TO

Ein Mechanismus mit vielen Möglichkeiten in Forth

Albert Nijhof
(übersetzt von Fred Behringer)
2002

1. Einleitung

Forth ist eine Low-Level- und gleichzeitig eine High-Level-Programmiersprache. Datenworte legen eine Adresse auf den Stack, wohin man dann anschließend etwas schreibt oder woraus man etwas ausliest. Bei Low-Level-Code gibt einem diese Arbeitweise alle Freiheiten, die man sich wünschen kann. High-Level-Code stellt andere Anforderungen. Die Adressen der Daten sind dann von geringerer Bedeutung, es geht um die Daten selbst. Adressen auf dem Stack sind dann lediglich ein vom System aufgezwungener Zwischenschritt und vor allen Dingen auch eine mögliche Fehlerquelle. Zudem machen die vielen Lese- und Schreibworte den Code fehleranfälliger und gleichzeitig schlechter lesbar.

Mit der Verwendung von VALUEs in Verbindung mit TO wurde ein kleiner Schritt in Richtung auf eine Lösung hin eingeleitet. VALUEs sind in ANS-Forth jedoch nicht mehr als eine Randerscheinung (lediglich den Locals ist TO auch bekannt), und der Standard bietet keine wirksame Methode, mehr Datenworte auf TO vorzubereiten oder neue Vorsetzworte ("Präfixe") zu definieren. Vor allem Strings flehen nach einer Möglichkeit der High-Level-Behandlung, denn der interaktive Umgang mit MOVE in Verbindung mit Adressen auf dem Stack läßt das wünschenswert erscheinen.

Die TO-Implementation, die ich Euch beschreiben möchte, verwende ich bereits seit 1990 (JinForth für den Atari ST). Die diesem Konzept zugrunde liegende Idee ist recht einfach und auf der Hand liegend. Sie liegt so sehr auf der Hand, daß sicher auch schon andere auf denselben Gedanken gekommen sind. Mir ist jedoch noch nichts darüber zu Ohren gekommen. Ich kann nur hoffen, daß auch für Euch noch etwas Neues dabei ist.

"Mein" TO ist fehlersicher und hat den "Nebeneffekt", daß der Programmierer auf ganz normale Forth-Art, ohne Patchen oder spezielle Syntax, für jede Anwendung unbeschränkt allerlei Präfixe für allerlei Worte (im Prinzip für alle Worte) definieren kann.

Die Aufgabe von störanfälligen STORE-Worten, wie `! 2! C! +! MOVE ON OFF ...` , wird von den stärker fehlersicheren Vorsetzworten ("Präfixen") übernommen, die natürlich sowohl innerhalb als auch außerhalb der Definitionen funktionieren. Leseworte vom Typ `@` werden überflüssig, da das Datenwort jetzt, da es sich um die Adresse nicht mehr zu kümmern braucht, die Leseaktion selbst ausführen kann.

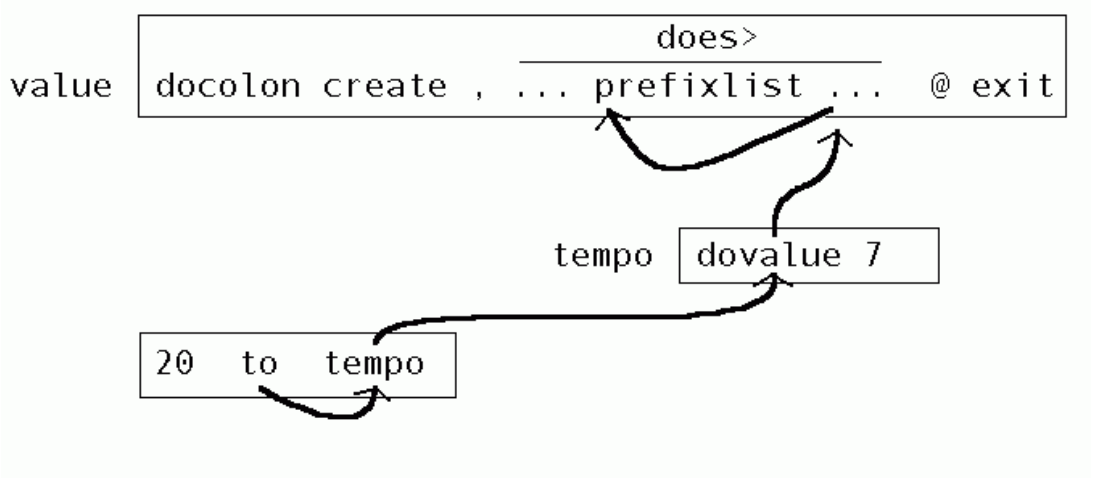
Allerdings muß der TO-Mechanismus vom Forth-Erbauer vorbereitet werden, ein kleiner, aber fundamentaler Eingriff: Einleitend zu DOES> und ;CODE-Routine wird jedesmal eine eindeutig bestimmte Wortliste compiliert. An sich hat das auf den Arbeitsablauf von Forth keinerlei Einfuß. Das System wird halt nur um einige zig Zellen größer - im allgemeinen (es scheint ein Forth zu geben, das für eine neue Wortliste 64 Kilobytes verlangt).

2. Der Mechanismus

Wir benötigen die folgenden vier Zutaten:

1. Ein Forth mit einem angepaßten DOES>
2. Das Wort PREFIXLIST
3. Das Wort PREFIX
4. Einsicht in den Mechanismus

```
: VALUE create , does> @ ;
7 value TEMPO
20 to tempo
```



1. Unmittelbar vor einer jeden Does-Routine compiliert DOES> eine eindeutig bestimmte Wortliste, die PREFIXLIST, in welcher die internen Aktionen definiert werden.

2. Ein Wort zum Auffinden der Präfixliste (in einem indirekt gefädelten Forth):
   ```
   : PREFIXLIST ( xt -- wid ) dup @ cell- @ swap ?doer ;
   ```
   ?DOER sieht nach, ob die CFA des Datenworts auf ein DOES> oder auf eine ;CODE-Routine zeigt.
   ```
   ' tempo prefixlist ( -- wid )
   ```

3. Mit PREFIX werden globale Präfixe definiert:
   ```
   : PREFIX ( "globaler name" "interner name" -- )
   prefix TO /to
   ```

4. TO sieht bei TEMPO nach, findet die Präfixliste und sucht darin nach /TO.

3. Anwendungen

Beispiel A ist trivial und kaum sinnvoll, gibt aber einen sehr guten Einblick in die Funktionsweise. Ich definiere Vorsetzworte ("Präfixe") für Variablen vom Typ BASE (Uservariablen). Man kann diese dann über GET und SET ansprechen, aber auch noch über `@` und `!` . Die beiden Methoden beißen sich nicht. Ist die interne Aktion immediate? Dann wird sie vom globalen Vorsetzwort ausgeführt. Das Datenwort-Token liegt dabei auf dem Stack.

Beispiel A

```
' base  prefixlist set-current

: /SET ( dataword-xt -- )  compile,  postpone ! ; immediate
: /GET ( dataword-xt -- )  compile,  postpone @ ; immediate

forth definitions
prefix SET /set
prefix GET /get
```

`( get base` wird `base` `@` `)`

```
decimal
: HEX 16 set base ;
hex get base decimal . [rtn] 16  ok
```

Man sieht, daß nur das Compilezeit-Verhalten von /GET und /SET codiert wird. GET und SET arbeiten jedoch auch interaktiv. Und wie das? Dafür sorgt FLYER (mehr darüber später).

Beispiel B

```
: VALUE ( x "name" -- ) create , does> @ ;
prefix TO /to

0 value TEST
' test prefixlist set-current
forget test

: /TO ( dataword-xt -- ) >body postpone literal
                         postpone ! ; immediate
forth definitions
```

`( to test` wird `literal test-body` `!` `)`

Beispiel C

In Beispiel C ist die interne Aktion nicht immediate: Das globale Vorsetzwort compiliert die interne Aktion und setzt die BODY-Adresse des Datenworts dahinter (Inline-Adresse).

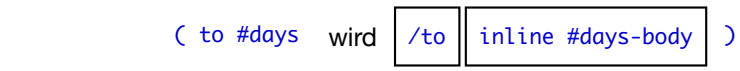
```
7 value #DAYS

' #days  prefixlist set-current

: /TO  ( x "inlinea" -- )  inline-address ! ;
: /+TO ( x "inlinea" -- )  inline-address +! ;

forth definitions

prefix +TO /+to
prefix EXTRA /+to
```




```
1 +to #days     #days . [rtn] 8  ok
12 extra #days  #days . [rtn] 20  ok

prefix NACH /to
0 nach #days    #days . [rtn] 0  ok
```

4. Does> anpassen

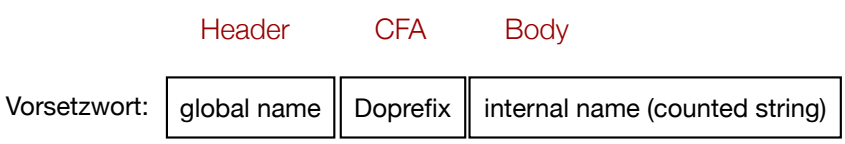
Jedes DOES> und ;CODE compiliert eine PREFIXLIST in die Zelle unmittelbar vor der Doer-Routine. Dafür müssen sie angepaßt werden. Die Abänderungen (Hinzufügungen) im Code sind unterstrichen:

```
: (;CODE) r> n + youngest ! ;
: DOES> postpone (;code) wordlist ,
        does-intro, ; immediate
: ;CODE postpone (;code) wordlist ,
        postpone [ assembler ; immediate
```

Das n in n + entspricht der Anzahl von Bytes, die durch wordlist , compiliert werden. Das Komma nach wordlist ist wahrscheinlich überflüssig, da WORDLIST selbst in den meisten Forth-Systemen die "wid" schon compiliert.

5. Prefix

Mit PREFIX ("global name" "internal name" --) werden Vorsetzworte compiliert.



- "global name" ist der Name des neuen Vorsetzwortes.
- "internal name" wird als Counted String im Body des Vorsetzwortes abgelegt.
- Die Adresse von DOPREFIX (DOES-Teil von PREFIX) kommt in die CFA des Vorsetzwortes. Doprefix geht zur Präfixliste des Datenwortes und sucht nach "internal name".

Die Funktionsweise von DOPREFIX:

1. Geh zum Datenwort, finde dessen Präfixliste und suche darin nach der internen Aktion. Die Token des Datenwortes und der internen Aktion liegen jetzt auf dem Stack.
2. Verwende FLYER (siehe folgenden Abschnitt), wodurch dann das interaktive Verhalten nicht mehr codiert zu werden braucht.
3. Ist die interne Aktion Immediate?
    - NEIN: Leite die Standardbehandlung ein: COMPILE, >BODY , d.h., compiliere die interne Aktion und danach (inline) die Body-Adresse des Datenwortes.
    - JA: Führe sie aus.

Der Code: (mit Kommentar entsprechend dem Beispiel TO TEMPO)

```
: PREFIX ( "name1" "name2" -- )
create immediate
bl parse
string, align
does> ( DOPREFIX )            ( TO-body )
' swap                        ( TEMPO-xt TO-body )
over prefixlist               ( TEMPO-xt TO-body Wid )
>r count r>                   ( TEMPO-xt "/TO" Wid )
search-wordlist               ( TEMPO-xt /TO-xt? -1,1,0 )
dup
if flyer                      ( TEMPO-xt /TO-xt )
   0<                         \ Not Immediate?
   if compile, >body , exit   \ Standardaktion
   then execute exit          \ Immediate
then -32 throw ;              \ /TO not found.
```

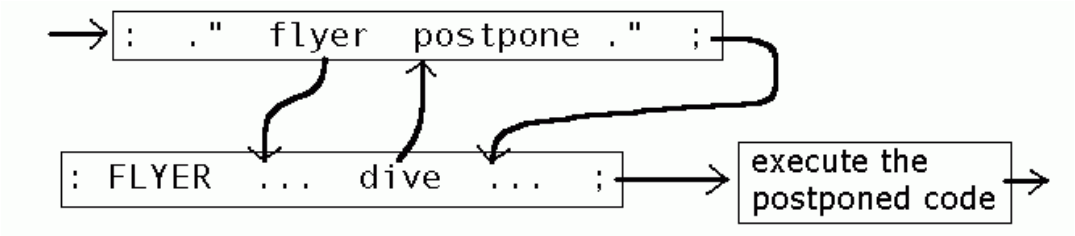
Eine immediate interne Aktion gibt uns große Flexibilität. Eine nicht immediate interne Aktion ist starr, ermöglicht aber oft eine effizientere Codierung in Assembler mit Inline-Adressen.

6. Flyer

In State-Smart-Worten sorgt FLYER dafür, daß man nur die Compilezeit-Aktion zu programmieren braucht, da das interaktive Verhalten damit auch schon bestimmt ist. Mit FLYER kann man also State-Smart-Worte auf einheitliche Weise behandeln.

```
: ." flyer postpone ." ; immediate
```

Angenommen, das urspüngliche ." funktioniert nur innerhalb von Definitionen. Dann funktioniert das neue ." mit dem darin enthaltenen FLYER interaktiv und in Definitionen.




```
: DIVE ( -- ) r> r> 2>r ; \ IP <--> Return address.

: FLYER ( -- )
state @ if exit then
here r> 2>r
] dive               \ Execute rest of caller and return.
postpone exit
postpone [
r@ here - allot ;    \ Execute the newly compiled code.
```

Dieser vereinfachte "Mini-FLYER" genügt für DOPREFIX. Das FLYER, das ich selbst verwende, befördert HERE während des Compilierens in einen Ringpuffer, worin dem zu compilierenden Code und den Daten ein längeres Leben beschert bleibt. Das braucht man für S" und beispielsweise für Locals. Man vergleiche den gleich folgenden Code.

```
.... value FLYBUF
256  value FLYBUFLEN
0    value THERE
0    value FLYING? \ Flag

: HERE/THERE ( -- ) there
flying? 0= dup to flying?
if flybuflen over flybuf -
   u<         \ Overflow?
   if drop flybuf then
then here dup to there
- allot ;
```

```
: DIVE ( -- ) r> r> 2>r ;

: FLYER  ( -- ) state @
if exit then flying? here/there
if abort" Flyer nesting" then
here r> 2>r ] !compilersecurity
dive
postpone exit postpone [
here/there ?compilersecurity ;
```

THERE is the alternative HERE in the circular flyer buffer.
Keep 264 bytes free for buffer overflow.

7. Anwendung auf Strings

```
: STRING ( maxlen "name" -- )
create dup c,         \ maxlen
0 c,                  \ actual length
allot align           \ for the string itself
does> char+ count ;



10 string X
' x prefixlist set-current
forget x

: /TO ( a n "inlinea" -- )
inline-address count rot umin swap 2dup c! char+ swap move ;
code /+TO ( a n "inlinea" -- ) "ass" END-CODE
code /INCR  (ch "inlinea" -- ) "ass" END-CODE

forth definitions
prefix INCR /incr
```

Diese internen Aktionen kann man zweckmäßiger und wahrscheinlich auch leichter in Assembler codieren. Je nach Wunsch können sie auch völlig fehlersicher gemacht werden.

```
16 string M
s" ANS"     to m   m type [rtn] ANS ok
char -    incr m   m type [rtn] ANS- ok
s" Forth"  +to m   m type [rtn] ANS-Forth ok
m        extra m   m type [rtn] ANS-ForthANS-For ok

: SUBSTRING ( a n pos len -- a n )
>r over umin /string r> umin ;

m 4 5 substring type [rtn] Forth ok
here 0 to m   m type [rtn] ok
```

Eine Anwendung auf andere Datentypen und mit anderen Vorsetzworten überlasse ich Eurer Phantasie.