Einführung in Forth-83

Teil 5

**Dr. Hartmut Pfüller (Leiter),
Dr. Wolfgang Drewelow, Dr. Bernhard Lampe
Ralf Neuthe, Egmont Woitzel
Wilhelm-Pieck-Universität Rostock,
Sektion Technische Elektronik**

5. Die qualitative Erweiterung von Forth

Beim Vergleich mit anderen Programmiersprachen fällt auf, daß im Kern von Forth keine Definitionsworte für Datenstrukturen wie Felder oder Records enthalten sind. Die Bedeutung von Datenstrukturen ist aber für die Programmierung unbestritten, so daß die Frage steht, wie solche Probleme in Forth rationell zu behandeln sind.
Anstatt eine mehr oder weniger fest vorgefertigte Menge von starren Datentypen vorzugeben, stellt Forth dem Programmierer Werkzeuge zum Eigenbau von Datenstrukturen zur Verfügung. Er kann damit seine Datenstrukturen für das Problem maßschneidern und muß nicht das Problem solange umformen, bis es in eine angebotene Datenstruktur paßt.
Zu dieser für Forth typischen Vorgehensweise gehört auch die Möglichkeit, spezielle Werkzeuge zu schaffen, die eine rationelle und problemnahe Behandlung der selbsterzeugten Datenstrukturen sichern. Welchen Weg schlägt Forth ein, um dieses Ziel zu erreichen? Das Prinzip ist vergleichbar mit der Vorgehensweise bei der im Teil 3 beschriebenen Ausgabeformatierung. Auch die komplexen Definitionsworte wie **:** oder **VARIABLE** basieren auf elementaren Bestandteilen, die dem Programmierer sämtlich einzeln zugänglich sind und die zur Konstruktion neuer definierender Worte benutzt werden können.

5.1. Die Verwaltung des Wörterbuchspeichers

5.1.1. Funktionen

Eine Orientierung darüber, wo für den nächsten Wörterbucheintrag freier Speicherplatz ist, erhält man dadurch, daß das Wort **HERE** die entsprechende Adresse zum Stapel schafft. Wenn diese Adresse in einer (nicht standardisierten) Variablen **DP** (Directionary Pointer, deutsch: Wörterbuchzeiger) geführt wird, könnte HERE so implementiert sein:

```
: HERE ( ===> addr) DP @ ;
```

Um das Wörterbuch am Ende zu verlängern, kann das Wort **ALLOT** benutzt werden. ALLOT erwartet eine Zahl auf dem Datenstapel und reserviert dem Wert entsprechend viele Bytes für das Wörterbuch von der durch HERE bezeichneten Stelle ab. Der Inhalt der mit ALLOT reservierten Speicherplätze wird nicht gesetzt, er bleibt so, wie er sich zufällig bis zu diesem Zeitpunkt ergeben hatte. Systemintern könnte ALLOT etwa so definiert sein:

```
: ALLOT ( n ===> ) DP +! ;
```

Ebenfalls eine Vergrößerung des Wörterbuchs erreicht man mit den beiden Worten

```
,     C,
```

Das Wort **,** schreibt die auf dem Datenstapel befindliche 16-Bit-Zahl ans Wörterbuch hintan; das Wörterbuch wird damit um 2 Byte länger, wobei der Inhalt dieser 2 Byte durch die vom Stapel geholte Zahl gesetzt wird. Entsprechend vergrößert **C,** das Wörterbuch um genau ein Byte, wobei die 8 Bit, die dessen Inhalt bestimmen, auch wieder als Wert vom Stapel geholt werden. Die beiden Worte könnten im System beispielsweise so definiert sein:

```
: , ( 16b ===> ) HERE 2 ALLOT ! ;
: C, ( 8b ===> ) HERE 1 ALLOT C! ;
```

Beim Zugang zur physischen Ebene müssen von Fall zu Fall spezifische Schaltkreiseigenschaften berücksichtigt werden. Angenommen, der Programmierer wünscht, ins Wörterbuch eine Tabelle von Zeigern zu Interruptserviceroutinen einzutragen, und der Prozessor verlangt, daß die Tabelle bei einer durch 8 teilbaren Adresse beginnt. Die Variable DP kann dann durch eine Wortfolge wie

```
HERE NEGATE 8 MOD ALLOT
```

auf die nächste durch 8 teilbare freie Adresse des Wörterbuchs gesetzt werden. Mittels solcher und ähnlicher Techniken kann man beispielsweise die Interruptbehandlung vollständig von Forth aus organisieren.

5.1.2. Vektoren und Tabellen

Die genannten Mittel gestatten bereits die Vereinbarung von Datenbereichen im Wörterbuch. Beispielsweise kann durch die Sequenz

```
VARIABLE ZAEHLERFELD 8 ALLOT
```

ein Feld für 5 Werte von je 16 Bit im Wörterbuch reserviert werden; ein Wert direkt durch **VARIABLE** und die restlichen vier Werte durch **8 ALLOT**. **VARIABLE** legen bei der Definition von **ZAEHLERFELD** dessen Laufzeitverhalten fest: Beim Aufruf von ZAEHLERFELD wird die Basisadresse (die Adresse der 0. Komponente) auf den Datenstapel gelegt. Die übrigen Komponenten sind durch den Offset von 2, 4, 6, 8 adressierbar, für Index 3 etwa so:

```
ZAEHLERFELD 6 +
```

Die Initialisierung aller fünf Zähler mit Null kann nach

```
: RUECKSETZEN ( 'feld ===> )
   5 0 DO 0 OVER ! 2+
      LOOP DROP ;
```

durch die Wortfolge

```
ZAEHLERFELD RUECKSETZEN
```

gemeinsam ausgeführt werden. Durch die Zusammenfassung von Zählern in einem Feld können solche Operationen vereinfacht werden, die alle Zähler des Feldes betreffen. Dazu gehören neben der Initialisierung beispielsweise Vergleiche der Zählerstände oder deren Auswertung und Darstellung.
Im obigen Beispiel ist es eventuell nachteilig, daß die Initialisierung nur für Felder der festen Dimension 5 arbeitet. Will man Felder variabler Länge in ähnlicher Weise behandeln, so muß die zugeordnete Feldlänge ermittelt werden können. Das läßt sich erreichen, wenn die Felddimension mit abgespeichert wird (beispielsweise vor dem Feld selbst). Entsprechend der oben stehenden Variante könnte man

```
VARIABLE ZAEHLERFELD 10 ALLOT
5 ZAEHLERFELD !
```

schreiben. Noch einfacher ist jedoch die Konstruktion, wenn man als Definitionswort für den Wörterbucheintrag das (primitivere) Definitionswort **CREATE** verwendet. Es wirkt wie VARIABLE, reserviert selbst aber nach dem Namen noch keinen zusätzlichen Speicherplatz für Daten. Die Folge

```
CREATE ZAEHLERFELD 5 , 10 ALLOT
```

spiegelt die Struktur des Feldes gut wider; CREATE legt das Laufzeitverhalten von ZAEHLERFELD fest. Das mit

```
: INITIALISIEREN ( 'feld ===> )
   DUP @ 0 DO 2+ 0 OVER !
      LOOP DROP ;
```

definierte Wort initialisiert so ein Feld variabler Länge.
Fließpunktarithmetik und transzendente Funktionen sind nicht Bestandteil des Forthkerns, weil sich die Anforderungen der Anwender an Genauigkeit und Verarbeitungsgeschwindigkeit stark unterscheiden. Außerdem lassen sich viele Probleme mit ganzen Zahlen lösen, wenn man auf die (gefährliche) Bequemlichkeit des Rechnens ohne vorherige Abschätzung verzichtet. Als Beispiel soll die Funktionsberechnung unter Verwendung von Funktionstafeln erläutert werden, wobei man bei geeigneter Skalierung im Bereich der ganzen Zahlen bleibt.
Eine mit

```
CREATE SINUSTAFEL
        0 ,  175 ,  349 ,  523 ,  698 ,
      872 , 1045 , 1219 , 1392 , 1564 ,
     1737 , 1908 , 2079 , . . .
        .
        .
        .      9976 , 9986 , 9994 , 9999 ,
    10000 ,
```

erzeugte Tafel enthält die mit 10000 skalierten Sinuswerte zu den Winkeln von 0, 1, . . ., 90 Grad. Hier wird die Zweckmäßigkeit der Vergabe des Namens **,** für die direkte Eintragung ins Wörterbuch erkennbar. Ein Leser des Programmteils SINUSTAFEL muß nicht unbedingt mit dem an das Komma gebundenen Kompilationsvorgang vertraut sein; es genügt, die im gewöhnlichen Sprachgebrauch übliche Bedeutung des Kommas als Trennsymbol zu kennen. Durch die ähnlich geschickte Wahl der Namen kann der Programmierer die Lesbarkeit seiner Programme günstig beeinflussen. Das Aufsuchen des passenden Sinuswertes geschieht nach

```
: SINUS ( n ===> sin[n] )
   2 * SINUSTAFEL + @ ;
```

in der Form

```
45 SINUS . (cr) 7071 ok
```

Unter Ausnutzung der Periodizität können beliebige Winkel auf den angegebenen Argumentbereich zurückgerechnet werden. Durch einen Suchprozeß innerhalb der Tabelle ist auch die Arkussinusfunktion implementierbar. Das oben definierte Wort **SINUS** stellt die primitivste Form des Zugriffs auf die Tabelle dar. Durch lineare Interpolation ließen sich auch Zwischenwerte berechnen.

5.2. Definition von Wortklassen

Wenn mehrere Objekte die gleiche oder eine ähnliche Struktur haben, kann man, wie am Beispiel der Initialisierung deutlich wurde, Operationen so erzeugen, daß sie über allen Objekten einer solchen Familie ausführbar sind. Wünschenswert wäre nun, auch die Kompilationsvorschrift für zur gleichen Familie gehörende Objekte nur einmal erklären zu müssen. Das spart Programmspeicherplatz und erhöht die Übersichtlichkeit. Mit der einmal erzeugten Form können dann viele gleichartige Objekte „gegossen" werden.
Die Konstruktion einer „Gußform" wird ermöglicht, weil auch Definitionsworte (z. B. CREATE) selbst wieder in Definitionen verwendet werden dürfen. Auf diese Art kann der Anwender dann eigene Definitionsworte definieren, die bei ihrer Ausführung selbst wieder neue Worte eines (vom Anwender) bestimmten Typs erzeugen. So können mit einem vom Programmierer neu geschaffenen Definitionswort spezielle Eigenschaften für eine ganze Klasse von neuen Worten fixiert werden. Die zweckmäßige Festlegung von problemgerechten Wortklassen kann in hohem Maße die Qualität eines Programms bestimmen. Sie verlangt eine gründliche Analyse der Problemstellung, insbesondere unter dem Gesichtspunkt der Zerlegung in ähnlich geartete Teilprobleme. Nach Erklärung des Datentyps

```
: FELD ( n ===> )
  CREATE DUP , 2 * ALLOT ;
```

beispielsweise würde durch

```
5 FELD ZAEHLERFELD ( ===> addr)
```

ein Zählerfeld der Dimension 5 ins Wörterbuch eingetragen werden. FELD ist nun das Definitionswort für einen vom Programmierer neu geschaffenen Datentyp. Das Laufzeitverhalten von ZAEHLERFELD wird wiederum durch CREATE festgelegt. Nach Aufruf von ZAEHLERFELD würde die Parameterfeldadresse des ZAEHLERFELDes auf den Datenstapel gelegt werden; dort war bei der Definition durch FELD die Dimension eingetragen worden.

5.2.1. Fadencodeniveau

Mitunter wird es sinnvoll sein, wenn nach dem Aufruf eines mit FELD erzeugten Objektes nicht die Adresse der Dimension, sondern die Adresse des ersten Feldelements zum Stapel geliefert wird. Das Laufzeitverhalten, das dem neudefinierten Wort durch CREATE mitgegeben wurde, müßte zu diesem Zweck anders programmiert werden können. Speziell für diesen Zweck ist das Wort **DOES>** vorgesehen. Die in der Konstruktion

```
: name . . . CREATE . . . DOES> . . . ;
```

nach DOES> stehenden Worte werden bei Aufruf jedes mit *name* definierten Objektes ausgeführt. Dabei ist es wichtig zu wissen, daß unmittelbar vorher die Parameterfeldadresse des Objekts auf dem Stapel bereitgelegt wird. Die auf DOES> folgenden Worte können dann diese Information über den Beginn des Parameterfeldes verwenden, um damit ein gewünschtes Laufzeitverhalten des Objekts zu erzeugen.
Eine neue Klasse von Datenfeldern, deren Abkömmlinge zwar ihre Dimensionen enthalten, aber beim Aufruf die Adresse ihrer nullten Komponente auf den Stapel legen, wird mit

```
: VEKTOR ( n ===> )
  CREATE DUP , 2* ALLOT
  DOES> ( ===> 'x0 ) 2+ ;
```

beschrieben. Beim Stapelkommentar nach DOES> muß man immer berücksichtigen, daß unmittelbar vor dem Abarbeiten der DOES>-Passage noch die Parameterfeldadresse des definierten Objekts obenauf gelegt wird. Sie sollten etwas Zeit investieren, um sich klarzumachen, welche Konsequenzen es hat, daß die Passage hinter DOES> *nicht* bei der Ausführung von VEKTOR ausgeführt wird. Vielmehr sorgt DOES> dafür, daß bei jeder Definition eines Objekts mittels **VEKTOR** das Codefeld dieses Objekts mit einem Verweis zur DOES>-Passage überschrieben wird. Die Worte nach DOES> werden damit genau bei jedem Aufruf eines Objekts abgearbeitet. Dadurch kommt auch die Familieneigenschaft aller mit demselben Definitionswort erzeugten Objekte zustande.
Der Zugriff auf Komponenten von mit VEKTOR definierten Vektoren wie

```
5 VEKTOR ZAEHLERFELD ( ===> addr)
```

ließe sich durch

```
: KOMPONENTE ( 'x n ===> 'xn)
  2 * + ;
```

bewerkstelligen. So wird durch

```
ZAEHLERFELD 3 KOMPONENTE @
```

der Inhalt des Elements mit dem Index 3 auf den Datenstapel gelegt.
Die Konstruktion **CREATE . . . DOES>** macht es möglich, die Kompilationsphase für Objekte einer Wortklasse vollständig von der Phase der Ausführung getrennt zu betrachten und zu programmieren. Der Programmierer kann die aus seiner Sicht insgesamt erforderlichen Aktionen zerlegen und der jeweils entsprechenden Phase zuordnen (für den Kompilationsvorgang hinter CREATE und für den Ausführungsvorgang hinter DOES>).
Zur Erläuterung des Gebrauchs von Wortklassen soll nun ein Beispiel aus Teil 2 des Kurses etwas modifiziert behandelt werden:
In einer Variablen MATERIAL stehe (wie auch damals) die Adresse des Zählers für Teile des gerade aktuellen Materials. Die Reservierung von Speicherplatz für einen materialabhängigen Zähler und die Bereitstellung seiner Adressen bei Nennung des Materials werden jetzt in dem Definitionswort

```
: MATERIAL:
  VARIABLE ( installiert Zähler)
  DOES> MATERIAL ! ( lädt Zähler) ;
```

zusammengefaßt. Nach den Definitionen

```
MATERIAL: HOLZ
MATERIAL: PLAST
```

und den „Verschönerungen"

```
: DAZU ( n addr ===> ) +! ;
: ? ( addr ===> ) @ . ;
```

sind die in Teil 2 benutzten Aktionen wie

```
7 HOLZ TEILE DAZU
  PLAST TEILE ?
```

wieder verwendbar.
Will man verschiedenartige Teile nach Material sortiert zählen, so kann man für jedes Material mehrere „Körbe", das heißt ein Zählerfeld bereitstellen. Der zum jeweiligen Teil gehörende Korb ist durch einen Offset zum Anfang des Zählerfeldes des entsprechenden Materials erreichbar (Bild 5.1).
Das Definitionswort TEIL: speichert während der Definition eines Teils den vom Programmierer zugeordneten Offset n verdoppelt ab

```
VARIABLE  MATERIAL

:  TEIL:          ( n ===> )
   2  *  CONSTANT
         DOES>    ( ===> korbadresse)
            @    MATERIAL  @  +  ;

2  CONSTANT   #TEILE
0  TEIL:      KEILE
1  TEIL:      STIFTE

:  MATERIAL:   ( ===> )
   CREATE  #TEILE   0  DO  0  ,  LOOP
   DOES>       ( ===> )   MATERIAL  !  ;

MATERIAL:  HOLZ
MATERIAL:  PLAST
```

Bild 5.1 Beispiel zu CREATE . . DOES>

und bildet bei Nennung des jeweiligen Teils die Adresse des zugehörigen materialrichtigen Zählers. Damit ist die Schreibweise

```
HOLZ 12 KEILE DAZU 60 STIFTE DAZU PLAST
STIFTE ?
```

möglich. Wenn für #TEILE vorsorglich ein etwas größerer Wert vereinbart wird, können mittels TEIL: auch noch nachträglich weitere Arten von Teilen definiert werden. Die Konstruktion CREATE . . . DOES> ist einfach, leistungsfähig und ungewöhnlich. Wir empfehlen Ihnen, eigene Experimente mit betont unkomplizierten Schritten zu beginnen. Eventuell sind in der ersten Zeit erläuternde Hilfsausschriften nützlich, wie im folgenden Beispiel für ein Definitionswort, das Objekte erzeugen kann, die nichts anderes tun, als ihre eigenen Aktivierungen mitzuzählen:

```
: ZAEHLER ." vor Definition "
  CREATE  ." nach Definition " 0 ,
  DOES>   ." bei Aktivierung"
    1 OVER +! @ . ;
```

Nach der Erzeugung eines Objekts mit

```
ZAEHLER KLICK
```

zeigt KLICK bei jeder Aktivierung an, der wievielte Aufruf das war.

5.2.2. Mikrocodeniveau

Alternativ zum Fadencode kann der Ausführungsteil auch in Maschinencode notiert werden. Für diesen Fall ist das Wort **;CODE** anstelle von **DOES>** zu benutzen. Die auf ;CODE folgenden Worte müssen dann Be-

standteil des Forth-Assemblers sein, der vorher geladen worden sein muß. Falls kein Assembler verfügbar ist, kann auch der von Hand übersetzte Maschinencode direkt mit Hilfe von **,** oder **C,** eingetragen werden. Man wird ;CODE benutzen, wenn man auf die größere Rechengeschwindigkeit von Assemblerprogrammen angewiesen ist. Zum Beispiel könnte man die Zugriffszeit auf Werte in Tabellen verkürzen, wenn man sie mit dem Definitionswort nach Bild 5.2 erzeugt. Der Programmierer braucht auch hier einen Zugang zum Parameterfeld der letztlich zu definierenden Objekte. Während DOES> hierfür die Parameterfeldadresse (PFA) auf dem Stapel hinterläßt, regelt sich nach ;CODE der

```
:  TAFEL        ( xn-1 . . . x1 x0 n ===> )
     CREATE
          0  DO   ,  LOOP
     ;CODE      ( i ===> xi )
     DE       INC,                ( WA-Register auf PFA justieren)
     HL       POP,                ( Index i vom Stapel holen)
     HL HL    ADD,  DE HL  ADD,   ( Elementadresse berechnen)
     (HL)  E  LD,   HL     INC,   ( Element nach DE laden)
     (HL)  D  LD,
     DE       PUSH,               ( Element xi zum Stapel)
     NEXT  #  JP,                 ( zurueck zum Adressinterpreter)
     END-CODE
```

Bild 5.2 Beispiel zu CREATE . . . ;CODE

dafür jeweils eins zu eins – deren Codefeldadressen im Kompilat kann mittels der Vorrangworte durchbrochen werden, weil sie in der Regel nicht ins Wörterbuch gelangen und stattdessen sogar andere Worte ins Wörterbuch kompilieren können. Das würde der Programmierer zunächst gar nicht bemerken, wenn er von anderen Programmiersprachen her gewöhnt ist, daß ihm der Zugang zur Gestaltung des Kompilats in der Regel verschlossen bleibt.
Der versierte Forthprogrammierer wird von Fall zu Fall wünschen, Vorrangworte nicht zur Kompilationszeit arbeiten zu lassen, sondern sie entgegen ihrem Normalverhalten trotzdem zu kompilieren. Dafür gibt es die Möglichkeit, die Vorrangeigenschaft fallweise explizit zu unterdrücken.
Mit dem Wort **[COMPILE]** kann man die Kompilation eines Wortes erzwingen; die eventuelle Vorrangeigenschaft wird unterdrückt. Die erzwungene Kompilation betrifft nur das unmittelbar auf [COMPILE] folgende Wort. Damit [COMPILE] diese Wirkung erzielen kann, ist es selbst als Vorrangwort erklärt.
Als Gegenstück zur vorgezogenen Ausführung von Worten ist auch eine aufgeschobene Kompilation möglich. Das erreicht man durch eine Wortfolge der Form

: yyy . . . COMPILE xxx . . . ;

Der Effekt ist der, daß yyy bei seiner Ausführung eine Kompilation von xxx durchführt (also die Eintragung der **C**odefeld**a**dresse von xxx ans Wörterbuchende). Während der Definition von yyy werden die CFAs von **COMPILE** und anschließend von xxx ganz normal ins Wörterbuch eingetragen. Beim Aufruf von yyy sorgt dann COMPILE dafür, daß die CFA von xxx noch einmal an das aktuelle Ende des Wörterbuchs kompiliert wird. Typisch wird COMPILE in Vorrangworten verwendet, wie wir in einem Beispiel im nächsten Abschnitt zeigen werden.

Zugang zur PFA des definierten Objekts über das WA-Register der virtuellen Forthmaschine. Welches Prozessorregister in einer konkreten Implementation dafür verwendet wird, muß der zugehörigen Dokumentation entnommen oder anderweitig in Erfahrung gebracht werden. Selbstverständlich hat man sich mit der hier vorgeführten Version für das Definitionswort TAFEL auf den Prozessor Z 80 und meist auch noch auf einen bestimmten Assembler festgelegt.
Nach der Vorbereitung
10000 9999 . . . 175 0
91 TAFEL SINUS
kann die Sinusfunktion für einen bestimmten Winkel wieder mit
n SINUS
ermittelt werden. Solche Tafeln sind ein sehr bequemes und rechenzeiteffektives Verfahren, um Funktionen implementieren zu können.

5.3. Compilererweiterungen

5.3.1. Vorrangworte

Im Teil 2 des Kurses (s.MP 5/89) wurden Strukturierungsworte vorgestellt. Sie dienen im Kompilationsmodus zum Aufbau eines möglichst laufzeitrationellen Fadencodes und werden deshalb innerhalb einer Definition *ausgeführt*, während „normale" Worte im Gegensatz dazu in Fadencode *kompiliert* werden. Soll ein bestimmtes vom Programmierer definiertes Wort diese besondere Eigenschaft erhalten, so daß es trotz des Kompilationsmodus ausgeführt wird, so ruft man nach dessen Definition das Wort **IMMEDIATE** (deutsch sinngemäß: unverzüglich) auf. Das zuletzt definierte Wort wird dadurch zu einem *Immediate*-Wort oder *Vorrangwort*.
Die Ausführung von Vorrangworten wird gewissermaßen in die Kompilationszeit vorgezogen.
Das bisher betont einfache Prinzip der Kompilation von neuen Worten durch Zusammenstellung von Worten im Quelltext und durch –

5.3.2. Programmverzweigungen

Im Abschnitt 2.3 wurden Worte gezeigt, mit denen Programme nach den Regeln der strukturierten Programmierung gegliedert werden können. Bei der Erzeugung *eigener* Verzweigungsstrukturen ist es zuweilen effektiver, sie aus elementaren Bestandteilen neu aufzubauen, anstatt die aus dem Anwenderwortschatz geläufigen vorgefertigten Strukturierungsworte zu verwenden.
Zu den elementaren Bestandteilen der Verzweigungsstrukturen gehören die Worte

>MARK >RESOLVE <MARK <RESOLVE

Mit ihnen werden zur Kompilationszeit Sprungadressen in Verzweigungsstrukturen verwaltet. Es gehören jeweils eine Markierung (MARK) und eine Auflösung (RESOLVE) zusammen. Mit MARK wird eine Marke (Adresse) auf den Datenstapel gelegt, zu der später von derjenigen Position des Wörterbuchzeigers aus gesprungen werden muß, die bei Nennung von RESOLVE gerade aktuell war (oder wird). Die Richtung der Sprünge ist an den Namen MARK und RESOLVE durch < für rückwärts und > für vorwärts abzulesen.
Besonders einfach ist die Eintragung von Rückwärtssprüngen durch

: <MARK (===> addr) HERE ;
: <RESOLVE (addr ===>) , ;

nachvollziehbar. Bei Vorwärtsreferenzen wird die Zieladresse erst zu einem späteren Zeitpunkt ermittelt, so daß der Speicherplatz erst reserviert und später die Adresse dort eingetragen wird. Auch hier läßt sich der möglichen Implementierung

```
: >MARK ( ===> addr )
  HERE 2 ALLOT ;
: >RESOLVE ( addr ===> )
  HERE SWAP ! ;
```

die Arbeitsweise dieser Worte entnehmen. Da die Worte nur die Eintragung der Sprungadressen leisten, müssen sie noch mit Sprungbefehlen und Sprungbedingungen verknüpft werden (vergl. BRANCH, ?BRANCH in Teil 4 des Kurses).
Die Sprungadreßberechnungen sind so angelegt, daß sie sich verschachteln lassen. Damit dem Programmierer dabei keine Flüchtigkeitsfehler unterlaufen, prüfen die meisten Forth-Systeme während der Kompilation die Vollständigkeit der Verzweigungsstrukturen. Oft wird dazu bei Eröffnung der Verzweigung über die Adresse eine Kennzahl gelegt, deren Anwesenheit im Schließbefehl vor Ausführung der Adreßverknüpfung überprüft wird. Auf solche einfachen Sicherungsmaßnahmen sollte man bei eigenen Kompilationsworten nicht verzichten. Man kann eigene Kompilationsworte gerade mit dem Ziel schaffen, während der Kompilation weitgehend die Richtigkeit des Programms zu überprüfen.
Als Beispiel für neue Kompilationsworte soll eine inzwischen weit verbreitete Fallkonstruktion betrachtet werden. Sie verallgemeinert die Struktur **IF . . . ELSE . . . THEN** auf eine beliebige Anzahl von Fällen. Welcher Zweig des Programms abgearbeitet wird, soll sich aus dem Wert einer zur Abarbeitungszeit auf dem Datenstapel liegenden Zahl (Fallnummer) ergeben. Die Struktur wird in der Weise

```
: . . . CASE n1 OF . . . ENDOF
           n2 OF . . . ENDOF
           .  .  .  .
           nk OF . . . ENDOF
  . . . ENDCASE . . . ;
```

benutzt. Die vorkommenden Worte können wie in Bild 5.3 definiert werden. Alle vier definierten Strukturierungsworte werden jeweils durch IMMEDIATE zu Vorrangworten erklärt; mit COMPILE werden die erforderlichen Wörterbucheinträge erzwungen.
Die Kompilation eines die **CASE**-Konstruk-

```
:  CASE   ( ===> addr 5 ; Strukturanfang )
   CSP  @  SP@  CSP  !  5  ;          IMMEDIATE

:  OF     ( 5 ===> addr 6 ; Zweiganfang )
   5  ?PAIRS   ( Strukturs icherung )
   COMPILE  OVER    COMPILE  =
   COMPILE  ?BRANCH
   >MARK    COMPILE DROP    6  ;      IMMEDIATE

:  ENDOF   ( addr1 6 ===> addr2 5 ; Zweigende)
   6  ?PAIRS    ( Strukturs icherung )
   COMPILE   BRANCH
   >MARK  SWAP  >RESOLVE  5 ;         IMMEDIATE

:  ENDCASE   ( addr addr1 ... addrk ===>)
   5  ?PAIRS    ( Strukturs icherung )
   COMPILE   DROP
   ( Sprunge ans Strukturende nachtragen:)
     BEGIN  SP@  CSP  @  -  WHILE   >RESOLVE
     REPEAT
   CSP  ! ; ( CSP wiederherstellen)   IMMEDIATE
```

Bild 5.3 Codierung einer CASE-Verzweigung

tion enthaltenden Wortes soll nun kurz verfolgt werden:
Das Wort CASE eröffnet die Struktur, indem es den Wert der Variablen **CSP** auf den Stapel rettet, den aktuellen Stand des Stackpointers für einen späteren Vergleich in der Variablen CSP speichert und die willkürlich gewählte Kennzahl 5 zur späteren Kontrolle der Vollständigkeit der Struktur auf den Stapel legt. Danach wird die Fallnummer ins Wörterbuch kompiliert. Das darauf folgende **OF** kontrolliert die richtige Reihenfolge innerhalb der CASE-Konstruktion, indem die durch CASE auf dem Stapel hinterlassene 5 abgefragt wird. Falls **?PAIRS** nicht bekannt ist, kann man vorläufig

```
: ?PAIRS ( n1 n2 ===> )
  – IF ." falsche Struktur " THEN ;
```

definieren. Normalerweise sollte man allerdings abbrechen, wenn Strukturfehler erkannt worden sind. Im Teil 6 des Kurses wird auf Möglichkeiten zur Fehlerbehandlung eingegangen.
Anschließend werden mit der Wortfolge **OVER = ?BRANCH**, der Platz für die Sprungadresse und **DROP** kompiliert. Zum Abschluß von OF wird die Kennzahl 6 auf den Datenstapel gelegt. Nun folgt die Kompilation des Zweigprogramms. Danach muß der Datenstapel denselben Zustand haben, damit die Prüfung der Kennzahl 6 durch **ENDOF** positiv ausfällt. Von ENDOF wird außerdem der mit OF kompilierte bedingte Vorwärtssprung auf den nach ENDOF vorzunehmenden Wörterbucheintrag, das heißt die nächste Fallabfrage, gerichtet. Der Speicherplatz für die Adresse des im Falle der Übereinstimmung notwendigen unbedingten Sprungs ans Ende der Gesamtkonstruktion wird durch das in ENDOF stehende **>MARK** reserviert.
Das Wort **ENDCASE** löst die von den ENDOFs auf dem Datenstapel hinterlassenen Markierungsadressen (für die unbedingten Sprünge ans Ende der Konstruktion) auf. Durch Vergleich des Stapelzeigers mit dem in CSP abgelegten alten Wert wird geprüft, ob die richtige Anzahl von Adressen zugeordnet wurde.
Wir empfehlen Ihnen, die Kompilation und die Ausführung des folgenden, als Demonstration gedachten Wortes nachzuvollziehen:

```
: .X (n ===> )
  CASE 0 OF ." Zweig 0 " ENDOF
       1 OF ." Zweig 1 " ENDOF
       2 OF ." Zweig 2 " ENDOF
       ." Zweig x" ENDCASE ;
```

Tafel 5 Kurzbeschreibung der Forthworte

Name	Stapeleffekt	Beschreibung
Wörterbuchverwaltung		
,	(16b ===>)	16b als neuen Eintrag ans Wörterbuch anhängen
C,	(8b ===>)	8b als neuen Eintrag ans Wörterbuch anhängen
ALLOT	(n ===>)	reserviert die nächsten n Byte im Wörterbuch
HERE	(===>addr)	addr ist die nächste freie Adresse am Wörterbuchende
IMMEDIATE	(===>)	erklärt das zuletzt definierte Wort zum Vorrangwort
Definitionsworte		
CREATE	(===>)	erzeugt den Kopf für eine Variable
DOES>	(===>pfa)	Beginn des für alle Elemente der Wortklasse gleichen Laufzeitcodes (Hochcode)
CODE	(===>)	ähnlich wie DOES>, aber es folgt Assembler- oder Maschinencode
Kompilationsworte		
<MARK	(===>addr)	liefert die aktuelle Wörterbuchendeposition addr zum Stapel
<RESOLVE	(addr ===>)	benutzt die addr von <MARK zur Berechnung und Eintragung der Adresse eines Rückwärtssprunges
>MARK	(===>addr)	liefert die aktuelle Wörterbuchendeposition addr und reserviert Platz für eine Vorwärtssprungadresse
>RESOLVE	(addr ===>)	bildet aus der von >MARK gelieferten addr und der aktuellen Wörterbuchendeposition einen Vorwärtssprung und trägt diesen in den von >MARK reservierten Platz ein
COMPILE	(===>)	kompiliert in der Ausführungsphase das nachfolgende Wort
LITERAL	(Kompilation: n ===>) (Ausführung: ===>n)	kompiliert die auf dem Parameterstack liegende Zahl als Literal und liefert diese bei der Ausführung zum Stapel
STATE	(===>addr)	Variable, die während des Ausführungsmodus den Wert Null hat
[	(===>)	schaltet den Ausführungsmodus ein
]	(===>)	schaltet den Kompilationsmodus ein
[COMPILE]	(===>)	kompiliert das im Quelltext nächstfolgende Wort; auch dann, wenn es ein Vorrangwort ist

5.3.3. *Zustandssteuerung*

Grundsätzlich befindet sich das Forthsystem entweder im Modus *Kompilieren* oder im Modus *Ausführen*. Welcher Zustand gerade aktiv ist, kann der Systemvariablen **STATE** entnommen werden. Im Modus Ausführung hat die Variable STATE den Wert Null, ansonsten ist sie davon verschieden (implementationsabhängig). STATE ist für die Abfrage des Systemzustandes vorgesehen und darf vom Anwender nicht selbst verändert werden. Manchmal kommen in Programmen Rechenoperationen mit solchen Operanden vor, deren Werte schon zur Übersetzungszeit bekannt sind. In solchen Fällen wäre es günstig, die entsprechenden Operationen tatsächlich schon zur Übersetzungszeit zu erledigen, so daß im übersetzten Programm schon die fertigen Zwischenergebnisse verwendet werden. Die aufgewendete Rechenzeit spart man bei der wiederholten Abarbeitung des definierten Wortes mehrfach ein. Die Berechnungen erfordern den Zustand *Ausführung*, der während der Kompilation vorübergehend eingeschaltet werden müßte. Nach Berechnung des Wertes wird dann weiter kompiliert. Zur Umschaltung des Zustandes werden die beiden Worte

[ ]

benutzt. Die Notation läßt sich leicht merken, weil die in Klammern stehenden Ausdrücke als „Nebenrechnung" aufgefaßt werden können, die nicht kompiliert wird. Als Ergebnis der „Nebenrechnung" wird in der Regel ein Zahlenwert auf dem Datenstapel geliefert. Seine Kompilation kann mit dem Wort **LITERAL** erfolgen, das einen zur Compilationszeit auf dem Datenstapel liegenden Wert ins Wörterbuch einträgt. Beim Aufruf der Passage wird der Wert auf den Stapel zurückgebracht.

Beispielsweise kann für ein Objekt der Klasse **VEKTOR** mit dem Namen **VX** nach Abschnitt 5.2.1. der Zugriff auf die dritte Komponente anstatt durch

```
: zzz . . . VX 3 KOMPONENTE . . . ;
```

mit

```
: zzz . . . [ VX 3 KOMPONENTE ]
  LITERAL . . . ;
```

vorgenommen werden. Der komplizierter aussehende zweite Quelltext stellt die günstigere Lösung dar. Statt der 4∗16 Bit Speicherplatz im Wörterbuch werden nur 2∗16 Bit benötigt. Außerdem wird während der Laufzeit des Wortes zzz die Passage schneller abgearbeitet, weil die in Klammern stehenden Operationen bereits während der Übersetzung in den Fadencode erledigt worden sind.
Die gewöhnlichen Zahlen werden übrigens während der Kompilation in derselben Weise behandelt. Zur Ablage einer Zahl im Kompilat werden deshalb in der Regel zweimal 16 Bit benötigt, einmal für LITERAL und einmal für die Zahl selbst. Das ist ein höherer Aufwand als für Konstanten, wo nur die Adresse (16 Bit) zur CFA der Konstanten ins Wörterbuch geschrieben wird. Damit gibt es einen weiteren Grund für die Verwendung von Konstanten.

wird fortgesetzt