Programmieren Sie strukturiert! (Teil 3)

Wer strukturiert programmieren will, braucht als Werkzeuge vernünftige Befehlsstrukturen. Bisher haben Sie Programmbausteine kennengelernt, die den Programmablauf steuern. Nun geht es um Prozeduren mit mehr als einem Ein-/Ausgang: um mehrfache Kommunikation.

Die bisherigen Beispiele für Prozeduren enthielten jeweils keine oder nur eine einzige Variable, das heißt: entweder gar keinen Durchgang oder einen Eingang oder einen Ausgang oder einen Ein-/Ausgang für Information. Die Anzahl der Durchlässe ist jedoch prinzipiell nicht beschränkt. Im Prinzip können so viele Variablen benutzt werden, wie der Programmierer wünscht. Und jede Variable kann einem anderen Typ angehören: IN oder OUT oder IN/OUT. Auf diese Weise entstehen neben den grundlegenden Prozedurtypen alle möglichen Mischtypen. Ein Beispiel für einen solchen Mischtyp sehen Sie in Bild 1.

Prozeduren mit mehrfacher Kommunikation

Die Prozedur INPUT'MIT'VORGABE soll zur Eingabe auffordern können (zum Beispiel »Wollen Sie weitermachen?«), soll eine Antwortmöglichkeit vorgeben können (die, die am ehesten zu erwarten ist, zum Beispiel »Ja«) und so lange um Antwort bitten, bis tatsächlich eine Eingabe gemacht worden ist.

Auf dem Bildschirm soll also folgendes erscheinen:
»Wollen Sie weitermachen (Ja/Nein)? Ja«

Der Cursor blinkt auf dem J. Wenn Sie Ja eingeben wollen, brauchen Sie jetzt nur auf die RETURN-Taste zu drücken.

Gemischter Prozedurtyp:

```
INPUT'MIT'VORGABE
Anfang Block
    Länge Aufforderung+Vorgabe bestimmen
    Länge überprüfen, eventuell Vorgabe streichen
    Solange bis Eingabe ok
      Vorgabe drucken
      Länge überprüfen auf benötigte Anzahl Zeilen
      Benötigte Anzahl Cursor hoch drucken
      Aufforderung zur Eingabe drucken
      Auf Eingabe warten
      Eingabe überprüfen
      Falls nicht ok zurück zum Schleifenanfang
    Ende Schleife
    Ende Block
```

Bild 1. »Input« mit Eingabeaufforderung und Vorgabe

Die Prozedur braucht zwei Informationen von außen: Mit welchen Worten sie zur Eingabe auffordern und was sie vorgeben soll. Und sie gibt eine Information an die Außenwelt weiter: Nämlich das, was als Antwort eingegeben worden ist. Der Prozedurkopf wird deshalb drei Variablen enthalten, zwei für hereinkommende und eine für hinausgehende Daten. Eine Comal-Prozedur finden Sie in Listing 1.

Der Befehlsaufruf kann zum Beispiel lauten:

```
input'mit'vorgabe("Was nun? "," ",eingabe$)
```

Ein paar Hinweise zur Comal-Prozedur »input'mit'vorgabe«:

ZONE bestimmt die Anzahl der Zeichen zwischen zwei Druckspalten; in der Prozedur wird ZONE vorsichtshalber auf 0 gesetzt, damit die Vorgabe in Zeile 9070 an die richtige Stelle kommt, nämlich direkt hinter den Aufforderungstext. Am Ende erhält ZONE wieder den Wert, den es vor dem Aufruf der Prozedur hatte.

```
9000 PROC input'mit'vorgabe(aufforderung$,vorgabe$,REF
     eingabe$) CLOSED
9010  DIM up$ OF 2
9020  z:=ZONE
9030  ZONE 0
9040  ok:=FALSE
9050  laenge:=LEN(aufforderung$)+LEN(vorgabe$)
9060  REPEAT
0070   PRINT TAB(LEN(aufforderung$)+1),vorgabe$
9080   IF laenge < 40 THEN
9090    up$:=CHR$(145)
9100   ELSE
9110    up$:=CHR$(145)+CHR$(145)
9120   ENDIF
9130   PRINT up$
9140   INPUT aufforderung$: eingabe$
9150   IF eingabe$ <  "  " AND eingabe$ >  "  "
       THEN
9160    ok:=TRUE
9170   ELSE
9180    PRINT up$
9190   ENDIF
9200  UNTIL ok
9210  ZONE z
9220 ENDPROC input'mit'vorgabe
```

Listing 1. Comal-Prozedur zur Eingabe nach Aufforderung

CHR$(145) entspricht dem Steuerzeichen für »Cursor nach oben«.

In Comal kann man die Eingabeaufforderung als Variable dem INPUT-Befehl mitgeben, was in Basic ja nicht möglich ist (Zeile 9140).

Für die Eingabeschleife brauchten wir eigentlich eine LOOP-Schleife. Die steht aber in Comal 0.14 nicht zur Verfügung, weshalb wir sie (mit Hilfe einer REPEAT-UNTIL-Schleife) imitieren müssen.

Listing 2 zeigt eine entsprechende Prozedur in Basic.

```
40000 rem proc: input mit vorgabe (uauff$: in, uvo
      rg$: in, ueingabe$: out)
40010  ul=len(uauff$)+len(uvo rg$)
40020  if ul > 76 then uvo rg$="  "
40030  rem loop
40040    print tab(4+len(uauff$));uvo rg$
40050    up$=chr$(145):if ul > 36 then up$=up$+up$
40060    print up$;uauff$;
40070    poke 198,3:poke631,34:poke632,34:poke633,20
40080    input ueingabe$
40090    if ueingabe$ >  "  " then 40120
40100    printup$;
40110    goto 40040
40120  rem endloop
40130  uauff$="  " :uvo rg$="  "
40140 return
```

Listing 2. Basic-Programm zur Eingabe nach Aufforderung

Zeile 40070 bewirkt, daß hinter dem INPUT-Fragezeichen ein Anführungszeichen gedruckt wird, so daß auch Texte mit Komma und Doppelpunkt akzeptiert werden.

In Zeile 40130 werden die Aufforderungs- und die Vorgabevariablen geleert. Das macht es möglich, beim Aufruf der Prozedur eine oder beide Variablen wegzulassen, wenn sie nicht gebraucht werden. Beispiele:

```
uauff$="Wollen Sie weitermachen"uvo rg$="JA"
       gosub 40000in$=ueingabe$
uauff$="Was möchten Sie jetzt"gosub 40000: in$=ueingabe$
gosub 40000: in$=ueingabe$
```

Anmerkung: In Comal ist die Anzahl der möglichen Variablen, die im Prozedurkopf erscheinen können, durch die Zeilenlänge begrenzt (je kürzer die Variablennamen sind, um so größer ist ihre mögliche Anzahl). In Basic gilt diese Beschränkung nicht; der Befehlsaufruf ist hier ja eine Sequenz, die nicht unbedingt einzeilig zu sein braucht, sondern auch als Zeilenblock codiert werden kann (siehe Teil I), und ein solcher Zeilenblock kann so viele Zeilen enthalten wie nötig.

```
100 rem sequenz
110   uauff$="Wollen Sie weitermachen?":uvo rg$="Ja"
120   gosub 40000: rem input'mit'vorgabe
130   in$=ueingabe$
140 rem ende sequenz
```

Zusammenfassung: Prozedurtypen

Wir haben, Comal imitierend, für Basic-Prozeduren folgende Befehlsstruktur vorgeschlagen (Bild 2):

```
REM PROC Prozedurname (Variablen)
...
...
...
RETURN
```

Bild 2. Comal-artige Befehlstrukturen für Basic-»Prozeduren«

Bei Variablen geben wir die Richtung, in die die Information geht, mit IN, OUT oder IN/OUT an. Sie beginnen immer mit U (für »Unterprogramm«). Außerhalb von Prozeduren werden Variablen mit U nicht verwendet.

Aufgerufen werden Prozeduren mit
GOSUB Zeilennummer: REM Prozedurname

Wenn Informationen mitgegeben werden sollen, so werden sie den Prozedurvariablen unmittelbar vor dem GOSUB-Aufruf zugewiesen; wenn Informationen abgeholt werden sollen, werden diese den im Programm gültigen Variablen unmittelbar danach zugeordnet. Der Aufruf nimmt möglichst nur eine Zeile in Anspruch. Wo nötig, kann jedoch auch eine Sequenz aus mehreren Zeilen benutzt werden.

Vier grundlegende Prozedurtypen haben wir besprochen (Bild 3).

Prozedurtypen

Kommunikationstyp	Definition	Aufruf
1 keine Kommunikation		GOSUB ZEILE
2 Einwegkommunikation 1	UV: IN	UV=V:GOSUB ZEILE
3 Einwegkommunikation 2	UV: OUT	GOSUB ZEILE:V=UV
4 Zweiwegkommunikation	UV: IN/OUT	UV=V:GOSUB ZEILE:V=U

Abkürzungen:
V = Variable (die außerhalb der Prozedur gilt),
UV = Unterprogrammvariable

Bild 3. Vier grundlegende Prozedurtypen — so ruft man sie auf

Funktionen in Basic

Funktionen in Comal gehorchen denselben Regeln wie Prozeduren.

Beispiel: Die Funktion FRACTION soll den Bruchteil einer (positiven oder negativen) Zahl berechnen und »enthalten«.

```
9000 FUNC fraction(zahl) CLOSED
9010   bruchteil:=zahl-sgn(zahl)*int(abs(zahl))
9020   RETURN bruchteil
9030 ENDFUNC fraction
```

Funktionen unterscheiden sich in Comal von Prozeduren nur dadurch, daß sie den Befehl RETURN enthalten. Nach RETURN folgt der Wert, den die Funktion zur Verfügung stellen soll (als Zahl, als Variable, als mathematischer Ausdruck, sogar als Funktion). Die Funktion FRACTION kann so aufgerufen werden:

```
PRINT fraction (3.57)
```

Dieses Beispiel macht noch einmal deutlich, daß Funktionsbefehle ähnlich benutzt werden wie Variablen.

So wie Prozeduren, haben auch Funktionen in Comal einen deutlich erkennbaren Rahmen. Für Basic-Funktionen müssen wir wieder einen verabreden. Der Funktionskopf soll so aussehen:

```
REM FUNC Funktionsname (Variable)
```

Danach folgen eine oder mehrere Definitionen. Das Ende markieren wir mit

```
REM ENDFUNC
```

Damit sieht die Funktion FRACTION in unserem selbststrukturierten Basic so aus:

```
100 rem func: fraction (zahl: in)
110   def fn frac (zahl)=zahl-sgn(zahl)*int(abs(zahl))
120 rem endfunc
```

Aufgerufen wird die Funktion zum Beispiel so:

```
print fn frac(3.57)
```

Bitte beachten Sie die erste Klammer in der Definitionszeile 110. Sie erlaubt es, wie in Comal, eine Information in das Funktionsinnere mitzugeben. Wie in Comal ist auch diese Variable lokal. Das bedeutet: Wir können beim Aufruf der Funktion irgendeine andere Variable in die Klammer schreiben (oder eine Zahl, wie oben, oder einen mathematischen Ausdruck, oder einen anderen Funktionsaufruf).

Wir haben übrigens hier, und wir wollen das auch weiterhin tun, den Funktionsnamen in der eigentlichen Definitionszeile verkürzt. Das scheint legitim, da ein ausführlicher Name in der Kopfzeile erscheint (FRACTION) und so der Kurzname FRAC ohne Schwierigkeiten verstanden werden kann.

Während eine Comal-Funktion beliebig viele Zeilen umfassen kann, sind Funktionsdefinitionen in Basic auf eine einzige Zeile beschränkt. Das ist oft zu wenig. Manchmal allerdings gibt es einen Ausweg. Zwar können wir keine mehrzeiligen Definitionen erfinden, aber wir können mehrere Funktionen hintereinander definieren, wobei die vorhergehende Funktion in eine spätere eingebettet wird.

Das folgende Programm wandelt Kleinbuchstaben in Großbuchstaben um.

```
100 rem func ucase (zeichencode: in)
110   def fn buch (zc) = abs ((zc > 64 and zc < 93)
      or (zc > 192 and zc < 222))
120   def fn klein (zc) = abs ((zc < 128))
130   def fn ucase (zc) = zc + fn klein (zc) * fn buch
      (zc) * 128
140 rem endfunc
```

Die Funktion UCASE (upper case) geht in drei Schritten vor.

Zunächst wird die eingebettete Funktion BUCH bearbeitet; sie überprüft, ob der Zeichencode ZC einem Buchstaben entspricht. Dann prüft die Funktion KLEIN, ob es sich um einen Kleinbuchstaben handeln könnte. Wenn beide Funktionen grünes Licht geben, dann wird der Zeichencode um 128 erhöht und damit zum ASCII-Wert für einen Großbuchstaben.

Beispiel für einen Aufruf der Funktion UCASE:

```
print chr$(fn ucase(asc("a")))
```

Sehen Sie jetzt, daß es ganz sinnvoll ist, daß wir einen Rahmen für Funktionsdefinitionen erfunden haben (ich weiß, daß Sie zunächst daran zweifelten, ob sich der Aufwand überhaupt lohne)?

Lassen Sie uns nun kurz einige Funktionsbeispiele Revue passieren, um zu illustrieren, wie wir diesen Unterprogrammtyp in strukturierten Programmen einsetzen können. Wir wollen die Beispiele dabei wieder nach dem Gesichtspunkt ordnen, wie die Funktionen mit der Außenwelt kommunizieren. Diese Kommunikation ist allerdings in Basic eingeschränkt. Sie kann nur in einer Richtung stattfinden: von außen nach innen.

Funktionen ohne Kommunikation

Beispiel: Cursorspalte

```
9000 FUNC cursorspalte CLOSED
9010   RETRUN PEEK(211)+1
9020 ENDFUNC cursorspalte
```

Aufruf der Funktion CURSORSPALTE in Comal zum Beispiel:

```
IF cursorspalte > 39 THEN PRINT
```

Dieselbe Funktion in Basic:

```
100 rem func: cursorspalte
110   def fn cs(x)=peek(211)+1
120 rem endfunc
```

Aufruf der Funktion CURSORSPALTE in Basic:

```
if fn cs(0) > 39 then print
```

In Comal zeigt eine Funktion, die nicht mit der Außenwelt kommuniziert, dies deutlich dadurch, daß sie keine Klammer benutzt. Dagegen braucht eine Basic-Funktion stets ihre Klammer sowie Parameter in den Klammern. Wir benutzen für Funktionen dieses Typs das nichtssagende X in der Definition und den Wert 0 beim Aufruf.

Funktionen mit Einwegkommunikation

Beispiel: Umrechnung in Standard-ASCII

Die Funktion TRUE-ASCII soll den speziellen Commodore-ASCII-Code in Standard-ASCII umrechnen, wie ihn die meisten anderen Computer benutzen.

```
9000 FUNC true'ascii(c64'zeichen$) CLOSED
9010  c:=ORD(c64'zeichen$)
9020  IF c > = 65 AND c < = 90 THEN
9030   RETURN c+32
9040  ELIF c > = 193 AND c < = 219 THEN
9050   RETURN c-128
9060  ELSE
9070   RETURN c
9080  ENDIF
9090 ENDFUNC true'ascii
```

Diese Funktion (sie stammt aus der »Comal Library of Functions and Procedures«) kann in Basic nicht in derselben einfachen und verständlichen Weise codiert werden — Basic erlaubt nur mathematische Ausdrücke in Funktionsdefinitionen.

```
100 rem func: true ascii (c64-Zeichencode: in)
110   def fn true(z) = z + abs (z > = 65 and z < = 90)
      * 32
      - abs(z > = 193 and z < = 219) * 128
120 rem endfunc
```

Statt der Mehrfachverzweigung wie in Comal benutzen wir in der Basic-Definition zwei Boolesche Ausdrücke. Wenn der erste dieser beiden Ausdrücke wahr ist, dann hat er, im Verein mit ABS, den Wert 1. Mit 32 multipliziert, erhält man den Wert 32, der zum Commodore-ASCII-Wert addiert wird. Der zweite Ausdruck ergibt in diesem Fall 0. Das Umgekehrte gilt für den Fall, daß es sich um einen kleinen Buchstaben handelt und deshalb der zweite Ausdruck wahr ist.

Diese Funktion illustriert eine weitere Beschränkung in bezug auf Basic-Funktionen. Während Comal auch eine Textvariable als Eingang akzeptiert, läßt Basic nur Zahlenvariablen zu. Dies führt zu unterschiedlichen Befehlsaufrufen unserer Funktion TRUE'ASCII in den beiden Sprachen:

```
PRINT true'ascii("A")
(Comal)
print fn true (asc("A))
(Basic)
```

Funktionen mit mehrfacher Kommunikation

Comal-Funktionen akzeptieren so viele Variablen wie nötig sind (und Platz in der Programmzeile enthalten ist).

Ein Beispiel mit zwei Variablen:

Der größere Wert. Die Funktion MAX soll aus zwei Zahlen die größere ermitteln und ausgeben.

```
9000 FUNC max(zahl1,zahl2) CLOSED
9010  IF zahl1 > zahl2 THEN
9020   RETURN zahl1
9030  ELSE
9040   RETURN zahl2
9050  ENDIF
9060 ENDFUNC max
```

Hier zeigt sich erneut eine Beschränkung in den Möglichkeiten von Basic-Funktionen. In Basic kann nur eine einzige Variable mitgegeben werden. Man könnte nun so verfahren, daß man die eine Variable in Klammern mitgibt und die zweite erst beim Aufruf mitteilt zum Beispiel Z2=25: PRINT MAX (50). Als Ergebnis würde die Zahl 50 ausgedruckt, denn das ist die größere Zahl. Dies Verfahren wirkt jedoch erfahrungsgemäß verwirrend, und unser Ziel ist es ja nun gerade, mögliche Verwirrungen von vornherein zu vermeiden. Es scheint deshalb sinnvoll, wenn wir schon mehrere Variablen der Funktion mitgeben müssen, alle diese Variablen gleich zu behandeln. Das heißt aber, daß wir sie alle schon beim Aufruf der Funktion mitgeben. Der Funktion selbst werden keine Variablen mitgegeben. Wir müssen also denselben Funktionstyp benutzen wie bei Funktionen ohne Kommunikation.

Die Basic-Codierung der Funktion MAX lautet demnach so.

```
100 rem func: max(u1: in, u2: in)
110   def fn max(x) = abs(u1 > u2)*u1+abs(u1 < = u2)*u2
120 rem endfunc
```

Vor dem Aufruf weisen wir, ähnlich wie bei den Basic-Prozeduren, den Variablen U1 und U2 »von Hand« ihre Werte zu:

```
u1=238: u2=21: print fn max(0)
```

Wieder einmal sind wir also in der Situation, wo wir selber dafür sorgen müssen, daß Variablen nur lokal gelten. Damit dies gewährleistet ist, benutzen wir, wie schon bei den Prozeduren, Variablennamen mit U (für »Unterprogramm«).

Und wenn's mit Funktionen nicht geht?

Basic-Funktionen zeigen mancherlei Beschränkungen. Nicht jede Beschränkung kann aufgefangen werden. Was ist zu tun? Die Lösung ist einfach: Wenn eine Funktion nicht programmierbar ist, machen wir eine Prozedur. Wir erinnern uns: Funktionen sind nichts anderes als spezielle Prozeduren. Das ist zwar in Basic nicht ohne weiteres erkennbar, aber in Comal haben wir es dafür um so deutlicher sehen können.

Das Problem taucht nicht nur in Basic auf. Schauen wir uns kurz ein Beispiel an: **String umdrehen**

Ein String soll in umgekehrter Zeichenfolge ausgegeben werden. Zum Beispiel COMPUTER soll sich in RETUPMOC verwandeln. (Man braucht dies zum Beispiel, wenn man Wörter nach dem Wortende sortieren will, etwa für ein Reimlexikon.)

Die Steckmodulversion Comal 2.01 macht es leicht, dies zu programmieren, denn dort gibt es Funktionen, die Strings ausgeben:

```
9000 FUNC rueckwaerts$(original$) CLOSED
9010  umgedreht$=""
9020  FOR i#=LEN(ORIGINAL$) TO 1 STEP -1 DO
9030   umgedreht$:+original$(i#)
9040  ENDFOR i#
9050  RETURN umgedreht$
9060 ENDFUNC rueckwaerts$
```

Aufgerufen wird diese Funktion zum Beispiel so:

```
PRINT rueckwaerts$("Computer")
```

In der etwas einfacheren Diskettenversion Comal 0.14 gibt es »nur« numerische Funktionen, das heißt solche, die Zahlen ausgeben. Schon hier also müssen wir uns mit einer Prozedur begnügen:

```
9000 PROC umdrehen(original$, REF umgedreht$) CLOSED
9010  umgedreht$=""
9020  FOR i#=LEN(ORIGINAL$) TO 1 STEP -1 DO
9030   umgedreht$:=umgedreht$+original$(i#)
9040  ENDFOR i#
9050 ENDPROC umdrehen
```

Und so rufen wir die Prozedur auf:

```
umdrehen("Computer",neu$)
PRINT neu$
```

In Basic müssen wir natürlich erst recht eine Prozedur programmieren:

```
34000 rem proc: umdrehen (uo riginal$: in, umgedreht$:
      out)
34010   umgedreht$=""
34020   for ui= len(uo riginal$) to 1 step -1
34030     umgedreht$=umgedreht$+mid$(uo riginal$,ui,1)
34040   next
34050 return
```

Der Basic-Aufruf:

```
uo riginal$="Computer" : gosub 34000 : neu$=umgedreht$
print neu$
```

Zusammenfassung: Funktionstypen

Wir haben für Basic-Funktionen folgende Befehlsstruktur vorgeschlagen, wieder in Anlehnung an Comal (Bild 4).

```
REM FUNC Funktionsname (Variable)
  DEF FN Funktionsname (Variable)
  DEF FN...
  ...
  ...
REM ENDFUNC
```

Bild 4. Eine Comal-analoge Befehlsstruktur für Funktionen in Basic

Alle Variablen sind vom IN-Typ und werden entsprechend gekennzeichnet. Für Funktionen ohne Kommunikation benutzen wir X als Definitionsvariable und 0 als Klammerinhalt beim Aufruf. Bei Mehrfachkommunikation werden alle benötigten Informationen von Hand mitgegeben. Die dafür verwendeten Variablen beginnen wieder mit U, um sie lokal zu halten.

Drei Funktionstypen haben wir besprochen (Bild 5).

Funktionstypen		
Kommunikationstyp	**Definition**	**Aufruf z.B.**
1 keine Kommunikation	DEF FN Name (X)	Z=FN Name (0)
2 Einwegkommunikation	DEF FN Name (LV)	Z=FN Name (V)
3 Mehrfachkommunikation	DEF FN Name (X)	U1=Wert:U2=Wert : Z=FN Name (0)

Abkürzungen:
V = Variable, LV = Lokal gültige Variable
U1,U2 = Unterprogrammvariablen

Bild 5. Die drei Arten der Parameterübergabe in Funktionen

Weitere Informationen über Basic-Funktionen finden Sie in dem Aufsatz »Funktionen für Anfänger« im 64er, Ausgabe 5/85.

Noch ein Tip zum Schluß: Namen für Unterprogramme

Wir streben gut strukturierte Programme und ähnliches deshalb an, damit wir sie leichter lesen und verstehen können. Wie lesbar ein Programm ist, hängt entscheidend von den Namen ab, die wir unseren Variablen sowie unseren Prozeduren und Funktionen geben. Besonders wichtig sind dabei die letzteren. Der Name eines Unterprogramms soll so deutlich wie möglich ausdrücken, wofür das Unterprogramm da ist. Das ist aber, wie die Erfahrung zeigt, gar nicht immer so einfach. Deshalb hier ein paar Tips.

Beim Erfinden von Namen geht man am besten von ***Sätzen*** aus, die man dann auf die allerwesentlichsten Wörter ***reduziert***. Was für Sätze man zugrundelegt, das hängt ab von der Art des Unterprogramms. Welche Wörter wesentlich sind, hängt einmal davon ab, was genau das Unterprogramm bewirkt, zum anderen aber auch vom Gesamtzusammenhang des Programms, in dem das Unterprogramm ein Element bildet.

Erstens zu den Sätzen: Prozeduren sind Handlungsbefehle; sie tun etwas. Die Sätze, aus denen Prozedurnamen abgeleitet werden, sollten deshalb Handlungen benennen. Beispiel: »Die Prozedur übersetzt einen Text vom Deutschen ins Englische«.

Funktionen erzeugen ein Ergebnis. Bei den Booleschen Funktionen ist dieses Ergebnis entweder 1 oder 0, das heißt ja oder nein, stimmt oder stimmt nicht, TRUE oder FALSE (um mit Comal zu »sprechen«). Funktionen dieses Typs »antworten« auf eine Ja/Nein-Frage. Sätze, die als Grundlagen für Namen von Booleschen Funktionen dienen sollen, müssen deshalb vom Typ »Ja/Nein-Frage« sein, zum Beispiel »Ist der Artikel preiswert?«

Alle anderen Funktionen erzeugen Ergebnisse, die weder von der Anzahl noch vom Inhalt her grundsätzlich festlegbar sind. Sie ermitteln Antworten auf Wer? Was? Wo-Fragen. Beispiel: »Welche Farbe hat das Kleid?« Da kann die Antwort »rot« sein oder »grün«, »blau«, »gelb«, »braun«, etc.

Zweitens: Was genau bewirkt ein Unterprogramm und wie wirkt sich das auf den Befehlsnamen aus? Zunächst zu den Prozeduren: Welche Wörter jeweils in den Befehlsfolgen eingehen, hängt wesentlich davon ab, welcher Aspekt der Handlung im Mittelpunkt steht. Das kann die nackte Handlung sein, oder ihr Ergebnis, oder die Art und Weise, wie die Handlung abläuft, und anderes mehr. Beispiele: »Die Prozedur ***ersetzt*** in einem String einen Teilstring durch einen anderen.« — ERSETZEN; »Die Prozedur zeichnet einen ***Baum***.« — BAUM; »Die Prozedur ***druckt*** einen Text besonders ***langsam*** aus.« — LANGSAM DRUCKEN.

Für jede Funktion einen treffenden Namen

Boolesche Funktionen informieren darüber, ob etwas zutrifft oder nicht. Die zugrundeliegende Frage enthält meistens ein Adjektiv (»richtig« in »Ist die Antwort ***richtig***?«) oder ein Partizip (»gefunden« in »Ist das Wort ***gefunden*** worden?«). Boolesche Funktionen werden deshalb (meistens) mit Adjektiven (RICHTIG) oder Partizipien (GEFUNDEN) benannt. Ein Trick, falls Sie dabei Schwierigkeiten haben: Probieren Sie aus, ob Sie (fast menschensprachlich) sagen können: »IF richtig THEN...«, oder »IF NOT gefunden THEN...«. Wenn ja, haben Sie einen guten Namen, denn genau so werden Boolesche Funktionen oft verwendet.

Bei allen anderen Funktionen sind Substantive die angemessene Wahl oder Substantive mit vorausgehendem Adjektiv: »Was ist der ***Preis***?« — PREIS; »Welches ist die ***größere Zahl***?« — GROESSERE ZAHL.

Drittens: Welchen Einfluß hat der Gesamtzusammenhang eines Programms auf die Wahl des Befehlsnamens? Beispielsatz: »Die Prozedur ***druckt*** einen ***Text*** auf dem ***Bildschirm*** aus«.

Die Prozedur wird DRUCKEN heißen, wenn es ausschließlich Texte zu drucken gibt. Wenn aber auch Bilder ausgedruckt werden können, sollte man sie TEXT DRUCKEN nennen, damit man sie von einer anderen Prozedur, die möglicherweise BILD DRUCKEN heißt, unterscheiden kann. Wenn ein Drucker angeschlossen ist und der Text sowohl auf den Drucker als auch auf den Bildschirm gegeben werden kann, heißt die Prozedur vielleicht besser TEXT AUF BILDSCHIRM.

Übrigens sollten Verben nicht in der Befehlsform stehen, sondern in der Grundform — Prozeduren wie LIES WORT oder BAUE WORT AUF — mit wem reden die? Solche Namen sind wohl aus dem Englischen übersetzt (READ WORD und BUILD WORD) und falsch verstanden. Tatsächlich handelt es sich bei »read« und »build« um Infinitive, und Infinitive sollten wir auch im Deutschen benutzen: WORT LESEN und WORT AUFBAUEN.

Eine letzte Bemerkung: Zuweilen macht man Unterprogramme, um Befehle zu simulieren, die in anderen Basic-Versionen oder gar in anderen Programmiersprachen vorhanden sind. Dann benutzt man gern die dort üblichen Namen. Der Zweck der Unterprogramme ist dann (zumindest für den Eingeweihten) auf Anhieb verständlich. Beispiele: FLASH, PRINT AT, PLOT, CIRCLE, UCASE, MAX (statt GROESSERE ZAHL), LINE (statt LINIE) oder REPLACE (statt ERSETZEN).

Nur Vorschläge

Wer strukturiert programmieren will, braucht als Werkzeuge dazu vernünftige Befehlsstrukturen. Solche Befehlsstrukturen sind die Steuerbausteine (insbesondere Wiederholungsschleifen und Verzweigungen) und die Unterprogramm-Bausteine (Prozeduren und Funktionen). Wo diese von der Programmiersprache nicht zur Verfügung gestellt werden, muß man sich selber helfen und entsprechende Strukturen herstellen. Vorschläge dazu werden in dieser Serie gemacht.

Diese Vorschläge sind jedoch nicht dazu gedacht, sklavisch befolgt zu werden. Sie sollen vielmehr Anregungen sein für eigene Anstrengungen und eigene Überlegungen. Vielleicht haben Sie bessere Ideen.

Noch allerdings haben wir nicht alle Werkzeuge besprochen. Noch fehlen umfassendere Bausteinstrukturen, nämlich die, aus denen schließlich Programme zusammengesetzt sind. Sie sollen »Modulbausteine« heißen und sind das Thema des nächsten, des letzten Teils dieser Serie. (Prof. Burkhard Leuschner/cg/gk)