Albert Nijhof

De Programmeertaal FORTH

(ANS FORTH)

Cursus
Systematisch over icht
Losse artikelen
Index

Okt. 1992 (1)
Dec. 2001 (2)

Voorwoord

Arnhem, 18.11.92

"Forth is de computertaal die het best geheim gehouden is." (Dick Pountain)

In een tijd, waarin men ons wil doen geloven dat het belang en de waarde van iets verband houden met de hoeveelheid lawaai die het maakt, krijgt deze uitspraak een voor de hand liggende betekenis.

Toch twijfel ik eraan of Forth onbekend is. Misschien kun je beter zeggen dat Forth ongekend is. Het is een taal die in veel opzichten afwijkt van de gangbare computertalen. Met een terminologie die gebaseerd is op de "gekende" computertalen uitleggen wat Forth is, moet dáárom mislukken. Sterker nog, Forth is voor nieuwelingen waarschijnlijk gemakkelijker dan voor mensen met programmeerervaring.

Forth bestaat uit woorden, losse woorden. Uitdrukkingen als `PRINT(x+(y/2))` waarin het kommando `PRINT` pas betekenis krijgt als het laatste haakje-sluiten bereikt is, zijn er niet in Forth. "Je kunt het natuurlijk wel zo maken" is onder Forth-programmeurs een gevleugelde uitdrukking. Ook: "maar waarom zou je?". Nu begeef ik me echter op het uitleg-pad, terwijl ik zojuist het hopeloze daarvan heb aangetoond.

In de Verenigde Staten is men bezig met het formuleren van een standaard voor Forth, ANS-Forth (American National Standard Forth). Die zal in 1993 officiëel zijn. Ik baseer mij alvast op deze standaard, en besteed geen aandacht aan F.I.G.-Forth, 79-Forth en FORTH-83. De index achterin bevat alle ANS-Forth woorden. Verschillen tussen de dialekten zijn voor een ervaren Forth-programmeur geen enkel probleem. Voor een beginner vormen ze slechts verwarrende ballast. Forth is geen dure taal. Er is een Forth voor vrijwel ieder systeem. Met enige moeite kan daar "iets bovenop gezet worden" om er het gewenste dialekt van te maken. Neem kontakt op met de Forth Gebruikers Groep van de HCC als er toch problemen zijn.

Een technische opmerking: ANS-Forth houdt nadrukkelijk de mogelijkheid open dat de gehele getallen intern anders dan met het two's complement principe gerealiseerd worden. Ik veronderstel in deze cursus dat de cursist beschikt over een two's complement Forth.

De cursus is voor beginners, maar ik ga er van uit dat basisbegrippen zoals byte, bit en RAM bekend zijn. Dit is een Forth-cursus, geen computer-cursus. De tekst is nogal kompakt en soms pas begrijpelijk als de opdrachten aan de computer uitgevoerd zijn. Het is noodzakelijk om de cursus achter de computer te volgen.

Vragen en construktieve kritiek zijn welkom,

Albert Nijhof
Steijnstraat 13
6814 AK Arnhem
~~a.nijhof@kader.hobby.nl~~

Voorwoord 2

Arnhem, 18.12.2001

De tweede druk, hoewel in een nieuw jasje gestoken, is qua inhoud gelijk aan de eerste uitgebreid met de hoofdstukken 200 tot en met 210.

De enige echte afwijking:
Toen de eerste druk uitkwam lag ANS-Forth nog niet helemaal vast. Op het laatste moment heeft men de wijziging aangebracht dat een VALUE bij zijn definiëring toch zijn beginwaarde op Stack verwacht. Dat is in deze druk aangepast.

Het Cursusgedeelte is didactisch geordend, de rest van dit boek niet.
Het is niet mijn opzet geweest dat een lezer dit boek bladzijde voor bladzijde van begin tot eind door zou werken. Integendeel, ik raad hem aan om, als hij in de buurt van hoofdstuk 15 aangekomen is en zich al enigszins thuis begint te voelen in Forth, in het systematische deel en de losse artikelen te gaan grasduinen. Hoe eerder je zelfstandig in Forth leert denken hoe beter.

De Index achterin noemt alle ANS-Forth woorden, ook de woorden die in dit boek onbesproken blijven.

A. N.
~~a.nijhof@kader.hobby.nl~~

Inhoud van het cursusdeel

Forth bestaat uit woorden.
Als je een of meer woorden achter elkaar intypt en daarna op Return drukt, zal Forth die woorden uitvoeren in de volgorde waarin je ze getypt hebt.

In het vervolg zal ik met een haakje sluiten aan het begin van een regel aangeven dat je de tekst achter het haakje moet intypen tot en met `[rtn]`. Met `[rtn]` is de Return toets of de Enter toets bedoeld.

Voorbeelden:

```
) 65 EMIT [rtn] A ok
)
) 66 EMIT [rtn] B ok
)
) 56 . [rtn] 56  ok
```

Een getal wordt ook als een woord opgevat. `EMIT` kan vervolgens over dat getal beschikken, beschouwt het als een ASCII code en laat het bijbehorende karakter op het scherm verschijnen.

Ook de punt is een Forth woord. Hij maakt het getal zelf op het scherm zichtbaar.

```
) 65 EMIT 66 EMIT [rtn] AB ok
)
) 65 66 EMIT EMIT [rtn] BA ok
)
) 65 74 EMIT EMIT EMIT [rtn] JA  Empty Stack
)
) 1000 102 EMIT . [rtn] f1000  ok
```

Een woord dat een getal nodig heeft, pakt altijd het getal dat als laatste binnengekomen is.

Forth geeft een foutmelding als de getallen op zijn. De formulering ervan kan per Forth verschillen.

Ingevoerde getallen worden vastgehouden met een mechanisme dat in het engels Stack heet (stapelen). Getallen worden op volgorde van hun binnenkomst bewaard, en in omgekeerde volgorde weer afgegeven aan woorden die getallen verbruiken.

Bij het bespreken van een Forth woord is het gebruikelijk om achter dat woord tussen haakjes aan te geven wat zijn invloed op de Stack is:

`EMIT ( getal -- )` Haal het getal van de Stack af, beschouw het als een ASCII code en druk het bijbehorende karakter af.

`. ( getal -- )` Haal het getal van de Stack af en druk het af met een spatie erachter.

Nog een paar Forth woorden:

`KEY ( -- getal )` Wacht totdat er een toets ingedrukt is en zet de ASCII code van die toets als getal op Stack.

`CR ( -- )` Ga naar een nieuwe regel. Geen verandering op de Stack.

? Kun je het effekt voorspellen van:

```
) KEY . [rtn] ?

) KEY EMIT [rtn] ?

) CR KEY . [rtn] ?

) KEY CR . [rtn] ?

) KEY . CR [rtn] ?

) KEY KEY EMIT EMIT [rtn] ?
```

`+ ( x y -- z )` Haal twee getallen van de Stack af, tel ze op, en zet het resultaat weer op Stack.

```
) 1 4 + [rtn]  ok
) . [rtn] 5  ok
) 2 3 + 4 + . [rtn] 9  ok
) 2 3 4 + + . [rtn] 9  ok
```

`- ( x y -- z )` Aftrekken: `z = x - y`

`* ( x y -- z )` Vermenigvuldigen: `z = x * y`

```
) 1 4 - . [rtn] -3  ok
) 2 3 * . [rtn] 6  ok
```

Je kunt de volgorde van getallen op Stack veranderen:

`SWAP ( x y -- y x )` Verwissel de laatste twee getallen op Stack onderling.

```
) 1 4 SWAP - . [rtn] 3  ok
```

`DUP ( x -- x x )` Maak een extra exemplaar van het laatste getal.

```
) KEY DUP . EMIT [rtn] ?
```

? Als je moet uitrekenen hoeveel `500+14*(61-52)` is, welke van de onderstaande uitwerkingen zijn dan onjuist? En welke vind je het overzichtelijkst?

```
) 500 14 61 52 - * + . [rtn] ?

) 61 52 - 14 * 500 + . [rtn] ?

) 500 14 52 61 SWAP - * + . [rtn] ?

) 14 61 52 - * 500 + . [rtn] ?

) 500 14 61 DUP EMIT 52 - * + . [rtn] ?

) 500 14 61 52 - DUP . * DUP . + . [rtn] ?

) 61 52 - DUP . 14 * DUP . 500 + . [rtn] ?
```

WORDS (--) Laat de woorden zien die Forth kent. De nieuwste woorden staan voorop.

Dat laatste zinnetje wordt begrijpelijk als je weet dat je er woorden bij kunt maken:

Programmeren in Forth = Nieuwe woorden maken

Dat gaat als volgt:

```
) : PILS 105 * . ; [rtn] ok
```

Let er op dat woorden van elkaar gescheiden moeten zijn door een of meer spaties.

- Het eerste woord, de dubbele punt, leidt de definitie van een nieuw woord in.
- Het tweede woord is PILS

Je vermoedt al dat het geen bestaand Forth woord is. Het eerste woord achter een dubbele punt wordt de naam van het nieuw te vormen woord. Die naam kun je vrij kiezen (lengte hoogstens 31 karakters, geen spaties, want die werken als woordscheiding). Als je liever KOFFIE hebt, kan dat.

- Dan volgen er drie woorden die je al kent. Zij worden in een lijst achter de naam genoteerd met de bedoeling dat ze pas later, als je het nieuwe woord gaat gebruiken, in aktie komen.
- De puntkomma, tenslotte, sluit de definitie af. Forth is nu uitgebreid met het woord PILS .

```
) WORDS [rtn] ?
```

Technisch onderscheidt PILS zich niet van de bestaande Forth woorden. Het is geen tweederangs woord, en het werkt ook niet trager dan de bestaande woorden.

: ccc (--) Maak een definitie met de naam ccc . Met ccc bedoel ik een willekeurige naam.

; (--) Beëindig een definitie.

PILS (x --) Druk de rekening voor x pilsjes af.

```
) 1 PILS [rtn] ?
) 4 PILS [rtn] ?
) 10 PILS [rtn] ?
) : KRAT 24 * ; [rtn] ok
) 2 KRAT PILS [rtn] ? ( En als je genoeg gehad hebt: )
) FORGET PILS [rtn] ok
```

FORGET ccc (--) Verwijder het woord ccc uit Forth. Met ccc verdwijnen ook alle woorden die nieuwer zijn dan ccc.

`.S ( -- )` Laat zien welke getallen er op Stack staan zonder de Stack te veranderen (punt S).

? Typ onderstaande regels in, en stel vast wat de woorden `DROP` en `OVER` doen. Schrijf, iedere keer voordat je op `[rtn]` drukt, op een papier welk resultaat je verwacht.

```
) ARNHEM [rtn] ?
) 15 .S [rtn] ?
) DUP .S [rtn] ?
) + .S [rtn] ?
) 2 .S [rtn] ?
) OVER .S [rtn] ?
) + .S [rtn] ?
) EMIT .S [rtn] ?
) DROP .S [rtn] ?
) DROP .S [rtn] ?
```

`ARNHEM` is geen Forth woord en veroorzaakt een foutmelding. Dat is natuurlijk flauw. Maar, bij een foutmelding wordt de Stack altijd leeggemaakt, en daar ging het mij om.
? 32 is de ASCII code voor een spatie. Kun je die spatie op het scherm ontdekken?

In de laatste regel probeert `DROP` een getal van Stack te verwijderen. Maar de Stack is leeg. Dat veroorzaakt een foutmelding. De `.S` die erachter staat, wordt niet meer uitgevoerd.

`BL ( -- k )` Zet de ASCII code van de spatie (32) op Stack.

```
) : WIJD ( k -- ) EMIT BL EMIT ; [rtn] ok
```

Tekst tussen haakjes is ter informatie. Die hoef je dus niet in te typen. Je zou dat echter wel kunnen doen, want Forth slaat alles over wat tussen haakjes staat. Het haakje openen is een Forth woord en **moet daarom gevolgd worden door een spatie**, daarna kan de informatie komen.

```
) 12 ( onzin ) . .S [rtn] ?
)
) 65 WIJD 66 WIJD 67 WIJD [rtn] ?
```

`( ( tekst -- )` Sla de tekst t/m het eerstvolgende haakje sluiten over.

[wordt vervolgd]

`CONSTANT ccc ( x -- )` Definiëer een konstante met de waarde `x` en de naam `ccc`.

```
) 12 CONSTANT DOZIJN [rtn] ok
```

`DOZIJN ( -- 12 )` Zet de waarde van de konstante op Stack.

`VARIABLE ccc ( -- )` Definiëer een variabele met de naam `ccc`.

```
) VARIABLE BEDRAG [rtn] ok
```

`BEDRAG ( -- a )` Zet het geheugenadres van de variabele op Stack.

`! ( x a -- )` Schrijf de waarde `x` in het adres `a`.

```
) 200 BEDRAG ! [rtn] ok
```

Nu heeft `BEDRAG` de waarde 200.
Het woord `!` wordt uitgesproken als "store".

`@ ( a -- x )` Zet de inhoud van adres `a` op Stack. ("fetch")
`+! ( y a -- )` Tel `y` op bij de inhoud van adres `a`. ("plus store")

```
) BEDRAG @ . [rtn] 200  ok
) 10 BEDRAG +! [rtn] ok
) BEDRAG @ . [rtn] 210  ok
```

Let er op, dat het nieuwste getal op de Stack ALTIJD een variabele (een adres) is als je de woorden `! @` en `+!` gebruikt. Vooral `!` en `+!` zijn gevaarlijke woorden. Als je er fouten mee maakt, overschrijf je ongewild ergens iets in het geheugen, en dat kan "fatale" gevolgen hebben. Dus:
Gebruik `! @` en `+!` alleen direkt achter de naam van een variabele.

```
) : APPELS ( n -- ) 80 * BEDRAG +! ; [rtn] ok
) : PEREN  ( n -- ) 70 * BEDRAG +! ; [rtn] ok
) : TOTAAL ( -- ) BEDRAG @ . 0 BEDRAG ! ; [rtn] ok

) TOTAAL [rtn] ?
) TOTAAL [rtn] ?
) 5 APPELS 3 PEREN TOTAAL [rtn] ?
)
) : EIEREN 60 * BEDRAG +! ; [rtn] ok
) 2 PEREN 2 EIEREN TOTAAL [rtn] ?
) DOZIJN EIEREN 10 APPELS 10 PEREN TOTAAL [rtn] ?

) FORGET DOZIJN [rtn] ?
```

? Opgave.
Schrijf, uit het hoofd, tussen haakjes achter de tot nu toe geleerde Forth woorden wat hun effekt op de Stack is. `( ... -- ... )`
Vóór het dubbele minteken: de getallen die het woord verbruikt.
Achter het dubbele minteken: de getallen die het woord produceert.
Geef met `ccc` aan dat het woord een naam nodig heeft.

(Uit hoofdstuk 1 en 2)

```
EMIT
.
KEY
CR
```

(Uit hoofdstuk 3)

```
+
-
*
SWAP
DUP
```

(Uit hoofdstuk 4)

```
WORDS
:
;
FORGET
```

(Uit hoofdstuk 5)

```
.S
DROP
OVER
BL
(
```

(Uit hoofdstuk 6)

```
CONSTANT
VARIABLE
!
@
+!
```

In sommige Forth systemen laat de dubbele punt bij het definiëren een waarde achter op Stack die er door de puntkomma weer afgehaald wordt. Deze waarde wordt in het systematische deel achter deze kursus met `sys?` aangeduid.

Een snelle en veilige manier om te weten te komen of Forth een bepaald woord kent, is met het enkele aanhalingsteken (engels: Tick).
`' ccc ( -- s )` Zet de sleutel van het woord `ccc` op Stack. Als het woord niet gevonden wordt, komt er een foutmelding.
Elk woord heeft, behalve een naam, nog een uniek herkenningsgetal: de sleutel (engels: Token).

```
) ' DROP . [rtn] ?
) ' WORDS . [rtn] ?
) ' ARNHEM . [rtn] ?
) ' 0 . [rtn] ?
) ' 75 . [rtn] ?
```

De waarde van een sleutel is systeemafhankelijk. Getallen hebben over het algemeen geen naam en geen sleutel.

`EXECUTE ( s -- )` Voer het woord dat de sleutel `s` heeft uit.

```
) 3 4 .S [rtn] ?
) ' SWAP EXECUTE [rtn] ok
) .S [rtn] ?
```

Het is aan de programmeur om ervoor te zorgen dat de sleutel voor `EXECUTE` een geldige sleutel is, dwz. verkregen is met het woordje `'` (Tick).

Bij het lezen en uitvoeren van een tekst die je ingetypt hebt, kijkt Forth woord voor woord of hij het kan vinden in zijn woordenboek.
Gevonden: Forth voert het woord uit.
Niet gevonden: Forth gaat er van uit dat het een getal zal zijn, dat hij op Stack moet zetten. Pas als dit ook niet lukt, komt er een foutmelding.

Ter overdenking:

```
) : IT ; [rtn] ok
) : ZEVEN 7 ; [rtn] ok
) ZEVEN 2 * . [rtn] ?

) : 7 7 ; [rtn] ok
) 7 7 * . [rtn] ?

) ZEVEN CONSTANT SEVEN [rtn] ok
) SEVEN . [rtn] ?

) ' 7 EXECUTE . [rtn] ?

) : 8 100 ; [rtn] ok
) 8 1 + . [rtn] ?

) FORGET IT [rtn] ok
```

Tot nu toe heb je opdrachten aan Forth steeds rechtstreeks op het toetsenbord ingetypt. Als je een tikfout maakt bij het intypen van een definitie en op Return gedrukt hebt, kun je die fout niet zomaar herstellen. Je moet de definitie weer van voren af aan beginnen. In de komende hoofdstukjes worden de definities langer. Dan zou het aangenaam zijn om de mogelijkheid te hebben de tekst rustig in te typen en van fouten te ontdoen, voordat Forth hem gaat lezen en uitvoeren.
Daar zijn drie methodes voor. Lees ze door en kijk in de handleiding bij je Forth welke je kunt gebruiken:

Methode 1, Blokken, met een Editor binnen Forth.
Forth beschouwt de gehele floppy als opgedeeld in genummerde blokken, Screens, die direkt via hun volgnummer aanspreekbaar zijn. Screen nr. 0 is niet bruikbaar. De telling begint bij 1. Een Screen bestaat uit 16 regels, elk met ruimte voor 64 karakters.
`LIST ( n -- )` Laat de tekst van Screen nr. `n` zien.
`LOAD ( n -- )` Lees de tekst van Screen `n` en voer hem uit, alsof hij rechtstreeks via het toetsenbord ingetypt wordt.
`THRU ( n1 n2 -- )` Load de Screens `n1` t/m `n2` achter elkaar.
Deze methode negeert de File administratie van het Operating System. Het is dus zaak om de Block Disks zorgvuldig van de File Disks te scheiden. Voor de Editor kommando's verwijs ik naar de handleiding bij je Forth systeem.

Methode 2, Files, met een tekstverwerker binnen of buiten Forth.
Schrijf met een tekstverwerker naar een File en laat die File door Forth lezen en uitvoeren.
`INCLUDE ccc ( -- )` Lees de File die `ccc` heet, en voer hem uit.
Bij deze methode kun je je favoriete tekstverwerker gebruiken, maar daarvoor zul je Forth moeten verlaten. Kijk in je handleiding voor de details.

Methode 3, Block Files, met een Editor.
Deze methode lijkt op de blokken methode. Het enige verschil is, dat een beperkt aantal blokken met elkaar een File vormen, waardoor het Operating System de baas kan blijven.
Je kunt de woorden `LIST LOAD` en `THRU` gebruiken. Kijk in de handleiding bij je Forth voor de details en voor de Editor kommando's.

Voorbeeld:

```
) : NUL? ( x -- )
)   ." ("
)   0 =
)   IF ." ja"
)   ELSE ." nee"
)   THEN ." ) "
) ;
```

Nieuwe woorden:
`." ccc"` `( -- )` Druk de tekst `ccc` af. Omdat `."` (punt aanhaling) een woord is, moet hij door een spatie gescheiden worden van de af te drukken tekst. Het tweede aanhalingsteken geeft het teksteinde aan en maakt geen deel uit van de tekst. Er mogen nu wel spaties maar uiteraard geen aanhalingstekens in `ccc` voorkomen.

`=` `( x y -- vlag )` Test of `x` gelijk is aan `y` en laat een vlag achter op de Stack. De True vlag is gelijk aan -1 en de False vlag is gelijk aan 0.

`TRUE` `( -- x )` `x=-1` d.w.z. alle bits van `x` zijn gezet.
`FALSE` `( -- x )` `x=0` d.w.z. alle bits van `x` zijn nul.

`IF` `( vlag -- )` Haal een vlag van Stack, en reageer aldus:
`[true?]` **`IF`** `[zo ja, doe dan dit]`
`ELSE` `[zo nee, doe dan dit]`
`THEN` `[enz.]`
`IF` beschouwt ieder getal dat niet nul is als een True vlag.

De If-Else-Then-konstruktie kan ook zonder `ELSE` voorkomen. Voorbeeld:

```
) : TEST ( x -- )
)   DUP . ." is "
)   DUP 0 < IF ." kleiner dan " THEN
)   DUP 0 > IF ." groter dan " THEN
)   DROP ." nul " ;
) CR 3 TEST CR -5 TEST CR 0 TEST [rtn] ?
```

Nieuwe woorden:
`<` `( x y -- vlag )` Test of `x<y`.
`>` `( x y -- vlag )` Test of `x>y`.

? Opgave.
Maak een woord `CIJFER?` `( -- )` dat vaststelt of een ingetypte toets een cijfer is. Op het scherm moet de mededeling 'cijfer' of 'geen cijfer' verschijnen. Gebruik het woord `KEY` hierbij.

Voorbeeld:

```
) : TEL ( -- )
)   20 0 DO I . LOOP ;
```

DO (grenswaarde beginwaarde --) Start een lus. De teller begint met beginwaarde. Grens- en beginwaarde (let op hun volgorde) worden van Stack gehaald en intern opgeborgen.
LOOP (--) Verhoog de teller met 1 en spring terug naar het eerste woord dat achter DO staat, maar, ga verder als de teller gelijk is geworden aan de grenswaarde.
I (-- t) Zet de waarde van de teller van de Do-Loop op Stack. I is een afkorting van Index.

Het invoeren en het afdrukken van getallen ging tot nu toe in het tientallig (decimale) stelsel. Forth gebruikt daarbij de variabele BASE als grondtal. Door de waarde van BASE te veranderen kun je in andere talstelsels werken. HEX BINARY en DECIMAL zijn Forth woorden die omschakelen naar een bepaald talstelsel. (Als BINARY nog niet bestaat: : BINARY (--) 2 BASE ! ;).

```
) DECIMAL TEL [rtn] ?
) BASE @ . [rtn] ?
) HEX TEL [rtn] ?
) BASE @ . [rtn] ? (strikvraag)
) BINARY TEL [rtn] ?
) 10 DECIMAL . [rtn] ?
) 10 HEX . [rtn] ?
) 10 1 - . [rtn] ?
) 10 DECIMAL . [rtn] ?
```

$ of % aan het begin van een getal betekent dat dit ene getal in een bepaald talstelsel bedoeld is ongeacht de waarde van BASE op dat moment. (Dit gaat niet op in iedere Forth.)
? Met welke talstelsels korresponderen deze tekens?

```
) DECIMAL %10 . [rtn] ? (enz.)
```

? Leg uit wat het woord LIJST doet:

```
) : LIJST ( -- )
)   #17 0 DO CR
)   I HEX .   I DECIMAL .   I BINARY .
)   LOOP DECIMAL ;
) LIJST [rtn] ?
) %12 . [rtn] ? (strikvraag)
```

Voorbeeld:

```
) : TEL-DOOR ( x -- y )
)   BEGIN DUP .
)   1 +
)   KEY? UNTIL
)   KEY DROP ;
```

Nieuwe woorden:
`BEGIN ( -- )` Start een lus.
`UNTIL ( vlag -- )` Ga verder als de vlag True is, maar spring terug naar het eerste woord achter `BEGIN` als de vlag False is.
`KEY? ( -- vlag )` Test of er een toets ingedrukt is.
Als het antwoord True is, kan met `KEY` de ASCII code van die toets op Stack gezet worden.

```
) 0 TEL-DOOR . [rtn] ?
) 0 TEL-DOOR TEL-DOOR . [rtn] ?
```

voorbeeld:

```
) VARIABLE TEKEN
) #43 TEKEN !
) TEKEN @ EMIT [rtn] ?
) : TEKENS ( n -- )
)   BEGIN
)   DUP 0 > WHILE
)   1 -
)   TEKEN @ EMIT
)   REPEAT DROP ;
```

Nieuw:
`WHILE ( vlag -- )` Ga verder als de vlag True is, maar spring uit de lus, over `REPEAT` heen, als de vlag False is.

? Wat moet je doen om met `TEKENS` zes minnetjes te laten verschijnen?

? Kun je het woord `TEKENS` maken met behulp van Do-Loop in plaats van met Begin-While-Repeat ?

? Wat doet het woord `TEKST` ?

```
) : TEKST ( -- )
)   KEY CR
)   BEGIN KEY
)   DUP EMIT
)   OVER = UNTIL
)   DROP ; ( Ik hoop dat je hier uitkomt. )
```

Voor een mens is een getal een rijtje cijfers. Als Forth zo'n getal binnenkrijgt, vertaalt hij dat rijtje cijfers naar een bitpatroon en zet dat op Stack. Bij die vertaling speelt BASE een rol.
Als Forth een bitpatroon als getal moet afdrukken (met . bijvoorbeeld), moet de weg terug afgelegd worden.
Berekeningen voert Forth uit met getallen die al vertaald zijn naar bitpatronen. BASE heeft daar geen invloed meer op.

Hoe is nu het verband tussen 'menselijke' getallen en bitpatronen?
Voor het gemak ga ik even uit van een heel klein Forthje waarin de patronen uit slechts vier bits bestaan (in werkelijkheid meestal 16 of 32 bits). Hier volgen alle mogelijke 4-bits patronen:

```
0000 0001 0010 0011 0100 0101 0110 0111 (groen)
1000 1001 1010 1011 1100 1101 1110 1111 (rood)
```

De patronen die met een 0 beginnen noem ik groen, de andere, die met een 1 voorop, noem ik rood.

Forth kent twee methodes om de patronen naar getallen te vertalen.

1) Tekenloos, Unsigned. Geen negatieve getallen. Nul is het kleinste getal. Als alle bits gezet zijn, is het grootste getal bereikt. Rood is altijd groter dan groen:

```
0000 0001 0010 0011 0100 0101 0110 0111 (groene patronen)
  0    1    2    3    4    5    6    7  (getallen)

1000 1001 1010 1011 1100 1101 1110 1111 (rode patronen)
  8    9   10   11   12   13   14   15  (getallen)
```

2) Met teken, Signed. Het vooropstaande bit wordt opgevat als teken, 1 duidt op een minteken. Rode patronen worden negatieve getallen en zijn dus kleiner dan groene:

```
1000 1001 1010 1011 1100 1101 1110 1111 (rode patronen)
 -8   -7   -6   -5   -4   -3   -2   -1  (getallen)

0000 0001 0010 0011 0100 0101 0110 0111 (groene patronen)
  0    1    2    3    4    5    6    7  (getallen)
```

[wordt vervolgd]

Gewoonlijk werkt men in Forth met Signed getallen. Voor tekenloze getallen bestaan soms aparte woorden.
Naast . (punt) bestaat U. (u punt).
Het woord . (punt) vertaalt naar Signed getallen.
Het woord U. (u punt) vertaalt naar Unsigned getallen.

```
) -1 U. [rtn] ?
```

Naast < (kleiner dan) bestaat U< (u kleiner dan).

```
) -5 5 < . [rtn] ?
) -5 5 U< . [rtn] ?
```

Voor optellen en aftrekken hoeft er geen onderscheid gemaakt te worden tussen Signed en Unsigned. Dat is mogelijk doordat Forth daarbij geen kontrole uitvoert op het te groot of te klein worden van de uitkomst. Overflow en underflow worden niet gemeld, de programmeur moet dat zelf opvangen. Het gaat als bij een kilometerteller in de auto. Als het hoogste getal verschijnt, allemaal negens, en je rijdt toch verder, dan komt er geen melding op het dashboard in de trant van END-OF-AUTO-ERROR.
Voor beide getalsoorten geldt: Het grootst mogelijke getal plus 1 geeft het kleinst mogelijke getal als uitkomst.

```
) BINARY FALSE . [rtn] ?
) FALSE U. [rtn] ?
) TRUE . [rtn] ?
) TRUE U. [rtn] ?
) DECIMAL TRUE U. [rtn] ?
) 3 5 - U. [rtn] ?
```

? Hoeveel bits gaan er per getal op de Stack?

Een cel is de hoeveelheid geheugen die een getal als bitpatroon in beslag neemt.

CELLS (x -- y) y is het aantal bytes dat nodig is voor x cellen.

```
) 10 CELLS . [rtn] ?
```

? Hoeveel bytes gaan er per getal op de Stack?
? Lees de beschijving van het woord LOOP in hoofdstuk 11 nog eens nauwkeurig, en bedenk hoevaak er 'ha ' verschijnt als je het woord HOEVAAK? uit zou voeren.

```
) : HOEVAAK? 0 0 DO ." ha " LOOP ; [rtn]
 ( Bezint eer gij begint )
```

Forth is een woordenboek waar nieuwe woorden aan toegevoegd kunnen worden. Dat vraagt natuurlijk geheugenruimte. Met het woord HERE kun je zien hoever het woordenboek doorloopt, en, waar het geheugengebied dat nog vrij is, begint.

HERE (- a) a is het adres waar nieuwe woorden terecht komen. Bij het definiëren van een woord verandert dat adres.

```
) HERE . [rtn] ?
) : PUNT . ; [rtn] ok
) HERE . [rtn] ?
) HERE PUNT [rtn] ?
) FORGET PUNT [rtn] ok
) HERE . [rtn] ?
```

Een Forth woord bestaat in principe uit drie elementen:
1) de Name, waardoor het woord terug te vinden is,
2) de Code, een stukje programma dat het eigenlijke werk doet,
3) de Body, waar de gegevens, die Code nodig heeft, instaan.

CREATE ccc (--) Maak een woord ccc waarvan de Body nog leeg is.
Een woord dat met CREATE gemaakt is, doet niets anders dan het adres van zijn Body op Stack zetten.

```
) HERE . [rtn] ?
) CREATE DAAR [rtn]
) HERE . [rtn] ?
```

DAAR (-- Body) Zet het adres van de Body van DAAR op Stack.

```
) DAAR . [rtn] ?
```

Dat is dus HERE

```
) 10 CELLS ALLOT [rtn] ok
) HERE . [rtn] ?
```

ALLOT (n --) Reserveer n bytes bij HERE

Je beschikt nu over een een gebied van 10 cellen vanaf een adres dat DAAR heet. Je kunt ermee doen wat je wilt, Forth zal er uit zichzelf niet meer aankomen.

[wordt vervolgd]

DAAR levert een adres. Dan kun je er dus ook @ en ! op loslaten.

```
) DAAR @ . [rtn] ?
) 1992 DAAR ! [rtn] ok
) 1 DAAR +! [rtn] ok
) DAAR @ . [rtn] ?
) DAAR @ DAAR 8 + ! [rtn] ok
) 7 DAAR +! [rtn] ok
) DAAR @ . [rtn] ?
) DAAR 8 + @ . [rtn] ?
```

Je moet er zelf voor zorgen dat je niet buiten het gereserveerde gebied komt. Iets als

```
7 DAAR 100 - !
```

zal waarschijnlijk een verrassend, maar zeker een ongewenst effekt hebben.

DUMP (a n --) Laat vanaf adres a de inhoud van een gebied van n bytes zien.

```
) DAAR 40 DUMP [rtn] ?
```

ACCEPT (a n -- m) Ontvang maximaal n karakters via het toetsenbord en noteer die bij adres a.
m is het het aantal karakters dat werkelijk ingevoerd is. De Return toets sluit de invoer af en telt zelf niet mee.

```
) DAAR 10 ACCEPT [rtn] ...... ok
) . [rtn] ?
) DAAR 40 DUMP [rtn] ?
```

CELL+ (a1 -- a2) a2 is het adres dat één cel verder ligt dan a1.

```
) DAAR CELL+ 10 ACCEPT DAAR ! [rtn] ...... ok
) DAAR 40 DUMP [rtn] ?
```

ASCII codes voor karakters staan in het geheugen dichter opeengepakt dan bitpatronen voor getallen. In één cel gaan meestal 2 of 4 karakters.

CHAR+ (a1 -- a2) Adres a2 ligt één karakter verder dan a1.
CHARS (x -- y) Voor x karakters zijn y bytes nodig.
[wordt vervolgd]

`C@ ( a -- k )` Lees karakter `k` uit adres `a`.
`C! ( k a -- )` Zet karakter `k` in adres `a`.

```
) DAAR CHAR+ 10 ACCEPT DAAR C! [rtn] ok
) DAAR 11 CHARS DUMP [rtn] ?
) DAAR C@ . [rtn] ?
```

In bovenstaand voorbeeld zet `C!` een bitpatroon in `DAAR` dat aangeeft hoeveel karakters er ingevoerd zijn. De programmeur moet zelf bijhouden dat het hier om een (klein) getal gaat en niet om een karakter. `C!` en `C@` onderscheiden zich van `!` en `@` door de lengte van de bitpatronen die ze verplaatsen. Met de inhoud of de betekenis ervan houden ze zich niet bezig.
Met `C!` komt niet het volledige bitpatroon dat op Stack staat in RAM terecht; het voorste stuk wordt domweg genegeerd. Bij het teruglezen met `C@` komen daar weer nullen voor in de plaats.

```
) -1 DAAR 8 + C! [rtn] ok
) DAAR 8 + C@ [rtn] ok
) DUP HEX . [rtn] ?
) DUP BINARY . [rtn] ?
) DECIMAL . [rtn] ?
```

? Hoeveel bits leest `C@` ?

```
) DAAR CHAR+ DAAR C@ TYPE [rtn] ?
```

`TYPE ( a len -- )` Druk de tekst af die op adres `a` begint en uit `len` karakters bestaat.

Korte teksten zet men vaak in het geheugen als een getelde sliert (Counted String). Op de plaats van het eerste karakter staat om hoeveel karakters het gaat, daarachter komt de tekst. Daardoor is het niet nodig om aan het einde een speciaal teken neer te zetten.

```
) DAAR COUNT TYPE [rtn] ?
```

`COUNT ( a1 -- a2 k )` Zet de inhoud van de karaktercel bij adres `a1` op Stack. Het adres `a2` is één karakter verder dan `a1`. Je kunt met `COUNT` door een tekst heenlopen:

```
) DAAR COUNT . [rtn] ?
) COUNT EMIT [rtn] ?
) COUNT EMIT [rtn] ?
) COUNT EMIT [rtn] ? ( enzovoort )
) DROP [rtn] ok
```

Je zou COUNT aldus kunnen definiëren:

```
) : COUNT       ( a1 -- a2 k )
)   DUP         ( a1 a1 )
)   CHAR+ SWAP  ( a2 a1 )
)   C@          ( a2 k )
) ;
```

Tussen haakjes vermeld ik steeds de situatie op de Stack.

De meeste Forth woorden kun je definiëren met behulp van andere Forth woorden. Nog een voorbeeld ter bestudering:

```
) : TYPE ( a len -- )
)   DUP 0 >       ( a len groter-dan-nul? )
)   IF 0          ( a len 0 )
)      DO         ( a )
)         COUNT   ( a+ k )
)         EMIT    ( a+ )
)      LOOP       \ ga terug naar COUNT
)   ELSE  DROP    \ drop len
)   THEN  DROP    \ drop adres
) ;
```

\ (--) Negeer de rest van de regel. Dit geldt voor Forth, niet voor de lezer!

In het laatste voorbeeld zie je dat If-Then en Do-Loop samen kunnen gaan. Ze moeten dan wel zoals dat heet "genest" zijn. Je kunt middenin een If-Then struktuur aan een Do-Loop beginnen, maar die Do-Loop moet volledig afgehandeld zijn voordat je de If-Then afmaakt.
De volgorde .. IF .. DO .. THEN .. LOOP .. is dus niet mogelijk.
Algemeen geldt voor If-Else-Then, Begin-Until, Begin-While-Repeat en Do-Loop dat je ze in een definitie onbeperkt kunt nesten: Je mag op elk moment aan zo'n struktuur beginnen, maar, je mag alleen verder gaan aan de onafgewerkte struktuur die als laatste begon.

? Zoek de fouten:
1) .. IF .. IF .. THEN .. THEN ..
2) .. IF .. THEN .. IF .. THEN ..
3) .. BEGIN .. IF .. UNTIL .. THEN ..
4) .. DO .. IF .. BEGIN .. UNTIL .. THEN .. LOOP ..
5) .. IF .. IF .. THEN .. ELSE .. IF .. ELSE .. THEN .. THEN ..

```
) 29 10 /MOD .S [rtn] ?
) . . [rtn] ?
```

`/MOD ( x y -- r q )` Deel `x` door `y`.
q (een geheel getal!) is het resultaat, en `r` is de rest. Uitspraak: "slash mod".

```
) 29 10 / . [rtn] ?
) 29 10 MOD . [rtn] ?
```

`/ ( x y -- q )` Deel `x` door `y`.
`q` is het quotient, een geheel getal!
`MOD ( x y -- r )` Deel `x` door `y`.
`r` is de rest. MOD is een afkorting van de wiskunde term Modulo.

Bij deze vorm van delen zijn alleen gehele getallen betrokken.

- Je weet de prijzen van verschillende artikelen zonder BTW en je wilt de prijzen met BTW (17%) berekenen:

```
) : MET1 ( prijs-zonder-BTW -- )   100 / 117 * . ;
) : MET2 ( prijs-zonder-BTW -- )   117 * 100 / . ;
```

? Welke methode (proberen!) voldoet het beste, `MET1` of `MET2` ? Kun je de verschillen verklaren?

Beter is:

```
) : MET3 ( netto-prijs -- )   117 100 */ . ;
```

`*/ ( x y z -- x*y/z )` Het voordeel van `*/` boven een losse `*` en `/` is, dat bij `*/` het tussenresultaat `x*y` een getal (een bitpatroon) mag worden dat te groot (te lang) is voor de Stack. Uitspraak: "star slash".

Nog beter is:

```
) VARIABLE PERCENTAGE 17 PERCENTAGE ! [rtn]
) : MET ( netto-prijs -- eindbedrag)
)   100 PERCENTAGE @ +   100 */ ;
) : .MET MET . ;
```

`MET` is niet meer afhankelijk van de hoogte van de BTW. Je kunt er zelfs kortingen mee berekenen:

```
) -15 PERCENTAGE !
) 1745 .MET [rtn] ?
) 495 .MET [rtn] ?
) 1245 MET .MET [rtn] ? ( dubbele korting )
```

```
*/MOD ( x y z -- rest x*y/z )

) 4 5 6 */MOD . . [rtn] ?
```

Overzicht van de delingen tot nu toe: **`*/MOD`** **`*/`** **`/MOD`** **`/`** en **`MOD`**

In het volgende hoofdstuk wil ik een wat ingewikkelder woord gaan maken dat de verhouding tussen twee getallen afdrukt als een decimale breuk. Ter voorbereiding eerst nog een paar nieuwe woorden.

`AND ( x y -- z )` `x` en `y` worden bit voor bit ge-AND. Een z-bit is 1 als het x-bit en het y-bit beide 1 zijn. Zo ook:
`OR ( x y -- z )` Een z-bit is 1 als het x-bit en het y-bit niet beide 0 zijn.
`XOR ( x y -- z )` Een z-bit is 1 als het x-bit en het y-bit verschillend zijn.

```
) : NEGATIEF? ( x y -- vlag )
)   XOR 0 < ;
```

? Wat is het verband tussen de vlag die `NEGATIEF?` levert en de tekens van `x` en `y` ?

`2DROP ( x y -- )` Verwijder 2 getallen van de Stack.
`2DUP ( x y -- x y x y )`

Je kunt `2DROP` als volgt definiëren:

```
: 2DROP DROP DROP ;
```

? Hoe zal de definitie van `2DUP` eruit zien?

```
) : STIP ." ." ;
```

? Wat doet `STIP`?

`.R ( x n -- )` Druk het getal `x` af, rechtsgericht in een veld van `n` posities, zonder afsluitende spatie. Als het getal daar niet in past, wordt het toch volledig afgedrukt. (punt r)

```
) CR 1 4 .R [rtn] ?
) CR 20 4 .R [rtn] ?
) CR 300 4 .R [rtn] ?
) CR 2000 4 .R [rtn] ?
) CR 10000 4 .R [rtn] ?
) CR 7 0 .R 8 0 .R [rtn] ?
```

`ABS ( x -- y )` `y` is de absolute waarde van `x`.
[wordt vervolgd]

Hier komt het aangekondigde woord. Ik geef het tweemaal, eerst in kale vorm, daarna met uitleg erbij.

```
) VARIABLE DECIMALEN  9 DECIMALEN !
) : .BREUK ( x y -- ) \ Druk x/y af als
)   2DUP XOR 0<       \  decimale breuk
)   IF ." -"
)   THEN ABS
)   SWAP ABS
)   OVER
)   /MOD
)   1 .R
)   DECIMALEN @ 0 >
)   IF ." ."
)      DECIMALEN @ 0 DO
)         OVER
)         BASE @ SWAP
)         */MOD
)         1 .R
)      LOOP
)   THEN 2DROP ;
```

```
VARIABLE DECIMALEN \ Aantal af te drukken decimalen
9 DECIMALEN ! \ Bijvoorbeeld 9
: .BREUK ( x y -- ) \ Druk x/y af als decimale breuk
  2DUP XOR 0< ( x y vlag )  \ Negatief antwoord?
  IF ." -"                  \ Dan minteken afdrukken
  THEN ABS ( x y )          \ y is nu zeker positief
  SWAP ABS ( y x )          \ x is nu zeker positief
  OVER     ( y x y )
  /MOD     ( y rest x/y )   \ Deze x/y vormt het
                            \ gedeelte voor de komma
  1 .R          ( y rest )  \ Druk deze x/y af
  DECIMALEN @ 0 > ( y rest vlag )
  IF ." ."        ( y rest ) \ Druk decimale punt af
     DECIMALEN @ 0 DO ( y rest )
        OVER          ( y rest y )
        BASE @ SWAP   ( y rest grondtal y )
        */MOD       ( y nieuwe-rest volgend-cijfer )
    \ Vermenigvuldigen met het grondtal is hetzelfde
    \ als nul-aanhalen in een staartdeling
        1 .R          ( y nieuwe-rest )
     LOOP             ( y rest )
  THEN 2DROP ;
```

```
) 3 8 .BREUK [rtn] ?
) 30 DECIMALEN ! [rtn] ok
) 1 31 .BREUK [rtn] ?
```

Sommige woordparen komen zo vaak voor dat er aparte woorden voor zijn.

```
: 1+ 1 + ;    : 1- 1 - ;     : TUCK SWAP OVER ;
: 2* 2 * ;    : 2/ 2 / ;     : NIP SWAP DROP ;
: 0> 0 > ;    : 0< 0 < ;
: 0= 0 = ;    : 0<> 0 <> ;
```

? Geef het stackeffect van deze woorden.

In de praktijk zijn deze woorden vaak niet met de dubbele punt gemaakt, maar rechtstreeks in machinetaal geschreven. Dergelijke woorden noemt men primitieven (Lo-level woorden). Ze zijn afhankelijk van de microprocessor. Vooral als woorden een elementaire taak verrichten, en formuleerbaar zijn met slechts enkele assemblerkommando's, zullen het primitieven zijn. Voorbeelden: `+ - SWAP DUP DROP OVER ! @ +! = TRUE FALSE <> U< CELL+ C! C@` enz. Ook `COUNT` waar ik een paar hoofdstukken terug een Hi-level definitie van heb gegeven, zal meestal een primitieve zijn.

Dubbele punt woorden (Hi-level woorden) zijn gemaakt met reeds bestaande Forth woorden, en onafhankelijk van de microprocessor analyseerbaar. In sommige Forth's vind je daarvoor het woord `SEE`.
`SEE ccc ( -- )` `SEE` gevolgd door de naam van een Hi-level woord geeft inzage in de bouw van dat woord.

Naast `CONSTANT` en `VARIABLE` is er nog een mogelijkheid om getallen van een naam te voorzien:

`x VALUE ccc ( x -- )` Definiëer een value `ccc` met waarde `x`.
`ccc ( -- )` Zet de actuele waarde van `ccc` op Stack.
Met `y TO ccc` geef je `ccc` de waarde `y`. Je kunt `!` en `@` niet gebruiken bij value's omdat het adres onbekend is.

```
) 0 VALUE LENGTE
) 17 TO LENGTE [rtn] ok
) LENGTE . [rtn] ?
) LENGTE 1+ TO LENGTE [rtn] ok
) LENGTE . [rtn] ?
```

Vergelijk `CONSTANT VARIABLE` en `VALUE` met elkaar:

Definiëren	77 CONSTANT KON	VARIABLE VAR	12 VALUE VAL
Getal opslaan	Gaat niet.	24 VAR !	24 TO VAL
Getal teruglezen	KON	VAR @	VAL

De grootste gemene deler van twee getallen x en y is het grootste getal (GGD) waarvoor de delingen x/GGD en y/GGD opgaan, d.w.z. nul als rest hebben. Als je de breuk x/y wilt vereenvoudigen, moet je teller en noemer door hun GGD delen.

Algorithme (recept) om de GGD van twee getallen te vinden.
1) Noem het grootste getal g en het kleinste getal k.
2) Als g=0 dan GGD=1.
3) Als k=0 dan GGD=g.
4) Vervang g door de rest die onstaat als je g door k deelt.
5) Ga naar 1)

```
) : GGD ( x y -- ggd )
)   2DUP OR 0= IF DROP 1 THEN
)   ABS
)   SWAP ABS
)   2DUP < IF SWAP THEN
)   BEGIN DUP
)   WHILE
)      TUCK
)      MOD
)   REPEAT
)   DROP
) ;
```

? Beantwoord de volgende vier vragen door met de hand, d.w.z. met potlood en papier en zonder computer, het programma woord voor woord uit te voeren.
- Wat is het resultaat als x=0 en y=0 ?
- Wat is het resultaat als x=0 en y=5 ?
- Wat is het resultaat als x=-4 en y=0 ?
- Wat is het resultaat als x=3 en y=-6 ?

? Noteer achter iedere regel de situatie op de Stack.

? Geef nauwkeurig aan welke delen van het programma korresponderen met de vijf regels van de algorithme.

Dringend advies:
ZET BIJ HET BESTUDEREN VAN EEN DEFINITIE ALTIJD ACHTER ELKE REGEL WAT ER OP DE STACK STAAT.

Behalve de gewone Stack heeft Forth nog de Return Stack.
Als het ingewikkeld dreigt te worden op de gewone Stack is het handig om even 1 of 2 getallen op de Return Stack te zetten.
`>R ( x -- ) ( R: -- x )` Zet `x` op de Return Stack.
`R> ( -- x ) ( R: x -- )` Haal `x` weer van de Return Stack af.
`R@ ( -- x ) ( R: x -- x )` Lees de `x` die op de Return Stack staat.

Voorbeeld, eerst weer zonder kommentaar, daarna met.

```
) \ Vereenvoudig de breuk x1/y1 tot x2/y2.
) : VEREENVOUDIG ( x1 y1 -- x2 y2 )
)   2DUP GGD
)   TUCK
)   /
)   >R
)   /
)   R>
) ;
```

? Zet, voordat je verder leest, de situatie op Stack achter iedere regel.

```
\ Vereenvoudig de breuk x1/y1 tot x2/y2.
: VEREENVOUDIG ( x1 y1 -- x2 y2 )
  2DUP GGD     ( x1 y1 ggd )
  TUCK         ( x1 ggd y1 ggd )
  /            ( x1 ggd y2 )
  >R           ( x1 ggd )    ( R: y2 )
  /            ( x2 )        ( R: y2 )
  R>           ( x2 y2 )
;

) 100 110 VEREENVOUDIG .S [rtn] ?
) . . [rtn] ?
```

Omdat ik `VEREENVOUDIG` zo'n lang woord vind, geef ik hem een kortere naam.

```
) : VV VEREENVOUDIG ;
) 1234 2468 VV . . [rtn] ?
```

De woorden `:` en `;` maken gebruik van de Return Stack. `DO I` en `LOOP` meestal ook. Daarom gelden de volgende beperkingen:

1) `>R` en `R>` moeten genest zijn t.o.v. `: ;` en `DO LOOP`.
2) `>R` en `R>` mogen alleen binnen een definitie voorkomen.
3) In een Do-Loop geeft `I` tussen `>R` en `R>` een onjuiste waarde.

```
) : PLUS + ;
) : + ( x y -- )
)   2DROP CR ."  vandaag wordt er niet opgeteld " ;
```

Tijdens het definiëren van + zal Forth meedelen dat er al een definitie met de naam + bestaat. Dat is geen foutmelding. Het is slechts een mededeling voor de programmeur. Het woord wordt wel gebouwd.
Aangezien nieuwe woorden achteraan de woordenlijst worden toegevoegd, en Forth altijd achteraan begint te zoeken, kun je vanaf nu alleen nog de nieuwe betekenis van + aanroepen.

```
) 4 5 + . [rtn] ?
```

De oude betekenis bestaat nog wel en definities waar de oude + in voorkomt blijven op de oude manier werken.

```
) 4 5 PLUS . [rtn] ?
```

Na

```
) FORGET + [rtn] ok
```

zal de oude + weer aanspreekbaar zijn, want `FORGET` heeft alleen de nieuwe + gevonden.

```
) 4 5 + . [rtn] ?
```

? Wat zal het gevolg zijn van de volgende definitie?

```
: FORGET ."  Nooit! " ;
```

In de programmeertaal Forth heb je grote vrijheden. Het devies is: alles moet mogelijk zijn. De prijs daarvan is, dat je niet gewaarschuwd kunt worden als je iets onhandigs doet. De programmeur draagt de verantwoordelijkheid voor wat hij doet en moet daarom ook begrijpen wat hij doet.
De een vindt het vervelend dat je steeds moet beseffen wat je aan het doen bent, en wenst door een `PIEP - Syntax Error` of wat voor Error dan ook, of op een struktureler manier, tegen zichzelf beschermd te worden. De ander bejubelt het feit dat Forth de volledige toegang verschaft tot de computer, "je kunt overal aankomen".

Bij V+ (optellen van twee breuken) wordt het ook druk op de Stack. Ditmaal los ik dat op door Value's te gebruiken.

```
) 0 VALUE TELLER1 0 VALUE NOEMER1
) 0 VALUE TELLER2 0 VALUE NOEMER2
)
) : V+ ( t1 n1 t2 n2 -- t3 n3 ) \ tel 2 breuken op
)   TO NOEMER2 TO TELLER2
)   TO NOEMER1 TO TELLER1
)   NOEMER1 NOEMER2 GGD  >R
)   TELLER1 NOEMER2 R@   */
)   TELLER2 NOEMER1 R@   */ +
)   NOEMER1 NOEMER2 R>   */
)   VEREENVOUDIG ;
```

? Zet, voordat je verder leest, de Stacksituatie achter iedere regel.

```
: V+ ( t1 n1 t2 n2 -- t3 n3 )
\ bereken de GGD van  n1  en  n2
\ t3 = t1*n2/ggd + t2*n1/ggd  en  n3 = n1*n2/ggd
  TO NOEMER2 TO TELLER2 ( t1 n1 )
  TO NOEMER1 TO TELLER1 ( -- )
  NOEMER1 NOEMER2 GGD   ( ggd )
  >R                  ( -- )                  ( R: ggd )
  TELLER1 NOEMER2 R@  ( t1 n2 ggd )           ( R: ggd )
  */                  ( t1*n2/ggd )           ( R: ggd )
  TELLER2 NOEMER1 R@  ( t1*n2/ggd t2 n1 ggd ) ( R: ggd )
  */                  ( t1*n2/ggd t2*n1/ggd ) ( R: ggd )
  +                   ( t3 )                  ( R: ggd )
  NOEMER1 NOEMER2 R>  ( n1 n2 ggd )           ( R: -- )
  */                  ( t3 n3 )
  VEREENVOUDIG        ( t3 n3 )
;

) 2 7 3 14 V+ ( 2/7 + 3/14 = ) .S [rtn] ?
) 6 DECIMALEN ! .BREUK [rtn] ?
```

NEGATE (x -- -x) Verander x van teken.

```
) 23 NEGATE . [rtn] ?
```

? Wat doet V- ?

```
) : V- NEGATE V+ ;
```

Tenslotte V* en V/

```
) : V* ( t1 n1 t2 n2 -- t3 n3 )
)   TO NOEMER2
)   VEREENVOUDIG
)   TO TELLER2 TO NOEMER1
)   NOEMER2
)   VEREENVOUDIG
)   TO NOEMER2
)   TELLER2 *
)   NOEMER1 NOEMER2 *
) ;
)
) : V/ ( t1 n1 t2 n2 -- t3 n3 )
)   SWAP
)   V*
) ;
```

? Zet achter elke regel de situatie op Stack met kommentaar.

In plaats van Value's zal men onder deze omstandigheden eerder Locale Value's (Locals) toepassen. Locals worden gemaakt binnen de definitie van het woord dat die Locals gebruikt. Als de definite klaar is, zijn de Locals niet meer zichtbaar. Omdat het definiëren van Locals niet in alle Forth's op dezelfde manier gaat, is dit niet de plaats om er meer over uit te leggen.

```
) : V. .BREUK ; ( voor de uniformiteit )
```

Nu beschik je voor het rekenen met verhoudingen of breuken over de volgende woorden:

V+ (optellen)
V- (aftrekken)
V* (vermenigvuldigen)
V/ (delen)
V. (afdrukken)
DECIMALEN (aantal decimalen achter de komma)
VV (vereenvoudigen, na invoer van een breuk)

De invoer van 1,25 gaat aldus:

```
) 125 100 VV [rtn] ok
) .S V. [rtn] ?
```

Het is in Forth mogelijk om met getallen te werken die meer ruimte innemen dan een cel. Zulke getallen, Double Numbers, hebben slechts één waarde, maar nemen twee plaatsen op de Stack in beslag.

`S>D ( x -- xlo xhi )` Breid een Signed Single Number uit tot een Signed Double Number met dezelfde waarde. (xlo=x)
`D. ( xlo xhi -- )` Druk de laatste twee getallen op de Stack samen af als één dubbelgetal.

```
) 3 S>D .S [rtn] ?
) D. [rtn] ?
) -3 S>D .S D. [rtn] ?
```

Als je een punt gebruikt bij de invoer van een getal maakt Forth er een dubbelgetal van.

```
) 30. .S [rtn] ?
) D. [rtn] ?
) -30. .S D. [rtn] ?
) 3.0 .S D. [rtn] ?
) .30 .S D. [rtn] ?
```

De plaats van de punt in het getal is niet van belang.

`D>S ( xlo xhi -- x )` Kort een dubbelgetal in tot een gewoon getal. Omdat er geen foutmelding komt als de waarde niet in een cel past, kan de waarde hierdoor veranderen.

```
) 3. D>S .S . [rtn] ?
```

Meer woorden voor dubbelgetallen.

```
2DROP ( xlo xhi -- )
2DUP ( ? )
2OVER ( ? )
2SWAP ( ? )
2ROT ( ? )
D+ ( xlo xhi ylo yhi -- zlo zhi ) z is de som van x en y.
D- ( ? )
D2* ( xlo xhi -- ylo yhi )
D2/ ( xlo xhi -- ylo yhi )
D= ( ? )
D< ( ? )
DNEGATE ( ? )
```

? Zoek aan de hand van zelf te maken voorbeelden nauwkeurig uit wat bovenstaande woorden doen.

Een mogelijke definitie van S>D is

```
: S>D DUP 0< ;
```

Bij een negatief getal wordt er True voorgezet, anders False.

```
) 3 S>D .S [rtn] ?
) DNEGATE .S D. [rtn] ?
) -3 S>D .S [rtn] ?
) DNEGATE .S D. [rtn] ?
```

Definitie van D>S

```
: D>S DROP ;
```

De voorste cel wordt domweg weggegooid.

Voorbeeld.
Een woord om te onderzoeken of de waarde van een dubbelgetal in een cel past.

```
) : ENKEL? ( xlo xhi -- xlo xhi vlag )
)   OVER S>D    ( xlo xhi  xlo xhi' )
)   2OVER       ( xlo xhi  xlo xhi' xlo xhi )
)   D=          ( xlo xhi vlag )
) ;
```

Dat kan korter.

```
) : ENKEL? ( xlo xhi -- xlo xhi vlag )
)   OVER 0<      ( xlo xhi xhi' )
)   OVER = ;
```

? Wat is de betekenis van de vlag die ENKEL? levert?

Voor puzzelaars:
? Van een Unsigned Number maak je een dubbelgetal door er nul voor te zetten.
Leg uit, waarom MAX-N het grootste (positieve) Signed enkelgetal geeft.

```
) : MAX-N -1 0 D2/ DROP ;
```

? Leg uit, waarom MIN-N het kleinste (negatieve) Signed enkelgetal geeft.

```
) : MIN-N 0 1 D2/ DROP ;
```

? Maak zo ook de woorden MAX-DN en MIN-DN voor dubbelgetallen.
Hint: hoe zien deze getallen er hexadecimaal uit?

De kracht van de V-woorden uit vorige hoofstukken zit in het feit dat de antwoorden zo nauwkeurig zijn als je maar wilt. Bereken bijvoorbeeld eens
4/7 + 4/9 - 1/63

```
) 100 DECIMALEN ! [rtn] ok
) 4 7 VV 2DUP V. [rtn] ?
) 4 9 VV V+ [rtn] ok
) 1 63 VV V- V. [rtn] ?
```

Een zwak punt is, dat er al gauw de situatie kan onstaan, dat teller of noemer niet meer in een cel passen.

```
) 100 1 VV [rtn] ok
 Herhaal nu enige malen de volgende regel:
) 2DUP V* 2DUP V. [rtn] ?
```

Ergens in de reeks, afhankelijk van de celgrootte, zal er een onzinnig antwoord verschijnen. Dat wordt veroorzaakt door een Overflow bij de vermenigvuldiging * die tweemaal voorkomt in V*.

```
) 100 [rtn] ok
 Herhaal nu enige malen de volgende regel:
) DUP * DUP . [rtn] ?
```

Ziehier de boosdoener. Het vervelende is, dat je achteraf niet kunt vaststellen of het goed of fout gegaan is. Toch is er hoop:

M* (x y -- zlo zhi) Vermenigvuldig x met y en zet het antwoord z als dubbelgetal op Stack. Er treedt nooit een Overflow op. De M staat voor Mixed. Bij dit woord zijn Single en Double Numbers (gemengd) betrokken.

* zou als volgt gedefiniëerd kunnen zijn.

```
: * ( x y -- z ) M* D>S ;
```

Een beveiligde versie:

```
) : SAFE* ( x y -- z )
)   M* OVER 0<
)   <> ABORT" Overflow " ;
```

Nieuw:
<> (x y -- vlag) Test of y ongelijk aan x is.
ABORT" ccc" (vlag --) False: geen aktie.
True: stap uit het programma en druk de tekst ccc af.

ingezonden brief

Geachte heer Nijhof,

Tijdens een vergadering van de VFW, de vakbond van Forth woorden, kwam uw kursus ter sprake. In positieve zin overigens. Maar, laat ik meteen ter zake komen.

In hoofdstuk 4, bij de bespreking van het woord PILS, schrijft u dat bij het maken van definities de woorden in een lijst achter de nieuwe naam komen te staan. Dat is een wel erg simpele voorstelling van zaken, die sommigen van ons tekort doet. Het is inderdaad waar, dat de meeste Forth woorden zich zo passief opstellen, dat ze zich tijdens het Definiëren zonder meer in een tabel laten opnemen.
Echter, de leden van mijn afdeling, de Immediate words, denken er anders over. Wij laten ons niet zo maar in een tabel zetten. Sterker nog, sommigen van ons, waaronder ikzelf (mijn naam is Dot Quote), zijn van de Compile groep; wij zelf houden ons bezig met het samenstellen van die lijst. Persoonlijk heb ik bijvoorbeeld de taak om, als ik opgeroepen wordt, een typiste in de tabel te zetten, daarna de tekst die binnenkomt te doorzoeken tot ik een aanhalingsteken vind, en vervolgens dat stuk tekst achter de typiste te plaatsen. Wij noemen dat Compileren.

(Mag ik tussendoor even opmerken dat programmeurs erg slordig kunnen zijn. Het komt nl. voor dat er geen aanhalingsteken te bekennen is. Daar zeur ik dan niet over. Ik neem gewoon de gehele resterende tekst op. Men zij gewaarschuwd. Misschien kunt u daar in uw kursus iets over zeggen.)

Het is de bedoeling dat bij het uitvoeren van het nieuwe woord de typiste de tekst typt. Ik heb dus mijn ondergeschikten. Mijn werk is wel iets ingewikkelder dan u suggereert!
In mijn vorige werkkring had ik bovendien tot taak om, als ik buiten een definitie om opgeroepen werd, de tekst direkt zelf uit te typen. De Klantenservice had daar een handje van, en men had er helaas ook altijd haast. Maar daar waren destijds meer dingen niet goed geregeld.
Gelukkig hoef ik dat nu niet meer. Mijn kollega, **.** **(** (Dot Paren) heeft die taak van mij overgenomen, met dit verschil dat zij naar een haakje sluiten zoekt in plaats van naar een aanhalingsteken. Ik gun ieder zijn eigenaardigheden. Zij is wel een Immediate word, in dienst van de Klantenservice, en een snelle typiste, maar van Compileren heeft ze geen verstand. En ik ben nu Only Compiling.

Mocht u meer willen weten over de verschillende Immediate woorden, dan raad ik u aan om ze persoonlijk te raadplegen. Zij zijn gaarne bereid om u inzicht te verschaffen in hun werkzaamheden.

Hoogachtend,

[Namens de Immediate words:] **."** (Dot Quote)

```
) : DOT-QUOTE ." waterval " ; [rtn] ?
) SEE DOT-QUOTE [rtn] ?
) DOT-QUOTE [rtn] ?
)
) : DOT-PAREN .( Tot Ziens! ) ; [rtn] ?
) SEE DOT-PAREN [rtn] ?
) DOT-PAREN [rtn] ?
```

De programmatekst die Forth te verwerken krijgt, heet de invoerstroom.
`."` en `.(` lezen zelf tekst van de invoerstroom. Omdat ze Immediate zijn, doen ze dat ook tijdens het definiëren.

`CHAR ( ccc -- k )` Lees het volgende woord van de invoerstroom en zet de ASCII code van de eerste letter op Stack.

```
) CHAR A . [rtn] ?
) CHAR a . [rtn] ?
) CHAR . . [rtn] ?
```

Omdat `CHAR` niet Immediate is, doet hij tijdens het definiëren nog niets. Pas als het nieuwe woord uitgevoerd wordt, leest `CHAR` de invoerstroom.

```
) : LETTER ( ccc -- )
)   ." heeft ASCII code "
)   CHAR . ;
)   LETTER A [rtn] ?
)   LETTER a [rtn] ?
```

`CHAR` heeft een variant die `[CHAR]` heet.

`[CHAR] ( ccc -- )` Zet tijdens het compileren de ASCII code van de eerste letter van `ccc` als een getal in de definitie. Omdat `[CHAR]` Immediate is, komt hij tijdens het compileren in aktie. Buiten een definitie heeft hij geen betekenis.

```
) : A-KODE ( -- k )
)   65 ;
) : KODE-A ( -- k )
)   [CHAR] A ;
) SEE A-KODE [rtn] ?
) SEE KODE-A [rtn] ?
```

Voor lezers die geen `SEE` in hun Forth hebben: `A-KODE` en `KODE-A` leveren niet alleen hetzelfde resultaat, ze zijn ook hetzelfde gebouwd.

`[CHAR] A`

zet tijdens het definiëren het getal 65 in de definitie.

Je kunt zelf natuurlijk ook Immediate woorden maken.
`IMMEDIATE ( -- )` Maak het laatst gedefiniëerde woord Immediate.

```
) : [.S] ( -- )
)   .S ; IMMEDIATE
```

`.S` laat de getallen op Stack zien. `[.S]` doet hetzelfde, maar dan tijdens het compileren, en laat geen sporen na in de definitie.

```
) 5 5555 [rtn] ok
) : TEST1 .S ; [rtn] ?
) : TEST2 [.S] ; [rtn] ?
) TEST1 [rtn] ?
) TEST2 [rtn] ?
) .S [rtn] ?
) [.S] [rtn] ?
) SEE TEST1 [rtn] ?
) SEE TEST2 [rtn] ?
```

? Kun je aannemelijk maken waarom `;` een Immediate woord zal zijn?

Samenvattend:
1) Immediate woorden zijn woorden die zich niet laten compileren. Zij worden tijdens het definiëren uitgevoerd.
2) Compiler woorden zijn woorden die tijdens het definiëren (compileren) invloed op het nieuw te maken woord hebben. Om dat te kunnen moeten ze wel Immediate zijn. Buiten een definitie zijn ze zinloos.
`."` is een Compiler woord, en dus ook Immediate. Only Compiling.
`.(` is Immediate, maar geen Compiler woord en heeft geen effekt op definities.

`IF ELSE THEN BEGIN UNTIL WHILE REPEAT DO LOOP` enz. noemt men besturingswoorden. Zij organiseren tijdens het compileren de vertakkingen. Besturingswoorden zijn dus Compiler woorden, onbruikbaar buiten definities.

Nieuwe besturingswoorden.
`?DO ( n2 n1 -- )` Bijna hetzelfde als `DO`.
Het verschil is dat `?DO` naar het eerste woord achter `LOOP` springt als `n2` gelijk is aan `n1`.
Deze uitleg van `?DO` is heel slordig. Juister is:
`?DO ( -- )` (compilerend:) Installeer `?LUSJESMAN` in de definitie.
`?LUSJESMAN ( n2 n1 -- )` (uitvoerend:) Begin aan een lus, tenzij `n1=n2`.

`LEAVE ( -- )` Ga naar het eerste woord achter `LOOP` (alleen tussen `DO` en `LOOP` te gebruiken).

Woordspel.
Speler A verzint een woord dat speler B moet raden. Na iedere poging krijgt speler B te zien hoeveel letters van zijn woord in het geheime woord voorkomen en hoeveel er al op de juiste plaats staan.

Het geheime woord mag hoogstens 31 letters lang zijn.

```
) #31 CONSTANT MAXLEN
) CREATE GEHEIM$ MAXLEN 1+ CHARS ALLOT
```

Dit is de ruimte waar het geheime woord, voorafgegaan door een Count, komt te staan.

Het geheime woord wordt met hoofdletters ingetypt zonder dat het te zien is op het scherm.

```
) : GEHEIM ( geheim$ maxlen -- )
)   OVER SWAP
)   0 DO
)       KEY >R
)       [CHAR] Z R@ <
)       R@ [CHAR] A <
)       OR IF R> DROP LEAVE THEN ( geen hoofdletter )
)       CHAR+
)       R> OVER C!             ( noteer deze letter )
)       [CHAR] * EMIT
)    LOOP
)    OVER -
)    SWAP C! ;          ( noteer het aantal letters )
```

? Onder welke omstandigheden wordt de Do-Loop in `GEHEIM` beëindigd?

Nieuw:
`SPACE ( -- )` Druk een spatie af.
`SPACES ( n -- )` Druk `n` spaties af.

```
) : SPELER-A ( -- )
)   ." Het geheime woord:"
)   CR 4 SPACES
)   GEHEIM$ MAXLEN GEHEIM
)   SPACE ;
```

[wordt vervolgd]

```
) CREATE GOK$   MAXLEN 1+ CHARS ALLOT
```

Ruimte voor de gok van speler B (ook een getelde string).

```
) : SPELER-B ( -- )
)   GOK$ CHAR+ GEHEIM$ C@
)   ACCEPT
)   GOK$ C!
)   SPACE ;
)
) VARIABLE TELLER
```

`TELLER` is voor intern gebruik in de woorden `LETTERS?` en `PLAATSEN?`.

Bepaal hoeveel letters van `GOK$` ook in `GEHEIM$` voorkomen:

```
) : LETTERS? ( gok$ geheim$ -- n )
)   0 TELLER !
)   SWAP
)   COUNT 0
)   ?DO OVER
)       COUNT 0
)       ?DO OVER J CHARS + C@
)           OVER I CHARS + C@
)           = IF 1 TELLER +! LEAVE THEN
)       LOOP DROP
)    LOOP DROP
)    DROP TELLER @ ;
```

Wanneer twee Do-Loop's genest zijn, geeft `I` de teller van de binnenste, en `J` de teller van de buitenste lus.

Nieuw:
`ROT ( x y z -- y z x )`

Bepaal hoeveel letters in `GOK$` al op de juiste plaats staan:

```
) : PLAATSEN? ( gok$ geheim$ -- n )
)   0 TELLER !
)   CHAR+ SWAP
)   COUNT 0
)   ?DO COUNT ROT
)       COUNT ROT
)       = IF 1 TELLER +! THEN
)   LOOP 2DROP
)   TELLER @ ;
```

[wordt vervolgd]

Onderzoek de gok van speler B.

```
) : RESULTAAT ( -- plaatsen letters )
)   GOK$ GEHEIM$ PLAATSEN?
)   GOK$ GEHEIM$ LETTERS?
) ;
```

Speler B geeft het op door op de plaats van de eerste letter een vraagteken in te typen.

```
) : OPGEGEVEN? ( -- vlag )
)   GOK$ CHAR+ C@ [CHAR] ? =
) ;
```

Nieuw:
`PAGE ( -- )` Wis het gehele scherm en zet de cursor linksboven.

Nu het spel zelf.

```
) : SPEL ( -- )
)   PAGE SPELER-A
)   ." goede letters/op de juiste plaats"
)   1 1
)   DO CR I 3 .R SPACE
)      SPELER-B
)      OPGEGEVEN? IF ."  niet" LEAVE THEN
)      RESULTAAT
)      #12 .R                 ( aantal goede letters )
)      [CHAR] / EMIT
)      DUP .                ( aantal juiste plaatsen )
)      GEHEIM$ C@ = IF LEAVE THEN
)   LOOP ."  geraden "
)   GEHEIM$ COUNT TYPE ;
```

? Wat gebeurt er als speler A geen enkele hoofdletter ingetypt heeft?

Inhoud van het systematische over icht

De Stack

DEPTH .S DROP DUP SWAP OVER ROT 2DROP 2DUP 2SWAP 2OVER 2ROT NIP TUCK PICK ROLL ?DUP -ROT

`DEPTH ( -- n )`
`n` is het aantal cellen op Stack dat in gebruik is, `n` zelf niet meegeteld.

`.S ( -- )` Druk de inhoud van de Stack af zonder de Stack te veranderen.

Stack manipulaties

`DROP ( x -- )`

`DUP ( x -- x x )` "dupe"

`SWAP ( x y -- y x )`

`OVER ( x y -- x y x )`

`ROT ( x y z -- y z x )` "rote"

`2DROP ( x1 x2 -- )` "two drop"

`2DUP ( x1 x2 -- x1 x2 x1 x2 )` "two dupe"

`2SWAP ( x1 x2 y1 y2 -- y1 y2 x1 x2 )` "two swap"

`2OVER ( x1 x2 y1 y2 -- x1 x2 y1 y2 x1 x2 )` "two over"

`2ROT ( x1 x2 y1 y2 z1 z2 -- y1 y2 z1 z2 x1 x2 )` "two rote"

`NIP ( x y -- y )`

`TUCK ( x y -- y x y )`

PICK (x_n ... x_0 n -- x_n ... x_0 x_n)

ROLL (x_n ... x_0 n -- x_{n-1} ... x_0 x_n)

`?DUP ( x -- x x )` of `( x -- x )` "question dupe"
Dupliceer `x` alleen als hij niet nul is.

Soms vind je

`-ROT ( x y z -- z x y )`

De Return Stack

>R R> R@ 2>R 2R> 2R@ RDROP 2RDROP

Bij het binnenkomen en het verlaten van een Hi-level woord gebruikt Forth de Return Stack. Ook de lus-administratie van een Do-Loop staat vaak op de Return Stack.
De programmeur kan de Return Stack misbruiken om er getallen op te zetten of er vanaf te halen. Hij dient er voor te zorgen dat hij Forth daarbij niet in de wielen rijdt. `>R` en `R>` moeten altijd korrekt genest zijn ten opzichte van het gebruik dat Forth zelf van de Return Stack maakt.

Ik geef de situatie op de Return Stack aan met `( R: ... -- ... )`

`>R ( x -- ) ( R: -- x )` "to r" Only Compiling"
Zet `x` op de Return Stack.

`R> ( -- x ) ( R: x -- )` "r from" Only Compiling
Haal `x` van de Return Stack af.

`R@ ( -- x ) ( R: x -- x )` "r fetch" Only Compiling
Lees `x` van de Return Stack, zonder hem er vanaf te halen.

Analoog (let op de volgorde van de cellen):

`2>R ( x1 x2 -- ) ( R: -- x1 x2 )` "two to r" Only Compiling

`2R> ( -- x1 x2 ) ( R: x1 x2 -- )` "two r from" Only Compiling

`2R@ ( -- x1 x2 ) ( R: x1 x2 -- x1 x2 )` "two r fetch" Only Compiling

Soms vind je

`RDROP ( -- ) ( R: x -- )` "r drop" Only Compiling

`2RDROP ( -- ) ( R: x1 x2 -- )` "two r drop" Only Compiling

Schrijven naar adressen

! C! 2! +! ACCEPT FILL MOVE

! (x a --) "store"
Zet x in de cel met adres a.

C! (k a --) "c store"
Zet k als karakter (kap het voorste stuk eraf) op adres a.

2! (x y a --) "two store"
Zet y in adres a en x in de cel daar achter.

+! (x a --) "plus store"
Verhoog de waarde in de cel op adres a met x (Signed of Unsigned).
a moet Aligned zijn.

ACCEPT (a n -- n2)
Ontvang hoogstens n karakters via het toetsenbord en noteer die aaneengesloten op adres a.
n moet groter dan nul zijn en n2 is het aantal werkelijk ontvangen karakters. De Return toets sluit ACCEPT af, telt niet mee, en wordt ook niet genoteerd.

FILL (a n k --)
Vul n karaktercellen vanaf adres a met karakters k.
Bij n=0 gebeurt er niets.

MOVE (a1 a2 u --)
Kopieer het gebied van u bytes vanaf adres a1 naar het gebied van u bytes vanaf adres a2.
Deze operatie gaat ook goed als a2 binnen het te kopieren gebied ligt. u is Unsigned.

Lezen uit adressen

@ 2@ C@ COUNT C@+ @+

@ (a -- x) "fetch"
Zet de inhoud van de cel met adres a op Stack.

2@ (a -- y x) "two fetch"
Zet de inhoud van de 2 cellen vanaf adres a (x,y) op Stack, eerst y dan x (2@ is de tegenhanger van 2!).

C@ (a -- k) "c fetch"
Lees het karakter in adres a en zet het op Stack. Het bitpatroon wordt aan de voorkant aangevuld met nullen om op celbreedte te komen.

COUNT (a -- a2 k)
k is de ASCII code van het karakter in adres a en a2 is één karakter verder dan a.

Soms vind je

C@+ (a -- a2 k) "c fetch plus"
Synoniem met COUNT.

@+ (a -- a2 x) "fetch plus"
x is de inhoud van de cel op adres a en a2 is één cel verder dan a.

Adressen

ALIGNED CELL+ CELLS CHAR+ CHARS CELL- CHAR-

ALIGNED (a -- a2)
Het adres a wordt, indien nodig, verhoogd tot er een adres ontstaat waar een cel kan beginnen. Dit kan nodig zijn na CHAR+ of CHAR-.

CELL+ (a -- a2) "cell plus"
Het adres a2 is één cel verder dan a.

CELLS (x -- y)
x cellen nemen y bytes in beslag.

CHAR+ (a -- a2) "char plus"
Het adres a2 is één karakter verder dan a.

CHARS (x -- y) "chars"
x karakters nemen y bytes in beslag.

Soms vind je

CELL- (a -- a2) "cell minus"
Het adres a2 is één cel voor a.

CHAR- (a -- a2) "char minus"
Het adres a2 is één karakter voor a.

Rondom HERE

HERE ALIGN , C, ALLOT

HERE (-- a)
Adres a is het einde van het woordenboek en het begin van het vrije geheugen (de Data Space). HERE is niet geschikt om er gegevens te bewaren omdat Forth dat adres intensief gebruikt. Zie PAD.

ALIGN (--)
Als er bij HERE geen cel kan beginnen, verplaats HERE dan naar het eerstvolgende adres waar dat wel kan. Dit kan nodig zijn na C, of ALLOT.

, (x --) "comma"
Reserveer één cel bij HERE en zet er x in. De programmeur dient er voor te zorgen dat er bij HERE een cel kan beginnen.

C, (k --) "c comma"
Reserveer ruimte voor één karakter bij HERE en zet k daarin.

ALLOT (x --)
Reserveer x bytes bij HERE. De nieuwe HERE wordt gelijk aan HERE+x.
x mag negatief zijn.

Invoer van het toetsenbord

ACCEPT KEY KEY? EKEY EKEY? EKEY>CHAR

ACCEPT (a n -- n2)
Ontvang hoogstens n ASCII codes via het toetsenbord en noteer die aaneengesloten op adres a.
n moet groter dan nul zijn en n2 is het aantal werkelijk ontvangen karakters. De Returntoets sluit ACCEPT af, telt niet mee, en wordt ook niet genoteerd.

KEY (-- k)
Wacht tot er een toets ingedrukt is. k is de ASCII code van die toets. Druk niets af.

KEY? (-- vlag) "key question"
Test of er een toets ingedrukt is. Bij een True vlag kan met KEY de ASCII code opgehaald worden.

De woorden KEY en KEY? gaan op voor toetsen die een ASCII code hebben van 32 tot en met 126 ($20 t/m $7E). Bij ASCII codes buiten dit gebied hangt het van de implementatie af welke toetsen wel of niet opgepikt worden.
De onderstaande EKEY-woorden werken met een implementatie-afhankelijk codegetal (ia) voor de toetsen. Daarmee kunnen ook toetsen geidentificeerd worden die geen ASCII code hebben.

EKEY (-- ia) "e key"
Wacht tot er een toets ingedrukt is. Het codegetal van die toets is ia. Druk niets af.

EKEY? (-- vlag) "e key question"
Test of er een toets ingedrukt is. Bij een True vlag kan met EKEY het codegetal opgehaald worden.

EKEY>CHAR (ia -- ia false) of (ia -- k true) "e key to char"
Zet, als dat kan, het codegetal om in een ASCII code.

Van cijferreeks naar bitpatroon

BASE >NUMBER DECIMAL HEX BINARY

BASE (-- a)
In de cel met adres a staat het grondtal waarmee Forth rekent bij het omzetten van bitpatronen naar getallen en omgekeerd. Zie >NUMBER en <#.

>NUMBER (ulo uhi a n -- vlo vhi a2 n2) "to number"
Analyseer de string a,n karakter voor karakter, vooraan te beginnen. Staat er een geldig cijfer (dat hangt van BASE af), vermenigvuldig dan uhi,ulo met de waarde van BASE , en tel de waarde van het cijfer er bij op. Stop als er geen cijfer staat. a2,n2 is de string die er over blijft, beginnend met het niet-cijfer. Als n2=0 dan is de gehele string omgezet.

DECIMAL (--)
Zet BASE op tien.

HEX (--)
Zet BASE op zestien.

Soms vind je

BINARY (--)
Zet BASE op twee.

Uitvoer

PAGE CR AT-XY SPACE SPACES EMIT TYPE .(

PAGE (--)
Wis het scherm en zet de cursor linksboven (formfeed bij printer).

CR (--) "c r"
Ga naar het begin van een nieuwe regel.

AT-XY (x y --) "at x y"
Begin de eerstvolgende tekstuitvoer naar het scherm op positie x van regel y.
Bij x=0 en y=0 is dat linksbovenaan.

SPACE (--)
Druk een spatie af.

SPACES (n --)
Druk, als n groter dan nul is, n spaties af.

EMIT (k --)
Druk het karakter af met ASCII code k.

TYPE (a u --)
Druk, als u niet nul is, de tekst a,u af.

.(ccc (--) "dot paren" Immediate woord
Lees de invoerstroom tot en met het eerstkomende haakje sluiten en druk, ook tijdens het compileren, die tekst af (zonder het haakje sluiten).

Uitvoer van getallen

. .R U. U.R D. D.R DU. DU.R

. (x --) "dot"
Druk x af als een Signed getal met een spatie erachter.

.R (x n --) "dot r"
Druk x af als een Signed getal, rechtsgericht in een veld van n posities.

U. (u --) "u dot"
Druk u af als een Unsigned getal met een spatie erachter.

U.R (u n --) "u dot r"
Druk u af als een Unsigned getal, rechtsgericht in een veld van n posities.

D. (xlo xhi --) "d dot"
Druk xhi,xlo af als een Signed dubbelgetal met een spatie erachter.

D.R (xlo xhi n --) "d dot r"
Druk xhi,xlo af als een Signed dubbelgetal, rechtsgericht in een veld van n posities.

Soms vind je

DU. (ulo uhi --) "d u dot"
Druk uhi,ulo af als een Unsigned dubbelgetal met een spatie erachter.

DU.R (ulo uhi n --) "d dot r"
Druk uhi,ulo af als een Unsigned dubbelgetal, rechtsgericht in een veld van n posities.

Van bitpatroon naar cijferreeks

<# # #S HOLD SIGN #>

<# (ulo uhi -- ulo uhi) "less number sign"
Voorbereiding voor het omzetten van het bitpatroon van uhi,ulo (Unsigned) naar een leesbaar getal. Tijdens het omzetten kunnen # #S HOLD en SIGN hun werk doen en #> sluit het omzetten af.

(ulo uhi -- vlo vhi) "number sign"
Bepaal, gebruik makend van BASE , het laatste cijfer van het bitpatroon uhi,ulo en plaats dat vooraan de string in wording. vhi,vlo is uhi,ulo zonder zijn laatste cijfer. Alleen te gebruiken tussen <# en #>.

#S (ulo uhi -- 0 0) "number sign s"
Voer een # uit en herhaal dit totdat het dubbelgetal op Stack 0,0 is. Alleen te gebruiken tussen <# en #>.

HOLD (k --)
Voeg vooraan de string in wording het karakter k toe. Alleen te gebruiken tussen <# en #>.

SIGN (x --)
Voeg, alleen als x negatief is, vooraan de string in wording een minteken toe. Alleen te gebruiken tussen <# en #>.

#> (ulo uhi -- a n) "number sign greater"
Sluit het omzetten af. De string a,n is het resultaat.

Besturing I

IF THEN
IF ELSE THEN

IF (-- sys1) Compiler woord
Compilerend: Compileer een voorwaardelijke sprong en laat een boodschap achter voor ELSE of THEN
Uitvoerend: (vlag --)
Als vlag niet nul is, ga dan gewoon door.
Als vlag=0 ga dan verder achter ELSE (of achter THEN als er geen ELSE is).

ELSE (sys1 -- sys2) Compiler woord
Compilerend: Compileer een sprong, laat een boodschap achter voor THEN en vul met behulp van sys1 het doel van de IF-sprong in.
Uitvoerend: (--)
Ga verder achter THEN.

THEN (sys1 --) Compiler woord
Compilerend: Vul met behulp van sys1 het doel van de IF-sprong of de ELSE-sprong in.
Uitvoerend: (--)
Geen aktie.

(-- sys1) betekent dat er een boodschap achtergelaten wordt en
(sys1 --) betekent dat er een boodschap verwerkt wordt.
Dat kan via de Stack gaan, maar er zijn andere mogelijkheden.
Deze boodschappen zijn nodig omdat bij het compileren van een sprong het doel van de sprong nog niet bekend is (IF), of omdat het doel al bekend is voordat de sprong gecompileerd wordt (BEGIN).

Lussen zonder teller

BEGIN UNTIL
BEGIN WHILE REPEAT
BEGIN AGAIN

BEGIN (-- sys1) Compiler woord
Compilerend: Laat een boodschap achter.
Uitvoerend: (--)
Geen aktie.

UNTIL (sys1 --) Compiler woord
Compilerend: Compileer een voorwaardelijke sprong met behulp van sys1.
Uitvoerend: (vlag --)
Als vlag niet nul is, ga dan gewoon door achter UNTIL.
Als vlag=0 ga dan terug naar BEGIN.

WHILE (sys1 -- sys2 sys1) Compiler woord
Compilerend: Compileer een voorwaardelijke sprong, laat een boodschap achter en verwissel de volgorde van de laatste twee boodschappen (1 CS-ROLL).
Uitvoerend: (vlag --)
Als vlag niet nul is, ga dan gewoon door achter WHILE.
Als vlag=0 ga dan verder achter REPEAT.

REPEAT (sys2 sys1 --) Compiler woord
Compilerend: Compileer met behulp van sys1 een sprong naar BEGIN.
Vul met behulp van sys2 het doel van de WHILE-sprong in.
Uitvoerend: (--)
Ga terug naar BEGIN.

AGAIN (sys1 --) Compiler woord
Compilerend: Compileer met behulp van sys1 een sprong naar BEGIN.
Uitvoerend: (--)
Ga terug naar BEGIN.

Evenals in hoofdstuk 112 duidt het woordje sys1 op een boodschappensysteem al dan niet via de Stack.

Tellende lussen

DO ?DO LEAVE LOOP +LOOP I J

DO (-- sys1) Compiler woord
Compilerend: Compileer een administrateur voor een lus met teller en laat een boodschap achter.
Uitvoerend: (grens teller --)
Noteer de lusgegevens (meestal op de Return Stack).

?DO (-- sys1) "question do" Compiler woord
Compilerend: Compileer een administrateur voor een voorwaardelijke lus met teller en laat een boodschap achter.
Uitvoerend: (grens teller --)
Als teller=grens ga dan verder achter LOOP.
In de andere gevallen: noteer de lusgegevens (als bij DO).

LOOP (sys1 --) Compiler woord
Compilerend: Compileer met behulp van sys1 een verhoger.
Uitvoerend: (--)
Verhoog teller met 1 .
Ga terug naar het eerste woord achter DO behalve wanneer teller dezelfde waarde als grens gekregen heeft. Verlaat in dat geval de lus en ruim de lus-administratie op.

+LOOP (sys1 --) "plus loop" Compiler woord
Compilerend: Compileer met behulp van sys1 een opteller.
Uitvoerend: (x --)
Tel x op bij de teller (Signed).
Ga terug naar het eerste woord achter DO behalve wanneer teller de scheiding tussen grens en grens-1 gepasseerd is. Verlaat in dat geval de lus en ruim de lus-administratie op.

LEAVE (--) Compiler woord
Uitvoerend: (--)
Ga verder achter LOOP ongeacht de waarden van teller en grens en ruim de lus-administratie op.

I (-- x) "i" Only Compiling
Zet de teller op Stack. Alleen te gebruiken in een Do-Loop.

J (-- x) "j" Only Compiling
Zet de teller van de buitenste lus op Stack. Alleen te gebruiken binnenin twee geneste Do-Loop's in één definitie.

Besturing II

AHEAD UNLOOP CASE OF ENDOF ENDCASE CS-PICK CS-ROLL

AHEAD (-- sys1) Compiler woord
Compilerend: Compileer een voorwaartse sprong en laat een boodschap achter voor THEN.
Uitvoerend: (--)
Ga verder achter THEN. AHEAD is een onderdeel van ELSE.

UNLOOP (--) Compiler woord
Ruim de lus-administratie op.

CASE (-- sys1) Compiler woord
Compilerend: Laat een boodschap achter voor ENDCASE.
Uitvoerend: (--)
Geen aktie.

OF (sys1 -- sys1 sys2) Compiler woord
Compilerend: Compileer een voorwaardelijke sprong en laat een boodschap achter voor ENDOF.
Uitvoerend: (x y -- x) of (x y --)
Als x<>y zet dan x weer op Stack en ga dan verder achter ENDOF. Als x=y ga dan gewoon door achter OF.

ENDOF (sys1 sys2 -- sys3 sys1) Compiler woord
Compilerend: Compileer een voorwaartse sprong, laat een boodschap achter voor ENDCASE en vul met behulp van sys2 het doel van de OF-sprong in. Verwissel de volgorde van de laatste twee boodschappen (1 CS-ROLL).
Uitvoerend: (--)
Ga verder achter ENDCASE.

ENDCASE (sys3 ... sys1 --) Compiler woord
Compilerend: Vul alle openstaande ENDOF-sprongen in.
Uitvoerend: (x --)
Verwijder x van de Stack.

CS-PICK (? n -- ?) "c s pick"
Voer een PICK uit op de reeks onafgehandelde boodschappen.

CS-ROLL (? n -- ?) "c s roll"
Voer een ROLL uit met de reeks onafgehandelde boodschappen.

Bits

AND OR XOR INVERT LSHIFT RSHIFT

`AND ( g1 g2 -- h )`
`h` ontstaat door bit voor bit een logische And uit te voeren op `g1` en `g2`.

`OR ( g1 g2 -- h )`
`h` ontstaat door bit voor bit een logische Or uit te voeren op `g1` en `g2`.

`XOR ( g1 g2 -- h )` "x or"
`h` ontstaat door bit voor bit een logische Xor uit te voeren op `g1` en `g2`.

`INVERT ( g -- h )`
Keer alle bits van `g` om, dus
`g h AND` levert False, en
`g h OR` levert True.

`LSHIFT ( g n -- h )` "l shift"
Alle bits van `g` schuiven `n` plaatsen naar links. Op de vrijkomende plaatsen komen nullen. Bits die eruit vallen gaan verloren.

`RSHIFT ( g n -- h )` "r shift"
Alle bits van `g` schuiven `n` plaatsen naar rechts. Op de vrijkomende plaatsen komen nullen. Bits die eruit vallen gaan verloren.

Vlaggen

TRUE FALSE 0= 0< 0> 0<> = < > <>

TRUE (-- x)
Alle bits van x zijn gezet, dus x=-1.

FALSE (-- x)
Alle bits van x zijn nul, dus x=0.

0= (x -- vlag) "zero equals"
vlag is alleen True als x=0.

0< (x -- vlag) "zero less"
vlag is alleen True als x<0.

0> (x -- vlag) "zero greater"
vlag is alleen True als x>0.

0<> (x -- vlag) "zero not equals"
vlag is alleen True als x niet nul is.

= (x y -- vlag) "equals"
vlag is True alleen als x=y.

< (x y -- vlag) "less than"
vlag is alleen True als x<y.

> (x y -- vlag) "greater than"
vlag is True alleen als x>y.

<> (x y -- vlag) "not equals"
vlag is alleen True als x ongelijk aan y is.

Vlaggen II

U< U> D0= D0< D= D< WITHIN

`U< ( u v -- vlag )` "u less than"
`vlag` is alleen True als `u<v` (Unsigned).

`U> ( u v -- vlag )` "u greater than"
`vlag` is alleen True als `u>v` (Unsigned).

`D0= ( xlo xhi -- vlag )` "d zero equals"
`vlag` is alleen True als `xhi,xlo` gelijk aan nul is.

`D0< ( xlo xhi -- vlag )` "d zero less"
`vlag` is alleen True als `xhi,xlo` kleiner dan nul is.

`D= ( xlo xhi ylo yhi -- vlag )` "d equals"
`vlag` is alleen True als `xhi,xlo` gelijk aan `yhi,ylo` is.

`D< ( xlo xhi ylo yhi -- vlag )` "d less than"
`vlag` is alleen True als `xhi,xlo` kleiner dan `yhi,ylo` is.

`WITHIN ( x y1 y2 -- vlag )`
`vlag` is alleen True als `x-y1` kleiner is (Unsigned) dan `y2-y1`.
Anders gezegd:
`vlag` is alleen True als `x` ligt in het gebied `[y1...y2)`, met inbegrip van `y1` maar zonder `y2`.
Zo'n gebied `[y1...y2)` loopt circulair door de getallen (Signed of Unsigned, dat maakt niet uit): de twee gebieden G1 `[y1...y2)` en G2 `[y2...y1)` overlappen elkaar niet en bevatten samen ALLE getallen.

Max en Min

MAX MIN DMAX DMIN UMAX UMIN

`MAX ( x1 x2 -- y )`
`y` is gelijk aan de grootste van het paar `x1,x2`.

`MIN ( x1 x2 -- y )`
`y` is gelijk aan de kleinste van het paar `x1,x2`.

`DMAX ( xlo xhi ylo yhi -- zlo zhi )` "d max"
Als `MAX` maar dan met dubbelgetallen.

`DMIN ( xlo xhi ylo yhi -- zlo zhi )` "d min"
Als `MIN` maar dan met dubbelgetallen.

Soms vind je

`UMAX ( u1 u2 -- v )` "u max"
Als `MAX` maar dan met Unsigned getallen.

`UMIN ( u1 u2 -- v )` "u min"
Als `MIN` maar dan met Unsigned getallen.

Bewerkingen van één getal

1+ 1- ABS NEGATE DABS DNEGATE 2* 2/ D2* D2/

1+ (x -- y) "one plus"
y = x+1 Signed of Unsigned; Overflow wordt niet gesignaleerd.

1- (x -- y) "one minus"
y = x-1 Signed of Unsigned; Overflow wordt niet gesignaleerd.

ABS (x -- u) "abs"
u (Unsigned) is de absolute waarde van x (Signed).

NEGATE (x -- y)
Keer het teken van x om, dus y = 0-x.

DABS (xlo xhi -- ulo uhi) "d abs"
uhi,ulo (Unsigned) is de absolute waarde van xhi,xlo (Signed).

DNEGATE (xlo xhi -- ylo yhi) "d negate"
Keer het teken van xhi,xlo om.

2* (x -- y) "two star"
y = 2*x dwz. schuif het bitpatroon x een plaats naar links en zet een nul op de vrijgekomen plek.

2/ (x -- y) "two slash"
y = x/2 dwz. schuif het bitpatroon x een plaats naar rechts en geef de vrijgekomen plaats de waarde die hij voor het schuiven al had.

D2* (xlo xhi -- ylo yhi) "d two star"
Beschouw xhi,xlo als één lang bitpatroon, schuif het in zijn geheel een plaats naar links en zet een nul op de vrijgekomen plek.

D2/ (xlo xhi -- ylo yhi) "d two slash"
Beschouw xhi,xlo als één lang bitpatroon, schuif het in zijn geheel een plaats naar rechts en geef de vrijgekomen plek de waarde die hij voor het schuiven al had.

Rekenen

+ - * /MOD / MOD */MOD */

+ (x y -- z) "plus"
z is gelijk aan x+y. Overflow van z wordt niet gesignaleerd. x y en z zijn alle drie Signed of alle drie Unsigned.

- (x y - z) "minus"
z is gelijk aan x-y. x y en z zijn alle drie Signed of alle drie Unsigned. Overflow van z wordt niet gesignaleerd.

* (x y -- p) "star"
p is gelijk aan y maal x. Overflow van p wordt niet gesignaleerd.

/MOD (x y -- r q) "slash mod"
q is gelijk aan x gedeeld door y (Floored of Symmetrical). r is de rest die daarbij ontstaat.

/ (x y -- q) "slash"
Hetzelfde als /MOD NIP

MOD (x y -- r)
Hetzelfde als /MOD DROP

*/MOD (x y1 y2 -- r z) "star slash mod"
z is gelijk aan x maal y1 gedeeld door y2 (Floored of Symmetrical). r is gelijk aan de rest die daarbij ontstaat. Het tussenproduct x*y wordt als dubbelgetal berekend. Overflow van z wordt niet gesignaleerd.

*/ (x y1 y2 -- z) "star slash"
Hetzelfde als */MOD NIP

Rekenen met dubbelgetallen

D+ D- M* UM* FM/MOD SM/REM UM/MOD

D+ (xlo xhi ylo yhi -- zlo zhi) "d plus"
Tel twee dubbelgetallen op, Signed of Unsigned. Overflow van zhi,zlo wordt niet gesignaleerd.

D- (xlo xhi ylo yhi -- zlo zhi) "d minus"
Trek twee dubbelgetallen van elkaar af, Signed of Unsigned. Overflow van zhi,zlo wordt niet gesignaleerd.

M* (x y -- zlo zhi) "m star"
Vermenigvuldig twee getallen met elkaar en zet het resultaat als dubbelgetal op Stack.

UM* (u v -- wlo whi) "u m star"
Vermenigvuldig twee Unsigned getallen met elkaar en zet het resultaat als dubbelgetal op Stack.

FM/MOD (xlo xhi y -- r q) "f m slash mod"
Deel xhi,xlo door y (Floored Division). q is het quotient en r is de rest. Overflow van q wordt niet gesignaleerd.

SM/REM (xlo xhi y -- r q) "s m slash rem"
Deel xhi,xlo door y (Symmetrical Division). q is het quotient en r is de rest. Overflow van q wordt niet gesignaleerd.

UM/MOD (ulo uhi v -- r q) "u m slash mod"
Deel uhi,ulo door v (beide Unsigned). q is het quotient en r is de rest. Overflow van q wordt niet gesignaleerd.

De invoerstroom

TIB #TIB >IN WORD PARSE

TIB (-- a) "t i b"
TIB is het adres van de aktuele invoerbuffer. Het adres blijft niet steeds hetzelfde.

#TIB (-- a) "number t i b"
#TIB @ is de lengte van de aktuele invoerbuffer.

>IN (-- a) "to in"
>IN @ is de relatieve positie in de aktuele invoerbuffer waar het lezen van de invoerstroom voortgezet gaat worden.

WORD ccc (k -- a)
Lees de invoerstroom. Sla eerst alle karakters k over. Lees vervolgens de tekst tot en met het eerstvolgende karakter k of, als dat er niet is, tot het einde van de buffer. Zet >IN op de nu bereikte positie. De tekst (zonder inleidende en afsluitende karakters k) staat als een getelde string in adres a.
De opeenvolging BL WORD wordt gebruikt om namen van Forth woorden uit de invoerstroom op te vissen.

PARSE ccc (k -- a n)
Lees de invoerstroom tot en met het eerstvolgende karakter k of als dat er niet is, tot aan het einde van de buffer. Zet >IN op de nu bereikte positie. a,n is de gelezen tekst zonder het afsluitende karakter k.

Toepassingen van BL WORD

' [‘] POSTPONE [COMPILE] CHAR [CHAR] TO FORGET SEE

' ccc (-- s) "tick"
Lees een Forth woord uit de invoerstroom en bepaal zijn sleutel s.

[‘] ccc (--) "bracket tick" Compiler woord
Compilerend: Lees een Forth woord uit de invoerstroom, bepaal zijn sleutel en compileer die in de definitie.
Uitvoerend: (-- s)
Zet de sleutel op Stack.

POSTPONE ccc (--) Compiler woord
Compilerend: Lees een Forth woord uit de invoerstroom en compileer hem zodanig, dat het zich bij latere uitvoering gedraagt alsof er dan pas gecompileerd wordt.
Uitvoerend: (-- ?)
Handel alsof er nu pas gecompileerd wordt.

[COMPILE] ccc (--) "bracket compile" Compiler woord
Compilerend: Lees een Forth woord uit de invoerstroom en forceer dat dit woord gecompileerd wordt, ook al is het Immediate.
Uitvoerend: (--)
Voer ccc uit.

CHAR ccc (-- k)
Lees een woord uit de invoerstroom en zet de ASCII code van het eerste karakter op Stack.

[CHAR] ccc (--) "bracket char" Compiler woord
Compilerend: Lees een woord uit de invoerstroom en compileer de ASCII code van het eerste karakter.
Uitvoerend: (-- k)
Zet de ASCII code op Stack.

TO ccc (x --) Immediate woord
Lees een Forth woord uit de invoerstroom en geef het, als het een Value of een Local is, de waarde x.

FORGET ccc (--)
Lees een Forth woord uit de invoerstroom en verwijder dat woord, met alle nieuwere woorden, uit het woordenboek.

SEE ccc (--)
Lees een Forth woord uit de invoerstroom en geeft een analyse van dat woord.

Toepassingen van PARSE

(.(." S" ABORT"

(ccc (--) "paren" Immediate woord
Lees de invoerstroom tot en met het eerstkomende haakje sluiten en negeer die tekst, ook tijdens het compileren.

.(ccc (--) "dot paren" Immediate woord
Lees de invoerstroom tot en met het eerstkomende haakje sluiten en druk die tekst zonder het haakje sluiten af, ook tijdens het compileren.

." ccc (--) "dot quote" Compiler woord
Compilerend: Lees de invoerstroom tot en met het eerstkomende aanhalingsteken en compileer die tekst zonder het afsluitende aanhalingsteken.
Uitvoerend: (--)
Druk de tekst ccc af.

S" ccc (-- ?) "s quote"
a) Buiten definities (ccc - a n)
Lees de invoerstroom tot en met het eerstkomende aanhalingsteken, en verplaats die tekst zonder het aanhalingsteken naar een buffer (adres a). De lengte is n karakters.
b) Binnen definities (ccc --) Compiler woord
Compilerend: Lees de invoerstroom tot en met het eerstkomende aanhalingsteken, en compileer die tekst zonder aanhalingsteken in de definitie.
Uitvoerend: (-- a n) Zet adres en lengte van de tekst op Stack.

ABORT" ccc (--) "abort quote" Compiler woord
Compilerend: Lees de invoerstroom tot en met het eerstkomende aanhalingsteken en compileer die tekst zonder het afsluitende aanhalingsteken.
Uitvoerend: (vlag --)
Bij vlag=0 geen actie.
Bij vlag ongelijk aan nul: Breek het programma af en druk daarbij de tekst ccc af als boodschap.

Definiërende woorden

CREATE : ; CONSTANT VARIABLE VALUE

`CREATE ccc ( -- )`
Lees een naam uit de invoerstroom en definiëer een woord met die naam, waarvan de Body nog leeg is.
`ccc ( -- a )`
Zet het Body adres van `ccc` op Stack.

`: ccc ( -- sys1 )` "colon"
Lees een naam uit de invoerstroom en begin aan een Hi-level definitie van die naam. Laat een boodschap achter voor `;` (Semicolon). Het nieuwe woord kan pas gevonden worden in het woordenboek als het voltooid is door `;` (Semicolon).

`; ( sys1 -- )` "semicolon" Compiler woord
Compilerend: Compileer een woord om de definitie te verlaten, verwerk de boodschap van `:` (Colon), voltooi de definitie en stop met compileren.
Uitvoerend: `( -- )` Verlaat de definitie.

`CONSTANT ccc ( x -- )`
Lees een naam uit de invoerstroom en definiëer een konstante met die naam.
`ccc ( -- x )`
Zet `x` op Stack.

`VARIABLE ccc ( -- )`
Lees een naam uit de invoerstroom en definiëer een variabele met die naam.
`ccc ( -- a )`
Op adres `a` is ruimte voor de waarde van de variabele `ccc`.

`VALUE ccc ( x -- )`
Lees een naam uit de invoerstroom en definiëer een Value met de naam `ccc` en de waarde `x`.
`ccc ( -- x )`
Zet de waarde van de Value op Stack.
`TO ccc ( x -- )`
Zet de waarde `x` in Value `ccc`.

Compileren

[ ] COMPILE, LITERAL 2LITERAL RECURSE EXIT

`[` (--) "left bracket" Immediate woord
Stop met compileren. STATE wordt hierbij gewijzigd.

`]` (--) "right bracket"
Ga compileren. STATE wordt hierbij gewijzigd.

`COMPILE,` (s --) "compile comma"
Compileer het woord dat `s` als sleutel heeft in de definitie.

`LITERAL` (x --) Compiler woord
Compilerend: Compileer het getal `x`.
Uitvoerend: (-- x)
Zet `x` op Stack.

`2LITERAL` (xlo xhi --) Compiler woord
Compilerend: Compileer het dubbelgetal `xhi,xlo`.
Uitvoerend: (-- xlo xhi)
Zet het dubbelgetal op Stack.

`RECURSE` (--) Compiler woord
Tijdens het compileren wordt `RECURSE` gebruikt in plaats van de naam van het woord in wording. Die naam kan niet op de gewone manier gevonden worden omdat de definitie nog niet af is.

`EXIT` (--) Only Compiling
Uitvoerend: (--)
Verlaat de definitie.

Woorden definiëren die woorden definiëren

DOES> ;CODE

DOES> (sys1 -- sys2) "does" Compiler woord
Compilerend: Dit is het einde van de definitie en tegelijkertijd het begin van de definitie van een naamloze aktie.
Uitvoerend: (-- bodyadres)
Wijzig, alvorens de definitie te verlaten, een zojuist met CREATE gedefiniëerd woord zodanig, dat de naamloze aktie toegevoegd wordt aan de Code die CREATE al aan dat woord meegegeven had (en dat was: Body adres op Stack zetten).

;CODE (sys1 -- sys2) "semicolon code" Compiler woord
Compilerend: Dit is het einde van de definitie en tegelijkertijd het begin van Assemblercode voor een naamloze aktie.
Uitvoerend: (--)
Wijzig, alvorens de definitie te verlaten, een zojuist met CREATE gedefiniëerd woord zodanig, dat hij de naamloze aktie als Code meekrijgt in plaats van de aktie die CREATE aan dat woord meegegeven had.

De sleutel

' [‘] EXECUTE FIND >BODY >NAME >LINK BODY> NAME> LINK>

`' ccc ( -- s )` "tick"
Lees een woordnaam uit de invoerstroom en zet de sleutel van dat woord op Stack.

`['] ccc ( -- )` "bracket tick" Compiler woord
Compilerend: `ccc ( -- )` Voer een `'` (Tick) uit en compileer de gevonden sleutel.
Uitvoerend: `( -- s )`
Zet de sleutel op Stack.

`EXECUTE ( s -- )`
Voer het woord waarvan de sleutel `s` op Stack staat uit.

`FIND ( a -- a 0 )` of `( a -- s -1 )` of `( a -- s 1 )`
Zoek de sleutel `s` van het Forth woord waarvan de naam als een getelde string in adres `a` staat. Drie mogelijkheden:
`( a -- a 0 )` Het woord is niet te vinden.
`( a -- s -1 )` Het gevonden woord is niet Immediate.
`( a -- s 1 )` Het gevonden woord is Immediate.

`>BODY ( s -- a )` "to body"
`a` is het Body adres van het woord met sleutel `s` (alleen voor woorden die met `CREATE` gemaakt zijn).

Soms vind je

`BODY> ( Body-adres -- s )` "body from"
`NAME> ( Name-adres -- s )` "name from"

`>NAME ( s -- Name-adres )` "to name"

Woordenlijsten

FORTH DEFINITIONS ORDER WORDS PREVIOUS ALSO ONLY VOCABULARY

FORTH (--)
Vervang de eerste woordenlijst in de zoekvolgorde door FORTH. FORTH is een VOCABULARY. Zie onderaan deze bladzijde.

DEFINITIONS (--)
De eerste woordenlijst in de zoekvolgorde is de lijst waar vanaf nu de nieuwe definities in terecht komen.

ORDER (--)
Laat de zoekvolgorde (Vocabulary Stack) zien.

WORDS (--)
Laat de woorden zien van de eerste woordenlijst in de zoekvolgorde.

PREVIOUS (--)
Verwijder de eerste woordenlijst uit de zoekvolgorde.

ALSO (--)
Dupliceer de eerste woordenlijst in de zoekvolgorde.

ONLY (--)
Stel de basis zoekvolgorde in.

Soms vind je

VOCABULARY ccc (--)
Definiëer een nieuwe woordenlijst.
ccc (--)
Vervang de eerste woordenlijst in de zoekvolgorde door ccc.

Allerlei I

IMMEDIATE EVALUATE QUIT QUERY UNUSED

`IMMEDIATE ( -- )`
Maak het laatst gedefiniëerde woord Immediate, d.w.z. het jongste woord van de woordenlijst waar de nieuwe woorden in terecht komen.

`EVALUATE ( a n -- )`
Onderbreeek de lopende invoerstroom en beschouw de tekst `a,n` als de nieuwe invoerstroom. Ga, als deze is afgehandeld, terug naar de oude invoerstroom waar deze was blijven steken (geneste invoerstroom).

`QUIT ( -- )`
Maak de Return Stack leeg, ga in de niet-compileer toestand, en begin aan de volgende kringloop:
1) wacht op invoer van het toetsenbord,
2) verwerk de invoer,
3) druk `'ok'` af en ga naar 1).

De kringloop wordt slechts onderbroken bij een fout in 2)

`QUERY ( -- )`
Ontvang invoer van het toetsenbord, en maak de ingevoerde tekst tot invoerstroom.

`UNUSED ( -- u )`
`u` is het aantal bytes dat vrij is vanaf `HERE` .

Allerlei II

BL S>D D>S MS BLANK ERASE GET-DATE PAD

BL (-- 32) "b l"
32 is de ASCII code voor een spatie.

S>D (x -- xlo xhi) "s to d"
Vervang x door een dubbelgetal met dezelfde waarde.

D>S (xlo xhi -- x) "d to s"
Haal de voorste cel van het dubbelgetal xhi,xlo weg.

MS (n --) "m s"
Wacht n millisekonden.

BLANK (a n --)
Vul het geheugen vanaf adres a met n spaties.

ERASE (a n --)
Vul n karaktercellen vanaf adres a met ASCII code nul.

PAD (-- a)
PAD is het adres van een kladblokje voor de programmeur. PAD ligt meestal op een vaste afstand boven HERE wat tot gevolg heeft dat de weggeschreven gegevens beperkt houdbaar zijn. De lengte van het kladblokje is afhankelijk van de implementatie.

Fouten afhandelen

ABORT ABORT" CATCH THROW

ABORT (? --)
Stap uit het programma. Wis de Stack en de Return Stack, en voer QUIT uit. Druk hierbij niets af.

ABORT" ccc (--) "abort quote" Compiler woord
Compilerend: Lees de invoerstroom tot en met het eerstkomende aanhalingsteken en compileer die tekst zonder het afsluitende aanhalingsteken.
Uitvoerend: (vlag --)
Bij vlag=0: geen actie.
Bij vlag ongelijk aan nul: Breek het programma af en druk daarbij de tekst ccc af als boodschap.

CATCH (s -- ?? vlag)
Zie hoofdstuk 208 en 209.

THROW (vlag -- ?? vlag)
Zie hoofdstuk 208 en 209.

Losse artikelen

Signed en Unsigned getallen

Leo: Volgens mij zit er een fout in jouw Forth. Kijk eens wat er gebeurt:

```
2 4 MAX . [rtn] 4 ok
200 400 MAX . [rtn] 400 ok
```

Nog niks aan de hand. Maar nu:

```
20000 40000 MAX . [rtn] 20000 ok
```

Helemaal niet o.k. Dit is onzin! Ik weet wat `MAX ( x y - z )` moet doen. Het resultaat `z` is de grootste van het paar `x` en `y` en dat klopt hier echt niet. Is je dat nooit eerder opgevallen?

Theo: Tja, ik moet zeggen dat het er raar uitziet, en toch.. het ís correct.

Leo: Ik ben zelfs zo ijverig geweest om te proberen om `MAX` opnieuw en beter te definiëren:

```
: MIJNMAX ( x y - z )
  2DUP < IF NIP EXIT THEN DROP ;
```

Maar het gekke is dat die dezelfde fout maakt.

```
20000 40000 MIJNMAX . [rtn] 20000 ok
```

Theo: De fout hangt samen met de KLEINER-DAN.

Leo: Als je dat wel weet, waarom heb je hem dan niet verbeterd? Je bent toch altijd bezig om die Forth van jou sneller en beter te maken!

Theo: De KLEINER-DAN is niet fout, je hebt hem verkeerd gebruikt.

Leo: ??

Theo: Wat zeg je hier van:

```
20000 . [rtn]  20000 ok
40000 . [rtn]  -25536 ok
```

Leo: Je verwacht misschien dat dit verhelderend werkt?

Theo: En hiervan:

```
-25536 40000 - . [rtn]  0 ok
```

Leo: Als je twee verschillende getallen van elkaar aftrekt hoort er geen nul uit te komen, zou ik zo zeggen. Moet ik om Forth te leren ook nog mijn algemene inzichten in het rekenen met getallen bijstellen? Dat is te veel gevraagd.

Theo: Het zit zo:
Deze Forth werkt met de getallen van -32768 tot en met 32767. Dat komt doordat hij getallen opslaat in een patroon van 16 bits. Getallen die groter of kleiner zijn passen niet meer in die 16 bits, want je kunt met 16 bits hoogstens 65536 (= 2 in de macht 16) verschillende patronen maken.

Leo: Maar waarom slikt hij die 40000 dan toch?

Theo: Je hebt gelijk, in een andere programmertaal zou er waarschijnlijk een foutmelding komen. Toch heeft het zin dat hij die 40000 goedkeurt. Ik zal je uitleggen waarom.

Je kunt het hele getallengebied van -32768 via 0 naar 32767 doorlopen door er steeds 1 bij op te tellen. Aan de rand van het gebied gebeurt er dit:

```
32767 1 + . [rtn]  -32768 ok
```

Als je het grootste getal met 1 verhoogt komt het kleinste getal als antwoord te voorschijn! Er treedt een overflow op, d.w.z. er onstaat een bitpatroon van 17 bits, maar Forth laat die 17de positie gewoon weg waardoor de reeks patronen weer opnieuw begint en al de getallen als het ware op een cirkel komen te liggen. Gevolg: als je twee getallen bij elkaar optelt komt er altijd een antwoord uit. De programmeur moet zelf in de gaten houden of dat antwoord juist is, anders gezegd, zonder overflow tot stand gekomen is.
Naast dit systeem (de SIGNED getallen) bestaat er een tweede systeem (de UNSIGNED getallen) dat met precies dezelfde bitpatronen werkt. Hierin blijven de positieve getallen hetzelfde maar alle negatieve getallen worden over boord gezet om er aan de bovenkant als hoge positieve getallen weer bij geplakt te worden. De UNSIGNED getallen lopen van 0 naar 65535. Die hoge positieve getallen moeten ook ingevoerd kunnen worden en daarom werd 40000 dus geaccepteerd. Forth onthoudt alleen het getal als bitpatroon en niet de manier waarop het binnengekomen is.
Aan de randen van het gebied treedt weer het overflow effect op.

```
65535 1 + . [rtn]  0 ok
```

Maar nu terug:

```
0 1 - . [rtn]  -1 ok
```

Dit gaat fout, want.. de PUNT weet niet dat het hier om een UNSIGNED getal gaat.

```
65535 1 + U. [rtn]  0 ok
0 1 - U. [rtn]  65535 ok
```

De `+` en de `-` werken goed voor beide systemen. Voor afdrukken, maar ook voor vergelijken heb je aparte commando's nodig: `. U. < U< > U>`
En de oplossing voor je MAX -probleem is dus het definiëren van een `UMAX` met `U<` erin.

```
: UMAX ( ux uy - uz ) 2DUP U<
   IF NIP EXIT THEN DROP ;

20000 40000 UMAX U. [rtn]  40000 ok
```

Leo: Gaat dit in alle Forth's zo?

Theo: Ja, alleen de gebieden zijn soms anders. In een 32-bits Forth loopt SIGNED van -2147483648 naar 2147483647 en UNSIGNED van 0 naar 4294967295 (ruim vier miljard).
Drie weken geleden sprak ik iemand met een Forth waarin 100 groter is dan 200 en 127 plus 1 -128 is.
Wat voor Forth zou dat geweest zijn?

Over de Forth-woorden die getallen afdrukken.

Getallen afdrukken I

1. De acht punt-commando's

Ans-Forth, de nieuwe Amerikaanse Forth-standaard, kent de volgende zes woordjes voor het afdrukken van getallen:

```
.      ( x -- )           core
U.     ( u -- )           core
.R     ( x r -- )         core ext.
U.R    ( u r -- )         core ext.
D.     ( xlo xhi -- )     double
D.R    ( xlo xhi r -- )   double
```

De ANS-norm ordent de woorden in een aantal hoofdstukjes.
CORE is de aanduiding voor de Forth-kern.
CORE EXTENSIONS bevat woorden die als een direkte uitbreiding van de kern gezien worden.
De woorden in DOUBLE hebben betrekking op dubbelgetallen.
Alleen de woorden van de kern CORE zijn verplicht aanwezig, de andere zijn optioneel.

De letter `U` in de punt-commando's staat voor Unsigned, d.w.z. alle bits van het bitpatroon worden gebruikt om een positief getal samen te stellen. Bij het grootste Unsigned getal zijn alle bits gezet.
De commando's zonder U werken met Signed getallen. Het hoogste bit geeft daarbij aan of het om een positief of een negatief getal gaat.
Een bitpatroon kan dus op twee manieren als een getal vertaald worden naar een string.

```
) -1 . [rtn]
) -1 U. [rtn] \ grootste Unsigned getal
```

De commando's die dubbelgetallen behandelen beginnen met de letter `D`. Twee getallen op Stack vormen samen een dubbelgetal. Het bereik wordt daardoor veel groter. Om het grootste Unsigned dubbelgetal af te drukken is een commando `DU.` nodig dat in ANS-Forth ontbreekt.

```
) -1. D. [rtn]
) -1. DU. [rtn] \ grootste Unsigned dubbelgetal
```

De letter R in een comando geeft aan dat het getal 'rechtsgericht in een veld van r posities' afgedrukt wordt.

```
) -1  33 .R [rtn]
) -1  32 U.R [rtn]
) -1. 31 D.R [rtn]
) -1. 30 DU.R [rtn]
```

Ook DU.R wordt niet genoemd in de standaard.

2. Definities voor de acht punt-commando's

Het afdrukken van getallen gaat in twee fases:

A. Het omzetten (Signed of Unsigned) van het bitpatroon op Stack in een String;

B. Het afdrukken van de String, al dan niet 'rechtsgericht'.

Voor fase A maak ik twee woordjes:

```
) : DU.STRING (ulo uhi -- adr len )
)   <# #S #> ;
)
) : D.STRING ( xlo xhi -- adr len )
)   TUCK     \ Onthoud het teken
)   DABS
)   <# #S
)   ROT SIGN \ Zet eventueel een minteken
)            \ voor de String
)   #> ;
```

<# (ulo uhi -- ulo uhi) "less number sign"
Reserveer een buffer voor het omzetten van het bitpatroon uhi,ulo naar een leesbare String. Tijdens het omzetten kunnen #S en SIGN hun werk doen en #> sluit het omzetten af.

#S (ulo uhi -- 0 0) "number sign S"
Zet het dubbelgetal op Stack om in een string. (Alleen bruikbaar tussen <# en #>.)

#> (ulo uhi -- adr len) "number sign greater"
Sluit het omzetten af. De String adr,len is het resultaat van de omzetting.

SIGN (x --)
Voeg, alleen als x negatief is, vooraan de String in wording een minteken toe. (Alleen bruikbaar tussen <# en #>.)

In fase B heb ik naast `TYPE` een woord nodig dat een String rechtsgericht afdrukt, voor het maken van kolommen bijvoorbeeld.

```
) : RTYPE ( adr len r -- )
)   OVER - SPACES
)   TYPE ;
```

Nu is het eenvoudig om de acht punt-commando's te definiëren. Eerst de dubbelgetallen:

```
) : D.   ( xlo xhi -- )   D.STRING TYPE SPACE ;
) : DU.  ( ulo uhi -- )   DU.STRING TYPE SPACE ;
) : D.R  ( xlo xhi r -- ) >R D.STRING R> RTYPE ;
) : DU.R ( ulo uhi r -- ) >R DU.STRING R> RTYPE ;
```

Daarna de enkele getallen:

```
) : .    ( x -- )    S>D D. ;
) : U.   ( u -- )    0 DU. ;
) : .R   ( x r -- )  >R S>D R> D.R ;
) : U.R  ( u r -- )  >R 0 R> DU.R ;
```

[wordt vervolgd]

Over PICTURE om getallen af te drukken in een bepaald formaat en over HEXA BINA en DECI om even in een ander talstelsen iets af te drukken.

Getallen afdrukken II

3. PICTURE of het afdrukken door een sjabloon

Soms wil je getallen afdrukken in een formaat dat de punt-commando's niet aankunnen. Dat kun je dan altijd zelf programmeren met de woordjes <# # SIGN HOLD #S en #> maar het is handig om een flexibel woord te hebben dat zoiets voor elkaar krijgt. Ik stel voor om dat woord PICTURE te noemen.

```
PICTURE ( ulo uhi adr1 len1 -- adr2 len2 )
```

Het verwacht op Stack een dubbelgetal en bovendien adres en lengte van de String die als een sjabloon of mal aangeeft hoe het dubbelgetal in een String omgezet moet wordt.

Een paar voorbeelden. In de eerste kolom staan de sjabloon-strings, in kolom twee en drie het effekt ervan bij toepassing op de dubbelgetallen 1234.5678 en 9.5

sjabloon-string	1234.5678	9.5
S" #####"	45678	00095
S" F#.#.#.#"	F5.6.7.8	F0.0.9.5
S" ##/##/##"	34/56/78	00/00/95
S" ? # #"	123456 7 8	0 9 5
S" -#+"	-8+	-5+
S" ##,# km"	67,8 km	09,5 km
S" DM ?.##"	DM 123456.78	DM 0.95

De speciale tekens in de mal zijn het hekje en het vraagteken:
Het hekje # zet telkens één cijfer in de String;
Het vraagteken plaatst de resterende cijfers of een nul in de String. (Denk erom dat de String van achter naar voren opgebouwd wordt!)
Andere karakters van de mal worden letterlijk overgenomen.
Opmerking: het om te zetten getal wordt opgevat als Unsigned.

4. De definitie van PICTURE

```
) : PICTURE ( ulo uhi adr1 len1 -- adr2 len2 )
)   2>R
)   <#
)   BEGIN R> 1-            \ positie in sjabloon
)      S>D 0=
)   WHILE >R 2R@ + C@      \ karakter uit sjabloon
)      [CHAR] # OVER =
)      IF DROP #           \ een cijfer
)      ELSE [CHAR] ? OVER =
)         IF DROP #S       \ de resterende cijfers
)         ELSE HOLD
)         THEN
)      THEN
)   REPEAT R> 2DROP
)   #> ;
```

`# ( ulo1 uhi1 -- ulo2 uhi2 )` "number sign"
Bepaal, gebruik makend van `BASE`, het laatste cijfer van het bitpatroon `uhi1,ulo1` en plaats dat cijfer vooraan de String in wording. `uhi2,ulo2` is `uhi1,ulo1` zonder zijn laatste cijfer. (Alleen gebruiken tussen `<#` en `#>`.)

`HOLD ( k -- )`
Voeg vooraan de String in wording het karakter `k` toe. (Alleen gebruiken tussen `<#` en `#>`.)

Voorbeelden:

```
) 2.34567 S" fl. ?.##"
) PICTURE TYPE [rtn] fl. 2345.67 ok

) 2.34567 S" C#-##"
) PICTURE TYPE [rtn] C5-67 ok

) -2.34567 S" C#-##
) PICTURE TYPE [rtn] C0-49 ok \ of C7-29 ok
```

`PICTURE` levert een String die slechts kort geldig is. Bij de meeste Forth-systemen zal hij overschreven worden zodra er weer een omzetting begint. Probeer eens

```
) 1234. S" ####" PICTURE 56 . TYPE [rtn]
```

5. Zomaar een ideetje

```
) : (EVENTJES  ( ? )
)   STATE @ ABORT" Only executing "
)   BASE @ >R
)   EXECUTE ' EXECUTE
)   R> BASE ! ;

) : HEXA ['] HEX      (EVENTJES ; IMMEDIATE
) : DECI ['] DECIMAL  (EVENTJES ; IMMEDIATE
) : BINA ['] BINARY   (EVENTJES ; IMMEDIATE
```

Definiëer van te voren `BINARY` als het nog niet aanwezig is:

```
) : BINARY ( -- ) 2 BASE ! ;
```

Gebruik `HEXA` `DECI` en `BINA` interactief, onmiddellijk voor een woord dat getallen afdrukt:

```
) DECIMAL -1 DUP HEXA U. U. [rtn]
```

Maar ook

```
) : TEL ( -- ) 15 0
    DO I S>D S" #### " PICTURE TYPE LOOP ;
) HEXA TEL [rtn]
) BINA TEL [rtn]
```

Over het compile-time en run-time gedrag van `IF THEN BEGIN` enz.

Controlestructuren I

1. Hoe het simpel kan

Eerst de data, dan de operatie. Deze grondregel van Forth gaat ook op bij het nemen van beslissingen in een programma door middel van voorwaardelijke sprongen. Een vlag op stack is het 'data-voer' voor de 'operatoren' `IF` of `UNTIL` die vervolgens beslissen of er zal worden gesprongen. In veel programmeertalen is de constructie:

```
IF        \ Bepaal de vlagwaarde van 'koud'.
 koud     \ Formulering van de vlag
THEN      \ Verlaat de constructie als vlag=false
 das-om   \ Voorwaardelijk uit te voeren code
ENDIF     \ Eindpunt; zou ook ENDTHEN kunnen heten
```

In Forth is de waardebepaling van de vlag geen grammaticaal onderdeel van de constructie. Er is alleen de operator `IF` met zijn afsluiter `THEN`. De programmeur moet er voor zorgen dat er een vlag klaarstaat:

```
( koud?     \ Vlag op stack )
IF          \ Spring bij 'false' vooruit naar THEN
 das-om     \ Voorwaardelijk uit te voeren code
THEN        \ Eindpunt; zou ENDIF kunnen heten
```

De achterwaartse sprong met `UNTIL` gaat in Forth analoog:

```
BEGIN       \ Beginpunt
 ... warm?  \ De programmeur zorgt ervoor dat
            \   deze code een vlag produceert
UNTIL       \ Spring bij 'false' terug naar BEGIN
( das-af )
```

Je kunt deze constructies in elkaar nesten. Als je dat netjes doet, zoals in `... BEGIN ... IF ... THEN ... UNTIL ...` heet dat gestructureerd programmeren.
In `... BEGIN ... IF ... UNTIL ... THEN ...` zal Forth protesteren als hij `UNTIL` tegenkomt, want de `IF` is nog niet afgesloten.
En in `... IF ... IF ... THEN ... THEN ...` hoort de eerste `THEN` daarom bij de tweede `IF`.

2. De compiler

Als je een definitie hebt gemaakt met

```
aaa IF bbb THEN ccc
```

erin, en je decompileert dat, dan zul je iets te zien krijgen in de trant van

```
aaa, JOF adr, bbb, ccc
```

`IF` is geworden: `JOF` (jump on false) gevolgd door een (relatief) adres.
`THEN` is verdwenen. Logisch, want het adres verwijst, waarschijnlijk in de vorm van een offset, al naar `ccc`. Dat adres is nog niet bekend op het moment dat `IF` bereikt is.
Hoe de compiler dat voor elkaar krijgt?
Eigenlijk doet de compiler dat niet, dat doen `IF` en `THEN` zelf.

```
: IF ( -- ) ?COMPILING \ Uitleg volgt
  POSTPONE JOF
  HERE IFKENMERK COMPOST!
  0 , ; IMMEDIATE

: THEN ( -- ) ?COMPILING
  HERE COMPOST@
  IFKENMERK ?PAIRS
  OFFSET! ; IMMEDIATE
```

`?COMPILING` veroorzaakt een foutmelding als je niet aan het compileren bent.
`IFKENMERK` is een constante die dient ter herkenning.
`COMPOST! ( adr const -- )` stuurt post (een pakketje met twee getallen) naar de compiler.
`COMPOST@ ( -- adr const )` vraagt het laatst gestuurde pakketje weer terug.
`?PAIRS ( x y - )` geeft een foutmelding als `x <> y`.
`OFFSET! ( adr1 adr2 -- )` slaat `adr1` als offset op in `adr2`.

Alle niet-standaardnamen in dit verhaaltje heb ik zelf verzonnen. Ik gebruik ze slechts om duidelijk te maken wat er gebeurt. In iedere Forth zullen deze namen en ook de details anders zijn, maar de globale gang van zaken zal zijn als hierboven beschreven.

`BEGIN` en `UNTIL` zullen nu geen probleem meer vormen.

```
: BEGIN ( -- )
  HERE BEGINKENMERK COMPOST! ; IMMEDIATE
```

```
: UNTIL ( -- ) ?COMPILING
  COMPOST@ BEGINKENMERK ?PAIRS
  POSTPONE JOF
  HERE  0 ,
  OFFSET! ; IMMEDIATE
```

Onthoud vooral dat IF en BEGIN boodschappen achterlaten voor THEN en UNTIL.

3. Onvoorwaardelijke sprongen

IF...THEN en BEGIN...UNTIL hebben elk een tegenhanger, AHEAD...THEN en BEGIN...AGAIN, met onvoorwaardelijke sprongen. Zij gebruiken geen vlag, doen geen test en lijken op het eerste gezicht zinloos:

```
AHEAD ??? THEN ...
BEGIN ... AGAIN ???
```

AHEAD springt altijd vooruit naar THEN en AGAIN springt altijd terug naar BEGIN. Waarom dan de code ??? achter AHEAD en AGAIN die toch nooit bereikt wordt?

Deze structuren hebben over het algemeen alleen zin als ze ongestructureerd met andere controlestructuren gebruikt worden.
... IF ... ELSE ... THEN ... kun je bijvoorbeeld zien als
... IF ... AHEAD THEN ... THEN waarvan de eerste THEN bij IF hoort en de tweede THEN bij AHEAD. Foutief genest dus.
Je kunt dit met een kunstgreep kloppend krijgen als je, precies tussen AHEAD en THEN in, op een of andere manier de laatste twee postpakketjes bij de compiler van plaats kunt laten verwisselen.

Daar is een (standaard)woord voor: 1 CS-ROLL

```
: ELSE (--)
  POSTPONE AHEAD 1 CS-ROLL
  POSTPONE THEN ; IMMEDIATE
```

Je zou dezelfde truc met AGAIN kunnen uithalen:
... BEGIN ... IF ... AGAIN THEN ... wordt dan ... BEGIN ... IF ... AGAIN? ... met

```
: AGAIN? ( -- )
  POSTPONE AGAIN 1 CS-ROLL
  POSTPONE THEN ; IMMEDIATE
```

Maar hiervoor heeft men een andere oplossing bedacht. Meer daarover de volgende keer.

[wordt vervolgd]

Controlestructuren II

4. WHILE

WHILE is de oplossing waar ik de vorige keer op doelde. Het is een IF die zijn compilerpost niet helemaal bovenop, maar vlak onder de bovenste post legt. Je kunt WHILE daarom alleen **binnen** een andere Controlestructuur gebruiken. Die andere structuur moet weer afgesloten zijn voordat de bijbehorende THEN komt. WHILE nest zich kruislings, en dient ervoor om **uit** een structuur naar THEN te springen:

```
: WHILE ( -- )
  POSTPONE IF
  1 CS-ROLL ; IMMEDIATE

 .. BEGIN .. WHILE .. AGAIN .. THEN ..
      1        2        1        2
```

De cijfers geven aan hoe de woorden bij elkaar horen. WHILE is bruikbaar binnen alle andere controlestructuren. Voorbeelden:

```
a) .. BEGIN .. WHILE .. UNTIL .. THEN ..
        1        2        1        2

b) .. BEGIN .. WHILE .. AGAIN .. ELSE .. THEN ..
        1        2        1        2       2

c) .. BEGIN .. WHILE .. WHILE ..
        1        2        3
                   AGAIN .. THEN .. THEN ...
                     1        3       2
```

In c) legt WHILE-2 zijn post onder BEGIN-1, WHILE-3 legt zijn post ook onder BEGIN-1, dus bovenop WHILE-2. De beide WHILE's zijn onderling weer normaal genest.

Strikvraagje:
Wat is het verschil tussen OLIE en AZIJN?

```
: OLIE  aaa IF bbb IF ccc THEN THEN ddd ;
: AZIJN aaa IF bbb WHILE ccc THEN THEN ddd ;
```

Opmerking:
REPEAT (een standaardwoord) staat voor AGAIN THEN.

```
.. BEGIN .. WHILE .. REPEAT ..
```

5. DO-LOOP, CASE

Het paar DO-LOOP werkt run-time als een BEGIN-AGAIN met de volgende extra's:
1. DO (grens teller --) bouwt een tel-mechanisme.
2. Binnen de lus is de teller opvraagbaar met I.
3. LOOP past na iedere doorloop de teller aan.
4. LOOP beëindigt de lus als de teller het opgegeven gebied afgewerkt heeft.
Voor de preciezen onder u: het opgegeven gebied is afgewerkt als de teller het "tussenschot" tussen 'grens-minus-1' en 'grens' passeert. Deze redenering geldt ook voor positieve èn voor negatieve stapjes bij +LOOP.

```
 10 1 DO ...            ( 1 2 .. 9 klaar )
  0 0 DO ...            ( 0 1 .. -2 -1 klaar )
 20 0 DO ...  2 +LOOP   ( 0 2 .. 18 klaar )
-10 0 DO ... -1 +LOOP   ( 0 -1 .. -10 klaar )
  0 0 DO ... -1 +LOOP   ( 0 klaar )
```

Je gebruikt LEAVE om een DO-LOOP voortijdig af te breken:

```
.. DO .. IF LEAVE THEN .. LOOP aaa
```

LEAVE ruimt de tel-administratie op en veroorzaakt een sprong naar aaa, onmiddellijk achter LOOP.

Ook compile-time lijkt DO-LOOP op BEGIN-AGAIN. Je kunt daarom met WHILE een sprong uit een DO-LOOP compileren. Dat kan gunstig zijn als de voorwaardelijk afgebroken lus heel anders voortgezet moet worden dan de compleet afgwerkte lus:

```
.. DO .. WHILE LOOP aaa
         ELSE .. I .. UNLOOP THEN ..
```

aaa wordt uitgevoerd als LOOP de lus beëindigt heeft. Het stuk tussen ELSE en THEN komt uitsluitend na een voorwaardelijke onderbreking. Meestal is dit niet op een elegante wijze met LEAVE te schrijven.
Er vallen twee dingen op:
1. Er is **altijd expliciet een** UNLOOP **nodig** om run-time de tel-administratie op te ruimen.
2. I is bruikbaar **buiten de lus zolang de tel-administratie nog intact is**.
LEAVE (altijd) en LOOP (na de laatste doorloop) voeren impliciet een UNLOOP uit.

Tenslotte de CASE-constructie in Forth, een veelbesproken onderwerp. Is het die aandacht waard? Ik denk dat CASE volstrekt overbodig is. Forth heeft hier voldoende betere formuleringen voor. Zou het kunnen zijn dat de behoefte aan CASE vooral gevoeld wordt door aan Forth beginnende anderstaligen met de begrijpelijke neiging om een-op-een te vertalen vanuit hun vertrouwde omgeving (DE Forth-drempel m.i.)?

Voorbeeld van een programmeerstijl die voorwaardelijke sprongen vermijdt.

Zonder uitzondering

Als je `x` met `y` vermenigvuldigt is het antwoord `y*x` tenzij er een `0` of een `1` bij betrokken is:

```
: MAAL ( x y -- y*x ¦ x ¦ y )
  OVER 0= OVER 1 = OR IF DROP ELSE
  OVER 1 = OVER 0= OR IF NIP ELSE *
  THEN THEN ;
```

Maar het kan eenvoudiger:

```
: MAAL ( x y -- y*x ) * ;
```

Hiermee wil ik illustreren dat wanneer iets een uitzondering lijkt te zijn, er niet altijd een speciale behandeling voor nodig is. Maar zo simpel als met MAAL gaat dat meestal niet.
In onderstaand programmaatje heb ik zoveel mogelijk uitzonderingen weggewerkt `( **)` door ze om te buigen tot reguliere acties. De route die het programma afglegt is daardoor gemakkelijk te volgen want er staan geen voorwaardelijke sprongen meer in.
De enige concessie is `UNTIL` aan het eind van `SPEEL`.

```
\ ----- Schuifspelletje -----
\ De speler schuift de blokjes in de goede stand.

marker -SPEEL

: CHAR-ARRAY ( aantal  -- ) \ Definiërend woord
  create c,
  here 1- c@ 1+ allot align \ Max. lengte?    ( **)
  DOES> ( nr -- adr )
  count rot umin + ;        \ Range?          ( **)

16 CHAR-ARRAY BORD \ 15 tekens en een lege plek
0 value HIER       \ Positie vd lege plek (spatie)

: EINDSTAND ( -- )
  s" VIJGEBLAD*FORTH " 0 bord swap move
  15 to hier ;
```

```
: .BORD ( -- )
  0 5 at-xy
  0 bord 4 0
  do cr cr 33 spaces
     4 0 do count emit 3 spaces loop
     loop drop 7 spaces ;


\ De speler kiest de te schuiven letter.
\ Daarom moeten alle tekens verschillend zijn.
0 value TOETS
: >CODE ( ch -- c ) dup $20 and - ; \ Upper?  ( **)
: SPELER ( -- ) key >code to toets ;

\ Pos = positie t.o.v. HIER. Let op de bordrand.
: WEST?  ( -- pos ) hier 3 and 0<> -1 and ;   ( **)
: OOST?  ( -- pos ) hier 1+ 3 and 0<> 1 and ; ( **)
: NOORD? ( -- pos ) hier 3 > -4 and ;         ( **)
: ZUID?  ( -- pos ) hier 12 < 4 and ;         ( **)

: LETTER? ( pos -- pos ) \ De goede letter?
  dup hier + bord c@ >code
  toets = and ;                               ( **)

: DAAR ( -- nr ) \ Positie vd te schuiven letter
  noord? letter?
  west?  letter? or
  oost?  letter? or
  zuid?  letter? or hier + ;
```

V	I	J	G
E	B	L	A
D	*	F	O
R	T	H	

```
: SCHUIF ( nr -- )
  dup bord hier bord
  over c@ over c@
  -rot
  swap c! swap c!
  to hier ;


: SPEEL ( -- )
  page eindstand .bord 1997 ms
  32 0 do random $F and schuif .bord loop
  begin speler daar schuif .bord
        toets $1B =   \ [Escape]?
  until ;
```

Dit programma coördineert processen die in onderling verschillende tempi verlopen: ze beginnen tegelijk, lopen gelijkmatig, en zijn tegelijk klaar.

Robot-arm met (bijv.) vijf motoren

De voorzitter van de Forth gebruikersgroep knutselde aan een robot-arm met vijf motoren (twee in de schouder, één in de elleboog en twee in de pols) en vroeg me een algoritme te verzinnen om het ding soepel te laten bewegen.

- Maar ik heb absoluut geen verstand van hardware.

- Je stuurt een getal naar zo'n motor. Die gaat dan met een ruk naar de stand die overeenkomt met dat getal. Je hoeft de actuele stand van de motor dus niet te kennen. En de allerlaagste routine maak ik wel.

Gewapend met deze hardware kennis heb ik een programma bedacht, én, het blijkt te werken! Na aanpassing van het woord `>MOTOR` natuurlijk. Bij mij thuis, zonder de hardware, heb ik het resultaat alleen maar op het scherm bekeken (met `.HIER`).

Omdat ik geen specifieke kennis van de arm gebruik zit er geen beveiliging in tegen opdrachten voor onmogelijke posities. Je kunt zelfs het aantal motoren anders kiezen. Het programma coördineert processen die in onderling verschillende tempi verlopen: ze beginnen tegelijk, lopen gelijkmatig, en zijn tegelijk klaar. Er zijn dus andere toepassingen denkbaar.
Bijvoorbeeld een tekenfilm van Achilles en de Schilpad, of rechte lijnen trekken met de vijfdimensionale plotter die je altijd al had willen bouwen.

Het probleem

Stel, ik beweeg van stand `<0 0 0 0 0>` naar `<10 20 13 7 30>`. Dan draait de laatste motor over de grootste hoek (`30`). Die grootste hoek (`30`) wordt het uitgangspunt voor de algoritme: ik verdeel de beweging van elke motor in 30 etappes. En een motor doet per etappe óf slechts één stap (`PLOP`) óf niets (`PILI`).

De laatste motor doet in alle etappes 1 stap (30 keer `PLOP`). Dat is eenvoudig. Nu de andere motoren.
Afstand 10: 10 etappes 1 stap (`PLOP`) en 20 etappes niets (`PILI`), regelmatig verdeeld wordt dat: `PILI PLOP PILI` (10 maal).
Afstand 20 blijft ook eenvoudig: `PLOP PILI PLOP` (10 maal).
Afstand 13 is moeilijker: 13 keer `PLOP` en 17 keer `PILI`. Hoe verdeel je dat een beetje netjes over 30 etappes?

De oplossing

Voorzie de motoren van benzinetanks.

De brandstof

1) Per etappe verbruikt de motor (ook als hij niet beweegt!) 13 liter (het aantal PLOP's, de te draaien hoek).
2) Raakt de brandstoftank leeg, dan wordt er precies 30 liter bijgetankt (het aantal etappes, de grootste hoek).

De redenering

Per etappe verbruikt hij 13 liter, over 30 etappes:
30 * 13 liter. 30 * 13 = 13 * 30.
Hoevaak zal hij moeten tanken?
Juist, 13 keer.
En hoeveel stapjes moest hij ook weer? ...

Ziedaar de oplossing:

Als in een etappe een tank leeg raakt, wordt er getankt én gePLOPt. De overige (17) etappes zijn PILI. Haal nu die benzinetanks maar weer weg en onthoud de gedachtengang, want die wordt toegepast in de volgende code.

De Forth code

```
\ ROBOT-ARM
marker -ARM forth definitions decimal
: VARIABLES ( n  - ) create cells allot
  does> ( index body - adres ) swap cells + ;

5 dup constant #MOTOREN
dup VARIABLES HIER      \ Lijst v motorstanden
dup VARIABLES DAAR      \ Lijst v doelstanden
dup VARIABLES STAPJE    \ -1 of +1 (richtingen)
dup VARIABLES TANK      \ Hoeveelheid brandstof
dup VARIABLES VERBRUIK  \ Verbruik per etappe
cells   0 hier   swap 0 fill

0 value #ETAPPES        \ Voor de langste weg
20 value WACHT          \ Vertraging per etappe

: DOEL ( m0 m1 m2 m3 m4 - )   \ Doel vastleggen
  #motoren begin 1-
     tuck daar !
  ?dup 0= until ;
```

```
: VOORBEREIDING ( - )
  0                          \ Voor langste weg
  #motoren 0                 \ Per motor:
  do i daar @ i hier @
     2dup <
     dup 2* 1+ i stapje !    \ 1 of -1, richting
     if swap then -
     dup i verbruik !        \ afstand
     max                     \ grootste afstand?
  loop dup to #etappes
  2/ #motoren 0
  do dup i tank !            \ tanks halfvol
  loop drop ;

\ Voor de scherm-versie zonder de hard-ware
: >MOTOR ( nieuwe-stand motor# - ) 2drop ;

\ Om de motorstanden op 't scherm te zetten
: .HIER ( - ) cr #motoren 0
  do i hier @ 4 .R loop space ;

: ETAPPE ( - ) #motoren 0
  do i tank @ i verbruik @
     2dup <                  \ Brandstof tekort?
     if #etappes -           \ Dan bijtanken
        i stapje @ i hier +! \ Nieuwe motorstand
        i hier @ i >MOTOR    \ P-L-O-P
     then - i tank !
  loop ;

: BEWEEG ( - ) \ Het doel moet al vastgelegd zijn.
  voorbereiding
  #etappes 0
  ?do .HIER etappe
     key? if key drop leave then
     wacht ms
  loop .HIER ;

: GA ( doelposities - ) doel beweeg ;

\ Voorbeeld: 10 20 13 7 30 ga
```

CATCH en THROW I

Leo: De vorige keer heb je de begrippen Signed en Unsigned uit de doeken gedaan. Zou je me vandaag iets willen vertellen over THROW? Ik weet dat THROW een getal op stack verwacht. Als dat een nul is gebeurt er niets. Bij andere waarden verlaat Forth het programma en zet soms een foutmelding op het scherm met de mededeling dat dit message nummer zoveel is. Bijvoorbeeld, het resultaat van

```
-14 THROW
```

is

```
Compile-only word (message # -14)
```

Daar is vast meer over te vertellen.

Theo: Goed. Ik maak even een omweggetje. Je hebt EXECUTE al leren kennen. Kun je voorspellen wat het volgende zal doen?

```
17 ' dup execute .s
```

Leo: Dat is gemakkelijk. ' (Tick) zoekt DUP op, en EXECUTE voert die DUP uit. Het geheel werkt dus als DUP.

```
[rtn] ( 17 17 ) ok
```

Theo: Juist. Ik maak eerst even een woordje dat ... of nee, stel jij maar vast wat VB doet:

```
: VB begin depth 0> while drop repeat ;
```

Leo: Zolang DEPTH groter dan nul blijft wordt er gedropt, dus VB maakt de stack schoon.

Theo: O.K. Nu puzzel nr 1. Wat doet:

```
vb 1 ' dup catch .s
```

Ik vertel er bij, dat CATCH hier bijna hetzelfde effect heeft als EXECUTE. Het verschil is, dat CATCH na afloop nog een nul op de stack plaatst. Bedenk het antwoord voordat je op [rtn] drukt.

Leo: VB maakt de stack leeg, dan komt er een 1 en die wordt gedupt.

```
[rtn] ( 1 1 0 ) ok
```

O ja, en die extra nul. Waarom die nul trouwens?

Theo: Zal ik je laten zien.

```
vb 2 ' ; catch .s [rtn] ( 2 -14 ) ok
```

Als het extra getal van CATCH een nul is betekent dat, dat de operatie gelukt is. Als de operatie niet lukt geeft hij een foutnummer i.p.v. die nul.

Leo: Wat is er in dit geval dan niet gelukt?

Theo: Probeer hetzelfde maar eens met EXECUTE.

```
vb 2 ' ; execute .s [rtn]
 Compile-only word (message # -14)
```

Leo: Hm.
(2 minuten stilte)
De .S achter EXECUTE wordt niet uitgevoerd, en die achter CATCH wel.

Theo: Juist. EXECUTE gooit het bijltje er bij neer. CATCH vangt eventuele fouten op en zorgt ervoor dat de stack hersteld wordt, d.w.z. voor en na de mislukte operatie even diep is. Hij vervangt het token van de mislukte operatie door het foutnummer. Na gelukte operaties komt de goedkeuringsnul uiteraard bovenop het bereikte resultaat. Kijk maar bij vb 1.
De volgende:

```
vb 3 s" dup" ' evaluate catch .s
```

Leo: Dit doet hetzelfde als voorbeeld 1.

```
[rtn] ( 3 3 0 ) ok
```

Theo: Juist. Maar nu deze:

```
vb 4 s" vijg" ' evaluate catch .s
```

Leo: Die "vijg" kan hij niet evalueren, dus komt er achter de 4 een of ander foutnummertje op de stack.

```
[rtn] ( 4 0 0 -61 ) ok
```

Waar komen die twee nullen vandaan?

Theo: Bedenk dat na een fout de stack voor en na CATCH even diep is.

Leo: Och natuurlijk, de VIJG-string stond ook op de stack. Maar waarom is die in 0 0 veranderd?

Theo: EVALUATE gaat met de string aan de slag, gebruikt daarbij de stack, en op het moment van de fout stond daar kennelijk 0 0.
Nu komen de denkertjes:

```
vb 5 ' char catch 8 .s
```

Leo: CATCH voert CHAR uit, die de ASCII-code van 8 opzoekt, en zet er een nul bij.

```
[rtn] ( 5 56 0 ) ok
```

Theo: Juist. Voor het volgende voorbeeld moet je weten dat CATCH voordat hij aan werk gaat de actuele invoerstroom opslaat. Als er iets fout gaat herstelt hij de stack(s) èn de invoerstroom. Waarom? Denk er maar eens over na wat er zou gebeuren als je na een fout tijdens het laden van een file de invoerstroom op zijn beloop zou laten.

```
vb 6 ' ' catch 666 .s [rtn] ( 6 -13 666 ) ok
```

CATCH voert de Tick uit op 666. Dat lukt natuurlijk niet. Hij zet foutnummer -13 op stack, en herstelt de invoerstroom. Die wees naar direct achter CATCH. 666 wordt opnieuw gelezen en uitgevoerd en tenslotte is .S aan de beurt.

Het spijt me, maar ik moet er nu vandoor. Ik geef je een stuk of wat puzzeltjes die je met de kennis die je nu hebt zou moeten kunnen oplossen.

```
vb 7  ' s" catch Vijgeblaadje" .s
vb 8  ' ' catch vijg .s
vb 9  ' to catch bl .s
vb 10 ' dup ' catch execute .s
vb 11 ' dup ' catch catch .s
vb 12 ' dup ' catch ' catch catch .s
vb ( 13 ) ' drop catch .s
vb 14 ' throw catch .s
```

Tot ziens.

Leo: Tot ziens.

[wordt vervolgd]

CATCH en THROW II

Theo: Je gebruikt CATCH en THROW in complexe programma's die onderdelen bevatten waarvan de afloop niet altijd te voorzien is. Het kleine Vijgeblaadje is niet geschikt voor zo'n voorbeeld. Daarom neem ik mijn toevlucht tot een kunstgreep:
Ik beschouw de gehele Forth als een complex programma waar de gebruiker vele onvoorziene dingen mee kan doen, en waar dus vele THROW's in voorkomen. De gehele Forth zet ik in een CATCH, en daar bouw ik een klein programmaatje omheen zodat je kunt zien wat er gebeurt.
Doel van dit alles: de functie van THROW en CATCH demonstreren.

```
: .DEPTH ( - ) depth 0 .r ." : "
  depth 0<                      \ Indien nodig:
  if begin 0 depth 0= until     \ stackreparatie
  ." Stack underflow "          \ met melding.
  then ;

: .TOP2 ( - )            \ Toon, indien aanwezig,
  depth 1 >              \ de bovenste
  if over . then         \ 2 getallen die
  depth if dup . then ; \ die op stack staan.

: .TOP4 ( - )
  depth 4 > if ." ~ " then
  depth 2 > if 2>r .top2 2r> then
  .top2 ;

: STATE-TEKEN ( - char )
  state @ if [char] ] else [char] [ then ;

: .SITUATIE ( - )
  cr .depth .top4 state-teken emit space ;

create INVOERBUFFER 80 allot

: INVOER ( - adr len )
  invoerbuffer dup 80 accept space ;

: RISICO-PROGRAMMA ( - )
  .situatie invoer evaluate ;

: VB ( - )
  begin ['] risico-programma catch
     dup
     if dup . ." THROW is uitgevoerd "
     then drop
  again ;
```

----- Voorbeeld van een sessie -----

Vet gedrukte tekst is invoer, de rest is reactie van het programma.
[rtn] is de returntoets.
LET OP: DE MAATREGELEN DIE FORTH NA EEN FOUT PLEEGT TE NEMEN BLIJVEN NU ACHTERWEGE.

```
vb [rtn]
0: [ drop [rtn] -4 THROW is uitgevoerd
0: [ 12 [rtn]
1: 12 [ 1999 [rtn]
2: 12 1999 [ : zwap [rtn]
2: 12 1999 ] qwuerty [rtn]
                      -61 THROW is uitgevoerd
```

Met QWERTY kan EVALUATE niets aanvangen. Het is geen woord en ook geen getal, maar de compile-toestand blijft gehandhaafd, de stack blijft intact.

```
2: 12 1999 ] 2>r [rtn]
2: 12 1999 ] r> r> [rtn]
2: 12 1999 ] ; [rtn]
```

Ondanks de storing lijkt ZWAP toch gecompileerd te zijn.

```
2: 12 1999 [ zwap [rtn]
2: 1999 12 [ ' asdf [rtn]
                      -13 THROW is uitgevoerd
```

TICK kan het woord ASDF niet vinden.

```
2: 1999 12 [ -56 throw [rtn]
[rtn]  ok
.s [rtn] ( 1999 12 )  ok
```

Met VB kom je in een oneindige lus, waaruit je toch kunt ontsnappen met woorden als QUIT of BYE. Dat laatste is natuurlijk niet de bedoeling. -56 throw is een speciale THROW waarvoor afgesproken is dat hij QUIT uitvoert.
Probeer ook ABORT en TRUE ABORT" Hallo!"
ABORT behoort -1 THROW uit te voeren en ABORT" een -2 THROW.

De opgavan uit hoofdstuk 208

```
: VB begin depth 0> while drop repeat ;
```

```
vb 7 ' s" catch Vijgeblaadje" .s [rtn] ( 7 adr len 0 )
```

`adr len` is adres en lengte van de string Vijgeblaadje.

```
vb 8 ' ' catch vijg .s [rtn]
```

Forth geeft een foutmelding want hij probeert na de `CATCH` nogmaals `vijg` te vinden.

```
vb 9 ' to catch bl .s [rtn] ( 9 -32 32 )
```

`TO` kan niet op `BL` werken: foutnummer -32, vervolgens `BL` (=32).

```
vb 10 ' dup ' catch execute .s [rtn] ( 10 10 0 )
```

Dup-actie en goedkeuringsnul

```
vb 11 ' dup ' catch catch .s [rtn] ( 11 11 0 0 )
```

Dup-actie en 2 goedkeuringsnullen

```
vb 12 ' dup ' catch ' catch catch .s [rtn] ( 12 12 0 0 0 )
```

```
vb ( 13 ) ' drop catch .s [rtn] ( )
```

Drop-actie op een lege stack, daarna een goedkeuringsnul. Mijn Forth protesteert niet.

```
vb 14 ' throw catch .s [rtn] ( 14? 14 )
```

De eerste 14 kan overschreven zijn.

Over het manipuleren van de invoerstroom.

MANY TIMES

FORTH aan het werk

Als de gebruiker een regel intypt en op [rtn] drukt gaat Forth die regel als volgt te lijf:

Hij zoekt naar de eerste niet-spatie (hier begint vast en zeker een woord), vervolgens naar de eerstvolgende spatie (daar eindigt het woord). Hierbij fungeert de systeemvariabele >IN als aanwijsstokje.
Forth houdt namelijk ijverig bij in >IN welke positie op de regel hij aan het bekijken is.

Daarna zoekt hij het zojuist afgebakende woord op in zijn woordenboek en voert het uit (executeert het). Vindt Forth dat woord niet, dan haakt hij af en laat de rest van de regel voor wat het is.

Zo lang het lukt herhaalt Forth deze procedure: hij bepaalt het volgende woord, steeds gebruik makend van >IN, en voert dat woord uit. Tot aan het einde van de regel.

Omdat Forth geen syntax heeft, hoeft hij van te voren niet te onderzoeken of de regel wel correct is.
Eventuele typfouten komen vanzelf aan het licht, want Forth kan een verkeerd gespeld woord toch niet vinden.
Met programmeerfouten zit dat natuurlijk anders, maar dat is geen syntaxkwestie.

```
) >IN @ . [rtn] ?
 1234567... posities
```

De inhoud van >IN wordt opgehaald door @. Forth heeft de regel dan al gelezen tot en met @ inclusief de spatie erachter.

```
)       >IN @ . [rtn] ?
 1234567890123... posities

)       >IN @ . >IN @ . [rtn] ?
 12345678901234567890 1... posities
```

Forth vertrouwt er op dat de gebruiker geen woord intypt dat de inhoud van >IN stiekem verandert, maar zeg zoiets nooit tegen een Forth-programmeur...

Forthprogrammeur aan het werk

```
: MANY ( -- ) >IN @ STOP? AND >IN ! ;
```

Ho, wacht even, STOP? is geen standaard Forthwoord. Daar gaan we dus eerst even op in.

```
: STOP? ( - true/false )
  KEY? DUP IF DROP KEY
     BL OVER = IF DROP KEY THEN
     27 OVER = IF -28 THROW THEN
     BL <> THEN ;

) 1000 MS STOP? . [rtn] ?
```

Levert nul op. Maar voer dezelfde regel nog eens in, en druk onmiddellijk na [rtn] op een of andere toets. Drie mogelijkheden.

1. Je drukt op [Esc]: Forth protesteert en reageert met een foutmelding.

2. Je drukt op een andere toets, geen [Esc] maar ook geen [Spatie]: Forth drukt -1 (TRUE) af.

3. Je drukt op [Spatie]: Forth pauzeert, stopt alle actie, en wacht op nog een toets. Wordt dat weer een [Spatie] dan is het resultaat 0 (FALSE). Na [Esc] komt er weer protest, en de overige toetsen leveren -1 (TRUE).

Dit klinkt ontzettend ingewikkeld, maar als je het een paar keer uitprobeert wordt het gauw heel logisch. STOP? (geen standaard Forthwoord dus) zit vaak in woorden als WORDS, SEE, DUMP, woorden die tekst produceren die van het scherm af kan lopen. Met STOP? heb je de mogelijkheid om de zaak even stil te zetten of af te breken.

```
: MANY ( -- ) >IN @ STOP? AND >IN ! ;
```

Zolang STOP? hier nul oplevert, wordt >IN nul gemaakt: Forth wordt om de tuin geleid, 'denkt' steeds dat hij nog niets van de regel gelezen heeft en begint steeds weer vooraan de regel.

Met MANY kun je interactief lusjes maken:

```
) 888 . MANY DROP [rtn] ?
```

Tamelijk zinloos, maar:

```
) BL [rtn]
) DUP EMIT 1+ MANY DROP [rtn] ?
```

TIMES is een genuanceerdere variant van MANY waarbij je aangeeft hoevaak Forth het eerste stuk van de regel moet uitvoeren.

```
0 VALUE #TIMES          \ Teller
: TIMES ( n -- )
  #TIMES 1+ TUCK ( #times+1 n #times+1 )
  0 TO #TIMES           \ Zekerheid boven alles.
  = STOP? OR            \ n-de keer of stop-ingreep?
  IF DROP EXIT THEN     \ Dan klaar.
  TO #TIMES 0 >IN ! ; \ Teller verhogen
                        \ en nog'n rondje.

) BL [rtn]
) DUP EMIT 1+  96 TIMES DROP [rtn] ?
```

Het aantal keren dat Forth vooraan de regel moet beginnen geef je op stack mee aan TIMES. De value #TIMES is voor intern gebruik door TIMES. Als je die van te voren toch op -100 of erger nog, op 100 zet...

Inde

Alle Forth-woorden in alfabetische volgorde

-

-	3, 121	core
-ROT	101	niet standaard
-TRAILING	-	string

.

. (punt)	1, 110, 202	core
."	10, 31, 32, 125	core
.(	31, 32, 109, 125	core ext
.R	20, 110, 202	core ext
.S	5, 101	toolkit ext

/

/	19, 121	core
/MOD	19, 121	core
/STRING	-	string

0

0<	22, 117	core
0<>	22, 117	core ext
0=	22, 117	core
0>	22, 117	core ext

1

1+	22, 120	core
1-	22, 120	core

2

2!	103	core
2*	22, 120	core
2/	22, 120	core
2>R	102	core ext
2@	104	core
2CONSTANT	-	double
2DROP	20, 28, 101	core

2DUP	20, 28, 101	core
2LITERAL	127	double
2OVER	101	core
2R>	102	core ext
2R@	102	core ext
2RDROP	102	niet standaard
2ROT	28, 101	double ext
2SWAP	101	core
2VARIABLE	-	double

:

:	4, 126	core, toolkit ext
:NONAME	-	core ext, toolkit ext

;

;	4, 126	core
;CODE	128	toolkit ext

<

<	10, 117	core
<#	111, 202	core
<>	30, 117	core ext

=

=	10, 117	core

>

>	10, 117	core
>BODY	129	core
>FLOAT	-	float
>IN	123, 210	core
>LINK	129	niet standaard
>NAME	129	niet standaard
>NUMBER	108	core
>R	24, 102	core

?

?	-	toolkit ext

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

XOR	20, 116	core

[

[	127	core
[']	124, 129	core
[CHAR]	32, 124	core
[COMPILE]	124	core ext
[ELSE]	-	toolkit ext
[IF]	-	toolkit ext
[THEN]	-	toolkit ext

\

\	18	core ext, block ext

]

]	127	core

Albert Nijhof

De Programmeertaal FORTH

(ANS FORTH)

Cursus
Systematisch over icht
Losse artikelen
Index

Okt. 1992 (1)
Dec. 2001 (2)