der Datendatei anzuzeigen.

Beispiele für Eingabeprogramme wurden bereits in Abschnitt 3.1.2 bei der Darstellung der Funktionen für Zufallszahlen und der statistischen Verteilungsfunktionen gezeigt. Es waren dies Beispiele, in denen keine externe Datendatei existierte, sondern die Daten mit dem Eingabeprogramm erzeugt wurden.

Ein einfacher Anwendungsfall liegt vor, wenn Sie einfach mit Hilfe einer Formel etwas berechnen möchten. So haben Sie etwa festgestellt, dass bei einer bestimmten Fischgattung das Gewicht von der Länge nach der Formel

$$Gewicht = 0{,}0027 \cdot Länge^{3{,}436}$$

abhängt, und Sie möchten nun das vorhergesagte Gewicht für einen 21 cm langen Fisch bestimmen.

Sie benötigen zu dieser Berechnung offenbar keine Datendatei und starten die beiden Anweisungen

```
compute gewicht=0.0027*21**3.436.
execute.
```

Anstelle eines Ergebnisses erhalten Sie die Fehlermeldung „This command is not permitted before the beginning of file definition command".

Sie können den Fehler umgehen, wenn Sie vor Ausführung der compute-Anweisung entweder eine Datendatei laden oder aber einfach eine Zahl in ein Feld des Spreadsheets tippen. Eleganter allerdings ist die Formulierung eines Eingabeprogramms:

```
input program.
compute gewicht=0.0027*21**3.436.
end case.
end file.
end input program.
list gewicht.
```

Hinzugekommen ist die Anweisung *end case*. Sie markiert, dass die Variablendefinition für einen Fall abgeschlossen ist. Formal wurde also eine Datenmenge erzeugt, die aus einem Fall und einer Variablen (*gewicht*) besteht.

Meist wird in einem Eingabeprogramm die loop-Anweisung benutzt, mit deren Hilfe Tabellen erzeugt werden können, z. B. die t-Tabelle bei der Vorstellung der statistischen Verteilungsfunktionen. Die loop-Anweisung wird im folgenden Abschnitt erläutert.

3.8 Die loop-Anweisung

Sinnvolle Anwendungen der loop-Anweisung finden sich in Eingabeprogrammen, wenn z. B. eine Tabelle erstellt werden soll (siehe Erstellung einer t-Tabelle in Abschnitt 3.1.2).

Wir greifen das Beispiel am Ende des vorhergehenden Kapitels auf und wollen eine Tabelle erstellen, die zu den Fischlängen die Fischgewichte zuordnet. Dabei sollen Längen von 6 bis 30 cm mit der Schrittweite 2 cm vorgegeben werden (Datei fisch.sps):

```
input program.
loop laenge=6 to 30 by 2.
compute gewicht=0.0027*laenge**3.436.
end case.
end loop.
end file.
end input program.
list laenge,gewicht.
```

Die mit der loop-Anweisung begonnene Schleife wird mit der end-loop-Anweisung beendet. Im Allgemeinen sieht die Struktur wie folgt aus:

```
loop variable=start to ende by schrittweite if Bedingung.
  Anweisungen
end loop if Bedingung.
```

Wird die Schrittweite nicht angegeben, ist eine solche von 1 voreingestellt. Beginnt der Variablenname mit dem Zeichen #, handelt es sich um eine so genannte Arbeitsvariable, die nicht in die Datendatei übernommen wird (siehe Programmbeispiel zu den Zufallsfunktionen in Abschnitt 3.1.2). Die Bedingung in der loop- bzw. end-loop-Anweisung ist wahlfrei. Jeweils ein Bespiel für den Einsatz einer solchen Bedingung wird am Ende des Kapitels gegeben.

Die end-case-Anweisung im gegebenen Beispiel signalisiert, dass bei einem loop-Schritt jeweils ein neuer Fall (case) kreiert wird.

Die folgende Tabelle wird ausgegeben:

```
   LAENGE  GEWICHT

     6.00     1.27
     8.00     3.42
    10.00     7.37
    12.00    13.79
    14.00    23.41
    16.00    37.04
    18.00    55.52
    20.00    79.74
    22.00   110.64
    24.00   149.20
    26.00   196.43
    28.00   253.40
    30.00   321.18
```

Eine loop-Schleife kann durch Formulierung einer entsprechenden Bedingung vorzeitig verlassen werden. Dazu kann die Breakanweisung benutzt werden, die dann in eine mit *do if* und *end if* gebildete Struktur einzuschließen ist. Im gegebenen Programmbeispiel soll die Schleife vorzeitig verlassen werden, wenn das errechnete Gewicht größer als 200 wird.

```
input program.
loop laenge=6 to 30 by 2.
compute gewicht=0.0027*laenge**3.436.
do if gewicht>200.
break.
```

```
end if.
end case.
end loop.
end file.
end input program.
list laenge,gewicht.
```

Die ausgegebene Tabelle ist nun etwas kürzer:

```
  LAENGE  GEWICHT

    6.00     1.27
    8.00     3.42
   10.00     7.37
   12.00    13.79
   14.00    23.41
   16.00    37.04
   18.00    55.52
   20.00    79.74
   22.00   110.64
   24.00   149.20
   26.00   196.43
```

Ein treffendes Programmbeispiel für eine Loop-Programmierung mit vorzeitigem Abbruch ist die Rückzahlung eines Kredits. Wir wollen annehmen, ein Betrag von 5 000 Euro (Variable *kap*) soll bei einem jährlichen Zinssatz von 5,5 % (Variable *p*) mit einer monatlichen Rate von 400 Euro (Variable *rate*) zurückgezahlt werden.

Der monatliche Zinsbetrag ist dann

$$zins = \frac{kap \cdot p}{12 \cdot 100}$$

und die monatliche Tilgung

$$Tilgung = Rate - Zins$$

Um diesen Betrag vermindert sich dann jeweils der zurückzuzahlende Betrag. Versuchen Sie, das folgende Programm nachzuvollziehen (Datei kredit.sps):

```
comment Abzahlung eines Kredits.
input program.
compute kap=5000.
compute p=5.5.
compute rate=400.
end case.
```

```
loop monat=1 to 100.
compute kap=lag(kap).
compute p=lag(p).
compute rate=lag(rate).
compute zins=kap*p/1200.
compute tilg=rate-zins.
do if kap > tilg.
compute kap=kap-tilg.
else.
compute tilg=kap.
compute kap=0.
end if.
end case.
do if lag(kap) = 0.
break.
end if.
end loop.
end file.
end input program.
list monat,zins,tilg,kap.
```

Mit den ersten drei compute-Anweisungen werden die Variablen *kap*, *p* und *rate* vorbesetzt. Hier können im Bedarfsfall andere Werte eingesetzt werden. Die end-case-Anweisung markiert, dass der erste Fall damit abgeschlossen ist. Bei den folgenden Anweisungen der loop-Schleife ist zu beachten, dass zunächst auf die Variablenwerte des vorhergehenden Falles zurückgegriffen werden muss und daher die lag-Funktion einzusetzen ist.

Ferner ist zu beachten, dass der Rückzahlungsvorgang abzubrechen ist (die loop-Schleife zu verlassen ist), wenn der Tilgungsbetrag größer oder gleich dem noch zurückzuzahlenden Betrag wird.

Wenig elegant wirkt die loop-Anweisung bei der Angabe des Schleifenendwertes. Da der richtige Wert in diesem Beispiel nicht bekannt ist, ist ein genügend großer Wert einzusetzen, der dann aber wegen des greifenden Abbruchkriteriums nicht erreicht wird.

Der folgende Zahlungsverlauf wird ausgegeben:

```
   MONAT     ZINS     TILG      KAP

     .        .        .    5000.00
    1.00    22.92   377.08  4622.92
    2.00    21.19   378.81  4244.11
    3.00    19.45   380.55  3863.56
    4.00    17.71   382.29  3481.27
    5.00    15.96   384.04  3097.22
    6.00    14.20   385.80  2711.42
```

```
    7.00    12.43   387.57  2323.84
    8.00    10.65   389.35  1934.49
    9.00     8.87   391.13  1543.36
   10.00     7.07   392.93  1150.43
   11.00     5.27   394.73   755.71
   12.00     3.46   396.54   359.17
   13.00     1.65   359.17      .00
```

Die Abbruchbedingung kann im gegebenen Beispiel auch in die end-loop-Anweisung übernommen werden, was das Konstrukt mit der break-Anweisung überflüssig macht:

```
......

end case.
end loop if lag(kap)=0.
end file.
end input program.
list monat,zins,tilg,kap.
```

Nach dieser Formulierung der end-loop-Anweisung wird die Schleife verlassen, falls die Bedingung „$lag(kap) = 0$“ erfüllt ist.

Alternativ hierzu kann das Abbruchkriterium, allerdings sozusagen positiv formuliert, in die loop-Anweisung eingebaut werden:

```
.....

end case.
loop monat=1 to 100 if lag(kap)>0.
compute kap=lag(kap).

.....
```

Bei dieser Formulierung der loop-Anweisung wird die Schleife so lange durchlaufen, wie die Bedingung „$lag(kap) > 0$“ erfüllt ist.

Weitere Anwendungen von loop-Anweisungen finden Sie in den Beispielen des folgenden Kapitels.

3.9 Die vector-Anweisung

Die vector-Anweisung definiert einen Vektor von numerischen Variablen, was bei bestimmten Problemen nützlich sein kann. Prinzipiell gibt es zwei Varianten der vector-Anweisung:

1. Es wird ein Vektor mit neuen Variablen definiert.

2. Es wird ein Vektor aus bestehenden Variablen definiert.

Die erstgenannte Variante ist insbesondere in den beiden folgenden Situationen nützlich:

- Eine nominalskalierte Variable mit mehr als zwei Kategorien soll in entsprechend viele dichotome Variablen zerlegt werden. Das ist z. B. notwendig, wenn diese Variable in einer multiplen Regressionsanalyse als Einflussvariable angegeben werden soll (siehe Abschnitt 10.8.3).
- Bei einer Frage mit Mehrfachantworten soll die kategoriale Codierung durch die dichotome Codierung ersetzt werden (siehe Abschnitt 10.1.2).

Laden Sie die Datei umfrage.sav, die einige Ergebnisse einer Meinungsumfrage enthält. Die Variable *beruf* ist eine nominalskalierte Variable mit fünf Kategorien: 1 = Beamter, 2 = Angestellter, 3 = Arbeiter, 4 = Selbständiger, 5 = in Ausbildung. Diese Variable soll in fünf dichotome Variablen mit der Codierung 1 = ja, 2 = nein „zerlegt" werden (Datei vec1.sps):

```
vector b(5).
loop #i=1 to 5.
compute b(#i)=2.
end loop.
if nmiss(beruf)=0 b(beruf)=1.
execute.
variable labels b1 'Beamter?'/
  b2 'Angestellter?'/
  b3 'Arbeiter?'/
  b4 'Selbstaendiger?'/
  b5 'in Ausbildung?'.
value labels b1 to b5 1 'ja' 2 'nein'.
execute.
```

Die vector-Anweisung definiert einen Vektor mit fünf Variablen, die bis zur nächsten execute-Anweisung in indizierter Form anzusprechen sind: $b(1)$, $b(2)$, $b(3)$, $b(4)$, $b(5)$. Anstelle dieser Zahlen kann als Index auch ein Variablenname angegeben werden (hier: *#i*, *beruf*).

Nach Ausführung der execute-Anweisung stehen dann die Variablen *b*1, *b*2, *b*3, *b*4, *b*5 zur Verfügung, die mit entsprechenden Labels versehen werden können.

Im folgenden Beispiel werden die sechs Variablen *vn*1 bis *vn*6 der Datei kenia.sav, die nach der kategorialen Methode (Zahlen 1 bis 9) codiert sind (siehe mult-response-Anweisung, Abschnitt 10.1.2), in die neun Variablen *vornach*1 bis *vornach*9 umgewandelt (Datei vec2.sps):

```
vector vornach (9).
loop #I=1 to 9.
```

```
compute vornach(#i)=2.
end loop.
do repeat v=vn1 to vn6.
if nmiss(v)=0 vornach(v)=1.
end repeat.
execute.
```

Diesmal wird ein Vektor mit neun Variablen aufgebaut. Diese haben nach Ausführung der execute-Anweisung die Namen *vornach*1, *vornach*2, …, *vornach*9 und sind wieder mit 1 = ja, 2 = nein codiert.

Ein Vektor kann auch aus bestehenden Variablen definiert werden. Laden Sie die Datei haushalt.sav (siehe Kap. 9). Bei den fünf Variablen *kochen* bis *putzen* treten fehlende Werte auf; diese sollen durch die Codierung 3 ersetzt werden:

```
vector arbeit=kochen to putzen.
loop #i=1 to 5.
if nmiss(arbeit(#i))=1 arbeit(#i)=3.
end loop.
execute.
```

Diesmal wird ein Vektor aus bestehenden Variablen aufgebaut, damit diese dann mit einer loop-Anweisung behandelt werden können. Das lässt sich allerdings kürzer mit einer do-repeat-Anweisung formulieren:

```
do repeat arbeit=kochen to putzen.
if nmiss(arbeit)=1 arbeit=3.
end repeat.
execute.
```

Die Variante, einen Vektor aus bestehenden Variablen aufzubauen, spielt praktisch keine Rolle.

3.10 Systemvariablen

Systemvariablen stehen unmittelbar zur Verfügung. Ihr Name beginnt mit einem $. Die drei nützlichsten Systemvariablen sind:

$casenum	fortlaufende, bei 1 beginnende Fallnummerierung
$jdate	aktuelles Datum in Anzahl von Tagen seit dem 14. Oktober 1582
$time	aktuelle Zeit in Anzahl von Sekunden seit dem 14. Oktober 1582, 0 Uhr

Die Systemvariable *$jdate* kann Bedeutung erlangen in Zusammenhang mit den Datumsfunktionen *yrmoda* und *xdate.tday*, die Systemvariable *$time* in Zusammenhang mit den Zeitfunktionen *time.hms* und *xdate.time* (siehe Abschnitt 3.1.2).

3.11 Beispiele von SPSS-Programmen

Die Steuersprache von SPSS hat sich von anfänglichen einfachen Programmanweisungen inzwischen zu einer recht mächtigen „Programmiersprache" entwickelt, mit der nicht nur statistische Auswertungen einer gegebenen Datenmenge vorgenommen, sondern in gewissem Umfang auch andere allgemeine Probleme gelöst werden können. Dies soll an zwei Beispielen demonstriert werden.

Berechnung der tariflichen Einkommensteuer

Die tarifliche Einkommensteuer für das Jahr 2002 berechnet sich nach folgenden Rechenschritten:

- Das zu versteuernde Einkommen ist auf den nächsten durch 36 ohne Rest teilbaren vollen Euro-Betrag abzurunden, falls es nicht bereits durch 36 ohne Rest teilbar ist, und dann um 18 Euro zu erhöhen. Das Ergebnis wird x genannt.
- y ist ein Zehntausendstel des 7200 Euro übersteigenden Teils von x.
- z ist ein Zehntausendstel des 9216 Euro übersteigenden Teils von x.

Dann werden, in Abhängigkeit vom zu versteuernden Einkommen, die folgenden Formeln angewandt.

Einkommen	Einkommensteuer
bis 7235 Euro	0
7236 Euro bis 9251 Euro	$(768{,}85 \cdot y + 1990) \cdot y$
9252 Euro bis 55007 Euro	$(278{,}65 \cdot z + 2300) \cdot z + 432$
ab 55008 Euro	$0{,}485 \cdot x - 9872$

Die Berechnung der tariflichen Einkommensteuer in Abhängigkeit vom zu versteuernden Einkommen erfolgt mit dem folgenden Programm (Datei steuer.sps).

```
comment Berechnung der tariflichen Einkommensteuer.
input program.
loop euro=10000 to 60000 by 2500.
compute x=euro-mod(euro,36)+18.
compute y=(x-7200)/10000.
compute z=(x-9216)/10000.
do if euro <= 7235.
compute steuer=0.
else if euro <= 9251.
compute steuer=(768.85*y+1990)*y.
else if euro <=55007.
compute steuer=(278.65*z+2300)*z+432.
```

```
else.
compute steuer=0.485*x-9872.
end if.
compute steuer=trunc(steuer).
compute dsatz=steuer/euro*100.
compute ssatz=(steuer-lag(steuer))/(euro-lag(euro))*100.
end case.
end loop.
end file.
end input program.
list euro,steuer,dsatz,ssatz.
```

Nach dem aufmerksamen Studium dieses Kapitels sollten Sie in der Lage sein, dieses Programm zu verstehen. Die Variable *dsatz* ist der durchschnittliche Steuersatz, die Variable *ssatz* der Spitzensteuersatz im jeweils letzten Schritt. Die loop-Anweisung können Sie gegebenenfalls in einfacher Weise nach Ihren Bedürfnissen abändern.

Die Ausgabe des Programms ist im Folgenden wiedergegeben.

```
    EURO   STEUER    DSATZ    SSATZ

10000.00   611.00     6.11      .
12500.00  1219.00     9.75    24.32
15000.00  1853.00    12.35    25.36
17500.00  2532.00    14.47    27.16
20000.00  3235.00    16.18    28.12
22500.00  3984.00    17.71    29.96
25000.00  4757.00    19.03    30.92
27500.00  5564.00    20.23    32.28
30000.00  6418.00    21.39    34.16
32500.00  7294.00    22.44    35.04
35000.00  8218.00    23.48    36.96
37500.00  9164.00    24.44    37.84
40000.00 10158.00    25.40    39.76
42500.00 11173.00    26.29    40.60
45000.00 12238.00    27.20    42.60
47500.00 13322.00    28.05    43.36
50000.00 14440.00    28.88    44.72
52500.00 15610.00    29.73    46.80
55000.00 16798.00    30.54    47.52
57500.00 18020.00    31.34    48.88
60000.00 19225.00    32.04    48.20
```

Nach diesen doch recht unerfreulichen Ergebnissen soll im zweiten Beispiel ein sympathischeres Problem gelöst werden: die Berechnung des Datums des Ostersonntags zu einer gegebenen Jahreszahl.

Berechnung des Datums des Ostersonntags

Ostern wird nach einem Beschluss des Konzils von Nicäa im Jahr 325 am ersten Sonntag nach dem ersten Vollmond im Frühjahr gefeiert. Dazu hat der berühmte deutsche Mathematiker Carl Friedrich Gauß (1777–1855), Direktor der Sternwarte in Göttingen, folgenden Algorithmus zur Berechnung des Datums des Ostersonntags bei gegebener Jahreszahl (im folgenden Programm Variable *jahr*) entwickelt:

k = ganzzahliger Teil von $jahr/100$

p = ganzzahliger Teil von $k/3$

q = ganzzahliger Teil von $k/4$

$m = 15 + k - p - q$

$m1$ = Divisionsrest von $m/30$

$n = 4 + k - q$

$n1$ = Divisionsrest von $n/7$

a = Divisionsrest von $jahr/19$

b = Divisionsrest von $jahr/4$

c = Divisionsrest von $jahr/7$

$d = 19 + a + m1$

$d1$ = Divisionsrest von $d/30$

$e = 2 \cdot b + 4 \cdot c + 6 \cdot d1 + n1$

$e1$ = Divisionsrest von $e/7$

$x = 22 + d1 + e1$

Bei der Bestimmung von x gibt es zwei Ausnahmen.

- Falls $x = 57$ ist, wird $x = 50$ gesetzt.
- Falls $d1 = 28$ ist und $e1 = 6$ und der Divisionsrest von $(11 \cdot m + 11)/30$ kleiner als 19 ist, wird $x = 49$ gesetzt.

Der Ostersonntag fällt dann auf den x. März oder, falls $x > 31$ ist, auf den $(x - 31)$. April.

Das Osterdatum für die Jahre 2000 bis 2010 wird danach mit dem folgenden Programm (Datei ostern.sps) berechnet. Andere Jahreszahlen sind entsprechend in der loop-Anweisung anzugeben.

```
comment Berechnung des Datums des Ostersonntags.
input program.
loop jahr=2000 to 2010.
compute k=trunc(jahr/100).
```

```
compute p=trunc(k/3).
compute q=trunc(k/4).
compute m=15+k-p-q.
compute m1=mod(m,30).
compute n=4+k-q.
compute n1=mod(n,7).
compute a=mod(jahr,19).
compute b=mod(jahr,4).
compute c=mod(jahr,7).
compute d=19*a+m1.
compute d1=mod(d,30).
compute e=2*b+4*c+6*d1+n1.
compute e1=mod(e,7).
compute x=22+d1+e1.
if x=57 x=50.
if d1=28 AND e1=6 AND mod(11*m+11,30)<19 x=49.
do if x <= 31.
compute tag=x.
compute monat=3.
else.
compute tag=x-31.
compute monat=4.
end if.
compute odatum=date.mdy (monat,tag,jahr).
end case.
end loop.
formats odatum(date11).
end file.
end input program.
list odatum.
```

Die folgenden Datumsangaben werden ausgegeben:

```
      DATUM

23-APR-2000
15-APR-2001
31-MAR-2002
20-APR-2003
11-APR-2004
27-MAR-2005
16-APR-2006
08-APR-2007
23-MAR-2008
12-APR-2009
04-APR-2010
```

Mit etwas Phantasie können mit der SPSS-Syntax also auch Probleme gelöst werden, die nichts mit statistischen Auswertungen zu tun haben.