TO

Een mechaniekje met mogelijkheden in forth (2002)

Albert Nijhof

1. Inleiding

Forth is een low-level en tegelijkertijd een high-level programmeertaal. Datawoorden zetten een adres op stack waar je vervolgens iets naar toe schrijft of iets uit leest. Die werkwijze geeft je bij low-level code alle vrijheid die je je maar kunt wensen. Bij high-level code liggen de eisen anders. Dan zijn data-adressen minder van belang, het gaat om de data zelf. Adressen op stack zijn dan slechts een door het systeem opgedrongen tussenstap en vooral ook een potentiële bron voor fouten. Bovendien maken de vele lees- en schrijf-woordjes de code gevoeliger voor fouten en tegelijkertijd minder leesbaar.

Het gebruik van Value's met TO is een kleine stap in de richting van een oplossing. Maar Value's zijn in ANS-FORTH niet meer dan een incident (alleen de locals kennen ook TO) en de standaard biedt geen productieve methode om meer datawoorden geschikt te maken voor TO of om nieuwe voorzetsels te definiëren. Met name strings smeken om de mogelijkheid van een high-level benadering, want het interactief gebruik van MOVE in combinatie met adressen op stack vráágt om moeilijkheden.

De TO-implementatie die ik u ga beschrijven gebruik ik al sinds 1990 (JinForth voor Atari ST). Het idee dat er aan ten grondslag ligt is eenvoudig en zo voor de hand liggend, dat ik verwacht dat meer mensen op dezelfde gedachte zullen zijn gekomen, maar dat is mij niet bekend. Ik hoop dat er voor u nog iets nieuws aan is.

Deze TO is veilig en heeft als "neveneffect" dat de programmeur op een normale Forthmanier, zonder patchen en zonder speciale syntax, voor een applicatie onbeperkt allerlei voorzetsels kan definiëren voor allerlei woorden (in principe voor alle woorden).

De taak van kwetsbare store-woordjes zoals `! 2! C! +! MOVE ON OFF ...` wordt overgenomen door de veiliger voorzetsels die natuurlijk zowel binnen als buiten definities functioneren. Leeswoordjes van het type `@` worden overbodig want het datawoord kan zelf de lees-actie uitvoeren nu hij niet meer voor het adres hoeft te zorgen.

Helaas moet het TO-mechanisme wel door de Forth-bouwer voorbereid zijn, een kleine maar fundamentele ingreep: voorafgaand aan elke DOES> en ;CODE-routine wordt een unieke wordlist gecompileerd. Dat heeft op zichzelf geen enkele invloed op de werking van Forth. Hij wordt alleen enkele tientallen cells groter. Normaal

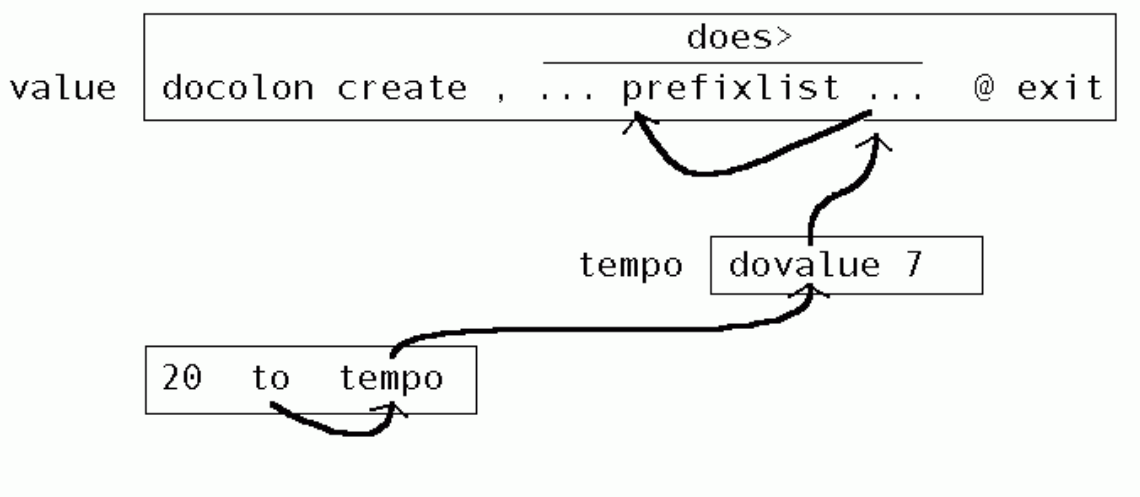
gesproken dan (er schijnt een Forth te bestaan die 64 Kilobytes nodig heeft voor een nieuwe wordlist).

2. Het mechaniekje

Deze vier ingrediënten zijn nodig:

1. Een Forth met aangepaste DOES>
2. Het woord PREFIXLIST
3. Het woord PREFIX
4. Inzicht in het mechaniekje

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



1. **Vlak voor iedere Does-routine compileert DOES> een unieke wordlist, de PREFIXLIST, waarin de interne acties gedefinieerd worden.**

2. Een woord om die prefixlist te vinden (in een indirect bedrade Forth):

   ```
   : PREFIXLIST ( xt -- wid ) dup @ cell- @ swap ?doer ;
   ```

   ?DOER controleert of het CFA van het datawoord wel naar een DOES> of ;CODE-routine wijst.

   ```
   ' tempo prefixlist ( -- wid )
   ```

3. Met PREFIX definieer je globale voorzetsels:

   ```
   : PREFIX ( 'globale naame' 'interne naam' -- )

   prefix TO /to
   ```

4. TO tickt TEMPO, vindt de prefixlist en zoekt daarin naar /TO.

3. Toepassingen

Voorbeeld A is triviaal en nauwelijks zinvol, maar laat wel heel goed zien hoe het functioneert. Ik definieer voorzetsels voor variabelen van het type BASE (uservariable) Het gevolg is dat je ze kunt aanspreken met `GET` en `SET` maar ook nog met `@` en `!` want de twee methodes bijten elkaar niet.
Is de interne actie immediate? Dan wordt hij door het globale voorzetsel geëxecuteerd. Het datawoord-token staat daarbij op stack.

Voorbeeld 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` wordt `base` `@` `)`

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

Je ziet dat alleen het compile time gedrag van /GET en /SET gecodeerd wordt. Toch werken GET en SET ook interactief. Hoe dat kan? Daar zorgt FLYER voor (waarover later).

Voorbeeld 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` wordt `literal test-body` `!` `)`

Voorbeeld C

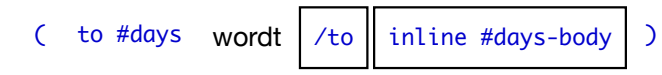
In voorbeeld C is de interne actie niet immediate: het globale voorzetsel compileert de interne actie en zet het adres van de datawoord-body erachter (inline adres).

```
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> aanpassen

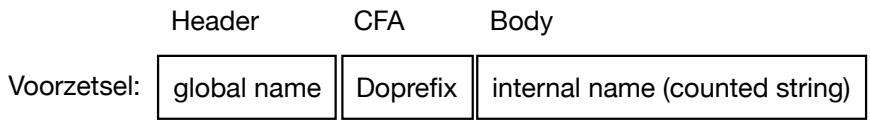
Door iedere DOES> en ;CODE wordt een PREFIXLIST gecompileerd in de cel vlak voor de Doer-routine. Daarvoor moeten ze aangepast worden. De veranderingen (toevoegingen) in de code zijn onderstreept:

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

De `n` in `n +` is gelijk aan het aantal bytes dat door `wordlist ,` gecompileerd wordt.

5. Doprefix

Met `PREFIX ( 'global name' 'internal name' -- )` definieer je voorzetsels.



'global name' is de naam van het nieuwe voorzetsel.
'internal name' wordt als counted string opgeslagen in de voorzetsel-body.
Het adres van DOPREFIX (DOES-deel van PREFIX) komt in het voorzetsel-CFA.
Doprefix gaat in de prefixlist van het datawoord op zoek naar 'internal name'.

Het werk van DOPREFIX:

1. Tick het datawoord, vind zijn Prefixlist en zoek daarin de interne actie. De tokens van datawoord en interne actie staan nu op stack.
2. Gebruik FLYER (zie het volgende hoofdstukje), waardoor je het interactieve gedrag niet hoeft te coderen.
3. Is de interne actie Immediate?
    - NEE: doe de standaardhandeling: `COMPILE, >BODY ,` d.w.z. compileer de interne actie en daarachter (inline) het adres van de datawoord-body.
    - JA: executeer hem.

De code:
(met commentaar aan de hand van het voorbeeld `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   \ Standaardaction
   then execute exit          \ Immediate
then -32 throw ;              \ /TO not found.
```

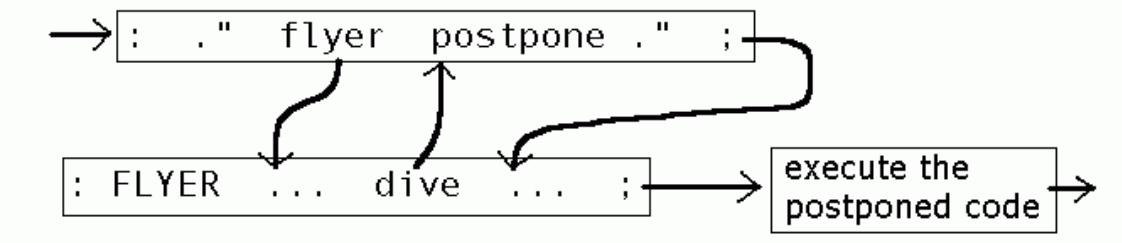
Een immediate interne actie geeft je grote flexibiliteit. Een niet immediate interne actie is star, maar maakt vaak een efficiënte codering in assembler mogelijk met een inline adres.

6. Flyer

In state smart woorden zorgt FLYER ervoor dat je alleen maar de compile time actie hoeft te coderen, want hun interactieve gedrag is daarmee ook al bepaald. Met FLYER kun je dus state smart woorden op uniforme wijze afhandelen.

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

Stel dat de oorspronkelijke `."` alleen binnen definities werkt. Dan werkt de nieuwe `."` met FLYER erin interactief en in definities.




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

: FLYER ( -- )
state @ if exit then \ No action while compiling.
here r> 2>r
] dive               \ Execute rest of caller and return.
postpone exit
postpone [
r@ here - allot ;    \ Execute the newly compiled code.
```

Deze vereenvoudigde FLYER voldoet voor DOPREFIX. De FLYER die ik zelf gebruik verplaatst HERE tijdens het compileren naar een circulaire buffer waarin de te compileren code en data een iets langer leven beschoren is. Dat is nodig voor `S"` en bijvoorbeeld voor locals. Zie de code hieronder.

```
.... value FLYBUF
256  value FLYBUFLEN
\ Keep 264 bytes free
\ for buffer overflow.
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.
```

7. Stringtoepassing

Een laatste voorbeeld

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

forth definitions
prefix INCR /incr
```

Je kunt deze interne acties doelmatiger en waarschijnlijk ook gemakkelijker in assembler coderen. Ze kunnen desgewenst volkomen veilig gemaakt worden.

```
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
```

Toepassingen met andere datatypes en andere voorzetsels laat ik aan uw fantasie over.