Programm-Strukturen#

Table of Contents

Wil Baden, auf den Sie in der englischsprachigen Literatur oft stoßen, hat in seinem Beitrag ESCAPING FORTH folgendes dargelegt: Es gibt vier Arten von Steueranweisungen :

- die Abfolge von Anweisungen,
- die Auswahl von Programmteilen,
- die Wiederholung von Anweisungen und Programmteilen,
- den Abbruch.

Die ersten drei Möglichkeiten sind zwingend notwendig und in den älteren Sprachen wie PASCAL ausschließlich vorhanden. Entsprechend steht im volksFORTH eine Anweisung für die Auswahl von Programmteilen zur Verfügung, wobei die Ausführung vom Resultat eines logischen Ausdrucks abhängig gemacht wird:

```
flag IF <Anweisungen> THEN
flag IF <Anweisungen> ELSE <Anweisungen> THEN
```

Soll dagegen im Programm ein Rücksprung erfolgen, um Anweisungen wiederho1t auszuführen, wird bei einer gegebenen Anzahl von Durchläufen diese Anweisung eingesetzt, wobei der aktuelle Index über I und J zur Verfügung steht:

```
<Grenzen> DO / ?DO <Anweisungen> LOOP
<Grenzen> DO / ?DO <Anweisungen> <Schrittweite> +LOOP
```

Wenn eine Wiederholung von Anweisungen ausgeführt werden soll, ohne daß die Anzahl der Durchläufe bekannt 1st, so ist eine Indexvariable mitzuführen oder sonstwie zum Resultat eines logischen Ausdrucks zu kommen. Die folgende Konstruktion ermöglicht eine Endlos-Schleife:

```
BEGIN <Anweisungen> REPEAT
```

Die Wiederholungsanweisungen sind insoweit symmetrisch, daß eine Anweisung so lange (while) ausgeführt wird, wie ein Ausdruck wahr ist, oder eine Anweisung wiederholt wird, bis (until) ein Ausdruck wahr wird.

```
BEGIN <Anweisungen> flag UNTIL
BEGIN <Anweisungen> flag WHILE <Anweisungen> REPEAT
```

Beide Möglichkeiten lassen sich in volksFORTH auch kombinieren, wobei auch mehrere (multiple) WHILE in einer Steueranweisung auftreten dürfen.

```
BEGIN <Anweisungen> flag WHILE <Anweisungen> flag UNTIL
```

Nun tritt in Anwendungen häufig der Fall auf, daß eine Steueranweisung verlassen werden soll, weil sich etwas ereignet hat.

Dann ist die vierte Situation, der Abbruch, gegeben, Die Programmiersprache "C" stellt dafür die Funktionen: *break*, *continue*, *return* und *exit* zur Verfügung; volksFORTH bietet hier *exit*, *leave*, *endloop*, *quit*, *abort*, *abort"* und *abort(* an. In FORTH wird EXIT dazu benutzt, um die Definition zu verlassen, in der es erscheint; LEAVE dagegen verläßt die kleinste umschließende DO...LOOP-Schleife.

Glossary#

Ab der Version 3.81.8 verfügt volksFORTH über eine zusätzliche Steueranweisung für den Compiler, die bedingte Kompilierung in der Form:

```
have <word> not .IF <action1> .ELSE <action2> .THEN
```

DieseWorte werden außerhalb von Colon-Definitionen eingesetzt und ersetzen das *\needs* früherer Versionen.

- have
- exit
- ?exit
- 0=exit
- if
- .IF
- then
- .THEN
- else
- .ELSE
- do
- ?do
- loop
- +loop
- I
- J
- leave
- endloop
- bounds
- begin
- repeat
- until
- while
- execute
- perform
- case?
- stop

Worte zur Fehlerbehandlung#

Diese arbeiten auch wie Steueranweisungen, wie die Definitionen von *ARGUMENTS* und *IS-DEPTH* zeigen: :

```
is-depth ( n ? )
  depth 1- ? abort" falsche Parameterzahl!" ;
```

IS-DEPTH überprüft den Stack auf eine gegebene Anzahl Stackelemente (depth) hin.

- abort
- 'abort
- abort"
- error"
- errorhandler
- (error
- r#
- scr
- quit
- ?pairs

Fallunterscheidung in FORTH#

Strukturierung mit IF ELSE THEN / ENDIF#

An dieser Stelle soll kurz die vielfältigen Möglichkeiten gezeigt werden, mit denen eine Fallunterscheidung in FORTH getroffen merden kann. Kennzeichnend für eine solche Programmsituation ist, daß von verschiedenen Möglichkeiten des Programmflusses genau eine ausgesucht werden soll.

Ausgehend von einer übersichtlichen Problemstellung, einem Spiel, werden die notwendigen Grundlagendefinitionen und die Entwicklung der oben beschriebenen Kontrollstruktur beschrieben.

Als Beispiel dient ein Spiel mit einfachen Regeln:

Bei diesem Trinkspiel, das nach dem Artikel "Ultimate CASE-Statement" (Vierte Dimension 2/87, Seite 40 ff) auch CRAPS genannt wird, geht es darum, einen Vorrat von gefüllten Gläsern unter den Mitspielern mit Hilfe des Würfels zu verteilen und leerzutrinken:

- Bei einer EINS wurde ein Glas aus dem Vorrat in der Tischmitte genommen und vor sich gestellt.
- Bei einer ZWEI oder einer DREI bekam der Nachbar/ die Nachbarin links ein Glas des eigenen Vorrates zugesehoben.
- Bei einer VIER oder einer FÜNF wurde dem Nachbarn/ der Nachbarin rechts ein Glas des eigenen Vorrates vorgesetzt.
- Bei einer SECHS wurden alle Gläser, die der Spieler / die Spielerin vor sich stehen hatte, leergetrunken.

Zuordnung ist also: 1=nehmen, 2/3=links, 4/5=rechts, 6=trinken und entsprechend der Augenzahl des Würfels soll eine der 6 möglichen Aktionen ausgeführt werden. Das Programm soll sich darauf beschränken, das Ergebnis dieses Würfelns einzulesen und auszuwerten. Daraufhin wird eine Meldung ausgegeben, welche der sechs Handlungen auszuführen ist.

Für ein solches Programm ist eine Zahleneingabe notwendig. Diese wurde hier mit dem Wort F83-NUMBERS realisiert:

```
: F83-number?  ( string -- d f )
   number? ?dup
   IF
     0<  IF extend THEN
     true exit
   THEN
   drop 0 0 false ;
```

```
: input#   (  string -- n )
   pad c/l 1- >expect
   pad F83-number? 2drop ;
```

Die Definition der Worter, die sechs oben genannten Aktionen symbolisch ausführen sollen, richtet sich nach den Spielregeln, die für jedes Würfelergebnis genau eine Handlung vorschreiben:

```
\ nehmen  trinken  links  rechts  schieben

: nehmen  bright ." ein Glas nehmen"          normal  2 spaces ;
: trinken bright ." alle Gläser austrinken"   normal  2 spaces ;
: links   bright ." ein Glas nach links"      normal  2 spaces ;
: rechts  bright ." ein Glas nach rechts"     normal  2 spaces ;

: schieben ;
```

SCHIEBEN ist eine Dummyprozedur, ein Füllsel, dessen Notwendigkeit sich erst sehr spät ergibt. Für den Dialog mit dern Anwender wird deflniert:

```
: Anfrage    cr ." Sollen Sie nehmen, trinken oder schieben?"
             cr ." Bitte Ihre Augenzahl und <cr> : " ;

: Glückwunsch  cr ." Viel Glück beim nächsten Wurf ..." ;
```

Das Wort *AUSWERTUNG* soll entsprechend einem Selektor genau eine von 6 möglichen Prozeduren ausführen. Also wird man prüfen, ob diese oder diese oder ... der Mogiichkeiten in Frage kommt. Hinzu kommt noch die Prüfung, ob der übergebene Parameter zwischen (between) 1 und 6 lag.

Die Def!nit!on von *BETWEEN* ist volksFORTH-gemäß recht kurz:

```
( Wert Untergrenze Obergrenze -- false   oder )
(                             -- true    wenn Untergrenze <= Wert <= Obergrenze )
: between   1+ uwithin ;

: Auswertung.1   ( wurfergebnis -- )
  dup 1 = IF  nehmen  ELSE
          dup 2 = IF links schieben ELSE
          dup 3 = IF links schieben ELSE
                  dup 4 IF rechts schieben ELSE
                  dup 5 IF rechts schieben ELSE
                        dup 6 = IF trinken THEN
                        THEN
                        THEN
                  THEN
                  THEN
          THEN
  1 6 between not IF invers ." Betrug!" normal THEN ;
```

Da eine solche Prüfung auf Gleichheit in der Programmierpraxis oft vorkommt, stellt das volks4TH dafür das Wort case? zur Verfügung. case? vergleicht die obersten beiden Stackwerte miteinander. Bei Ungleichheit bleibt der Testwert (Selektor) erhalten, so daß die Worte DUP und = dadurch ersetzt werden.

```
: Auswertung.3  ( Wurfergebnis -- )

   1 case? IF  nehmen           exit THEN
   2 case? IF  links  schieben  exit THEN
```

```
   3 case? IF  links  schieben  exit THEN
   4 case? IF  rechts schieben  exit THEN
   5 case? IF  rechts schieben  exit THEN
   6 case? IF  trinken          exit THEN

   drop    invers  ." Betrug!"  normal ;
```

Bei dieser Auswertung wird aus dem QueHtext zu wenig deutlich, daß bei ZWEI und DREI dieselbe Handlung ausgeführt wird, wie auch VIER und FÜNF die gleichen Aktionen zur Folge haben.

=OR prüft deshalb einen Testwert n2 auf Gleichheit mit einer unter einem Flag f1 liegenden Zahl n2. Das Ergebnis dieses Tests wird mit dem bereits vorliegenden Flag OR-verknüpft. Dieses neue Flag f2 und der Testwert nl werden Übergeben:

=OR Definition in Forth

```
  : =or  ( n1 f1 n2 -- n1 f2 )
    2 pick
    = or ;
```

=OR Definition in 8086 Assembler

```
   code 0or     ( n1 f1 n2 -- n1 f2 )
      A D xchg    D pop
      S W mov
        W ) A cmp
        0= ?[ -1 # D mov ]?
      next
   end-code
```

Dieses Wort bringt im Quelltext eine deutliche Verbesserung:

```
: Auswertung.4  ( Wurfergebnis -- )
  dup
  1 6 between IF
              dup 1 =           IF nehmen          THEN
              dup 2 = 3 =or     IF links  schieben THEN
              dup 4 = 5 =or     IF rechts schieben THEN
              dup 6 =           IF trinken         THEN
            ELSE
              invers ." Betrug!" normal
            THEN
   drop ;
```

Damit wurde ohne eine CASE-Anweisung eine sehr übersichtliche Steuerung des Programm-Flusses geschaffen. Die Plausibilitatsprüfung, ob die eingegeben Zahl zwischen 1 und 6 lag, ist hier an den Anfang gerückt und wird in einem einzigen ELSE-Zweig abgearbeitet.

Behandlung einer CASE ? Situation#

Strukturelles CASE#

Viele Programmiersprachen stellen eine CASE-Anweisung zur Verfügung, die wie in PASCAL mit Hilfe eines Fall-Indices eine Liste von Fall-Konstanten auswertet und eine entsprechende Anweisung ausführt. Obwohl ein solches CASE-Konstrukt ? wie oben gezeigt ? nicht notwendig ist, macht es Programme besser lesbar und liegt bei Problemstellungen wie der Auswertung eines gegebenen Index eigentlich näher. Dies ist in "Wil Baden - Ultimate CASE-Statement - VD 2/87, S.40 ff." ausführlich diskutiert worden, , wobei aber der ältere Esker-CASE (Dr. Charles Eaker -- Just in CASE

(FORTH DIM II/3)) von Dr. Charles Eaker sicherlich der bekanntere ist, der auch in der Literatur und in Quelltexten häufig Erwähnung und Verwendung findet.

Herr H. Sehnitter hat diesen Eaker-CASE für das volks4TH implementiert und dabei Veränderungen in der Struktur und Verbesserungen in der Anwendung vorgenommen.

```
\ caselist initlist >marklist >resolvlist

| variable caselist
| : initlist    ( list ? addr )
    dup @ swap off ;

| : >marklist    ( list ? )
    here over @ , swap ! ;

| : >resolvelist ( addr list? )
    BEGIN dup @
    WHILE dup dup @ dup @ rot ! >resolve
    REPEAT ! ;

\ case elsecase endcase

: CASE    caselist initlist 4 ; immediate restrict
: ELSECASE   4 ?pairs compile drop 6 ; immediate restrict
: ENDCASE    dup 4 =
               IF  drop compile drop
               ELSE 6 ?pairs
               THEN caselist >resolvelist
  ; immediate restrict

\ of endof

: OF    4 ?pairs  compile over
                  compile =
                  compile ?branch >mark
                  compile drop 5 ; immediate restrict

: ENDOF 5 ?pairs  compile branch caselist >marklist >resolve 4 ; immediate restrict
```

Diese Implementierung des Eaker-CASE stellt eine Verbesserung gegenüber dem Original dar, indem Herr Schnitter die Kontrollstruktur um ELSECASE erweitert hat. Seibstverständlich ist die neue Version vollkommen aufwärtskompatibel mit der Originalversion.

Verbesserung:

In der Originalversion der CASE-Struktur ist es nicht möglich, zwischen dem letzten ENDOF und ENDCASE einen Wert oder ein Flag auf den Stapel zu legen, da ENDCASE grundsätzlich den "Top of Stack" entfernte.

In der verbesserten Version bereinigt ELSECASE den Stapel. ELSECASE muß jedoch nicht aufgerufen werden; in diesem Fall kompiliert ENDCASE wie bisher ein DROP. Es ist jetzt möglich, zwischen den Worten ELSECASE und ENDCASE ? wie auch zwischen OF und ENDOF ? einen Wert auf den Stapel zu legen und diesen außerhalb der CASE-Kontrollstruktur zu verwenden.

Änderung:

Die Vorwärtsreferenzen werden nicht über den Stack aufgelöst, sondern über eine verkettete Liste. Die Variable **caselist** enthält die Startadresse für noch nicht bekannte Sprungadressen.

Die Schachtelungstiefe mehrerer CASE-Konstruktionen ist beliebig und wird durch **initlist** gelöst. **>marklist** füllt zur Kompilierzeit die Liste der Vorwärtsreferenzen und **>resolvelist** löst sie wieder auf.

Anwendungshinweis:

Wenn diese Definitionen außerhalb der Zusammenstellung des Arbeitssystems zugeladen werden, sollten nach dem Compilieren die Namen der mit | als headerless markierten Worte mit **clear** entfernt werden.

Das Beispiel einer Tastatuabfrage auf CTRL-Tasten zeigt (MS-DOS), wie dieses CASE-Konstrukt einzusetzen ist. Wichtig ist hierbei, daß das **OF** selbst die Gleichheit der beiden vorliegenden Werte prüft und in diesem Fall die Anweisungen zwischen **OF** und **ENDOF** ausführt.

```
: Control    bl word 1+ c@  $BF and  state @  IF [compile] Literal THEN : immediate

: Tastaurabfrage
  ." exit mot ctrl x" cr
  BEGIN key
      CASE control A OF ." action ^a " cr false ENDOF
           control B OF ." action ^b " cr false ENDOF
           control C OF ." action ^c " cr false ENDOF
           control D OF ." action ^d " cr false ENDOF
           control X OF ." exit 2"        true  ENDOF
      ELSECASE
           ." befehl unbekannt " CR false
      ENDCASE
  UNTIL ;
```

Mit dieser CASE-Anweisung läßt sich die Zuordnung der sechs Möglichkeiten zu den sechs Anweisungen ähnlich wie in PASCAL schreiben, lediglich Bereiche wie 0..255 als Fall-Konstanten sind nicht erlaubt.

```
: Auswertung.5 ( Wurfergebnis -- )
   CASE
      1 OF  nehmen          ENDOF
      2 OF  links schieben  ENDOF
      3 OF  links schieben  ENDOF
      4 OF  rechts schieben ENDOF
      5 OF  rechts schieben ENDOF
      6 OF  trinken         ENDOF
   ELSECASE
      invers ." Betrug!" normal
   ENDCASE ;
```

Das vollständige Programm kann so geschrieben werden, wobei die typische dreiteilung "Eingabe-Verarbeitung-Ausgabe" deutlich wird:

```
: craps  ( -- )
  cr Anfrage cr
     input#
     Auswertung
  cr Glückwunsch
;
```

Wil Baden hat in "Ultimate CASE Statement" ausgeführt, das eine CASE-Anweisung nur syntaktischer Zucker für ein Programm ist und letztendlich nichts weiter ist, als das Kompilieren einer verschachtelten IF...THEN-Anweisung. Eine solche Implemetierung fur das volksFORTH83 wurde von Herrn Klaus Schleisiek geschrieben:

```
\ CASE OF ENDOF ENDCASE

: CASE ( n1 -- n1 n1 ) dup ;
: OF       [compile] IF compile drop ; immediate restrict
: ENDOF    [compile] ELSE 4+ ; immediate restrict
: ENDCASE  compile drop BEGIN 3 case? WHILE >resolve REPEAT ; immediate restrict
```

Wil Badens Implementierung hält sich sehr eng an die logischen Grundlagen, wobei der Unterschied zum EAKER-CASE hauptsächlich darin besteht, daß hier jedes TRUE-Flag den Anweisungsteil zwischen OF und ENDOF ausführt; das OF nimmt keine Prüfung auf Gleichheit vor, sondern beliebige Ausdrücke können zu einem Flag führen, das dann von OF ausgewertet wird. So ist das Auswerten des Fall Index variabler als beim EAKER-CASE:

```
: Auswertung.6  ( Würfenwurf -- )
  dup
     1 6 between not
         IF invers ." Betrug!" normal drop exit THEN
  CASE 1 = OF nehmen             ENDOF
  CASE 6 = OF trinken            ENDOF
  CASE 4 < OF links schieben     ENDOF
  CASE 3 > OF rechts schieben    ENDOF
  ENDCASE ;
```

Hier bei dieser Konstruktion steht die Plausibilltätsprüfung ganz vorn, um den ELSECASE-Fall durch ein EXIT aus dem Wort zu erreichen. Wird keines der Worte aus der Auswahl-Liste ausgeführt, läßt sich mit BREAK eine andere Lösung erreichen:

```
: BREAK   compile exit
          [compile] THEN ; immediate restrict
```

Dadurch, daß BREAK ein EXIT aus dem Wort darstellt, wird ein (implizites) ELSECASE erreichen, indem man die Anweisungen der Auswahl-Liste mit OF und BREAK klammert und die Anweisungen für den ELSE-Fall nach ENDCASE aufführt:

```
: Auswertung.7 ( Würfelzahl -- )
  CASE 1 =           OF nehmen             BREAK
  CASE 2 = 3 =or     OF links schieben     BREAK
  CASE 4 = 5 =or     OF rechts schieben    BREAK
  CASE 6 =           OF trinken            BREAK
  ENDCASE  invers ." Betrug!" normal ;
```

Positionelles CASE#

Eine ganz anderen Lösungsansatz bietet ein positioneller CASE Konstrukt, bei dem die Fallunterscheidung durch den Fall-Index tabellarisch vorgenommen wird.

Bei den bisherigen Lösungen wurden immer eine Reihe von Vergleichen zwischen einem Fall-Index und einer Liste von Fall-Konstanten vorgenommen; nun wird der Fall-Index selbst benutzt, die gewünschte Prozedur auszuwählen. Die Verwendung des Fall-Index als Selektor bringt auch Vorteile in der Laufzeit, da die Vergleiche entfallen.

Wenn FORTH-Worte in Tabellen abgelegt werden sollen, stellt sich das Problem, daß ein FORTH-Wort bei seinem Aufruf normalerweise die einkompilierten Worte ausführt. Bei einer Tabelle ist das nicht erwünscht; dort ist sinnvollerweise gefordert, daß die Startadresse der Tabelle übergeben wird, um den Fall-Index als Offset in diese Tabelle zu nutzen.

Dies läßt sich in volksFORTH entweder auf die traditionelle Weise mit] und [oder dem volksFORTH-spezifischen **Create:** lösen:

```
Create Glas
    ]  nehmen  links schieben
       rechts schieben  trinken  [

Create: Glas
        nehmen
        links schieben
        rechts schieben
        trinken ;
```

Diese Tabelle Glas macht auch deutlich, welche Punktion das Dummy-Wort **schieben** außer einer besseren Lesbarkeit noch hat: Es löst die Schwierigkeit, daß 6 möglichen Wurfergebnissen nur 4 mögliche Aktionen gegenüberstehen.

Die Art und Weise des Zugriffs in BEWEGEN entspricht dem Zugriff auf eine Zahl in einem eindimensionalen Feld, einem Vektor:

```
: bewegen  ( addr n -- cfa )
  2* + perform ;

: richtig ( n -- 0 <= n <= 3 )
  swap
  1 max  6 min
  3 case? IF 2 1- exit THEN
  5 case? IF 4 1- exit THEN
  1- ;
```

Dieses Wort RICHTIG läßt zwar Werte kleiner als 1 und größer als 6 zu, justiert sie aber auf den Bereich zwischen 1 und 6. Auch hier müßte eine Möglichkeit geschaffen werden, ein Wurfergebnis außerhalb der 6 Möglichkeiten als Betrugsversuch zurückzuweisen!

Die Verbindung von Tabelle und Zugriffsprozedur wird von dem Wort :Does> vorgenommen:

```
\ :Does> für  Create <name> :Does> <action> ;  ks 25 aug 88

| : (does>  here >r [compile] Does> ;
  : :DOES>  last @ 0= Abort" without reference"
    (does>  corrent @  context !  hide 0 ] ;
```

Dieses Wort **:DOES>** weist dem letzten über **Create** definierten Wort einen Laufzeit-Teil zu. Dieses Wort wurde von Klaus Schleisiek programmiert auch hier gilt der Hinweis, nach dem Kompilieren das mit | als headerless deklarierte Wort durch **clear** zu löschen.

```
Create: Auswertung.8
          nehmen
            links schieben
            rechts schieben
          trinken ;

:DOES> richtig bewegen ;
```

Ohne :DOES> sind die Tabelle und die Zugriffsprozeduren voneinander unabhängige Worte:

```
: CRAPS1
  cr Anfrage cr
    input#
    Glas richtig bewegen
  cr  Glückwunsch ;
```

Entschließt man sich dagegen, sowohl Tabelle als auch Zugriffsprozedur in einem Wort zu definieren, so ergibt sich das gewohnte Erscheinungsbild:

```
: CRAPS
  cr Anfrage cr
     input#
     Auswertung
  cr  Glückwunsch ;
```

Bei häufigerem Einsatz solcher Tabellen bietet sich der Einsatz von "positional CASE defining words" an. Auch hier wiederum zuerst die volks4TH-gemäße Lösung, danach die traditionelle Variante:

```
: Case:  ( -- )
   Create: Does>  ( pfa -- ) swap 2* + perform ;

\ Alternative Definition für CASE:
: Case:
   : Does>  ( pfa -- ) swap 2* + perform ;
```

Eine sehr elegante Möglichkeit, die Fehlerbehandlung im Falle eines unglaubwürdigen Fall-Indexes zu handhaben, bietet das Wort **Associative:**. Dieses Wort **Associative:** durchsucht eine Tabelle nach einer Übereinstimmung zwischen einem Zahlenwert auf dem Stack und den Zahlenwerten in der Tabelle und liefert den Index der gefundenen Zahl (match) zurück. Im Falle eines Mißerfolgs (mismatch) wird der größtmögliche Index +1 (out of range = maxIndex + 1) übergeben:

```
: Associative:   ( n -- )
  Constant  Does>   ( n -- index )
  dup @ -rot
  dup @ 0
  DO 2+ 2dup @ =
     IF 2drop drop I 0 0  LEAVE THEN
  LOOP 2drop ;

6 Associative:  Auswerten

           1,
         2 , 3 ,
         4 , 5 ,
           6 ,

      Case: Handlen     \ besteht aus

            nehmen
      links       links
      rechts     rechts
           trinken
    schimpfen ;
```

Statt der Primitivabsicherung über MIN und MAX wird eine "out of range" Fehlerbehandlung namens **schimpfen** an der Tabellenposition maxIndex +1 durchgeführt.

Einsatzmöglichkeiten#

Dieser letzte Teil der Ausführungen über die Möglichkeiten, eine CASE-Situation zu handhaben, greift Anregungen aus der Literatur (E. Floegel, FORTH Handbuch (S. 109) und W. Waigaard, Menus in FORTH, Elektroniker, 9/88 (S.109 ff.)) auf.

Dazu werden zwei Worte definiert:

- CLS - löscht den gesamten Bildschirm und
- CELLS - macht die Berechnung des Tabellenzugriffs deutlicher

```
: cls    full page ;
: cells  2* ;
```

Das Inhaltliche und die tabellarische Struktur bleiben unverändert, lediglich die Behandlung einer "out of range" Situation wird diesmal mit **min** und **max** und zweimaligem Eintragen der Fehler-Routine **schimpfen** verwirklicht.

```
Create: Handlung
          schimpfen   nehmen links links
                      rechts recht trinken  schimpfen ;

\ Die Ausführung einer Liste nach Floegel 7/86
: auswählen   ( addr n -- *cfa )  2 arguments
   swap 0 max          \ out of range MIN
        7 min          \ out of range MAX
        cells + ;

: auswerten  ( n -- ) 1 arguments
   Handlung  auswählen  perform ;

: .all ( -- )
   8 0 DO cr I dup . auswerten  2 spaces LOOP ;
```

AUSWÄHLEN übergibt bei gegebenem Vektor und gegebenem Index einen Zeiger auf die "code field address" (cfa) des entsprechenden Wortes. **AUSWERTEN** führt das so ausgewählte Wort aus und **.ALL** diente nur zur Kontrolle. Solch ein Wort, das angelegte Datenstrukturen auf dem Bildschirm darstellt, sollte in der Entwicklungsphase eines Programmes immer dabei sein.

Eine weitere M5glichkeit, Werte in einen Vektor einzutragen, hat Herr Floegel in seinem Buch "FORTH Handbuch" dargestellt:

```
Create Tabelle  8 cells allot
       :DOES> ( i -- addr ) swap cells + ;

' schimpfen   0 Tabelle !
' nehmen      1 Tabelle !
' links  dup  2 Tabelle !
              3 Tabelle !
' rechts dup  4 Tabelle !
              5 Tabelle !
' trinken     6 Tabelle !
' schimpfen   7 Tabelle !

: auswerten  ( i -- )  0 max  7 min  Tabelle perform ;

: .action    ( i -- )
   Tabelle @ >name bright .name normal ;

: .Tabelle ( -- )  cr  8 0 DO cr I .action LOOP ;
```

Hier besteht mit **.ACTION** und **.TABELLE** die Möglichkeit, sich den Vektor darstellen zu lassen. In ähnlicher Weise werden auch im Kommandozeilen-Editor CED die neuen Aktionen in die Eingabe-Vektoren eingetragen.

Eine geringfügige Modifikation aus "W. Waigaard, Menus in FORTH" soll die VerknQpfung eines Vektors von Worten und einer Menü-Option zeigen:

```
Create function
    ] noop noop noop noop
      noop noop noop noop [
    :DOES> ( i -- addr )
      swap  0 max 7 min  cells + ;
```

function ist ein execution vector, der mit **NOOP** vorbesetzt ist. Zur Laufzeit liefert er die Adresse des indizierten Elementes zurück.

```
: .action ( i addr -- )
  @ >name  bright .name normal ;
```

.WORD gibt den Namen eines Wortes aus, dessen CFA in eine Adresse eingetragen wurde.

```
: option   ( i -- )
  R>
  dup 2+ >R                  \ i *w.addr
  @                          \ i w.addr
  stash swap  function !     \ i w.addr i addr
  function .action ;         \ i addr
```

option holt die Adresse des auf **option** folgenden Wortes. Das Wort soll nicht ausgeführt werden, sondern das nachfolgende. Nur der Pointer auf das Wort soll ausgewertet werden. Nach dem übergebenen Index wird der Pointer in **function** eingetragen. Der Name des so eingetragenen Wortes wird angezeigt!

```
\ Menü      jrp 06feb89
: Menue
   0 option schimpfen
   1 option nehmen
   2 option links
   3 option links
   4 option rechts
   5 option rechts
   6 option trinken
   7 option schimpfen ;
```

Wenn das Wort **MENUE** aufgerufen wird, werden nicht nur die Optionen in die Tabelle eingetragen, sondern auch namentlich auf dem Bildschirm dargestellt. Diese Technik bietet sich für eine Menüzeile an fester Bildschirmposition an, ähnlich der Statuszeile des volksFORTH. Zum Ändern solcher Menüpunkte bieten sich die Funktionstasten an:

MS-DOS

```
: fkey ( -- )
  key &58 + abs  function perform ;
```

FKEY liefert beim Druck einer Funktionstaste einen Wert von -59 bis -68 zurück. Dieser wird für 10 Funktionstasten in den Bereich von -1 bis -10 skaliert und der Absolutwert gebildet.

Rekursion#

Bevor die Technik der Rekursion für das volksFORTH dargestellt wird, soll ein anderes Wort **.LASTNAME** zeigen, daß das Wort **LAST** mit dem in der Literatur oft anzutreffenden **LATEST** identisch ist: Beide Worte liefern die "name field address" (nfa) des zuletzt definierten Wortes im CURRENT-Vokabular. Das Wort **LAST'** dagegen liefert die "code field address" (cfa) des zuletzt definierten Wortes.

```
: .lastname  last @ .name ;
```

Die Rekursion ist eine Technik, bei der ein Wort sich immer wieder selbst aufruft. Eines der bekannten Beispiele dafür ist die Berechnung der Fakultät einer positiven ganzen Zahl. Hierbei ergibt sich n! aus dem Produkt aller ihrer Vorgänger.

Im volksFORTH ist der Selbstaufruf eines Wortes durch **RECURSIVE** gekennzeichnet, so daß sich ein Programm zur Fakultätsberechnung wie folgt präsentiert:

```
: fakultät  ( +n -- n! )
   recursive
   dup 0< IF  drop ." keine negativen Zahlen! " exit
          THEN
   ?dup 0= IF 1   \ spezialfall 0
           ELSE dup 1- fakultät *
           THEN ;

 cr  4 fakultät .
 cr  5 fakultät .
 cr  6 fakultät .
```

Allerdings findet sich ? vor allem in der fig-FORTH-Literatur ? ein Wort **MYSELF**, das mit dem in FORTH83-Umgebungen anzutreffenden **RECURSE** identisch ist. Da auch diese Konstruktion, bei der MYSELF/RECURSE als Platshalter für den Wortnamen dienen, gerne eingesetzt wird, werden die möglichen Definitionen und eine weitere Form von FAKULTÄT gezeigt:

```
: myself  last @ name> , ; immediate

: myself  last' , ; immediate

: recurse  [compile] myself ; immediate

' myself Alias recurse immediate

: fakultät ( +n -- n! )
  dup 0<  IF ." keine negativen Zahlen erlaubt!"
          ELSE  ?dup 0= IF 1
                        ELSE dup 1- myself *
                        THEN
          THEN ;
```

Bei der Verwendung von **RECURSE** wird lediglich **MYSELF** dadurch ersetzt:

```
...
          ELSE  ?dup 0= IF 1
                        ELSE dup 1- recurse *
                        THEN
          THEN ;
...
```