Vorwort 1

Arnheim, 18.11.92

"Forth ist die bestgeheimgehaltene Computersprache." (Dick Pountain)

In einer Zeit, da man uns glauben machen will, dass die Wichtigkeit und der Wert einer Sache mit dem Maß an Aufsehen einhergeht, das sie hervorruft, bekommt dieser Ausspruch eine auf der Hand liegende Bedeutung.

Ich bezweifle jedoch, dass Forth unbekannt ist. Vielleicht sollte man Forth besser als Sprache bezeichnen, von der man zwar gehört hat, die man aber nicht wirklich kennt. Forth ist eine Sprache, die in vielerlei Hinsicht von den gängigen Computersprachen abweicht. Mit den Computersprachen, die man "kennt", erklären, was Forth ist, muss daher missglücken. Stärker noch, Forth ist für Neulinge wahrscheinlich leichter als für Leute mit Programmiererfahrung.

Forth besteht aus Worten, aneinander gereihten Worten.
Ausdrücke wie `PRINT(x+(y/2))`, in welchen der Befehl `PRINT` erst dann seine Bedeutung erlangt, wenn die letzte Schlussklammer erreicht ist, gibt es in Forth nicht. "Sie können das natürlich so auch machen", ist unter Forth-Programmierern ein geflügeltes Wort. Und dann: "Aber warum wollen Sie das?". Jetzt begebe ich mich aber auf den Erklärungspfad, nachdem ich das Hoffnungslose daran soeben angedeutet habe.

In den Vereinigten Staaten ist man mit dem Formulieren eines Forth-Standards, dem ANS-Forth (American National Standard Forth), beschäftigt. Das wird 1993 offiziell werden. Ich nehme schon jetzt diesen Standard als Grundlage und lasse F.I.G.-Forth, 79-Forth und FORTH-83 außer Acht. Das Stichwortverzeichnis hinten im Buch enthält alle ANS-Forth-Worte. Unterschiede zwischen den Dialekten bereiten dem erfahrenen Forth-Programmierer keinerlei Schwierigkeiten. Für den Anfänger stellen sie jedoch einen recht verwirrenden Ballast dar. Forth ist keine fest umrissene Sprache. Es gibt ein eigenes Forth für fast jedes System. Mit einiger Mühe lässt sich da "etwas draufsetzen", um den gewünschten Dialekt herzustellen. Nehmen Sie Kontakt mit der Forth-Gebruikers-Groep der HCC auf, falls Sie doch Probleme haben sollten.

Eine technische Anmerkung: ANS-Forth hält nachdrücklich die Möglichkeit offen, die gesamte Zahlendarstellung intern anders als über das Zweierkomplement-Prinzip zu verwirklichen. Ich setze in diesem Kursbegleitbuch voraus, dass der Leser über ein Forthsystem mit Zweierkomplement-Darstellung verfügt.

Der Kurs ist für Anfänger gedacht, aber ich gehe davon aus, dass Grundbegriffe wie Byte, Bit und RAM bekannt sind. Das ist ein Forth-Kurs, kein Computer-Kurs. Der Text ist ziemlich kompakt und mitunter erst dann verständlich, wenn die Programmieraufgaben am Computer ausgeführt werden. **Der Kurs sollte unbedingt mit dem Computer in Griffbereitschaft absolviert werden.**

Fragen und konstruktive Kritik bitte an:

Albert Nijhof, `anij@hccnet.nl`

Vorwort 2

Arnheim, 18.12.2001

Die zweite Fassung (des holländischen Originals), obwohl in ein neues Gewand gekleidet, ist vom Inhalt her der ersten gleich geblieben, erweitert um die Kapitel 200 bis 211.

Die einzige wirkliche Abweichung:
Als die zweite Fassung erschien, lag ANS-Forth noch nicht ganz fest. Im letzten Moment wurde es dahingehend geändert, dass ein VALUE bei seiner Definition nun doch einen Anfangswert auf dem Stack erwartet. Das wurde in der vorliegenden Fassung angepasst.

Der auf den Kurs bezogene Teil des Buches ist didaktisch geordnet, der Rest nicht. Es war nicht meine Absicht, den Leser das Buch Seite für Seite von Anfang bis Ende durcharbeiten zu lassen. Im Gegenteil rate ich ihm, wenn er etwa bei Kapitel 15 angekommen ist und beginnt, sich in Forth zu Hause zu fühlen, dass er sich dann im systematischen Teil und in den "Losen Artikeln" umschaut. Je früher man lernt, selbständig in Forth zu denken, desto besser.

Im Stichwortverzeichnis am Ende des Buches stehen alle ANS-Forth-Worte, auch die, die in diesem Buch nicht besprochen werden.

Albert Nijhof, `anij@hccnet.nl`

Vorwort 3

Arnheim und München, 1.11.2003

Das holländische Original wurde in enger E-Mail-Zusammenarbeit mit dem Autor, Albert Nijhof, von Fred Behringer übersetzt.

Der Text wurde an manchen Stellen ergänzt, erweitert oder angepasst. Alle Änderungsvorschläge stammen entweder vom Autor oder vom Übersetzer und wurden zwischen beiden abgesprochen.

Für eventuelle fehlerhafte Übertragungen und Unachtsamkeiten in den Veränderungen zeichnet der Übersetzer verantwortlich.

Für die Übertragung der ersten Fassung der Übersetzung von der alten in die neue deutsche Schreibweise und für Hilfe bei der elektronischen Aufbereitung danken wir Herrn Rolf Schöne, Leiter des Forth-Büros (Verwaltung der Forth-Gesellschaft). Anstoß gegeben und zur moralischen Unterstützung beigetragen haben: Friederich Prinz, Redakteur der "Vierten Dimension" (Zeitschrift der Forth-Gesellschaft), und Dipl.-Ing. Klaus Zobawa, Gruppe "Küstenforth" in der Forth-Gesellschaft.

Albert Nijhof, Vorstandsmitglied der HCC-Forth-gebruikersgroep
http://www.forth.hccnet.nl/

Prof. Dr. Fred Behringer, Vorstandsmitglied der Forth-Gesellschaft
http://www.forth-ev.de/

A

0 - 36

Inhalt des Kursteils

1. EMIT .
2. Der Stack KEY CR
3. + - * SWAP DUP
4. WORDS Definitionen FORGET
5. .S DROP OVER BL (
6. CONSTANT VARIABLE ! @ +!
7. Überblick
8. Tick EXECUTE
9. Editor
10. Flags IF ELSE THEN TRUE FALSE
11. DO LOOP I HEX $xx #xx %xx
12. BEGIN UNTIL KEY? WHILE REPEAT
13. Signed und Unsigned
14. Zellen
15. HERE Name Code Body CREATE ALLOT
16. DUMP ACCEPT
17. Characters TYPE
18. Kontrollstrukturen ineinander schachteln
19. Teilen
20. AND OR XOR
21. .BRUCH
22. Primärworte TUCK NIP SEE VALUE TO
23. Algorithmus für GGT
24. >R R> R@ VEREINFACHE VF
25. Neu definieren
26. V+ NEGATE V-
27. V* V/ Local-Values V.
28. Doppeltgenaue Zahlen I
29. Doppeltgenaue Zahlen II
30. M* <> ABORT"
31. Immediate-Words in Kürze
32. CHAR [CHAR]
33. IMMEDIATE Steuerungsworte ?DO LEAVE
34. SPACE SPACES
35. J ROT
36. PAGE Spiel

Forth besteht aus Worten.
Wenn Sie ein oder mehrere Worte nacheinander eintippen und danach Return drücken, führt Forth die Worte in der Reihenfolge, in der Sie sie eingetippt haben, aus.

Im Folgenden werde ich mit einer Schlussklammer am Anfang einer Zeile andeuten, dass Sie den Text nach der Klammer bis einschließlich dem `[rtn]` eintippen sollen. Mit `[rtn]` ist die Return-Taste gemeint.

Beispiele:

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

Eine Zahl wird auch als ein Wort aufgefasst. `EMIT` kann dann über diese Zahl verfügen, betrachtet sie als einen ASCII-Code und lässt das zugehörige Zeichen auf dem Bildschirm erscheinen.

Auch der Punkt ist ein Forth-Wort. Er macht die Zahl selbst auf dem Bildschirm sichtbar.

```
) 65 EMIT 66 EMIT [rtn] AB ok
)
) 65 66 EMIT EMIT [rtn] BA ok
)
) 65 74 EMIT EMIT EMIT [rtn] JA  Empty Stack
)
) 1000 45 EMIT . [rtn] -1000  ok
```

Ein Wort, das eine Zahl benötigt, nimmt stets diejenige Zahl auf, die als letzte eingegeben wurde.

Forth gibt eine Fehlermeldung aus, wenn die Zahlen aufgebraucht sind. Die Formulierung dieser Meldung ist von Forth zu Forth verschieden.

Eingegebene Zahlen werden mit einem Mechanismus festgehalten, der im Englischen Stack heißt (stapeln). Zahlen werden in der Reihenfolge ihrer Eingabe aufbewahrt und in der umgekehrten Reihenfolge wieder an Worte abgegeben, die Zahlen verbrauchen.

Bei der Besprechung von Forth-Worten ist es üblich, hinter dem betreffenden Wort in Klammern dessen Einfluss auf den Stack anzudeuten:

`EMIT ( zahl -- )` Hol die Zahl vom Stack, betrachte sie als einen ASCII-Code und zeige das zugehörige Zeichen an.

`. ( zahl -- )` Hol die Zahl vom Stack und zeige sie mit einem angehängten Zwischenraum an.

Noch ein paar Forth-Worte:

`KEY ( -- zahl )` Warte, bis eine Taste gedrückt ist, und lege den ASCII-Code dieser Taste als Zahl auf den Stack.

`CR ( -- )` Gehe zu einer neuen Zeile. Keine Veränderung auf dem Stack.

? Können Sie die Wirkung voraussagen von:

```
) KEY . [rtn] ?

) KEY EMIT [rtn] ?

) CR KEY . [rtn] ?

) KEY CR . [rtn] ?

) KEY . CR [rtn] ?

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

`+ ( x y -- z )` Hole zwei Zahlen vom Stack herunter, zähle sie zusammen und lege das Ergebnis wieder auf den Stack.

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

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

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

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

Sie können die Reihenfolge von Zahlen auf dem Stack verändern:

`SWAP ( x y -- y x )` Vertausche die letzten beiden Zahlen auf dem Stack.

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

`DUP ( x -- x x )` Stelle von der letzten Zahl ein Extra-Exemplar her.

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

? Wenn Sie ausrechnen sollen, wieviel `500+14*(61-52)` ist, welche der unten stehenden Lösungswege sind dann nicht richtig? Und welche finden Sie am übersichtlichsten?

```
) 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 ( -- )` Lass die Worte sehen, die Forth kennt. Die neuesten Worte stehen ganz vorn.

Der letzte Satz wird verständlich, wenn Sie wissen, dass Sie Worte hinzuerfinden können: **Programmieren in Forth = Neue Worte machen**

Das geht wie folgt:

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

Beachten Sie, dass Worte durch einen oder mehrere Zwischenräume voneinander getrennt werden müssen.

- Das erste Wort, der Doppelpunkt, leitet die Definition eines neuen Wortes ein.
- Das zweite Wort ist `PILS`
Wie Sie schon vermuten werden, ist das kein bestehendes Forth-Wort. Das erste Wort nach dem Doppelpunkt wird der Name des neu zu bildenden Wortes. Diesen Namen können Sie frei wählen (Länge höchstens 31 Zeichen, keine Zwischenräume, denn die dienen der Worttrennung). Wenn Ihnen `KAFFEE` besser gefällt, geht das auch.
- Dann folgen da drei Worte, die Sie bereits kennen. Sie werden in einer Liste hinter dem Namen aufgeführt und bewirken, dass sie erst später, wenn Sie das neue Wort verwenden, in Aktion treten.
- Der Strichpunkt am Ende schließt die Definition ab. Forth wurde nun um das Wort `PILS` erweitert.

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

Technisch unterscheidet sich `PILS` nicht von den bestehenden Forth-Worten. Es ist kein zweitrangiges Wort und es arbeitet auch nicht langsamer als die bestehenden Worte.

`: ccc ( -- )` Stelle eine Definition mit Namen `ccc` her. Unter `ccc` verstehe ich einen willkürlichen Namen.

`; ( -- )` Beende eine Definition.

`PILS ( x -- )` Gib die Rechnung für `x` Pils aus.

```
) 1 PILS [rtn] ?
) 4 PILS [rtn] ?
) 10 PILS [rtn] ?
) : KASTEN 20 * ; [rtn] ok
) 2 KASTEN PILS [rtn] ? ( Und wenn Sie genug hatten: )
) FORGET PILS [rtn] ok
```

`FORGET ccc ( -- )` Entferne das Wort `ccc` aus Forth. Mit `ccc` verschwinden auch alle Worte, die neuer als `ccc` sind.

`.S ( -- )` Lass sehen, welche Zahlen sich auf dem Stack befinden, ohne den Stack zu verändern (Punkt S).

? Tippen Sie die unten stehenden Zeilen ein und stellen Sie fest, was die Worte `DROP` und `OVER` tun. Schreiben Sie jedes Mal, bevor Sie auf `[rtn]` drücken, auf einen Zettel, welches Resultat Sie erwarten.

```
) ARNHEIM [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] ?
```

`ARNHEIM` ist kein Forth-Wort und verursacht eine Fehlermeldung. Das ist natürlich mager. Aber Fehlermeldungen leeren auch immer den Stack, und darum ging es mir hier.

? 32 ist der ASCII-Code für einen Zwischenraum. Können Sie diesen Zwischenraum auf dem Bildschirm entdecken?

In der letzten Zeile macht `DROP` den Versuch, eine Zahl vom Stack zu entfernen. Der Stack ist jedoch leer. Das verursacht eine Fehlermeldung. Das dahinter stehende `.S` wird nicht mehr ausgeführt.

`BL ( -- c )` Lege den ASCII-Code des Zwischenraums (32) auf den Stack.

```
) : WEIT ( c -- ) EMIT BL EMIT ; [rtn] ok
```

Text zwischen Klammern dient nur der Information. Die brauchen Sie natürlich nicht einzutippen. Sie können das aber sehr wohl auch tun, denn Forth überspringt alles, was zwischen Klammern steht. Die öffnende Klammer ist ein Forth-Wort und **es muss daher ein Zwischenraum auf sie folgen.** Danach kann dann die Information kommen.

```
) 12 ( irgendwas Unsinniges ) . .S [rtn] ?
)
) 65 WEIT 66 WEIT 67 WEIT [rtn] ?
```

`( ( text -- )` Überspringe den Text bis einschließlich der ersten Schlussklammer, die folgt.

[wird fortgesetzt]

`CONSTANT ccc ( x -- )` Definiere eine Konstante mit dem Wert `x` und dem Namen `ccc`.

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

`DUTZEND ( -- 12 )` Lege den Wert der Konstanten auf den Stack.

`VARIABLE ccc ( -- )` Definiere eine Variable mit dem Namen `ccc`.

```
) VARIABLE BETRAG [rtn] ok
```

`BETRAG ( -- a )` Lege die Speicheradresse der Variablen auf den Stack.

`! ( x a -- )` Schreibe den Wert `x` in die Adresse `a`.

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

Nun hat `BETRAG` den Wert 200.
Das Wort `!` wird als "store" ausgesprochen.

`@ ( a -- x )` Lege den Inhalt von Adresse `a` auf den Stack. ("fetch")
`+! ( y a -- )` Addiere `y` zum Inhalt von Adresse `a`. ("plus store")

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

Achten Sie darauf, dass die neueste Zahl auf dem Stack IMMER eine Variable (eine Adresse) ist, wenn Sie die Worte `! @` und `+!` verwenden. Vor allem `!` und `+!` sind gefährliche Worte. Wenn Sie dabei Fehler machen, überschreiben Sie ungewollt etwas irgendwo im Speicher, und das kann "fatale" Folgen haben. Also:
`! @` und `+!` nur direkt hinter dem Namen einer Variablen verwenden.

```
) : AEPFEL ( n -- ) 80 * BETRAG +! ; [rtn] ok
) : BIRNEN ( n -- ) 70 * BETRAG +! ; [rtn] ok
) : TOTAL  ( -- ) BETRAG @ . 0 BETRAG ! ; [rtn] ok

) TOTAL [rtn] ?
) TOTAL [rtn] ?
) 5 AEPFEL 3 BIRNEN TOTAL [rtn] ?
)
) : EIER 60 * BETRAG +! ; [rtn] ok
) 2 BIRNEN 2 EIER TOTAL [rtn] ?
) DUTZEND EIER 10 AEPFEL 10 BIRNEN TOTAL [rtn] ?

) FORGET DUTZEND [rtn] ?
```

? Aufgabe.
Schreiben Sie aus dem Gedächtnis heraus hinter den bis jetzt gelernten Forth-Worten zwischen Klammern auf, welche Wirkung sie auf den Stack haben.
`( ... -- ... )`
Vor dem doppelten Minuszeichen stehen die Zahlen, die das Wort verbraucht.
Hinter dem doppelten Minuszeichen stehen die Zahlen, die das Wort erzeugt.
Geben Sie mit `ccc` an, dass das Wort einen Namen benötigt.

(Aus Kapitel 1 und 2)

```
EMIT
.
KEY
CR
```

(Aus Kapitel 3)

```
+
-
*
SWAP
DUP
```

(Aus Kapitel 4)

```
WORDS
:
;
FORGET
```

(Aus Kapitel 5)

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

(Aus Kapitel 6)

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

In einigen Forth-Systemen hinterlässt der Doppelpunkt beim Definieren einen Wert auf dem Stack, der vom Semikolon dann wieder abgeholt wird. Dieser Wert wird im systematischen Teil hinter diesem Kurs mit sys? angedeutet.

Eine schnelle und sichere Art herauszubekommen, ob Forth ein bestimmtes Wort kennt, geht über das einzelne Anführungszeichen (englisch: Tick).
`' ccc ( -- s )` Lege den Wortschlüssel von `ccc` auf den Stack. Wenn das Wort nicht gefunden wird, erscheint eine Fehlermeldung.
Jedes Wort hat außer seinem Namen noch eine eindeutige Kennzahl: den Schlüssel (englisch: Token).

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

Der Wert des Schlüssels ist systemabhängig. Zahlen haben im Allgemeinen keinen Namen und keinen Schlüssel.

`EXECUTE ( s -- )` Führe das Wort, dessen Schlüssel `s` ist, aus.

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

Es liegt beim Programmierer, dafür zu sorgen, dass der Schlüssel vor `EXECUTE` ein gültiger Schlüssel ist, d.h., dass er über das Forth-Wort `'` (Tick) erhalten wurde.

Beim Lesen und Ausführen eines Textes, den Sie eingetippt haben, sieht Forth Wort für Wort nach, ob es das Wort in seinem Wörterbuch finden kann.
Gefunden: Forth führt das Wort aus.
Nicht gefunden: Forth geht davon aus, dass es eine Zahl ist, die es auf den Stack legen soll. Erst wenn auch das nicht gelingt, erscheint eine Fehlermeldung.

Zum Überlegen:

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

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

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

) ' 7 EXECUTE . [rtn] ?

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

) FORGET IT [rtn] ok
```

Bisher haben Sie Anweisungen an Forth immer direkt von der Tastatur aus eingetippt. Wenn Sie beim Eintippen einer Definition einen Tippfehler begangen und Return gedrückt haben, können Sie diesen Fehler nicht so ohne weiteres ausbessern. Sie müssen die Definition wieder ganz von vorn anfangen. In den kommenden Kapiteln werden die Definitionen länger. Dann werden Sie es als angenehm empfinden, wenn Sie eine Möglichkeit haben, den Text ruhig einzutippen und von Fehlern zu befreien, bevor Forth ihn liest und ausführt. Hierfür gibt es drei Methoden. Lesen Sie sich diese Methoden durch und schauen Sie in der Anleitung zu Ihrem Forth nach, welche Sie verwenden können.

Methode 1: Blöcke, mit einem Editor innerhalb von Forth.
Forth betrachtet die gesamte Floppy als in nummerierte Blöcke, Screens, aufgeteilt, die direkt über ihre Blocknummern angesprochen werden können. Screen Nr. 0 ist nicht verwendbar. Die Zählung beginnt bei 1. Ein Screen besteht aus 16 Zeilen, jede mit Platz für 64 Zeichen.
`LIST ( n -- )` Lass den Text von Screen Nr. `n` sehen.
`LOAD ( n -- )` Lies den Text von Screen `n` und führe ihn so aus, als ob er geradewegs von der Tastatur aus eingetippt wird.
`THRU ( n1 n2 -- )` Load die Screens `n1` bis einschließlich `n2` hintereinanderweg.
Diese Methode ignoriert die Dateiverwaltung des Betriebssystems. Es ist also wichtig, die Blockdisketten sorgfältig von den Dateidisketten zu trennen. Für die Editor-Befehle verweise ich auf die Anleitung zu Ihrem Forth-System.

Methode 2: Dateien, mit einem Textbearbeiter innerhalb oder außerhalb von Forth.
Schreiben Sie mit einem Textverarbeitungssystem auf eine Datei und lassen Sie diese Datei von Forth lesen und ausführen.
`INCLUDE ccc ( -- )` Lies die Datei, die `ccc` heißt, und führe sie aus.
Bei dieser Methode können Sie Ihr bevorzugtes Textverarbeitungssystem verwenden, müssen dafür aber Forth verlassen. Wegen näherer Einzelheiten schauen Sie in Ihrem Handbuch nach.

Methode 3: Block-Dateien, mit einem Editor.
Diese Methode ähnelt der Block-Methode. Der einzige Unterschied besteht darin, dass eine bestimmte Anzahl von Blöcken miteinander eine Datei bilden, sodass das Betriebssystem die Oberhand behalten kann.
Sie können die Worte `LIST LOAD` und `THRU` verwenden. Wegen der näheren Einzelheiten und der Editor-Befehle schauen Sie bitte in der Anleitung zu Ihrem Forth nach.

Beispiel:

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

Neue Worte:
`." ccc" ( -- )` Gib den Text `ccc` aus. Da `."` (Punkt-Anführung) ein Wort ist, muss es vom auszugebenden Text durch einen Zwischenraum getrennt werden. Das zweite Anführungszeichen gibt das Textende an und ist nicht Bestandteil des Textes. Es dürfen sehr wohl Zwischenräume, aber auf gar keinen Fall Anführungszeichen in `ccc` vorkommen.

`= ( x y -- flag )` Prüfe, ob `x` gleich `y` ist, und hinterlasse ein Flag auf dem Stack. Das True-Flag ist gleich -1 und das False-Flag gleich 0.

`TRUE ( -- x )` `x=-1`, d.h., alle Bits von `x` sind gesetzt.
`FALSE ( -- x )` `x=0`, d.h., alle Bits von `x` sind null.

`IF ( flag -- )` Hole ein Flag vom Stack und reagiere wie folgt:
`[true?] IF [wenn ja, dann tue Folgendes]`
`ELSE [wenn nein, dann tue Folgendes]`
`THEN [usw.]`
`IF` betrachtet jeden Wert, der nicht gleich null ist, als ein True-Flag.

Die If-Else-Then-Konstruktion kann auch ohne `ELSE` vorkommen. Beispiel:

```
) : TEST ( x -- )
)   DUP . ." ist "
)   DUP 0 < IF ." kleiner als " THEN
)   DUP 0 > IF ." größer als " THEN
)   DROP ." null " ;
) CR 3 TEST CR -5 TEST CR 0 TEST [rtn] ?
```

Neue Worte:
`< ( x y -- flag )` Prüfe, ob `x<y` ist.
`> ( x y -- flag )` Prüfe, ob `x>y` ist.

? Aufgabe.
Erzeugen Sie ein Wort `ZIFFER? ( -- )`, das bei einer gedrückten Taste feststellt, ob es eine Ziffer ist. Auf dem Bildschirm sollte die Mitteilung 'Ziffer' oder 'keine Ziffer' erscheinen. Machen Sie dabei von dem Wort `KEY` Gebrauch.

Beispiel:

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

`DO ( grenzwert anfangswert -- )` Starte eine Schleife. Der Zähler beginnt mit dem Anfangswert. Grenz- und Anfangswert (man beachte deren Reihenfolge) werden vom Stack geholt und intern aufbewahrt.
`LOOP ( -- )` Erhöhe den Zähler um 1 und springe zum ersten Wort zurück, das hinter `DO` steht, gehe jedoch weiter, wenn der Zähler gleich dem Grenzwert geworden ist.
`I ( -- z )` Lege den Wert des Do-Loop-Zählers auf den Stack. `I` ist eine Abkürzung für Index.

Die Ein- und Ausgabe von Zahlen ging bis jetzt im Dezimalsystem (Basis 10) vor sich. Forth verwendet dabei die Variable `BASE` als Basiszahl. Indem Sie den Wert von `BASE` verändern, können Sie in anderen Zahlensystemen arbeiten. `HEX BINARY` und `DECIMAL` sind Forth-Worte, die in ein bestimmtes Zahlensystem umschalten. Wenn Sie noch kein `BINARY` haben:

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

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

`# $` oder `%` am Anfang einer Zahl bedeutet, dass die betreffende Zahl ungeachtet des momentanen Wertes von `BASE` in einem bestimmten Zahlensystem genommen werden soll. (Das trifft nicht auf jedes Forth zu.)

? Mit welchen Zahlensystemen korrespondieren diese Zeichen?

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

? Erklären Sie, was das Wort `LISTE` macht:

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

Beispiel:

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

Neue Worte:
`BEGIN ( -- )` Starte eine Schleife.
`UNTIL ( flag -- )` Mach weiter, wenn das Flag True ist, aber springe auf das erste Wort nach `BEGIN` zurück, wenn das Flag False ist.
`KEY? ( -- flag )` Prüfe, ob eine Taste gedrückt wurde.
Wenn die Antwort True lautet, kann mit `KEY` der ASCII-Code der Taste auf den Stack gelegt werden.

```
) O ZAEHL-DURCH . [rtn] ?
) O ZAEHL-DURCH ZAEHL-DURCH . [rtn] ?
```

Beispiel:

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

Neu:
`WHILE ( flag -- )` Mach weiter, wenn das Flag True ist, aber springe über `REPEAT` hinweg aus der Schleife heraus, wenn das Flag False ist.

? Was müssen Sie tun, um mit `SYMBOLE` sechs Minuszeichen erscheinen zu lassen?
? Können Sie das Wort `SYMBOLE` mit Hilfe von Do-Loop anstelle von Begin-While-Repeat erzeugen?
? Was macht das Wort `TEXT` ?

```
) : TEXT ( -- )
)   KEY CR
)   BEGIN KEY
)   DUP EMIT
)   OVER = UNTIL
)   DROP ; ( Ich hoffe, dass Sie hier herauskommen. )
```

Für einen Menschen ist eine Zahl eine Reihe von Ziffern. Wenn Forth eine solche Zahl hereinbekommt, übersetzt es die Ziffernfolge in ein Bitmuster und legt das auf den Stack. Bei der Übersetzung spielt `BASE` eine Rolle.
Wenn Forth ein Bitmuster als Zahl ausgeben soll (mit `.` beispielsweise), muss der umgekehrte Weg zurückgelegt werden.
Berechnungen führt Forth mit Zahlen aus, die bereits in Bitmuster übersetzt sind. `BASE` hat darauf keinen Einfluss mehr.

Wie ist nun die Beziehung zwischen 'menschlichen' Zahlen und Bitmustern?
Der Einfachheit halber gehe ich von einem ganz kleinen Forth aus, in welchem die Bitmuster aus nur vier Bits bestehen (in Wirklichkeit meist 16 oder 32 Bits). Hier folgen alle möglichen 4-Bit-Muster:

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

Die Muster, die mit einer 0 beginnen, nenne ich grün, die anderen, die vorn eine 1 haben, nenne ich rot.

Forth kennt zwei Methoden, die Muster in Zahlen zu übersetzen.

1) Vorzeichenlos, Unsigned. Keine negativen Zahlen. Null ist die kleinste Zahl. Wenn alle Bits gesetzt sind, ist die größte Zahl erreicht. Rot ist stets größer als grün:

```
0000 0001 0010 0011 0100 0101 0110 0111 (grüne Muster)
  0    1    2    3    4    5    6    7  (Zahlen)

1000 1001 1010 1011 1100 1101 1110 1111 (rote Muster)
  8    9   10   11   12   13   14   15  (Zahlen)
```

2) Mit Vorzeichen, Signed. Das anführende Bit wird als Vorzeichen aufgefasst, 1 deutet auf ein Minuszeichen. Rote Muster werden negative Zahlen und sind folglich kleiner als grüne:

```
1000 1001 1010 1011 1100 1101 1110 1111 (rote Muster)
 -8   -7   -6   -5   -4   -3   -2   -1  (Zahlen)

0000 0001 0010 0011 0100 0101 0110 0111 (grüne Muster)
  0    1    2    3    4    5    6    7  (Zahlen)
```

[wird fortgesetzt]

Gewöhnlich arbeitet man in Forth mit Signed-Zahlen. Für vorzeichenlose Zahlen gibt es in einigen Fällen Extra-Worte.
Neben dem . (Punkt) gibt es U. (u Punkt).
Das Wort . (Punkt) übersetzt nach Signed-Zahlen.
Das Wort U. (u Punkt) übersetzt nach Unsigned-Zahlen.

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

Neben < (kleiner als) gibt es U< (u kleiner als).

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

Beim Zusammenzählen und Abziehen braucht zwischen Signed und Unsigned kein Unterschied gemacht zu werden. Das ist so, weil Forth keine Überprüfung des Ergebnisses auf zu groß oder zu klein vornimmt. Überlauf und Unterlauf werden nicht gemeldet, der Programmierer muss das selbst auffangen. Das geht genauso wie beim Kilometerzähler im Auto. Wenn die höchste Zahl erscheint, lauter Neunen, und Sie fahren doch weiter, dann wird am Armaturenbrett keine Meldung im Stile von END-OF-AUTO-ERROR angezeigt.
Für beide Sorten von Zahlen gilt: Die größtmögliche Zahl plus 1 liefert die kleinstmögliche Zahl als Ergebnis.

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

? Wie viele Bits werden pro Zahl auf den Stack gelegt?

Eine Zelle ist die Menge an Speicherplatz, die eine Zahl als Bitmuster beansprucht.

CELLS (x -- y) y ist die Anzahl von Bytes, die für x Zellen benötigt werden.

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

? Wie viele Bytes werden pro Zahl auf den Stack gelegt?

? Lesen Sie die Beschreibung des Wortes LOOP in Kapitel 11 noch einmal genau nach und überlegen Sie, wie oft ein 'ha' erscheinen würde, wenn Sie auf die Idee kämen, das Wort WIEOFT? auszuführen.

```
) : WIEOFT? 0 0 DO ." ha " LOOP ; [rtn]
  ( Erst denken, dann handeln )
```

Forth ist ein Wörterbuch, an das neue Wörter (Forth-Worte) angefügt werden können. Das erfordert natürlich Speicherplatz. Mit dem Wort `HERE` können Sie sich anschauen, bis wohin das Wörterbuch geht und wo der Speicherbereich, der noch frei ist, beginnt.

`HERE ( - a ) a` ist die Adresse, wo neue Worte landen. Beim Definieren eines Wortes verändert sich diese Adresse.

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

Ein Forth-Wort besteht im Prinzip aus drei Elementen:

1. dem Namen, über den das Wort wiedergefunden werden kann,
2. dem Code, ein Stückchen Programm, das die eigentliche Arbeit verrichtet,
3. dem Body, der die Daten, die der Code benötigt, enthält.

`CREATE ccc ( -- )` Erzeuge ein Wort `ccc`, dessen Body noch leer ist.
Ein Wort, das mit `CREATE` erzeugt wurde, macht nichts anderes, als die Adresse seines Body auf den Stack zu legen.

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

`DORT ( -- body )` Lege die Adresse vom Body von `DORT` auf den Stack.

```
) DORT . [rtn] ?
```

Das ist also `HERE`

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

`ALLOT ( n -- )` Reserviere `n` Bytes bei `HERE`

Sie verfügen nun über einen Bereich von 10 Zellen ab einer Adresse, die `DORT` heißt. Sie können damit tun, was Sie wollen, Forth wird diesen Bereich von sich aus nicht mehr antasten.

[wird fortgesetzt]

`DORT` liefert eine Adresse. Sie können darauf dann also auch `@` und `!` ansetzen.

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

Sie müssen selbst dafür sorgen, dass Sie aus dem reservierten Bereich nicht hinausgelangen. So etwas wie

```
7 DORT 100 - !
```

wird wahrscheinlich einen überraschenden, aber sicher einen unerwünschten Effekt haben.

`DUMP ( a n -- )` Lass ab Adresse `a` den Inhalt eines Bereiches von `n` Bytes sehen.

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

`ACCEPT ( a n -- m )` Nimm maximal `n` Zeichen über die Tastatur entgegen und schreibe diese nach Adresse `a`. `m` ist die Zahl der tatsächlich eingegebenen Zeichen, die Länge des Eingabestrings (`m<n` oder `m=n`). Die Return-Taste schließt die Eingabe ab und zählt selbst nicht mit.

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

`CELL+ ( a1 -- a2 )` `a2` ist die Adresse, die um genau eine Zelle weiter liegt als `a1`.

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

ASCII-Codes für Zeichen liegen im Speicher dichter aneinander gepackt als Bitmuster für Zahlen. In eine einzige Zelle gehen meistens 2 oder 4 Zeichen.

`CHAR+ ( a1 -- a2 )` Adresse `a2` liegt um genau ein Zeichen weiter als `a1`.
`CHARS ( x -- y )` Für `x` Zeichen sind `y` Bytes nötig.

[wird fortgesetzt]

`C@ ( a -- c )` Lies Zeichen `c` aus Adresse `a`.
`C! ( c a -- )` Lege Zeichen `c` nach Adresse `a`.

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

Im oben stehenden Beispiel legt `C!` ein Bitmuster nach `DORT`, das angibt, wie viele Zeichen eingegeben wurden. Der Programmierer muss selbst in Erinnerung behalten, dass es sich hier um eine (kleine) Zahl handelt, und nicht um ein Zeichen. `C!` und `C@` unterscheiden sich von `!` und `@` durch die Länge der Bitmuster, die sie in Bewegung setzen. Mit dem Inhalt oder dessen Bedeutung beschäftigen sie sich nicht.
Mit `C!` gelangt nicht das gesamte Bitmuster, das auf dem Stack liegt, ins RAM; das vordere Stück wird einfach ignoriert. Beim Zurücklesen mit `C@` erscheinen dort dann wieder Nullen.

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

? Wie viele Bits liest `C@` ?

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

`TYPE ( a len -- )` Gib den Text aus, der an Adresse `a` beginnt und aus `len` Zeichen besteht.

Kurze Texte setzt man häufig als eine gezählte Zeichenkette (Counted String) in den Speicher. An der Stelle des ersten Zeichens steht, um wie viele Zeichen es hier geht, danach kommt der Text. Dadurch ist es nicht mehr nötig, an das Ende ein spezielles Zeichen zu setzen.

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

`COUNT ( a1 -- a2 c )` Lege den Inhalt der Zeichenzelle bei Adresse `a1` auf den Stack. Die Adresse `a2` liegt um genau ein Zeichen weiter als `a1`. Sie können mit `COUNT` einen Text durchlaufen:

```
) DORT COUNT . [rtn] ?
) COUNT EMIT [rtn] ?
) COUNT EMIT [rtn] ?
) COUNT EMIT [rtn] ? ( und so weiter )
) DROP [rtn] ok
```

Sie könnten `COUNT` also wie folgt definieren:

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

Zwischen den Klammern gebe ich stets die Stackverhältnisse an.

Die meisten Forth-Worte können Sie mit Hilfe von anderen Forth-Worten definieren. Zur Übung noch ein Beispiel:

```
) : TYPE ( a len -- )
)   DUP 0 >      ( a len groesser-als-null? )
)   IF 0         ( a len 0 )
)      DO        ( a )
)         COUNT  ( a+ c )
)         EMIT   ( a+ )
)      LOOP      \ zurueck nach COUNT
)   ELSE DROP    \ drop len
)   THEN DROP    \ drop Adresse a
) ;
```

`\ ( -- )` Ignoriere den Rest der Zeile. Das gilt für Forth, nicht für den Leser!

Am letzten Beispiel sehen Sie, dass If-Then und Do-Loop gleichzeitig auftreten können. Die beiden Strukturen müssen dann aber das sein, was man "geschachtelt" nennt. Man kann mitten in einer If-Then-Struktur eine Do-Loop beginnen, die Do-Loop muss dann aber vollständig abgeschlossen sein, bevor man das If-Then beendet.
Die Reihenfolge `.. IF .. DO .. THEN .. LOOP ..` ist also nicht möglich.
Allgemein gilt für If-Else-Then, Begin-Until, Begin-While-Repeat und Do-Loop, dass Sie diese in einer Definition uneingeschränkt schachteln können: Sie dürfen jederzeit mit solch einer Struktur beginnen, Sie dürfen aber immer nur an jener noch nicht verarbeiteten Struktur weiterarbeiten, die als letzte begann.

? Suchen Sie die Fehler:

```
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 )` Teile `x` durch `y`.
`q` (eine ganze Zahl!) ist das Ergebnis und `r` ist der Rest. Gesprochen: "slash mod".

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

`/ ( x y -- q )` Teile `x` durch `y`.
`q` ist der Quotient, eine ganze Zahl!
`MOD ( x y -- r )` Teile `x` durch `y`.
`r` ist der Rest. MOD ist eine Abkürzung für den mathematischen Ausdruck "modulo".

Bei dieser Art des Teilens sind nur ganze Zahlen betroffen.

- Sie kennen die Preise verschiedener Artikel ohne Mehrwertsteuer und Sie wollen die Preise mit Mehrwertsteuer (16%) berechnen:

```
) : MIT1 ( preis-ohne-MWST -- )   100 / 116 * . ;
) : MIT2 ( preis-ohne-MWST -- )   116 * 100 / . ;
```

? Welche Methode (ausprobieren!) arbeitet am besten, `MIT1` oder `MIT2`? Können Sie den Unterschied erklären?

Besser ist:

```
) : MIT3 ( netto-preis -- )   116 100 */ . ;
```

`*/ ( x y z -- x*y/z )` Der Vorteil des zusammengesetzten Operators `*/` gegenüber einzeln angewandtem `*` und `/` liegt darin, dass bei `*/` das Zwischenergebnis `x*y` eine Zahl (ein Bitmuster) werden darf, die für den Stack zu groß (zu lang) wäre. Gesprochen: "star slash".

Noch besser ist:

```
) VARIABLE PROZENTSATZ 16 PROZENTSATZ ! [rtn]
) : MIT ( netto-preis -- endbetrag)
)   100 PROZENTSATZ @ +   100 */ ;
) : .MIT MIT . ;
```

`MIT` ist nicht mehr von der Höhe der MWST abhängig. Sie können damit auch Abzüge mitberechnen:

```
) -15 PROZENTSATZ !
) 1745 .MIT [rtn] ?
) 495 .MIT [rtn] ?
) 1245 MIT .MIT [rtn] ? ( doppelter Abzug )
```

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

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

Überblick über die bisherigen Teilungsworte: `*/MOD */ /MOD / MOD`

Im folgenden Kapitel will ich ein etwas komplizierteres Wort angehen, welches das Verhältnis zweier Zahlen als Dezimalbruch ausgibt. Zur Vorbereitung erst noch ein paar neue Worte.

`AND ( x y -- z )` `x` und `y` werden Bit für Bit durch ein logisches Und verknüpft. Ein Bit in z ist 1 dann und nur dann, wenn das zugehörige x-Bit und das zugehörige y-Bit beide 1 sind. Entsprechend:
`OR ( x y -- z )` Bitweises logisches Oder. Ein Bit in z ist 0 dann und nur dann, wenn das zugehörige x-Bit und das zugehörige y-Bit beide 0 sind.
`XOR ( x y -- z )` Bitweises logisches exklusives Oder. Ein Bit in z ist 1 dann und nur dann, wenn das zugehörige x-Bit vom zugehörigen y-Bit verschieden ist.

```
) : NEGATIV? ( x y -- flag )
)   XOR 0 < ;
```

? Wie hängt das Flag, das `NEGATIV?` liefert, mit den Bitmustern von `x` und `y` zusammen?

`2DROP ( x y -- )` Entferne 2 Zahlen vom Stack.
`2DUP ( x y -- x y x y )`

Sie können `2DROP` wie folgt definieren:
`: 2DROP DROP DROP ;`
? Wie wird wohl die Definition von `2DUP` aussehen?

```
) : PUNKT ." ." ;
```
? Was macht `PUNKT`?

`.R ( x n -- )` Gib die Zahl `x` aus, rechtsbündig in einem Feld von `n` Stellen, ohne abschließenden Zwischenraum. Wenn die Zahl dort nicht hineinpasst, wird sie trotzdem vollständig ausgegeben. (Punkt 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` ist der Absolutwert von `x`.

[wird fortgesetzt]

Hier kommt das angekündigte Wort. Ich bringe es zweimal, erst in kahler Form, danach mit hinzugefügten Erklärungen.

```
) VARIABLE DEZIMALSTELLEN  9 DEZIMALSTELLEN !
) : .BRUCH ( x y -- )    \ Gib x/y als Dezimalbruch aus
)   2DUP XOR 0<
)   IF ." -"
)   THEN ABS
)   SWAP ABS
)   OVER
)   /MOD
)   1 .R
)   DEZIMALSTELLEN @ 0 >
)   IF ." ."
)      DEZIMALSTELLEN @ 0 DO
)         OVER
)         BASE @ SWAP
)         */MOD
)         1 .R
)      LOOP
)   THEN 2DROP ;

VARIABLE DEZIMALSTELLEN \ Anzahl auszugebender Dezimalstellen
9 DEZIMALSTELLEN !  \ Beispielsweise 9
: .BRUCH ( x y -- ) \ Gib x/y als Dezimalbruch aus
  2DUP XOR 0< ( x y flag )  \ Negatives Ergebnis?
  IF ." -"                  \ Dann Minuszeichen ausgeben
  THEN ABS ( x y )          \ y ist nun sicher positiv
  SWAP ABS ( y x )          \ x ist nun sicher positiv
  OVER     ( y x y )
  /MOD     ( y rest x/y )   \ Dieses x/y bildet den
                            \ Teil vor dem Komma
  1 .R     ( y rest )       \ Gib dieses x/y aus
  DEZIMALSTELLEN @ 0 > ( y rest flag )
  IF ." ."             ( y rest ) \ Gib Dezimalpunkt aus
     DEZIMALSTELLEN @ 0 DO ( y rest )
        OVER           ( y rest y )
        BASE @ SWAP    ( y rest basiszahl y )
        */MOD          ( y neuer-rest naechste-ziffer )
    \ Malnehmen mit der Basiszahl ist dasselbe wie das
    \ Herunterholen einer Null bei nicht aufgegangener Teilung
        1 .R           ( y neuer-rest )
     LOOP              ( y rest )
  THEN 2DROP ;

) 3 8 .BRUCH [rtn] ?
) 30 DEZIMALSTELLEN ! [rtn] ok
) 1 31 .BRUCH [rtn] ?
```

Manche Wortpaare kommen so häufig vor, dass es dafür Extra-Worte gibt.

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

? Beschreiben Sie die Stackwirkung dieser Worte.

In der Praxis werden diese Worte häufig nicht mit dem Doppelpunkt gebildet, sondern direkt in Maschinensprache geschrieben. Solche Worte nennt man "primitives" (Lo-Level-Worte). Sie sind vom Mikroprozessor abhängig. Vor allen Dingen, wenn Worte elementare Aufgaben verrichten und mit nur wenigen Assemblerbefehlen formuliert werden können, werden es Primitives sein. Beispiele: `+ - SWAP DUP DROP OVER ! @ +! = TRUE FALSE <> U< CELL+ C! C@` usw. Auch `COUNT`, von dem ich ein paar Kapitel weiter vorn eine Hi-Level-Definition gegeben habe, wird meistens ein Primitive sein.

Doppelpunkt-Worte (Hi-Level-Worte) sind aus bereits bestehenden Forth-Worten aufgebaut und unabhängig vom Mikroprozessor analysierbar. In manchen Forth-Systemen finden Sie zur Analyse das Wort `SEE`.
`SEE ccc ( -- ) SEE` mit einem Hi-Level-Wort im Gefolge gibt Einsicht in den Aufbau des betreffenden Wortes.

Neben `CONSTANT` und `VARIABLE` gibt es noch eine weitere Möglichkeit, Zahlen mit einem Namen zu versehen:

`x VALUE ccc ( x -- )` Definiere einen Value `ccc` mit dem Wert `x`.
`ccc ( -- )` Lege den momentanen Wert von `ccc` auf den Stack.
Mit `y TO ccc` setzen Sie `ccc` auf den Wert `y`. Sie können `!` und `@` bei Values nicht verwenden, da die Adresse unbekannt ist.

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

Vergleich von `CONSTANT VARIABLE VALUE`:

Definieren	`77 CONSTANT KON`	`VARIABLE VAR`	`12 VALUE VAL`
Zahl ablegen	Geht nicht.	`24 VAR !`	`24 TO VAL`
Zahl auslesen	`KON`	`VAR @`	`VAL`

Der größte gemeinsame Teiler zweier Zahlen `x` und `y` ist die größte Zahl (GGT), für welche die Divisionen `x/GGT` und `y/GGT` aufgehen, d.h. den Rest null liefern. Wenn Sie den Bruch `x/y` vereinfachen (kürzen) wollen, müssen Sie Zähler und Nenner durch ihren GGT teilen.

Algorithmus (Rezept) zum Auffinden des GGTs zweier Zahlen.

1. Nenne die größte Zahl `g` und die kleinste `k`.
2. Wenn `g=0` ist, dann `GGT=1`.
3. Wenn `k=0` ist, dann `GGT=g`.
4. Ersetze `g` durch den Rest, der entsteht, wenn `g` durch `k` geteilt wird.
5. Gehe nach 1.

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

? Beantworten Sie die folgenden vier Fragen, indem Sie das Programm per Hand, d.h. mit Bleistift und Papier und ohne Computer, Wort für Wort ausführen.

1. Wie sieht das Ergebnis aus, wenn `x=0` und `y=0` ist?
2. Wie sieht das Ergebnis aus, wenn `x=0` und `y=5` ist?
3. Wie sieht das Ergebnis aus, wenn `x=-4` und `y=0` ist?
4. Wie sieht das Ergebnis aus, wenn `x=3` und `y=-6` ist?

? Vermerken Sie hinter jeder Zeile die Verhältnisse auf dem Stack.

? Geben Sie genau an, welche Teile des Programms den fünf Vorschriften des Algorithmus entsprechen.

Dringender Rat: **Merken Sie beim Untersuchen einer Definition hinter jeder Zeile stets an, was auf dem Stack liegt.**

Außer dem gewöhnlichen Stack hat Forth noch den Return-Stack. Wenn es auf dem gewöhnlichen Stack kompliziert zu werden droht, kann man auch ein oder zwei Zahlen kurz auf den Return-Stack legen.

`>R ( x -- ) ( R: -- x )` Lege `x` auf den Return-Stack.
`R> ( -- x ) ( R: x -- )` Hole `x` wieder vom Return-Stack herunter.
`R@ ( -- x ) ( R: x -- x )` Lies das `x` auf dem Return-Stack aus.

Beispiel, zunächst wieder ohne Kommentar, danach mit.

```
) \ Vereinfache den Bruch x1/y1 zu x2/y2.
) : VEREINFACHE ( x1 y1 -- x2 y2 )
)   2DUP GGT
)   TUCK
)   /
)   >R
)   /
)   R>
) ;
```

? Merken Sie, bevor Sie weiterlesen, hinter jeder Zeile die Verhältnisse auf dem Stack an.

```
\ Vereinfache den Bruch x1/y1 zu x2/y2.
: VEREINFACHE ( x1 y1 -- x2 y2 )
  2DUP GGT    ( x1 y1 ggt )
  TUCK        ( x1 ggt y1 ggt )
  /           ( x1 ggt y2 )
  >R          ( x1 ggt )   ( R: y2 )
  /           ( x2 )       ( R: y2 )
  R>          ( x2 y2 )
;

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

Da mir `VEREINFACHE` recht lang vorkommt, gebe ich ihm einen kürzeren Namen.

```
) : VF VEREINFACHE ;
) 1234 2468 VF . . [rtn] ?
```

Die Worte `:` und `;` machen vom Return-Stack Gebrauch. `DO I` und `LOOP` zumeist auch. Darum gelten die folgenden Beschränkungen:

1. `>R` und `R>` müssen in Bezug auf `:` `;` und `DO LOOP` paarweise aufeinander abgestimmt sein.
2. `>R` und `R>` dürfen nur innerhalb einer Definition vorkommen.
3. In einer Do-Loop liefert `I` zwischen `>R` und `R>` nicht den richtigen Wert.

```
) : PLUS + ;
) : + ( x y -- )
)   2DROP CR ."  Heute wird nicht addiert " ;
```

Während der Definition von + teilt Forth mit, dass es da schon eine Definition namens + gibt. Das ist keine Fehlermeldung. Es ist lediglich eine Mitteilung an den Programmierer. Das Wort wird sehr wohl aufgebaut.
Da neue Worte an die Wortliste ganz hinten angefügt werden und Forth immer ganz hinten zu suchen beginnt, können Sie von nun an nur noch die neue Bedeutung von + aufrufen.

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

Die alte Bedeutung bleibt aber immer noch bestehen und Definitionen, in welchen das alte + vorkommt, arbeiten weiterhin in gewohnter Weise.

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

Nach

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

ist das alte + wieder ansprechbar, denn FORGET hat nur das neue + gefunden.

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

? Was bewirkt die folgende Definition?

```
: FORGET ."  Nie! " ;
```

In der Programmiersprache Forth haben Sie große Freiheiten. Die Devise lautet: Alles muss möglich sein. Der Preis dafür ist, dass Sie niemals gewarnt werden, wenn Sie etwas Ungeschicktes machen. Der Programmierer trägt die Verantwortung dafür, was er tut, und muss daher auch genau wissen, was er tut.
Der eine empfindet es als unangenehm, wenn er sich stets bewusst machen muss, was er da gerade tut, und wünscht sich durch ein PIEP - Syntax Error oder was auch immer für Error oder auf eine strukturelle Art, vor sich selbst beschützt zu werden. Der andere bejubelt gerade den Umstand, dass ihm Forth den vollständigen Zugang zum Computer verschafft, "man kommt überall ran".

Bei `V+` (Addition zweier Brüche) wird es auf dem Stack auch eng. Diesmal löse ich das Problem durch die Verwendung von Values.

```
) 0 VALUE ZAEHLER1 0 VALUE NENNER1
) 0 VALUE ZAEHLER2 0 VALUE NENNER2
)
) : V+ ( z1 n1 z2 n2 -- z3 n3 ) \ addiere 2 Brüche
)   TO NENNER2 TO ZAEHLER2
)   TO NENNER1 TO ZAEHLER1
)   NENNER1  NENNER2 GGT  >R
)   ZAEHLER1 NENNER2 R@   */
)   ZAEHLER2 NENNER1 R@   */ +
)   NENNER1  NENNER2 R>   */
)   VEREINFACHE ;
```

? Halten Sie, bevor Sie weiterlesen, hinter jeder Zeile die Stackverhältnisse fest.

```
: V+ ( z1 n1 z2 n2 -- z3 n3 )
\ Berechne den GGT von n1 und n2
\ z3 = z1*n2/ggt + z2*n1/ggt  und  n3 = n1*n2/ggt
  TO NENNER2 TO ZAEHLER2 ( z1 n1 )
  TO NENNER1 TO ZAEHLER1 ( -- )
  NENNER1  NENNER2 GGT   ( ggt )
  >R                  ( -- )                  ( R: ggt )
  ZAEHLER1 NENNER2 R@ ( z1 n2 ggt )           ( R: ggt )
  */                  ( z1*n2/ggt )           ( R: ggt )
  ZAEHLER2 NENNER1 R@ ( z1*n2/ggt z2 n1 ggt ) ( R: ggt )
  */                  ( z1*n2/ggt z2*n1/ggt ) ( R: ggt )
  +                   ( z3 )                  ( R: ggt )
  NENNER1  NENNER2 R> ( n1 n2 ggt )           ( R: -- )
  */                  ( z3 n3 )
  VEREINFACHE         ( z3 n3 )
;

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

`NEGATE ( x -- -x )` Ändere das Vorzeichen von `x`.

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

? Was macht `V-` ?

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

Zum Schluss `V*` und `V/`

```
) : V* ( z1 n1 z2 n2 -- z3 n3 )
)   TO NENNER2
)   VEREINFACHE
)   TO ZAEHLER2 TO NENNER1
)   NENNER2
)   VEREINFACHE
)   TO NENNER2
)   ZAEHLER2 *
)   NENNER1 NENNER2 *
) ;
)
) : V/ ( z1 n1 z2 n2 -- z3 n3 )
)   SWAP
)   V*
) ;
```

? Man halte hinter jeder Zeile die Stackverhältnisse mit Kommentar fest.

An Stelle von Values wird man unter diesen Umständen eher Lokale Values (Locals) anwenden. Locals werden innerhalb der Definition des Wortes, das diese Locals verwendet, angefertigt. Wenn die Definition fertig ist, sind die Locals nicht mehr sichtbar. Da das Definieren von Locals nicht in allen Forth-Systemen auf gleiche Weise vor sich geht, ist es hier nicht angebracht, mehr darüber zu erklären.

```
) : V. .BRUCH ; ( der Eindeutigkeit halber )
```

Jetzt verfügen Sie zum Rechnen mit Verhältnissen oder Brüchen über die folgenden Worte:

`V+` (zusammenzählen)
`V-` (abziehen)
`V*` (malnehmen)
`V/` (teilen)
`V.` (ausgeben)
`DEZIMALSTELLEN` (Zahl der Dezimalstellen hinter dem Komma)
`VF` (vereinfachen, nach Eingabe eines Bruches)

Die Eingabe von 1,25 geht also folgendermaßen vonstatten:

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

In Forth ist es möglich, mit Zahlen zu arbeiten, die mehr als eine Zelle an Platz einnehmen. Solche Zahlen, doppeltgenaue Zahlen, haben nur einen einzigen Wert, nehmen aber zwei Plätze auf dem Stack in Beschlag.

`S>D ( x -- xlo xhi )` Erweitere eine einfachgenaue Signed-Zahl zu einer doppeltgenauen Signed-Zahl gleichen Wertes. (xlo=x)
`D. ( xlo xhi -- )` Gib die letzten beiden Einträge auf dem Stack als eine einzige doppeltgenaue Zahl aus.

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

Wenn Sie bei Eingabe einer Zahl einen Punkt verwenden, macht Forth eine doppeltgenaue Zahl daraus.

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

Der Platz des Punktes in der Zahl ist nicht von Belang.

`D>S ( xlo xhi -- x )` Kürze eine doppeltgenaue Zahl zu einer gewöhnlichen Zahl. Da hierbei keine Fehlermeldung auftaucht, wenn der Wert nicht in eine Zelle passt, kann sich der Wert dadurch verändern.

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

Mehr Worte für doppeltgenaue Zahlen.

```
2DROP ( xlo xhi -- )
2DUP ( ? )
2OVER ( ? )
2SWAP ( ? )
2ROT ( ? )
D+ ( xlo xhi ylo yhi -- zlo zhi ) z ist die Summe von x und y.
D- ( ? )
D2* ( xlo xhi -- ylo yhi )
D2/ ( xlo xhi -- ylo yhi )
D= ( ? )
D< ( ? )
DNEGATE ( ? )
```

? Finden Sie anhand von selbst gemachten Beispielen genau heraus, was die oben stehenden Worte machen.

Eine mögliche Definition von S>D ist

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

Bei einer negativen (einfachgenauen) Zahl wird dabei True vorangesetzt, ansonsten False.

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

Definition von D>S. Die vordere Zelle wird einfach weggeworfen:

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

Beispiel. Ein Wort, mit dem untersucht werden kann, ob der Wert einer doppeltgenauen Zahl in eine Zelle passt.

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

Das geht auch kürzer.

```
) : EINFACH? ( xlo xhi -- xlo xhi flag )
)   OVER 0<     ( xlo xhi xhi' )
)   OVER = ;
```

? Was bedeutet das von EINFACH? gelieferte Flag?

? Für Knobler: aus einer Unsigned Number machen Sie eine doppeltgenaue Zahl, indem Sie Null davor setzen. Erklären Sie, warum MAX-N den Wert der größten (positiven) einfachgenauen Zahl liefert.

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

? Erklären Sie, warum MIN-N den Wert der kleinsten (negativen) einfachgenauen Zahl liefert.

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

? Bilden Sie so auch die Worte MAX-DN und MIN-DN für doppeltgenaue Zahlen. Hinweis: Wie sehen diese Zahlen hexadezimal aus?

Die Stärke der V-Worte aus den vorigen Kapiteln besteht darin, dass die Ergebnisse so genau sind, wie Sie es nur wollen. Berechnen Sie beispielsweise
`4/7 + 4/9 - 1/63`

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

Ein Schwachpunkt besteht darin, dass ganz leicht die Situation entstehen kann, dass Zähler oder Nenner nicht mehr in eine Zelle passen.

```
) 100 1 VF [rtn] ok
```
Wiederholen Sie nun einige Male die folgende Zeile:
```
) 2DUP V* 2DUP V. [rtn] ?
```

Irgendwo in der Berechnungsfolge wird, abhängig von der Zellgröße, ein unsinniges Ergebnis auftreten. Das wird durch einen Überlauf beim Malnehmen `*`, das in `V*` zweimal vorkommt, verursacht.

```
) 100 [rtn] ok
```
Wiederholen Sie nun einige Male die folgende Zeile:
```
) DUP * DUP . [rtn] ?
```

Siehe da, der Übeltäter. Das Unangenehme dabei ist, dass Sie hinterher nicht feststellen können, ob es gut oder schlecht gelaufen ist. Doch es besteht Hoffnung:

`M* ( x y -- zlo zhi )` Nimm `x` mit `y` mal und leg das Ergebnis `z` als doppeltgenaue Zahl auf den Stack. Es tritt nie ein Überlauf auf. Das `M` steht für Mixed. Bei diesem Wort sind Single und Double Numbers (gemischt) beteiligt.

`*` könnte wie folgt definiert sein.

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

Eine abgesicherte Version:

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

Neu:
`<>` `( x y -- flag )` Prüfe, ob `y` ungleich `x` ist.
`ABORT" ccc" ( flag -- )` False: Keine Aktion.
True: Steige aus dem Programm aus und gib den Text `ccc` aus.

- Eingesandter Brief -

Sehr geehrter Herr Nijhof,

Anlässlich einer Versammlung des FFW, des Fachverbandes der Forth-Worte, kam Ihr Kurs zur Sprache. Im positiven Sinne übrigens. Aber lassen Sie mich gleich zur Sache kommen.

In Kapitel 4, bei der Besprechung des Wortes PILS, schreiben Sie, dass beim Anfertigen einer Definition die Worte in einer Liste hinter dem neuen Namen zu liegen kommen. Das ist aber doch eine recht einfache Darstellung der Dinge, die einigen von uns nicht gerecht wird. Es ist in der Tat wahr, dass sich die meisten Forth-Worte so passiv gebärden, dass sie sich beim Definieren ohne weiteres in eine Tabelle aufnehmen lassen.
Jedoch denken die Leute aus meiner Abteilung, die Immediate-Words, anders darüber. Wir lassen uns nicht so ohne weiteres in eine Tabelle setzen. Und mehr noch, einige von uns, darunter ich selbst (mein Name ist Dot Quote), sind von der Compile-Gruppe; wir selbst beschäftigen uns mit dem Zusammenstellen solcher Listen. Ich persönlich habe zum Beispiel die Aufgabe, immer wenn ich aufgerufen werde, eine Schreibkraft in die Tabelle zu setzen, danach den hereinkommenden Text so lange zu durchsuchen, bis ich ein Anführungszeichen finde, und daraufhin dann das betreffende Stück Text hinter die Schreibkraft zu platzieren. Wir nennen das Compilieren.

(Darf ich zwischendurch mal schnell bemerken, dass Programmierer recht schlampig sein können. Es kommt nämlich vor, dass kein Anführungszeichen zu erkennen ist. Da nörgle ich dann nicht viel rum. Ich nehme dann einfach den gesamten verbleibenden Text auf. Man sei gewarnt. Vielleicht können Sie in Ihrem Kurs darüber etwas sagen.)

Absicht dabei ist, dass bei Ausführung des neuen Wortes die Schreibkraft den Text schreibt. Ich habe also meine Untergebenen. Meine Arbeit ist doch etwas komplizierter, als Sie es erscheinen lassen!
In meinem vorigen Wirkungskreis hatte ich obendrein die Aufgabe, wenn ich außerhalb einer Definition aufgerufen wurde, den Text direkt selbst auszugeben. Das war so die Art des Kundendienstes, und man war leider auch immer stets in Eile. Aber da waren seinerzeit mehr Dinge nicht gut geregelt.
Gott sei Dank habe ich das nun nicht mehr nötig. Meine Kollegin, .((Dot Paren), hat diese Aufgabe von mir übernommen, mit dem einzigen Unterschied, dass sie anstelle eines Anführungszeichens nach einer abschließenden runden Klammer sucht. Ich gönne jedem seine Eigenheiten. Sie ist zwar ein Immediate-Wort, im Kundendienst, und eine schnelle Schreibkraft, aber vom Compilieren hat sie keine Ahnung. Und ich, Dot Quote, bin jetzt Only Compiling.

Wenn Sie über die verschiedenen Immediate-Worte mehr wissen wollen, dann empfehle ich Ihnen, sie persönlich zu befragen. Sie sind gern bereit, Ihnen in das, was sie tun, Einblick zu verschaffen.

Hochachtungsvoll!

[Für die Immediate-Words:] ." (Dot Quote)

```
) : DOT-QUOTE ." Wasserfall " ; [rtn] ?
) SEE DOT-QUOTE [rtn] ?
) DOT-QUOTE [rtn] ?
)
) : DOT-PAREN .( Auf Wiedersehen! ) ; [rtn] ?
) SEE DOT-PAREN [rtn] ?
) DOT-PAREN [rtn] ?
```

Der Programmtext, den Forth zum Verarbeiten kriegt, heißt der Input-Strom. `."` und `.(` lesen selbst Text aus dem Input-Strom. Da sie Immediate sind, tun sie das auch während des Definierens.

`CHAR ( ccc -- c )` Lies das folgende Wort aus dem Input-Strom und lege den ASCII-Code des ersten Buchstabens auf den Stack.

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

Da `CHAR` nicht Immediate ist, tut es zur Zeit des Definierens noch nichts. Erst wenn das neue Wort ausgeführt wird, liest `CHAR` den Input-Strom.

```
) : BUCHSTABE ( ccc -- )
)   ." hat ASCII-Code "
)   CHAR . ;
)   BUCHSTABE A [rtn] ?
)   BUCHSTABE a [rtn] ?
```

`CHAR` hat eine Variante, die `[CHAR]` heißt.

`[CHAR] ( ccc -- )` Setze zur Zeit des Definierens den ASCII-Code des ersten Buchstabens von `ccc` als Zahl in die Definition. Da `[CHAR]` Immediate ist, tritt es zur Zeit des Compilierens in Aktion. Außerhalb einer Definition hat es keine Bedeutung.

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

Für Leser, die kein `SEE` in ihrem Forth haben: `A-KODE` und `KODE-A` liefern nicht nur dasselbe Ergebnis, sie sind auch genauso aufgebaut.

```
[CHAR] A
```

setzt während des Definierens die Zahl 65 in die Definition.

Sie können natürlich auch selbst Immediate-Worte machen.
`IMMEDIATE ( -- )` Mach das zuletzt definierte Wort Immediate.

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

`.S` lässt die Zahlen auf dem Stack sehen. `[.S]` macht dasselbe, jedoch während des Compilierens, und hinterlässt in der Definition keine Spuren.

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

? Können Sie eine einleuchtende Begründung dafür finden, warum `;` ein Immediate-Wort sein wird?

Zusammenfassend:

1. Immediate-Worte sind Worte, die sich nicht compilieren lassen. Sie werden zur Zeit des Definierens ausgeführt.
2. Compiler-Worte sind Worte, die zur Zeit des Definierens (Compilierens) das neu zu bildende Wort beeinflussen. Um das tun zu können, müssen sie natürlich Immediate sein. Außerhalb einer Definition sind sie sinnlos.

`."` ist ein Compiler-Wort, und folglich auch Immediate. Only Compiling.
`.(` ist Immediate, aber kein Compiler-Wort und hat keinen Effekt auf Definitionen.

`IF ELSE THEN BEGIN UNTIL WHILE REPEAT DO LOOP usw.` nennt man Steuerungsworte. Sie organisieren zur Compilierzeit die Verzweigungen. Sie sind also Compiler-Worte, außerhalb von Definitionen unbrauchbar.

Neue Steuerungsworte.
`?DO ( n2 n1 -- )` Beinahe dasselbe wie `DO`.
Der Unterschied besteht darin, dass `?DO` auf das erste Wort nach `LOOP` springt, wenn `n2` gleich `n1` ist.
Diese Erklärung von `?DO` ist sehr schlampig. Richtiger wäre:
`?DO ( -- )` (compilierend:) Installiere `?SCHLEIFENMACHER` in die Definition.
`?SCHLEIFENMACHER ( n2 n1 -- )` (ausführend:) Fang eine Schleife an, es sei denn, dass `n1=n2` ist.

`LEAVE ( -- )` Gehe zum ersten Wort nach `LOOP` (nur zwischen `DO` und `LOOP` zu verwenden).

Wortspiel

Spieler A denkt sich ein Wort aus, das Spieler B erraten muss. Nach jedem Versuch bekommt Spieler B zu sehen, wie viele Buchstaben seines Wortes im zu erratenden Wort vorkommen und wie viele bereits auf dem richtigen Platz stehen.

Das geheime Wort darf höchstens 31 Buchstaben lang sein.

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

Das ist der Platz, in welchem das geheime Wort, angeführt von einem Count, zu liegen kommt.

Das geheime Wort wird in Großbuchstaben eingetippt, ohne dass es auf dem Bildschirm zu sehen ist.

```
) : GEHEIM ( geheim$ maxlen -- )
)   OVER SWAP
)   0 DO
)       KEY >R
)       [CHAR] Z R@ <
)       R@ [CHAR] A <
)       OR IF R> DROP LEAVE THEN ( kein Grossbuchstabe )
)       CHAR+
)       R> OVER C!          ( notiere diesen Buchstaben )
)       [CHAR] * EMIT
)    LOOP
)    OVER -
)    SWAP C! ;        ( notiere die Zahl der Buchstaben )
```

? Unter welchen Umständen wird die Do-Loop in `GEHEIM` beendet?

Neu:
`SPACE ( -- )` Gib einen Zwischenraum aus.
`SPACES ( n -- )` Gib `n` Zwischenräume aus.

```
) : SPIELER-A ( -- )
)   ." Das geheime Wort:"
)   CR 4 SPACES
)   GEHEIM$ MAXLEN GEHEIM
)   SPACE ;
```

[wird fortgesetzt]

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

Platz für den Rateansatz des Spielers B (auch ein "counted string").

```
) : SPIELER-B ( -- )
)   RATEANSATZ$ CHAR+ GEHEIM$ C@
)   ACCEPT
)   RATEANSATZ$ C!
)   SPACE ;
)
) VARIABLE ZAEHLER
```

`ZAEHLER` ist für den internen Gebrauch in den Worten `BUCHSTABEN?` und `PLAETZE?` vorgesehen.
Bestimme, wie viele Buchstaben von `RATEANSATZ$` auch in `GEHEIM$` vorkommen:

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

Wenn zwei Do-Loops ineinander geschachtelt sind, gibt `I` den Zählerstand der inneren und `J` den Zählerstand der äußeren Schleife wieder.

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

Bestimme, wie viele Buchstaben in `RATEANSATZ$` bereits auf dem richtigen Platz stehen:

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

[wird fortgesetzt]

Untersuchung des Rateansatzes von Spieler B.

```
) : RESULTAT ( -- plaetze buchstaben )
)   RATEANSATZ$ GEHEIM$ PLAETZE?
)   RATEANSATZ$ GEHEIM$ BUCHSTABEN?
) ;
```

Spieler B gibt auf, indem er an den Platz des ersten Buchstabens ein Fragezeichen setzt.

```
) : AUFGEGEBEN? ( -- flag )
)   RATEANSATZ$ CHAR+ C@ [CHAR] ? =
) ;
```

Neu:
`PAGE ( -- )` Lösche den gesamten Bildschirm und setze den Cursor nach links oben.

Jetzt das Spiel selbst.

```
) : SPIEL ( -- )
)   PAGE SPIELER-A
)   ." Stimmende Buchstaben/auf dem richtigen Platz"
)   1 1
)   DO CR I 3 .R SPACE
)      SPIELER-B
)      AUFGEGEBEN? IF ."  nicht" LEAVE THEN
)      RESULTAT
)      #12 .R           ( Zahl stimmender Buchstaben )
)      [CHAR] / EMIT
)      DUP .                ( Zahl richtiger Plaetze )
)      GEHEIM$ C@ = IF LEAVE THEN
)   LOOP ."  geraten "
)   GEHEIM$ COUNT TYPE ;
```

? Was passiert, wenn Spieler A keinen einzigen Großbuchstaben eingetippt hat?

B

100 - 133

Inhalt der systematischen Übersicht

101. Der Stack
102. Der Return-Stack
103. Schreiben nach Adressen
104. Lesen aus Adressen
105. Adressen
106. Rund um `HERE`
107. Eingabe von der Tastatur
108. Von Ziffernfolge nach Bitmuster
109. Ausgabe
110. Ausgabe von Zahlen
111. Von Bitmuster nach Ziffernfolge
112. Steuerung I
113. Schleifen ohne Zähler
114. Schleifen mit Zähler
115. Steuerung II
116. Bits
117. Flags I
118. Flags II
119. Max und Min
120. Bearbeitung einer einzelnen Zahl
121. Rechnen
122. Rechnen mit doppeltgenauen Zahlen
123. Der Input-Strom
124. Anwendung von `BL WORD`
125. Anwendung von `PARSE`
126. Definierende Worte
127. Compilieren
128. Worte definieren, die Worte definieren
129. Der Wortschlüssel
130. Wortlisten
131. Verschiedenes I
132. Verschiedenes II
133. Fehler behandeln

Der Stack

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

`DEPTH ( -- n )`
`n` ist die Zahl der belegten Zellen auf dem Stack, `n` selbst nicht mitgerechnet.

`.S ( -- )` Gib den Stackinhalt aus, ohne den Stack zu verändern.

Stack-Manipulationen

`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 )` oder `( x -- x )` "question dupe"
Dupliziere `x` nur dann, wenn es nicht null ist.

Zuweilen findet man

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

Der Return-Stack

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

Beim Eintritt in ein High-Level-Wort und beim Verlassen eines solchen verwendet Forth den Return-Stack. Auch die Schleifenverwaltung einer Do-Loop steht häufig auf dem Return-Stack.
Der Programmierer kann den Return-Stack zum Drauflegen oder Herunterholen von Zahlen missbrauchen. Er muss dafür sorgen, dass er dabei Forth nicht in die Quere kommt. `>R` und `R>` müssen in Hinsicht auf den Gebrauch, den Forth selbst vom Return-Stack macht, stets korrekt aufeinander abgestimmt sein.

Die Situation auf dem Return-Stack gebe ich durch
`( R: .. -- .. )` wieder.

`>R ( x -- ) ( R: -- x )` "to r" Only Compiling
Lege `x` auf den Return-Stack.

`R> ( -- x ) ( R: x -- )` "r from" Only Compiling
Hole `x` vom Return-Stack herunter.

`R@ ( -- x ) ( R: x -- x )` "r fetch" Only Compiling
Lies `x` vom Return-Stack aus, ohne es dort wegzunehmen.

Analog (man beachte die Reihenfolge der Zellen):

`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

Zuweilen findet man

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

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

Schreiben nach Adressen

`! C! 2! +! ACCEPT FILL MOVE`

`! ( x a -- )` "store"
Lege `x` in die Zelle mit der Adresse `a`.

`C! ( c a -- )` "c store"
Lege `c` als Zeichen (mit abgeschnittenem vorderen Teil) auf die Adresse `a`.

`2! ( x y a -- )` "two store"
Lege `y` auf Adresse `a` und `x` in die Zelle danach.

`+! ( x a -- )` "plus store"
Erhöhe den Wert in der Zelle an Adresse `a` um `x` (Signed oder Unsigned).
`a` muss Aligned sein.

`ACCEPT ( a n -- n2 )`
Empfange höchstens `n` Zeichen über die Tastatur und lege sie ab Adresse `a` hintereinanderweg ab.
`n` muss größer als null sein und `n2` ist die Anzahl tatsächlich empfangenener Zeichen. Die Return-Taste schließt `ACCEPT` ab, zählt nicht mit und wird auch nicht notiert.

`FILL ( a n c -- )`
Fülle `n` Zeichenzellen ab Adresse `a` mit dem Zeichen `c`.
Bei `n=0` geschieht nichts.

`MOVE ( a1 a2 u -- )`
Kopiere den Bereich von `u` Bytes ab Adresse `a1` in den Bereich von `u` Bytes ab Adresse `a2`.
Diese Operation geht auch dann gut, wenn `a2` innerhalb des zu kopierenden Bereichs liegt. `u` ist Unsigned.

Lesen aus Adressen

`@ 2@ C@ COUNT C@+ @+`

`@ ( a -- x )` "fetch"
Lege den Inhalt der Zelle an Adresse `a` auf den Stack.

`2@ ( a -- y x )` "two fetch"
Lege den Inhalt der 2 Zellen ab Adresse `a ( x,y )` auf den Stack, erst `y`, dann `x` (`2@` ist das Gegenstück zu `2!`).

`C@ ( a -- c )` "c fetch"
Lies das Zeichen in Adresse `a` und lege es auf den Stack. Das Bitmuster wird vorn mit Nullen aufgefüllt, um auf Zellbreite zu kommen.

`COUNT ( a -- a2 c )`
`c` ist der ASCII-Code des Zeichens in Adresse `a`, und die Adresse `a2` liegt um genau ein Zeichen weiter als `a`.

Zuweilen findet man

`C@+ ( a -- a2 c )` "c fetch plus"
Synonym zu `COUNT`.

`@+ ( a -- a2 x )` "fetch plus"
`x` ist der Inhalt der Zelle auf Adresse `a`, und die Adresse `a2` liegt um genau eine Zelle weiter als `a`.

Adressen

`ALIGNED CELL+ CELLS CHAR+ CHARS CELL- CHAR-`

`ALIGNED ( a -- a2 )`
Die Adresse `a` wird, falls nötig, bis zu einer Adresse `a2` erhöht, an welcher eine Zelle beginnen kann. Das kann nach `CHAR+` oder `CHAR-` nötig werden. Es gibt Forth-Systeme, bei denen Zellen prozessorbedingt beispielsweise nur an geraden Adressen beginnen dürfen.

`CELL+ ( a -- a2 )` "cell plus"
Die Adresse `a2` liegt genau eine Zelle hinter `a`.

`CELLS ( x -- y )`
`x` Zellen beanspruchen `y` Bytes.

`CHAR+ ( a -- a2 )` "char plus"
Die Adresse `a2` liegt genau ein Zeichen hinter `a`.

`CHARS ( x -- y )` "chars"
`x` Zeichen beanspruchen `y` Bytes.

Zuweilen findet man

`CELL- ( a -- a2 )` "cell minus"
Die Adresse `a2` liegt genau eine Zelle vor `a`.

`CHAR- ( a -- a2 )` "char minus"
Die Adresse `a2` liegt genau ein Zeichen vor `a`.

Rund um HERE

HERE ALIGN , C, ALLOT

HERE (-- a)
Die Adresse a kennzeichnet das Ende des Wörterbuchs und den Anfang des freien Speicherbereichs (des Data Space). HERE eignet sich nicht zur Aufbewahrung von Daten, da Forth diese Adresse intensiv verwendet. Siehe PAD.

ALIGN (--)
Wenn es bei HERE nicht erlaubt ist, eine Zelle beginnen zu lassen, dann setze HERE auf den Wert der nächstbesten Adresse, an der das zulässig ist. Das kann nach C, oder ALLOT nötig werden. Siehe auch die bei ALIGNED eingefügte Bemerkung.

, (x --) "comma"
Reserviere eine Zelle bei HERE und lege x dorthin. Es liegt beim Programmierer, HERE so einzurichten, dass dort eine Zelle beginnen kann.

C, (c --) "c comma"
Reserviere Platz für ein Zeichen bei HERE und lege c dorthin.

ALLOT (x --)
Reserviere x Bytes bei HERE. Das neue HERE nimmt den Wert HERE+x an. x darf negativ sein.

Eingabe von der Tastatur

`ACCEPT KEY KEY? EKEY EKEY? EKEY>CHAR`

`ACCEPT ( a n -- n2 )`
Empfange höchstens `n` Zeichen über die Tastatur und lege sie ab Adresse `a` hintereinanderweg ab.
`n` muss größer als null sein und `n2` ist die Anzahl tatsächlich empfangenener Zeichen. Die Return-Taste schließt `ACCEPT` ab, zählt nicht mit und wird auch nicht notiert.

`KEY ( -- c )`
Warte, bis eine Taste gedrückt wurde. `c` ist der ASCII-Code dieser Taste. Gib kein Echo aus.

`KEY? ( -- flag )` "key question"
Prüfe, ob eine Taste gedrückt wurde. Im Falle `flag=True` kann der ASCII-Code mit `KEY` abgeholt werden.

Die Worte `KEY` und `KEY?` sind für Tasten bestimmt, die einen ASCII-Code zwischen 32 und einschließlich 126 ($20 bis $7E) liefern. Bei ASCII-Codes außerhalb dieses Bereichs hängt es von der Implementierung ab, welche Tasten angenommen werden und welche nicht.
Die unten stehenden `EKEY`-Worte arbeiten mit einer implementierungsabhängigen Codezahl (`ia`) für die Tasten. Damit können auch Tasten identifiziert werden, die keinen ASCII-Code haben.

`EKEY ( -- ia )` "e key"
Warte, bis eine Taste gedrückt wurde. `ia` ist die Codezahl dieser Taste. Gib kein Echo aus.

`EKEY? ( -- flag )` "e key question"
Prüfe, ob eine Taste gedrückt wurde. Im Falle `flag=True` kann die Codezahl mit `EKEY` abgeholt werden.

`EKEY>CHAR ( ia -- ia false )` oder `( ia -- c true )` "ekey to char"
Setze die Codezahl, wenn möglich, in einen ASCII-Code um.

Von Ziffernfolge zu Bitmuster

`BASE >NUMBER DECIMAL HEX BINARY`

`BASE ( -- a )`
In der Zelle mit Adresse `a` steht die Basiszahl, mit der Forth beim Umwandeln von Bitmustern in Zahlen und umgekehrt rechnet. Siehe auch bei `>NUMBER` und `<#` nach.

`>NUMBER ( ulo uhi a n -- vlo vhi a2 n2 )` "to number"
Analysiere zunächst einmal den String `a,n` von links nach rechts, Zeichen für Zeichen. Steht da eine gültige Ziffer (was von `BASE` abhängt), dann multipliziere `uhi,ulo` mit dem Wert von `BASE` und addiere dazu den Wert der betreffenden Ziffer. Halte an, wenn an der betrachteten Stelle keine Ziffer steht. `a2,n2` ist der String, der dabei übrig bleibt, beginnend mit der Nichtziffer. Wenn `n2=0` ist, dann ist der gesamte String umgewandelt.

`DECIMAL ( -- )`
Setze `BASE` auf 10.

`HEX ( -- )`
Setze `BASE` auf 16.

Zuweilen findet man

`BINARY ( -- )`
Setze `BASE` auf 2.

Ausgabe

`PAGE CR AT-XY SPACE SPACES EMIT TYPE .(`

`PAGE ( -- )`
Lösch den Bildschirm und setz den Cursor nach links oben (formfeed beim Drucker).

`CR ( -- )` "c r"
Geh zum Anfang einer neuen Zeile.

`AT-XY ( x y -- )` "at x y"
Beginn die nächste Textausgabe auf dem Bildschirm auf Position `x` von Zeile `y`. Bei `x=0` und `y=0` ist das links oben.

`SPACE ( -- )`
Gib einen Zwischenraum aus.

`SPACES ( n -- )`
Gib im Falle von `n` größer als null `n` Zwischenräume aus.

`EMIT ( c -- )`
Gib das Zeichen, das den ASCII-Code `c` hat, aus.

`TYPE ( a u -- )`
Gib im Falle von `u` ungleich null den Text `a,u` aus.

`.( ccc ( -- )` "dot paren" Immediate-Wort
Lies den Input-Strom bis zur ersten Schlussklammer und gib, auch im Compilier-Modus, den gelesenen Text aus. Der auf keinen Fall zu vergessende Zwischenraum nach `.(` und die Schlussklammer werden dabei nicht mit ausgegeben.

Zahlenausgabe

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

`. ( x -- )` "dot"
Gib `x` als Signed-Zahl mit einem Zwischenraum dahinter aus.

`.R ( x n -- )` "dot r"
Gib `x` als Signed-Zahl aus, rechtsbündig in einem Feld von `n` Stellen.

`U. ( u -- )` "u dot"
Gib `u` als Unsigned-Zahl mit einem Zwischenraum dahinter aus.

`U.R ( u n -- )` "u dot r"
Gib `u` als Unsigned-Zahl aus, rechtsbündig in einem Feld von `n` Stellen.

`D. ( xlo xhi -- )` "d dot"
Gib `xhi,xlo` als doppeltgenaue Signed-Zahl mit einem Zwischenraum dahinter aus.

`D.R ( xlo xhi n -- )` "d dot r"
Gib `xhi,xlo` als doppeltgenaue Signed-Zahl aus, rechtsbündig in einem Feld von `n` Stellen.

Zuweilen findet man

`DU. ( ulo uhi -- )` "d u dot"
Gib `uhi,ulo` als doppeltgenaue Unsigned-Zahl mit einem Zwischenraum dahinter aus.

`DU.R ( ulo uhi n -- )` "d dot r"
Gib `uhi,ulo` als Unsigned-Zahl aus, rechtsbündig in einem Feld von `n` Stellen.

Von Bitmuster zu Ziffernfolge

```
<# # #S HOLD SIGN #>
```

`<# ( ulo uhi -- ulo uhi )` "less number sign"
Vorbereitung zur Umwandlung des Bitmusters von `uhi,ulo` (Unsigned) in eine lesbare Zahl. Während der Umwandlung können `# #S HOLD` und `SIGN` ihre Arbeit tun, und `#>` schließt die Umwandlung ab.

`# ( ulo uhi -- vlo vhi )` "number sign"
Ermittle unter Verwendung von `BASE` die letzte Ziffer des Bitmusters `uhi,ulo` und lege diese an den Anfang des aufzubauenden Strings. `vhi,vlo` ist `uhi,ulo` ohne dessen letzte Ziffer. Nur zwischen `<#` und `#>` zu verwenden.

`#S ( ulo uhi -- 0 0 )` "number sign s"
Führe ein `#` aus und wiederhole das so lange, bis die doppeltgenaue Zahl auf dem Stack `0,0` ist. Nur zwischen `<#` und `#>` zu verwenden.

`HOLD ( c -- )`
Füge vorn an den aufzubauenden String das Zeichen `c` an. Nur zwischen `<#` und `#>` zu verwenden.

`SIGN ( x -- )`
Füge vorn an den aufzubauenden String genau dann, wenn `x` negativ ist, ein Minuszeichen an. Nur zwischen `<#` und `#>` zu verwenden.

`#> ( ulo uhi -- a n )` "number sign greater"
Schließe die Umwandlung ab. Der String `a,n` ist das Ergebnis.

Steuerung I

```
IF THEN
IF ELSE THEN
```

`IF ( -- sys1 )` Compiler-Wort
Compilierend: Compiliere einen Vorwärtssprung und hinterlasse eine Nachricht für ELSE oder THEN
Ausführend: `( flag -- )`
Wenn `flag` nicht null ist, fahre in der Ausführung einfach fort.
Wenn `flag` gleich null ist, mache hinter `ELSE` (oder hinter `THEN`, falls kein `ELSE` vorhanden ist) weiter.

`ELSE ( sys1 -- sys2 )` Compiler-Wort
Compilierend: Compiliere einen Sprung, hinterlasse eine Nachricht für `THEN` und füge mit Hilfe von `sys1` das Ziel des IF-Sprungs ein.
Ausführend: `( -- )`
Mache hinter `THEN` weiter.

`THEN ( sys1 -- )` Compiler-Wort
Compilierend: Füge mit Hilfe von `sys1` das Ziel des IF-Sprungs oder des ELSE-Sprungs ein.
Ausführend: `( -- )`
Keine Aktion.

`( -- sys1 )` bedeutet, dass eine Nachricht hinterlassen wird, und
`( sys1 -- )` bedeutet, dass eine Nachricht verarbeitet wird.
Das kann über den Stack geschehen, es gibt aber auch andere Möglichkeiten.
Diese Nachrichten sind nötig, da beim Compilieren eines Sprungs das Sprungziel noch nicht bekannt ist (`IF`) oder da das Sprungziel bereits bekannt ist, bevor der Sprung überhaupt compiliert wird (`BEGIN`).

Schleifen ohne Zähler

```
BEGIN UNTIL
BEGIN WHILE REPEAT
BEGIN AGAIN
```

`BEGIN ( -- sys1 )` Compiler-Wort
Compilierend: Hinterlasse eine Nachricht.
Ausführend: `( -- )`
Keine Aktion.

`UNTIL ( sys1 -- )` Compiler-Wort
Compilierend: Compiliere einen bedingten Sprung mit Hilfe von `sys1`.
Ausführend: `( flag -- )`
Wenn `flag` nicht null ist, dann mache normal nach `UNTIL` weiter.
Wenn `flag` gleich null ist, gehe zu `BEGIN` zurück.

`WHILE ( sys1 -- sys2 sys1 )` Compiler-Wort
Compilierend: Compiliere einen bedingten Sprung, hinterlasse eine Nachricht und vertausche die Reihenfolge der letzten beiden Nachrichten (`1 CS-ROLL`).
Ausführend: `( flag -- )`
Wenn `flag` nicht null ist, dann mache normal nach `WHILE` weiter.
Wenn `flag` gleich null ist, mache nach `REPEAT` weiter.

`REPEAT ( sys2 sys1 -- )` Compiler-Wort
Compilierend: Compiliere mit Hilfe von `sys1` einen Sprung nach `BEGIN`.
Füge mit Hilfe von `sys2` das Ziel des `WHILE`-Sprungs ein.
Ausführend: `( -- )`
Gehe zu `BEGIN` zurück.

`AGAIN ( sys1 -- )` Compiler-Wort
Compilierend: Compiliere mit Hilfe von `sys1` einen Sprung nach `BEGIN`.
Ausführend: `( -- )`
Gehe zu `BEGIN` zurück.

Genau wie in Kapitel 112 deutet das Wörtchen `sys1` auf ein Nachrichtensystem hin, das auch anders als über den Stack laufen kann.

Schleifen mit Zähler

`DO ?DO LEAVE LOOP +LOOP I J`

`DO ( -- sys1 )` Compiler-Wort
Compilierend: Compiliere einen Zählschleifenverwalter und hinterlasse eine Nachricht.
Ausführend: `( grenze zaehler -- )`
Halte die Schleifenparameter fest (meist auf dem Return-Stack).

`?DO ( -- sys1 )` "question do" Compiler-Wort
Compilierend: Compiliere einen Zählschleifenverwalter und hinterlasse eine Nachricht.
Ausführend: `( grenze zaehler -- )`
Wenn `zaehler` und `grenze` gleich sind, dann mache hinter `LOOP` weiter.
Anderenfalls: Halte die Schleifenparameter fest (wie bei `DO`).

`LOOP ( sys1 -- )` Compiler-Wort
Compilierend: Compiliere mit Hilfe von `sys1` einen Inkrementierer.
Ausführend: `( -- )`
Erhöhe `zaehler` um 1.
Gehe zum ersten Wort nach `DO` zurück, es sei denn, dass `zaehler` denselben Wert wie `grenze` bekommen hat. Verlasse im letzteren Fall die Schleife und räume die Schleifenverwaltung auf.

`+LOOP ( sys1 -- )` "plus loop" Compiler-Wort
Compilierend: Compiliere mit Hilfe von `sys1` einen Addierer.
Ausführend: `( x -- )`
Addiere `x` zum Zähler (Signed).
Gehe zum ersten Wort nach `DO` zurück, es sei denn, dass `zaehler` die Trennlinie zwischen `grenze` und `grenze-1` überschritten hat. Verlasse im letzteren Fall die Schleife und räume die Schleifenverwaltung auf.

`LEAVE ( -- )` Only Compiling
Mache ohne Rücksicht auf den Wert von `zaehler` und `grenze` hinter `LOOP` weiter und räume die Schleifenverwaltung auf.

`I ( -- x )` "i" Only Compiling
Setze den Zählerwert auf den Stack. Nur in einer Do-Loop zu verwenden.

`J ( -- x )` "j" Only Compiling
Setze den Zählerwert der äußeren Schleife auf den Stack. Nur innnerhalb von zwei Do-Loops in ein und derselben Definition zu verwenden.

Steuerung II

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

`AHEAD ( -- sys1 )` Compiler-Wort
Compilierend: Compiliere einen Vorwärtssprung und hinterlasse für `THEN` eine Nachricht.
Ausführend: `( -- )`
Mache nach `THEN` weiter. `AHEAD` ist ein Teilstück von `ELSE`.

`UNLOOP ( -- )` Only Compiling
Räume die Schleifenverwaltung auf.

`CASE ( -- sys1 )` Compiler-Wort
Compilierend: Hinterlasse für `ENDCASE` eine Nachricht.
Ausführend: `( -- )`
Keine Aktion.

`OF ( sys1 -- sys1 sys2 )` Compiler-Wort
Compilierend: Compiliere einen bedingten Sprung und hinterlasse für `ENDOF` eine Nachricht.
Ausführend: `( x y -- x )` oder `( x y -- )`
Wenn `x<>y` ist, dann lege `x` wieder auf den Stack und mache nach `ENDOF` weiter. Wenn `x=y` ist, dann mache normal nach `OF` weiter.

`ENDOF ( sys1 sys2 -- sys3 sys1 )` Compiler-Wort
Compilierend: Compiliere einen Vorwärtssprung, hinterlasse eine Nachricht für `ENDCASE` und füge mit Hilfe von `sys2` das Ziel des `OF`-Sprungs ein. Vertausche die Reihenfolge der letzten beiden Nachrichten (`1 CS-ROLL`).
Ausführend: `( -- )`
Mache nach `ENDCASE` weiter.

`ENDCASE ( sys3 .. sys1 -- )` Compiler-Wort
Compilierend: Füge alle angefallenen `ENDOF`-Sprünge ein.
Ausführend: `( x -- )`
Entferne `x` vom Stack.

`CS-PICK ( ? n -- ? )` "c s pick"
Führe auf die Reihe der unerledigten Nachrichten ein `PICK` aus.

`CS-ROLL ( ? n -- ? )` "c s roll"
Führe mit der Reihe der unerledigten Nachrichten ein `ROLL` aus.

Bits

`AND OR XOR INVERT LSHIFT RSHIFT`

`AND ( g1 g2 -- h )`
`h` entsteht dadurch, dass man auf `g1` und `g2` Bit für Bit ein logisches And ausübt.

`OR ( g1 g2 -- h )`
`h` entsteht dadurch, dass man auf `g1` und `g2` Bit für Bit ein logisches Or ausübt.

`XOR ( g1 g2 -- h )` "x or"
`h` entsteht dadurch, dass man auf `g1` und `g2` Bit für Bit ein logisches Xor ausübt.

`INVERT ( g -- h )`
Kehre alle Bits von `g` um, also
`g h AND` liefert False und
`g h OR` liefert True.

`LSHIFT ( g n -- h )` "l shift"
Alle Bits von `g` werden um `n` Stellen nach links verschoben. Auf die frei werdenden Stellen kommen Nullen. Bits, die herausfallen, gehen verloren.

`RSHIFT ( g n -- h )` "r shift"
Alle Bits von `g` werden um `n` Stellen nach rechts verschoben. Auf die frei werdenden Stellen kommen Nullen. Bits, die herausfallen, gehen verloren.

Flags I

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

`TRUE ( -- x )`
Alle Bits von `x` sind gesetzt, also `x=-1`.

`FALSE ( -- x )`
Alle Bits von `x` sind null, also `x=0`.

`0= ( x -- flag )` "zero equals"
`flag` ist nur dann True, wenn `x=0` ist.

`0< ( x -- flag )` "zero less"
`flag` ist nur dann True, wenn `x<0` ist.

`0> ( x -- flag )` "zero greater"
`flag` ist nur dann True, wenn `x>0` ist.

`0<> ( x -- flag )` "zero not equals"
`flag` ist nur dann True, wenn `x` ungleich `0` ist.

`= ( x y -- flag )` "equals"
`flag` ist nur dann True, wenn `x=y` ist.

`< ( x y -- flag )` "less than"
`flag` ist nur dann True, wenn `x<y` ist.

`> ( x y -- flag )` "greater than"
`flag` ist nur dann True, wenn `x>y` ist.

`<> ( x y -- flag )` "not equals"
`flag` ist nur dann True, wenn `x` ungleich `y` ist.

Flags II

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

`U< ( u v -- flag )` "u less than"
`flag` ist nur dann True, wenn `u<v` (Unsigned).

`U> ( u v -- flag )` "u greater than"
`flag` ist nur dann True, wenn `u>v` (Unsigned).

`D0= ( xlo xhi -- flag )` "d zero equals"
`flag` ist nur dann True, wenn `xhi,xlo` gleich null ist.

`D0< ( xlo xhi -- flag )` "d zero less"
`flag` ist nur dann True, wenn `xhi,xlo` kleiner als null ist.

`D= ( xlo xhi ylo yhi -- flag )` "d equals"
`flag` ist nur dann True, wenn `xhi,xlo` gleich `yhi,ylo` ist.

`D< ( xlo xhi ylo yhi -- flag )` "d less than"
`flag` ist nur dann True, wenn `xhi,xlo` kleiner als `yhi,ylo` ist.

`WITHIN ( x y1 y2 -- flag )`
`flag` ist nur dann True, wenn `x-y1` kleiner (Unsigned) als `y2-y1` ist.
Anders gesagt:
`flag` ist nur dann True, wenn `x` im Bereich `[y1..y2)` liegt, wobei `y1` noch zum Bereich gehört, `y2` jedoch nicht mehr.
Ein solcher Bereich `[y1..y2)` durchläuft die Zahlen (Signed oder Unsigned spielt keine Rolle) kreisförmig: Die zwei Bereiche G1 `[y1..y2)` und G2 `[y2..y1)` überlappen sich nicht und umfassen zusammen ALLE Zahlen.

Max und Min

`MAX MIN DMAX DMIN UMAX UMIN`

`MAX ( x1 x2 -- y )`
`y` ist gleich der größten der beiden Zahlen `x1` und `x2`.

`MIN ( x1 x2 -- y )`
`y` ist gleich der kleinsten der beiden Zahlen `x1` und `x2`.

`DMAX ( xlo xhi ylo yhi -- zlo zhi )` "d max"
Wie `MAX`, aber mit doppeltgenauen Zahlen.

`DMIN ( xlo xhi ylo yhi -- zlo zhi )` "d min"
Wie `MIN`, aber mit doppeltgenauen Zahlen.

Zuweilen findet man

`UMAX ( u1 u2 -- v )` "u max"
Wie `MAX`, aber mit Unsigned-Zahlen.

`UMIN ( u1 u2 -- v )` "u min"
Wie `MIN`, aber mit Unsigned-Zahlen.

Bearbeitung einer einzelnen Zahl

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

`1+ ( x -- y )` "one plus"
`y = x+1` Signed oder Unsigned; Overflow wird nicht vermeldet.

`1- ( x -- y )` "one minus"
`y = x-1` Signed oder Unsigned; Overflow wird nicht vermeldet.

`ABS ( x -- u )` "abs"
`u` (Unsigned) ist der Absolutwert von `x` (Signed).

`NEGATE ( x -- y )`
Kehr das Vorzeichen von `x` um, also `y = 0-x` .

`DABS ( xlo xhi -- ulo uhi )` "d abs"
`uhi,ulo` (Unsigned) ist der Absolutwert von `xhi,xlo` (Signed).

`DNEGATE ( xlo xhi -- ylo yhi )` "d negate"
Kehr das Vorzeichen von `xhi,xlo` um.

`2* ( x -- y )` "two star"
`y = 2*x` , d.h., schiebe das Bitmuster `x` eine Stelle nach links und belege die frei gewordene Stelle mit dem Bitwert null.

`2/ ( x -- y )` "two slash"
`y = x/2` , d.h., schiebe das Bitmuster `x` eine Stelle nach rechts und gib der frei gewordenen Stelle den Wert, den sie vor der Verschiebung bereits hatte.

`D2* ( xlo xhi -- ylo yhi )` "d two star"
Betrachte `xhi,xlo` als ein einziges langes Bitmuster, schiebe es insgesamt eine Stelle nach links und belege die frei gewordene Stelle mit dem Bitwert null.

`D2/ ( xlo xhi -- ylo yhi )` "d two slash"
Betrachte `xhi,xlo` als ein einziges langes Bitmuster, schiebe es insgesamt eine Stelle nach rechts und gib der frei gewordenen Stelle den Wert, den sie vor der Verschiebung bereits hatte.

Rechnen

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

`+ ( x y -- z )` "plus"
`z` ist gleich `x+y`. Overflow von `z` wird nicht vermeldet. `x y z` sind alle drei Signed oder alle drei Unsigned.

`- ( x y - z )` "minus"
`z` ist gleich `x-y`. `x y z` sind alle drei Signed oder alle drei Unsigned. Overflow von `z` wird nicht vermeldet.

`* ( x y -- p )` "star"
`p` ist gleich `y` mal `x`. Overflow von `p` wird nicht vermeldet.

`/MOD ( x y -- r q )` "slash mod"
`q` ist gleich `x` geteilt durch `y` (Floored oder Symmetrical). `r` ist der Rest, der dabei entsteht.

`/ ( x y -- q )` "slash"
Dasselbe wie `/MOD NIP`

`MOD ( x y -- r )`
Dasselbe wie `/MOD DROP`

`*/MOD ( x y1 y2 -- r z )` "star slash mod"
`z` ist gleich `x` mal `y1` geteilt durch `y2` (Floored oder Symmetrical). `r` ist gleich dem Rest, der dabei entsteht. Das Zwischenprodukt `x*y1` wird als doppeltgenaue Zahl berechnet. Overflow von `z` wird nicht vermeldet.

`*/ ( x y1 y2 -- z )` "star slash"
Dasselbe wie `*/MOD NIP`

Rechnen mit doppeltgenauen Zahlen

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

D+ (xlo xhi ylo yhi -- zlo zhi) "d plus"
Zähle zwei doppeltgenaue Zahlen zusammen, Signed oder Unsigned. Overflow von zhi,zlo wird nicht vermeldet.

D- (xlo xhi ylo yhi -- zlo zhi) "d minus"
Ziehe zwei doppeltgenaue Zahl voneinander ab, Signed oder Unsigned. Overflow von zhi,zlo wird nicht vermeldet.

M* (x y -- zlo zhi) "m star"
Nimm zwei einfachgenaue Zahlen miteinander mal und lege das Ergebnis als doppeltgenaue Zahl auf den Stack.

UM* (u v -- wlo whi) "u m star"
Nimm zwei einfachgenaue Unsigned-Zahlen miteinander mal und lege das Ergebnis als doppeltgenaue Zahl auf den Stack.

FM/MOD (xlo xhi y -- r q) "f m slash mod"
Teile xhi,xlo durch y (Floored Division). q ist der Quotient und r ist der Rest. Overflow von q wird nicht vermeldet.

SM/REM (xlo xhi y -- r q) "s m slash rem"
Teile xhi,xlo durch y (Symmetrical Division). q ist der Quotient und r ist der Rest. Overflow von q wird nicht vermeldet.

UM/MOD (ulo uhi v -- r q) "u m slash mod"
Teile uhi,ulo durch v (beide Unsigned). q ist der Quotient und r ist der Rest. Overflow von q wird nicht vermeldet.

Der Input-Strom

`TIB #TIB >IN WORD PARSE`

`TIB ( -- a )` "t i b"
`TIB` ist die Adresse des momentanen Eingabepuffers. Die Adresse bleibt nicht immer dieselbe.

`#TIB ( -- a )` "number t i b"
`#TIB @` ist die Länge des momentanen Eingabepuffers.

`>IN ( -- a )` "to in"
`>IN @` ist die relative Stelle im momentanen Eingabepuffer, an welcher das Lesen des Input-Stroms fortgesetzt wird.

`WORD ccc ( c -- a )`
Lies den Input-Strom. Übergehe erst alle Zeichen `c`. Lies daraufhin den Text bis einschließlich zum ersten danach auftretenden Zeichen `c` oder, falls es kein solches gibt, bis zum Ende des Puffers. Setze `>IN` auf die jetzt erreichte Stelle. Der Text (ohne einleitende oder abschließende Zeichen `c`) steht als counted string in Adresse `a`.
Die Aneinanderreihung `BL WORD` wird zum Herausfischen von Forth-Worten aus dem Input-Strom verwendet.

`PARSE ccc ( c -- a n )`
Lies den Input-Strom bis einschließlich zum ersten auftretenden Zeichen `c` oder, falls es kein solches gibt, bis zum Ende des Puffers. Setze `>IN` auf die jetzt erreichte Stelle. An Adresse `a` liegt der gelesene Text ohne das abschließende Zeichen `c`. `n` ist die Länge des Textes

Anwendungen von BL WORD

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

' ccc (-- s) "tick"
Lies ein Forth-Wort aus dem Input-Strom und ermittle seinen Schlüssel s.

['] ccc (--) "bracket tick" Compiler-Wort
Compilierend: Lies ein Forth-Wort aus dem Input-Strom, ermittle seinen Schlüssel und compiliere diesen in die Definition.
Ausführend: (-- s)
Lege den Schlüssel auf den Stack.

POSTPONE ccc (--) Compiler-Wort
Compilierend: Lies ein Forth-Wort aus dem Input-Strom und compiliere es so, dass es sich bei späterer Ausführung so verhält, als ob erst dann compiliert wird.
Ausführend: (-- ?)
Handle so, als ob erst jetzt compiliert wird.

[COMPILE] ccc (--) "bracket compile" Compiler-Wort
Compilierend: Lies ein Forth-Wort aus dem Input-Strom und erzwinge dessen Compilation, auch wenn es ein Immediate-Wort ist.
Ausführend: (--)
Führe ccc aus.

CHAR ccc (-- c)
Lies ein Wort aus dem Input-Strom und lege den ASCII-Code des ersten Zeichens auf den Stack.

[CHAR] ccc (--) "bracket char" Compiler-Wort
Compilierend: Lies ein Forth-Wort aus dem Input-Strom und compiliere den ASCII-Code des ersten Zeichens.
Ausführend: (-- c)
Lege den ASCII-Code auf den Stack.

TO ccc (x --) Immediate-Wort
Lies ein Forth-Wort aus dem Input-Strom und setze es, wenn es ein Value oder ein Local ist, auf den Wert x.

FORGET ccc (--)
Lies ein Forth-Wort aus dem Input-Strom und entferne dieses Wort einschließlich aller neueren Worte aus dem Wörterbuch.

SEE ccc (--)
Lies ein Forth-Wort aus dem Input-Strom und gib eine Analyse dieses Wortes aus.

Anwendungen von PARSE

```
( .( ." S" ABORT"
```

`( ccc ( -- )` "paren" Immediate-Wort
Lies den Input-Strom bis einschließlich zur ersten auftretenden Abschlussklammer und ignoriere den Text, auch während des Compilierens.

`.( ccc ( -- )` "dot paren" Immediate-Wort
Lies den Input-Strom bis einschließlich zur ersten auftretenden Abschlussklammer und gebe den Text ohne die Abschlussklammer aus, auch während des Complierens.

`." ccc ( -- )` "dot quote" Compiler-Wort
Compilierend: Lies den Input-Strom bis einschließlich zum ersten auftretenden Anführungszeichen und compiliere den Text ohne das abschließende Anführungszeichen.
Ausführend: `( -- )`
Gib den Text `ccc` aus.

`S" ccc ( -- ? )` "s quote"

a) Außerhalb von Definitionen `( ccc -- a n )`
Lies den Input-Strom bis einschließlich zum ersten auftretenden Anführungszeichen und platziere diesen Text ohne das Anführungszeichen in einen Puffer (Adresse `a`). Die Länge beträgt `n` Zeichen.

b) Innerhalb von Definitionen `( ccc -- )` Compiler-Wort
Compilierend: Lies den Input-Strom bis einschließlich zum ersten auftretenden Anführungszeichen und compiliere diesen Text ohne Anführungszeichen in die Definition.
Ausführend: `( -- a n )` Lege Adresse und Länge des Textes auf den Stack.

`ABORT" ccc ( -- )` "abort quote" Compiler-Wort
Compilierend: Lies den Input-Strom bis einschließlich zum ersten auftretenden Anführungszeichen und compiliere diesen Text ohne das abschließende Anführungszeichen.
Ausführend: `( flag -- )`
Bei `flag` gleich null keine Aktion.
Bei `flag` ungleich null: Brich das Programm ab und gib dabei den Text `ccc` als Meldung aus.

Definierende Worte

`CREATE : ; CONSTANT VARIABLE VALUE`

`CREATE ccc ( -- )`
Lies einen Namen aus dem Input-Strom und definiere ein Wort mit diesem Namen, dessen Body noch leer ist.
`ccc ( -- a )`
Lege die Body-Adresse von `ccc` auf den Stack.

`: ccc ( -- sys1 )` "colon"
Lies einen Namen aus dem Input-Strom und fange mit einer High-Level-Definition dieses Namens an. Hinterlasse eine Nachricht für ; (Semicolon). Das neue Wort kann erst dann im Wörterbuch gefunden werden, wenn es von ; (Semicolon) fertig gestellt wurde.

`; ( sys1 -- )` "semicolon" Compiler-Wort
Compilierend: Compiliere ein Wort zum Verlassen der Definition, verarbeite die Nachricht von : (Colon), beende die Definition und höre auf zu compilieren.
Ausführend: `( -- )` Verlasse die Definition.

`CONSTANT ccc ( x -- )`
Lies einen Namen aus dem Input-Strom und definiere eine Konstante mit diesem Namen.
`ccc ( -- x )`
Lege `x` auf den Stack.

`VARIABLE ccc ( -- )`
Lies einen Namen aus dem Input-Strom und definiere eine Variable mit diesem Namen.
`ccc ( -- a )`
Auf Adresse `a` ist Platz für den Wert der Variablen `ccc`.

`VALUE ccc ( x -- )`
Lies einen Namen aus dem Input-Strom und definiere einen Value mit dem Namen `ccc` und dem Wert `x`.
`ccc ( -- x )`
Lege den Wert des Values auf den Stack.
`TO ccc ( x -- )`
Lege den Wert `x` in den Value `ccc`.

Compilieren

`[ ] COMPILE, LITERAL 2LITERAL RECURSE EXIT`

`[ ( -- )` "left bracket" Immediate-Wort
Verlasse den Compilier-Modus. `STATE` wird dabei verändert.

`] ( -- )` "right bracket"
Gehe in den Compilier-Modus. `STATE` wird dabei verändert.

`COMPILE, ( s -- )` "compile comma"
Compiliere das Wort, das `s` zum Schlüssel hat, in die Definition.

`LITERAL ( x -- )` Compiler-Wort
Compilierend: Compiliere die Zahl `x`.
Ausführend: `( -- x )`
Lege `x` auf den Stack.

`2LITERAL ( xlo xhi -- )` Compiler-Wort
Compilierend: Compiliere die doppeltgenaue Zahl `xhi,xlo`.
Ausführend: `( -- xlo xhi )`
Lege die doppeltgenaue Zahl auf den Stack.

`RECURSE ( -- )` Compiler-Wort
Während des Compilierens wird anstelle des Namens des im Entstehen begriffenen Wortes `RECURSE` verwendet. Auf die gewohnte Art kann der Name nicht gefunden werden, da die Definition noch nicht fertig ist.

`EXIT ( -- )` Only Compiling
Verlasse die Definition.

Worte definieren, die Worte definieren

`DOES> ;CODE`

`DOES> ( sys1 -- sys2 )` "does" Compiler-Wort
Compilierend: Das ist das Ende der Definition und gleichzeitig der Anfang der Definition einer namenlosen Aktion.
Ausführend: `( -- bodyadresse )`
Verändere vor dem Verlassen der Definition ein gerade mit `CREATE` definiertes Wort so, dass die namenlose Aktion an den Code, der von `CREATE` dem Wort schon mitgegeben wurde (und das war: Body-Adresse auf den Stack legen), angefügt wird.

`;CODE ( sys1 -- sys2 )` "semicolon code" Compiler-Wort
Compilierend: Das ist das Ende der Definition und gleichzeitig der Anfang von Assembler-Code für eine namenlose Aktion.
Ausführend: `( -- )`
Verändere vor dem Verlassen der Definition ein gerade mit `CREATE` definiertes Wort so, dass es anstelle der von `CREATE` dem Wort übergebenen Aktion die namenlose Aktion als Assembler-Code mitbekommt.

Der Wortschlüssel

```
' ['] EXECUTE FIND >BODY BODY>
```

`' ccc ( -- s )` "tick"
Lies einen Wortnamen aus dem Input-Strom und lege den Schlüssel dieses Wortes auf den Stack.

`['] ccc ( -- )` "bracket tick" Compiler-Wort
Compilierend: `ccc ( -- )` Führe ein `'` (Tick) aus und compiliere den gefundenen Schlüssel als Zahl in die Definition.
Ausführend: `( -- s )`
Lege den Schlüssel auf den Stack.

`EXECUTE ( s -- )`
Führe das Wort, dessen Schlüssel auf dem Stack liegt, aus.

`FIND ( a -- a 0 )` oder `( a -- s -1 )` oder`( a -- s 1 )`
Suche den Schlüssel `s` des Forth-Wortes, dessen Name als counted string an der Adresse `a` liegt. Drei Möglichkeiten:
`( a -- a 0 )` Das Wort ist nicht zu finden.
`( a -- s -1 )` Das gefundene Wort ist nicht Immediate.
`( a -- s 1 )` Das gefundene Wort ist Immediate.

`>BODY ( s -- a )` "to body"
`a` ist die Body-Adresse des Wortes mit Schlüssel `s` (nur für Worte, die mit `CREATE` erzeugt wurden).

Zuweilen findet man

`BODY> ( body-adresse -- s )` "body from"

Wortlisten

FORTH DEFINITIONS ORDER WORDS PREVIOUS ALSO ONLY VOCABULARY

FORTH (--)
Ersetze die erste Wortliste der Suchordnung durch FORTH. FORTH ist ein VOCABULARY. Siehe weiter unten auf dieser Seite.

DEFINITIONS (--)
Die erste Wortliste in der Suchordnung ist nun die Liste, in welche ab jetzt die neuen Definitionen aufgenommen werden.

ORDER (--)
Zeige die Suchordnung (Vocabulary-Stack) an.

WORDS (--)
Zeige die Worte der ersten Wortliste der Suchordnung an.

PREVIOUS (--)
Entferne die erste Wortliste aus der Suchordnung.

ALSO (--)
Dupliziere die erste Wortliste in der Suchordnung.

ONLY (--)
Schalte die Grundsuchordnung ein.

Zuweilen findet man auch

VOCABULARY ccc (--)
Definiere eine neue Wortliste
ccc (--)
Ersetze die erste Wortliste in der Suchordnung durch ccc.

Verschiedenes I

`IMMEDIATE EVALUATE QUIT QUERY UNUSED`

`IMMEDIATE ( -- )`
Mache das zuletzt definierte Wort zu einem Immediate-Wort, d.h. das jüngste Wort derjenigen Wortliste, in welcher die neuen Worte landen.

`EVALUATE ( a n -- )`
Unterbrich den laufenden Input-Strom und betrachte den Text `a,n` als den neuen Input-Strom. Wenn dieser abgearbeitet ist, gehe wieder zum alten Input-Strom, an die Stelle, wo derselbe unterbrochen wurde, zurück (geschachtelter Input-Strom).

`QUIT ( -- )`
Leere den Return-Stack, gehe in den Nicht-Compilier-Zustand und beginne den folgenden Kreislauf:

1. warte auf Tastatur-Eingaben,
2. verarbeite die Eingaben,
3. gib "`ok`" aus und gehe nach 1.

Der Kreislauf wird lediglich bei einem Fehler in 2. unterbrochen.

`QUERY ( -- )`
Empfange Tastatur-Eingaben und mache den aufgenommenen Text zum Input-Strom.

`UNUSED ( -- u )`
`u` ist die Zahl der Bytes, die ab `HERE` frei sind.

Verschiedenes II

`BL S>D D>S MS BLANK ERASE PAD`

`BL ( -- 32 )` "b l"
`32` ist der ASCII-Code für einen Zwischenraum.

`S>D ( x -- xlo xhi )` "s to d"
Ersetze `x` durch eine doppeltgenaue Zahl vom selben Wert.

`D>S ( xlo xhi -- x )` "d to s"
Entferne die vordere Zelle der doppeltgenauen Zahl `xhi,xlo` vom Stack.

`MS ( n -- )` "m s"
Warte `n` Millisekunden.

`BLANK ( a n -- )`
Fülle den Speicher ab Adresse `a` mit `n` Zwischenräumen.

`ERASE ( a n -- )`
Fülle `n` Zeichenzellen ab Adresse `a` mit dem ASCII-Code 0.

`PAD ( -- a )`
`PAD` ist die Adresse eines Schmierblocks für den Programmierer. `PAD` liegt meistens in einem festen Abstand oberhalb von `HERE`, was zur Folge hat, dass die dort abgespeicherten Daten nur beschränkt haltbar sind. Die Länge des Schmierblocks hängt von der Implementation ab.

Fehler behandeln

`ABORT ABORT" CATCH THROW`

`ABORT ( ? -- )`
Verlasse das Programm. Lösche Stack und Return-Stack und führe `QUIT` aus. Gib dabei nichts aus.

`ABORT" ccc ( -- )` "abort quote" Compiler-Wort
Compilierend: Lies den Input-Strom bis einschließlich zum ersten ankommenden Anführungszeichen und compiliere den Text ohne das abschließende Anführungszeichen.
Ausführend: `( flag -- )`
Bei `flag` gleich null keine Aktion.
Bei `flag` ungleich null: Brich das Programm ab und gib dabei den Text `ccc` als Nachricht aus.

`CATCH ( s -- ?? flag )`
Siehe Kapitel 208 und 209.

`THROW ( flag -- ?? flag )`
Siehe Kapitel 208 und 209.

C

200 - 211

Lose Artikel

Die in diesem Teil des Buches wiedergegebenen Artikel erschienen in zwangloser Folge im "Vijgeblaadje", der Vereinszeitschrift der "Forth-gebruikersgroep" (holländische Forth-Gesellschaft).

201. Signed- und Unsigned-Zahlen

202. Zahlen ausgeben I

    Über die Forth-Worte, die Zahlen ausgeben.

203. Zahlen ausgeben II

    Über `PICTURE` zur Ausgabe von Zahlen in einem bestimmten Format und über `HEXA BINA DECI` zur gelegentlichen Ausgabe einer Zahl in einem anderen Zahlensystem.

204. Kontrollstrukturen I

    Über das Compile-Time- und Run-Time-Verhalten von `IF THEN BEGIN` usw.

205. Kontrollstrukturen II

206. Ohne Ausnahme

    Beispiel eines Programmierstils, der bedingte Sprünge vermeidet.

207. Roboterarm

    Dieses Programm koordiniert Prozesse, die in untereinander verschiedenen Zeitverhältnissen verlaufen: Sie beginnen gleichzeitig, laufen gleichmäßig und sind gleichzeitig fertig.

208. CATCH und THROW I

209. CATCH und THROW II

210. MANY TIMES

    Über das Manipulieren des Input-Stroms.

211. HX DM BN

    `BASE` für die Dauer eines einzelnen Wortes ändern.

Signed- und Unsigned-Zahlen

Leo: Meiner Meinung nach steckt in deinem Forth ein Fehler. Sieh doch mal, was da passiert:

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

Bis jetzt noch nichts. Aber nun:

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

Das ist überhaupt nicht o.k. Absoluter Unsinn! Ich weiß genau, was bei `MAX ( x y -- z )` herauskommen muss. Das Ergebnis ist die größere der beiden Zahlen `x` und `y`, und das stimmt hier ganz und gar nicht. Ist dir das noch nie aufgefallen?

Theo: Tja, ich muss sagen, das sieht wirklich seltsam aus ... aber richtig ist es trotzdem.

Leo: Ich bin selbst so fleißig gewesen und habe versucht, `MAX` neu und besser zu definieren:

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

Das Verrückte an dieser Definition ist aber, dass sie denselben Fehler macht.

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

Theo: Der Fehler hängt mit dem KLEINER-ALS zusammen.

Leo: Wenn du das schon weißt, warum hast du es dann nicht verbessert? Du bist doch stets bemüht, dein Forth schneller und besser zu machen!

Theo: Das KLEINER-ALS ist kein Fehler, du hast es einfach falsch verwendet.

Leo: ??

Theo: Was sagst du hierzu:

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

Leo: Du erwartest wahrscheinlich, dass mir die Sache jetzt klarer wird?

Theo: Und hierzu:

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

Leo: Wenn man zwei verschiedene Zahlen voneinander abzieht, würde ich sagen, kann das nicht 0 ergeben. Muss ich zum Erlernen von Forth auch noch mein Allgemeinwissen über das Rechnen mit Zahlen revidieren? Das wäre zuviel verlangt.

Theo: Da steckt Folgendes dahinter:
Dieses Forth arbeitet mit Zahlen von -32768 bis einschließlich 32767. Das kommt daher, weil es Zahlen in ein Muster von 16 Bits packt. Zahlen, die größer oder kleiner sind, passen nicht mehr in diese 16 Bits, denn mit 16 Bits kannst du höchstens 65536 (= 2 hoch 16) verschiedene Bitmuster erzeugen.

Leo: Aber warum schluckt dann dieses Forth die 40000 trotzdem?

Theo: Da hast du Recht, in einer anderen Programmiersprache würde hier wahrscheinlich eine Fehlermeldung kommen. Es hat aber wiederum Sinn, dass das Forth-System die 40000 gutheißt. Ich werde dir erklären, warum.

Du kannst den gesamten Zahlenbereich von -32768 über 0 nach 32767 durchlaufen, indem du fortwährend 1 hinzuaddierst. Am Rande des Bereichs geschieht Folgendes:

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

Wenn du die größte Zahl um 1 erhöhst, kommt als Ergebnis die kleinste Zahl zum Vorschein! Es tritt ein Überlauf auf, d.h., es entsteht ein Bitmuster von 17 Bits, doch Forth lässt die 17. Stelle normalerweise weg, wodurch die Bitmusterreihe wieder von vorn beginnt und die Zahlengesamtheit tatsächlich auf einem Kreis angeordnet erscheint. Die Folge: Wenn man zwei Zahlen miteinander addiert, erhält man auf jeden Fall ein Ergebnis. Der Programmierer muss selbst im Auge behalten, ob das erhaltene Ergebnis richtig ist, anders gesagt, ob es ohne Überlauf zustande gekommen ist.
Neben diesem System (den SIGNED-Zahlen) gibt es noch ein zweites System (die UNSIGNED-Zahlen), das mit genau denselben Bitmustern arbeitet. Die positiven Zahlen bleiben dabei dasselbe, aber alle negativen Zahlen werden über Bord geworfen und am oberen Ende als höhere positive Zahlen wieder angefügt. Die UNSIGNED-Zahlen laufen von 0 nach 65535. Diese höheren positiven Zahlen müssen auch eingegeben werden können und darum wurde 40000 also akzeptiert. Forth merkt sich lediglich die Zahl als Bitmuster, und nicht die Art und Weise, wie die Zahl eingegeben wurde.
An den Bereichsrändern tritt wieder der Überlauf-Effekt auf.

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

Aber nun zurück:

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

Das geht schief, denn der PUNKT weiß nicht, dass es hier um eine UNSIGNED-Zahl geht.

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

+ und - arbeiten in beiden Systemen gut. Für Ausgabezwecke, aber auch für Vergleiche, benötigt man Extra-Befehle: . U. < U< > U>
Und die Lösung deines MAX-Problems liegt also in der Definition eines UMAX, das U< enthält.

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

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

Leo: Geht das in allen Forth-Systemen so?

Theo: Ja, nur sind die Bereiche manchmal anders. In einem 32-bit-Forth läuft SIGNED von -2147483648 bis 2147483647 und UNSIGNED von 0 bis 4294967295 (ungefähr vier Milliarden).
Vor drei Wochen sprach ich mit jemandem, der mit einem Forth arbeitet, in welchem 100 größer als 200 ist und 127 plus 1 den Wert -128 ergibt. Was für ein Forth wird das wohl gewesen sein?

Über die Forth-Worte, die Zahlen ausgeben.

Zahlen ausgeben I

1. Die acht Punkt-Befehle

ANS-Forth, der neue amerikanische Forth-Standard, kennt die folgenden sechs Worte zur Ausgabe von Zahlen:

```
.     ( 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
```

Die ANS-Norm ordnet die Worte in eine Anzahl von Kapiteln ein.
`Core` deutet den Forth-Kern an.
`Core Extensions` enthält Worte, die als direkte Erweiterung des Kerns gesehen werden können.
Die Worte in `Double` beziehen sich auf doppeltgenaue Zahlen.
Obligatorisch sind nur die Worte des Kerns `Core`, die anderen sind optional.

Der Buchstabe `U` in den Punkt-Befehlen steht für Unsigned, d.h., alle Bits des Bitmusters werden zur Zusammenstellung einer positiven Zahl verwendet. Bei der größten Unsigned-Zahl sind alle Bits gesetzt.
Die Befehle ohne U arbeiten mit Signed-Zahlen. Das höchste Bit gibt dabei an, ob es um eine positive oder negative Zahl geht.
Ein Bitmuster kann also auf zwei Arten in einen Zahlenstring übersetzt werden.

```
) -1 . [rtn]
) -1 U. [rtn] \ groesste Unsigned-Zahl
```

Die Befehle, die doppeltgenaue Zahlen behandeln, beginnen mit dem Buchstaben `D`. Zwei Zahlen auf dem Stack bilden zusammen eine doppeltgenaue Zahl. Der Bereich wird dadurch viel größer. Um die größte doppeltgenaue Unsigned-Zahl auszugeben, wird ein Befehl `DU.` benötigt, der in ANS-Forth fehlt.

```
) -1. D. [rtn]
) -1. DU. [rtn] \ groesste doppeltgenaue Unsigned-Zahl
```

Der Buchstabe `R` in einem Befehl gibt an, dass die Zahl 'rechtsbündig in einem Feld von `r` Stellen' ausgegeben wird.

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

Auch `DU.R` wird im ANS-Standard nicht genannt.

2. Definitionen für die acht Punkt-Befehle

Die Ausgabe von Zahlen geschieht in zwei Phasen:

A. Die Umwandlung (Signed oder Unsigned) des Bitmusters auf dem Stack in einen String;
B. Die Ausgabe des Strings, 'rechtsbündig' oder nicht.

Für Phase A bilde ich zwei Worte:

```
) : DU.STRING (ulo uhi -- adr len )
)   <# #S #> ;
)
) : D.STRING ( xlo xhi -- adr len )
)   TUCK     \ Vorzeichen merken
)   DABS
)   <# #S
)   ROT SIGN \ Eventuell ein Minuszeichen
)            \ vor den String setzen
)   #> ;
```

`<# ( ulo uhi -- ulo uhi )` "less number sign"
Reserviere einen Puffer für die Umwandlung des Bitmusters `uhi,ulo` in einen lesbaren String. Während der Umwandlung können `#S` und `SIGN` ihre Arbeit tun, und `#>` schließt die Umwandlung ab.

`#S ( ulo uhi -- 0 0 )` "number sign S"
Wandle die doppeltgenaue Zahl auf dem Stack in einen String um. (Nur zwischen `<#` und `#>` verwendbar.)

`#> ( ulo uhi -- adr len )` "number sign greater"
Schließe die Umwandlung ab. Der String `adr,len` ist das Ergebnis der Umwandlung.

`SIGN ( x -- )`
Füge, nur falls `x` negativ ist, vorn an den im Entstehen begriffenen String ein Minuszeichen an. (Nur zwischen `<#` und `#>` verwendbar.)

In Phase B benötige ich neben `TYPE` noch ein Wort, das einen String rechtsbündig ausgibt, beispielsweise um Spalten zu erzeugen.

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

Nun ist es einfach, die acht Punkt-Befehle zu definieren. Erst die doppeltgenauen Zahlen:

```
) : 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 ;
```

Danach die einfachgenauen Zahlen:

```
) : .    ( 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 ;
```

[wird fortgesetzt]

Über PICTURE zur Ausgabe von Zahlen in einem bestimmten Format und über HEXA BINA DECI zur gelegentlichen Ausgabe einer Zahl in einem anderen Zahlensystem.

Zahlen ausgeben II

3. PICTURE oder das Ausgeben über Schablonen

Mitunter wollen Sie Zahlen in einem Format ausgeben, das die Punkt-Befehle nicht beherrschen. Das können Sie dann jederzeit mit den Worten <# # SIGN HOLD #S #> selbst programmieren. Es wäre aber nicht ungeschickt, ein flexibles Wort an der Hand zu haben, das genau das fertig bringt. Ich schlage vor, ein solches Wort PICTURE zu nennen.

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

Es erwartet eine doppeltgenaue Zahl auf dem Stack und darüber hinaus die Adresse und die Länge des Strings, der als Schablone oder Vorgabefeld angibt, wie die doppeltgenaue Zahl in einen String umgesetzt werden soll.

Ein paar Beispiele. In der ersten Spalte stehen die Schablonenstrings, in Spalte zwei und drei deren Auswirkung bei Anwendung auf 1234.5678 und 9.5 als doppeltgenaue Zahlen.

```
Schablonenstring  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" EUR ?.##"      EUR 123456.78    EUR 0.95
```

Besondere Bedeutung in dem Schablonenfeld haben der Gartenzaun und das Fragezeichen:
Der Gartenzaun # setzt jedes Mal genau eine Ziffer in den String.
Das Fragezeichen setzt die verbleibenden Ziffern oder eine Null in den String. (Denken Sie daran, dass der String von hinten nach vorn aufgebaut wird!)
Andere Zeichen des Schablonenfeldes werden buchstabengetreu übernommen.
Bemerkung: Die umzuwandelnde Zahl wird als Unsigned aufgefasst.

4. Die Definition von PICTURE

```
) : PICTURE ( ulo uhi adr1 len1 -- adr2 len2 )
)   2>R
)   <#
)   BEGIN R> 1-            \ Stelle in der Schablone
)      S>D 0=
)   WHILE >R 2R@ + C@      \ Zeichen aus der Schablone
)      [CHAR] # OVER =
)      IF DROP #           \ Eine Ziffer
)      ELSE [CHAR] ? OVER =
)         IF DROP #S       \ Die restlichen Ziffern
)         ELSE HOLD
)         THEN
)      THEN
)   REPEAT R> 2DROP
)   #> ;
```

`# ( ulo1 uhi1 -- ulo2 uhi2 )` "number sign"
Bestimme unter Verwendung von `BASE` die letzte Ziffer des Bitmusters `uhi1,ulo1` und füge diese Ziffer vorn an den im Aufbau befindlichen String an. `uhi2,ulo2` ist `uhi1,ulo1` ohne dessen letzte Ziffer. (Nur zwischen `<#` und `#>` verwenden.)

`HOLD ( c -- )`
Füge vorn an den im Aufbau befindlichen String das Zeichen `c` an. (Nur zwischen `<#` und `#>` verwenden.)

Beispiele:

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

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

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

`PICTURE` liefert einen String, der nur von kurzem Bestand ist. Bei den meisten Forth-Systemen wird er überschrieben werden, sobald eine neue Umwandlung beginnt. Probieren Sie

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

5. Nur mal so 'ne Idee

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

) : HEXA ['] HEX      (MAL-EBEN ; IMMEDIATE
) : DECI ['] DECIMAL  (MAL-EBEN ; IMMEDIATE
) : BINA ['] BINARY   (MAL-EBEN ; IMMEDIATE
```

Definieren Sie zuvor `BINARY` (wenn es noch nicht vorhanden ist):

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

Verwenden Sie `HEXA DECI BINA` interaktiv, unmittelbar vor einem Wort, das Zahlen ausgibt:

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

Aber auch

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

Über das Compile-Time- und Run-Time-Verhalten von `IF THEN BEGIN` usw.

Kontrollstrukturen I

1. Wie es einfach geht

Erst die Daten, dann der Ausführungsbefehl. Diese Grundregel von Forth findet auch beim Treffen von Entscheidungen in Programmen mit Hilfe von bedingten Sprüngen Anwendung. Ein Flag auf dem Stack ist das 'Datenfutter' für die 'Operatoren' `IF` oder `UNTIL`, die daraufhin entscheiden, ob gesprungen werden soll oder nicht. In vielen Programmiersprachen haben wir die Konstruktion:

```
IF        \ Bestimme den Flag-Wert von 'KALT'.
 KALT     \ Formulierung des Flags
THEN      \ Verlasse die Konstruktion, wenn flag=false
 SCHAL-UM \ Bedingt auszufuehrender Code
ENDIF     \ Endpunkt; koennte auch ENDTHEN heissen
```

In Forth gehört die Bestimmung des Flag-Wertes nicht zur Grammatik der Konstruktion. Es gibt lediglich den Operator `IF` mit seinem Abschlusselement `THEN`. Der Programmierer muss dafür sorgen, dass ein Flag bereitsteht:

```
( KALT?   \ Flag auf Stack )
IF        \ Spring bei 'false' vorwaerts nach THEN
 SCHAL-UM \ Bedingt auszufuehrender Code
THEN      \ Endpunkt; koennte auch ENDIF heissen
```

Der Rückwärtssprung mit `UNTIL` geht in Forth analog:

```
BEGIN      \ Anfangspunkt
 ..  WARM? \ Der Programmierer sorgt dafuer, dass
           \   dieser Code ein Flag produziert
UNTIL      \ Spring bei 'false' zurueck nach BEGIN
( SCHAL-AB )
```

Sie können diese Strukturen ineinander schachteln. Wenn Sie das sauber tun, wie in `.. BEGIN .. IF .. THEN .. UNTIL ..` , heißt das strukturierte Programmierung.
In `.. BEGIN .. IF .. UNTIL .. THEN ..` wird Forth protestieren, wenn es auf `UNTIL` trifft, denn das `IF` ist noch nicht abgeschlossen.
Und in `.. IF .. IF .. THEN .. THEN ..` gehört das erste `THEN` darum zum zweiten `IF`.

2. Der Compiler

Wenn Sie eine Definition konstruiert haben, die

```
aaa IF bbb THEN ccc
```

enthält, und wenn Sie das dann decompilieren, dann werden Sie etwas in der Art von

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

zu sehen bekommen. `IF` ist zu `JOF` (jump on false) geworden, auf was dann eine (relative) Adresse folgt. `THEN` ist verschwunden. Logisch, denn die Adresse verweist, sehr wahrscheinlich in Form eines Offsets, schon auf `ccc`. Diese Adresse ist in dem Moment, in dem `IF` erreicht wird, noch nicht bekannt. Wie der Compiler das schafft? Eigentlich macht der Compiler das gar nicht. Das machen `IF` und `THEN` selbst.

```
: IF ( -- ) ?COMPILING \ Erklaerung folgt
  POSTPONE JOF
  HERE IFMARKE COMPOST!
  0 , ; IMMEDIATE

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

`?COMPILING` verursacht eine Fehlermeldung, wenn Sie nicht beim Compilieren sind.
`IFMARKE` ist eine Konstante, die der Wiedererkennung dient.
`COMPOST! ( adr const -- )` schickt Post (ein Päckchen mit zwei Zahlen) an den Compiler.
`COMPOST@ ( -- adr const )` verlangt das zuletzt geschickte Päckchen wieder zurück.
`?PAIRS ( x y -- )` gibt eine Fehlermeldung aus, wenn `x<>y` ist.
`OFFSET! ( adr1 adr2 -- )` bewahrt `adr1` als Offset in `adr2` auf.

Alle Nichtstandardnamen in dieser Geschichte habe ich mir selbst ausgedacht. Ich verwende sie nur, um deutlich zu machen, was da passiert. In jedem Forth werden diese Namen und auch die Details anders sein, der generelle Gang der Handlung wird aber genau so sein, wie oben beschrieben.

`BEGIN` und `UNTIL` sollten nun kein Problem mehr darstellen.

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

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

Merken Sie sich vor allem, dass `IF` und `BEGIN` Nachrichten für `THEN` und `UNTIL` hinterlassen.

3. Unbedingte Sprünge

`IF .. THEN` und `BEGIN .. UNTIL` haben jedes für sich ein Gegenstück, `AHEAD .. THEN` und `BEGIN .. AGAIN`, mit unbedingten Sprüngen. Die Gegenstücke verwenden kein Flag, überprüfen nichts und scheinen auf den ersten Blick sinnlos zu sein:

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

`AHEAD` springt immer vorwärts auf `THEN` zu, und `AGAIN` springt immer zurück nach `BEGIN`. Warum dann der Code `??` nach `AHEAD` und `AGAIN`, der doch nie erreicht wird?

Diese Strukturen haben im Allgemeinen nur dann einen Sinn, wenn sie unstrukturiert mit anderen Kontrollstrukturen zusammen verwendet werden.
`.. IF .. ELSE .. THEN ..` können Sie beispielsweise ansehen als
`.. IF .. AHEAD THEN .. THEN` , wovon das erste `THEN` zu `IF` gehört und das zweite `THEN` zu `AHEAD`. Falsch geschachtelt also.
Sie können das mit einem Kunstgriff zum Funktionieren bringen, wenn Sie nämlich den Compiler auf die eine oder andere Art dazu veranlassen, genau zwischen `AHEAD` und `THEN` die letzten beiden Nachrichtenpäckchen ihre Plätze tauschen zu lassen.

Dafür gibt es ein (Standard-)Wort: `1 CS-ROLL`

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

Denselben Trick können Sie auch mit `AGAIN` aus dem Hut zaubern:
`.. BEGIN .. IF .. AGAIN THEN ..` wird dann
`.. BEGIN .. IF .. AGAIN? ..` mit

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

Aber dafür hat man eine andere Lösung erdacht. Mehr darüber das nächste Mal.

[wird fortgesetzt]

Kontrollstrukturen II

4. WHILE

`WHILE` ist die Lösung, auf die ich das letzte Mal abzielte. Es ist ein `IF`, das seine Post an den Compiler nicht ganz nach oben, sondern direkt unter die oberste Post legt. Sie können `WHILE` daher nur **innerhalb** einer anderen Kontrollstruktur verwenden. Die andere Struktur muss wieder abgeschlossen sein, bevor das zugehörige `THEN` kommt. `WHILE` schachtelt sich über Kreuz und dient dazu, **aus** einer Struktur nach `THEN` zu springen:

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

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

Die Ziffern geben an, wie die Worte zueinander gehören. `WHILE` ist innerhalb aller anderen Kontrollstrukturen verwendbar. Beispiele:

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

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

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

In c) legt `WHILE-2` seine Post unter `BEGIN-1` ab, `WHILE-3` legt seine Post auch unter `BEGIN-1` ab, also oberhalb von `WHILE-2`. Die beiden WHILEs sind untereinander wieder normal geschachtelt.

Fangfrage:
Was ist der Unterschied zwischen `OEL` und `ESSIG`?

```
: OEL   aaa IF bbb IF ccc THEN THEN ddd ;
: ESSIG aaa IF bbb WHILE ccc THEN THEN ddd ;
```

Bemerkung:
`REPEAT` (ein Standardwort) steht für `AGAIN THEN`.

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

5. DO-LOOP, CASE

Das Paar DO-LOOP arbeitet zur Run-Time wie ein BEGIN-AGAIN mit den folgenden Zusätzen:

1. `DO ( grenze zaehler -- )` baut einen Zählmechanismus auf.
2. Innerhalb der Schleife ist der Zähler über `I` abfragbar.
3. `LOOP` passt den Zähler nach jedem Durchlauf an.
4. `LOOP` beendet die Schleife, wenn der Zähler den vorgegebenen Bereich abgearbeitet hat. Für die ganz Genauen unter Ihnen: Der vorgegebene Bereich ist abgearbeitet, wenn der Zähler die "Grenzlinie" zwischen 'grenze-minus-1' und 'grenze' überschreitet. Diese Überlegung gilt auch für sowohl positive als auch negative Schritte bei `+LOOP`.

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

Man verwendet `LEAVE` zum vorzeitigen Abbruch einer DO-LOOP:

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

`LEAVE` räumt die Zählverwaltung auf und leitet einen Sprung nach `aaa`, unmittelbar hinter `LOOP`, ein.

Auch zur Compile-Time gleicht DO-LOOP dem BEGIN-AGAIN. Sie können daher mit `WHILE` einen Sprung aus einer DO-LOOP compilieren. Das kann günstig sein, wenn die bedingt abgebrochene Schleife ganz anders fortgesetzt werden soll als die vollständig abgearbeitete Schleife:

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

`aaa` wird ausgeführt, wenn `LOOP` die Schleife beendet hat. Das Stückchen zwischen `ELSE` und `THEN` kommt einzig und allein nach einer bedingten Unterbrechung zum Tragen. Meistens lässt sich das mit `LEAVE` nicht elegant schreiben. Zwei Dinge fallen auf:

1. Es ist **immer explizit ein** `UNLOOP` **nötig,** um im Run-Time-Modus die Zählverwaltung aufzuräumen. `LEAVE` (immer) und `LOOP` (nach dem letzten Durchlauf) führen implizit ein `UNLOOP` aus.
2. `I` ist **außerhalb der Schleife verwendbar, solange die Zählverwaltung noch intakt ist**.

Zum Schluss das CASE-Konstrukt in Forth, ein viel besprochener Gegenstand. Ist er derartige Aufmerksamkeit wert? Ich denke, dass CASE völlig überflüssig ist. Forth hat dafür genügend bessere Formulierungen. Könnte es sein, dass der Bedarf an CASE vor allem von jenen Forth-Anfängern gefühlt wird, die von anderen Sprachen her kommen und die die begreifliche Neigung haben, aus ihrer vertrauten Umgebung heraus eins-zu-eins zu übersetzen (m.E. die Forth-Hemmschwelle schlechthin)?

Beispiel eines Programmierstils, der bedingte Sprünge vermeidet.

Ohne Ausnahme

Wenn man `x` mit `y` malnimmt, lautet das Ergebnis `y*x`, es sei denn, dass dabei eine `0` oder eine `1` beteiligt ist:

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

Völlig richtig, es geht aber auch einfacher:

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

Hiermit will ich demonstrieren, dass nicht immer, wenn etwas eine Ausnahme zu sein scheint, eine Sonderbehandlung dafür nötig ist. Aber so einfach wie mit `MAL` geht das natürlich nicht immer.
Im unten stehenden Programm habe ich so viele Sonderfälle wie möglich dadurch beseitigt, `( **)`, dass ich sie zu Normalfällen zurechtgebogen habe. Es treten keine bedingten Sprünge mehr auf, und der Weg, den das Programm durchläuft, kann dadurch leichter verfolgt werden.
Das einzige Zugeständnis ist `UNTIL` am Ende von `SPIEL`.

```
\ ----- Schiebespiel -----
\ Der Spieler schiebt die Steinchen
\ in die "richtige" Reihenfolge.

MARKER -SPIEL

: CHAR-ARRAY ( anzahl  -- ) \ Definierendes Wort
  CREATE C,
  HERE 1- C@ 1+ ALLOT ALIGN \ Max. Laenge?    ( **)
  DOES> ( nr -- adr )
  COUNT ROT UMIN + ;        \ Range?          ( **)

16 CHAR-ARRAY BRETT \ 15 Zeichen und ein leeres Feld
0 VALUE HIER        \ Leerfeldposition (Zwischenraum)

: ENDSTAND ( -- )
  S" VIJGEBLAD*FORTH " 0 BRETT SWAP MOVE
  15 TO HIER ;

\ Anm. d. Uebers.:
\ VIJGEBLAD = Vereinszeitschrift der Forth-gebruikersgroep,
\ der niederlaendischen Forth-Gesellschaft.
```

```
: .BRETT ( -- )
  0 5 AT-XY
  0 BRETT 4 0
  DO CR CR 33 SPACES
     4 0 DO COUNT EMIT 3 SPACES LOOP
     LOOP DROP 7 SPACES ;

\ Der Spieler waehlt den zu verschiebenden Buchstaben.
\ Darum muessen alle Zeichen verschieden sein.
0 VALUE TASTE
: >CODE ( ch -- c ) DUP $20 AND - ; \ Upper?  ( **)
: SPIELER ( -- ) KEY >CODE TO TASTE ;

\ Pos = Position bezueglich HIER. Brettrand beachten.
: WEST? ( -- pos ) HIER 3 AND 0<> -1 AND ;    ( **)
: OST?  ( -- pos ) HIER 1+ 3 AND 0<> 1 AND ;  ( **)
: NORD? ( -- pos ) HIER 3 > -4 AND ;          ( **)
: SUED? ( -- pos ) HIER 12 < 4 AND ;          ( **)

: BUCHSTABE? ( pos -- pos ) \ Der richtige Buchstabe?
  DUP HIER + BRETT C@ >CODE
  TASTE = AND ;                               ( **)

: DORT  ( -- nr ) \ Position des zu verschiebenden Buchstabens
  NORD? BUCHSTABE?
  WEST? BUCHSTABE? OR
  OST?  BUCHSTABE? OR
  SUED? BUCHSTABE? OR HIER + ;
```

```
: SCHIEB ( nr -- )
  DUP BRETT HIER BRETT
  OVER C@ OVER C@
  -ROT
  SWAP C! SWAP C!
  TO HIER ;
```

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

```
: SPIEL ( -- )
  PAGE ENDSTAND .BRETT 1997 ms
  32 0 DO RANDOM $F AND SCHIEB .BRETT LOOP
  BEGIN SPIELER DORT SCHIEB .BRETT
        TASTE $1B = \ [Escape]?
  UNTIL ;
```

Dieses Programm koordiniert Prozesse, die in gegenseitig verschiedenen Tempi ablaufen: Sie beginnen gleichzeitig, laufen gleichmäßig und sind zur gleichen Zeit fertig.

Roboterarm mit (z.B.) fünf Motoren

Der Vorsitzende der Forth-gebruikersgroep bastelte an einem Roboterarm mit fünf Schrittmotoren (zwei in der Schulter, einer im Ellenbogen und zwei im Handgelenk) und bat mich, mir einen Algorithmus auszudenken, mit dem sich das Ding geschmeidig bewegen lässt.

Aber ich verstehe überhaupt nichts von Hardware.

Du schickst so 'nem Motor eine Zahl. Der geht dann mit einem Ruck in eine Stellung, die dieser Zahl entspricht. Die momentane Stellung des Motors brauchst du also nicht zu kennen. Und den untersten Programmteil mach ich selbst.

Mit solcherart Hardwarekenntnissen bewaffnet, habe ich mir ein Programm überlegt, das tatsächlich zu funktionieren scheint! Nach Anpassung des Wortes `>MOTOR` natürlich. Bei mir zu Hause, ohne die Hardware, habe ich mir das Ergebnis lediglich auf dem Bildschirm betrachtet (mit `.HIER`).

Da ich keine speziellen Kenntnisse über den Arm voraussetze, gibt es auch keine Absicherung gegen den Versuch unmöglicher Stellungen. Selbst die Anzahl der Motoren können Sie anders wählen. Das Programm koordiniert Prozesse, die in gegenseitig verschiedenen Tempi verlaufen: Sie beginnen gleichzeitig, laufen gleichmäßig und sind zur gleichen Zeit fertig. Es sind also auch andere Anwendungen denkbar.
Beispielsweise ein Zeichentrickfilm über Achilles und die Schildkröte, oder das Ziehen gerader Linien mit dem fünfdimensionalen Plotter, den Sie schon immer mal bauen wollten.

Das Problem

Angenommen, ich bewege mich vom Zustand `<0 0 0 0 0>` nach `<10 20 13 7 30>`. Dann läuft der letzte Schrittmotor über den größten Winkel (`30`). Dieser größte Winkel (`30`) wird der Ausgangspunkt des Algorithmus: Ich verteile die Bewegung der einzelnen Motoren über 30 Etappen. Und jeder Motor macht pro Etappe entweder nur einen Schritt (`PLOP`) oder nichts (`PILI`).

Der letzte Motor macht in allen Etappen 1 Schritt (30-mal `PLOP`). Das ist einfach. Nun die anderen Motoren.
Abstand 10: 10 Etappen 1 Schritt (`PLOP`) und 20 Etappen nichts (`PILI`).
Gleichmäßig verteilt, wird das: `PILI PLOP PILI` (10-mal).
Abstand 20 bleibt auch einfach: `PLOP PILI PLOP` (10-mal).
Abstand 13 ist schwieriger: 13-mal `PLOP` und 17-mal `PILI`. Wie verteilt man das möglichst schön über 30 Etappen?

Die Lösung

Man sehe für die Motoren Benzintanks vor.

Der Kraftstoff

1. Pro Etappe verbraucht der Motor (auch wenn er nicht läuft!) 13 Liter (die Anzahl PLOPs, die Winkeldrehung).
2. Ist der Kraftstofftank nahezu leer, dann werden genau 30 Liter hinzugetankt (die Anzahl Etappen, der größte Winkel).

Die Überlegung

Pro Etappe verbraucht er 13 Liter, über 30 Etappen:
30 * 13 Liter. 30 * 13 = 13 * 30. Wie oft wird er tanken müssen?
Richtig, 13-mal. Und wie viele Schritte sollten es noch gleich sein? ...

Und hier die Lösung:

Wenn in einer Etappe nichts mehr im Tank ist, wird getankt und gePLOPt. Die übrigen (17) Etappen sind PILI. Nun lasse man den Benzintank beiseite und merke sich den Gedankengang, denn der wird im folgenden Code verwendet.

Der Forth-Code

```
\ ROBOTER-ARM
 MARKER -ARM FORTH DEFINITIONS DECIMAL
 : VARIABLES ( n -- ) CREATE CELLS ALLOT
   DOES> ( index body -- adresse ) SWAP CELLS + ;

 5 DUP CONSTANT #MOTOREN
 DUP VARIABLES HIER       \ Liste der Motorstellungen
 DUP VARIABLES DORT       \ Liste der Zielstellungen
 DUP VARIABLES SCHRITT    \ -1 oder +1 (Richtungen)
 DUP VARIABLES TANK       \ Kraftstoffmenge
 DUP VARIABLES VERBRAUCH  \ Verbrauch pro Etappe
 CELLS   0 HIER   SWAP 0 FILL

 0 VALUE #ETAPPEN         \ Fuer den laengsten Weg
 20 VALUE WARTE           \ Verzoegerung pro Etappe

 : ZIEL ( m0 m1 m2 m3 m4 -- ) \ Ziel festlegen
   #MOTOREN BEGIN 1-
      TUCK DORT !
   ?DUP 0= UNTIL ;
```

```
: VORBEREITUNG ( -- )
  0                           \ Fuer laengsten Weg
  #MOTOREN 0                  \ Pro Motor:
  DO I DORT @ I HIER @
     2DUP <
     DUP 2* 1+ I SCHRITT !    \ 1 oder -1, Richtung
     IF SWAP THEN -
     DUP I VERBRAUCH !        \ Abstand
     MAX                      \ Groesster Abstand?
  LOOP DUP TO #ETAPPEN
  2/ #MOTOREN 0
  DO DUP I TANK !             \ Tanks halb voll
  LOOP DROP ;

\ Fuer die Bildschirmversion, ohne die Hardware
: >MOTOR ( neue-stellung motor# -- ) 2DROP ;

\ Um die Motorstellungen auf den Bildschirm zu bringen
: .HIER ( -- ) CR #MOTOREN 0
  DO I HIER @ 4 .R LOOP SPACE ;

: ETAPPE ( -- ) #MOTOREN 0
  DO I TANK @ I VERBRAUCH @
     2DUP <                   \ Kraftstoff zu knapp?
     IF #ETAPPEN -            \ Dann auftanken
        I SCHRITT @ I HIER +! \ Neuer Motorstand
        I HIER @ I >MOTOR     \ P-L-O-P
     THEN - I TANK !
  LOOP ;

: BEWEGE ( -- ) \ Das Ziel muss schon festgelegt sein.
  VORBEREITUNG
  #ETAPPEN 0
  ?DO .HIER ETAPPE
     KEY? IF KEY DROP LEAVE THEN
     WARTE MS
  LOOP .HIER ;

: GEHE ( zielpositionen -- ) ZIEL BEWEGE ;

\ Beispiel: 10 20 13 7 30 GEHE
```

```
\ ----- Beispiel -----

  10  20  13   7  30  GEHE [rtn]
   0   0   0   0   0
   0   1   0   0   1
   1   1   1   0   2
   1   2   1   1   3
   1   3   2   1   4
   2   3   2   1   5
   2   4   3   1   6
   2   5   3   2   7
   3   5   3   2   8
   3   6   4   2   9
   3   7   4   2  10
   4   7   5   3  11
   4   8   5   3  12
   4   9   6   3  13
   5   9   6   3  14
   5  10   6   3  15
   5  11   7   4  16
   6  11   7   4  17
   6  12   8   4  18
   6  13   8   4  19
   7  13   9   5  20
   7  14   9   5  21
   7  15  10   5  22
   8  15  10   5  23
   8  16  10   6  24
   8  17  11   6  25
   9  17  11   6  26
   9  18  12   6  27
   9  19  12   7  28
  10  19  13   7  29
  10  20  13   7  30  ok
```

CATCH und THROW I

Leo: Das letzte Mal hast du die Begriffe Signed und Unsigned erläutert. Könntest du mir heute etwas über `THROW` erzählen? Ich weiß, dass `THROW` eine Zahl auf dem Stack erwartet. Wenn das eine Null ist, passiert nichts. Bei anderen Werten verlässt Forth das Programm und setzt manchmal eine Fehlermeldung auf den Bildschirm mit der Mitteilung, dass das die Message Nummer soundso ist. Das Ergebnis beispielsweise von

```
-14 THROW
```

ist

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

Darüber gibt es bestimmt mehr zu erzählen.

Theo: Gut. Ich hol mal etwas aus. Du hast `EXECUTE` bereits kennen gelernt. Kannst du vorhersagen, was das Folgende tut?

```
17 ' DUP EXECUTE .S
```

Leo: Das ist leicht. `'` (Tick) sucht `DUP` auf, und `EXECUTE` führt dieses `DUP` aus. Das Ganze arbeitet also wie `DUP`.

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

Theo: Genau. Ich mach erst mal noch ein Wort, das ... oder nein, krieg mal heraus, was `BS` macht:

```
: BS BEGIN DEPTH 0> WHILE DROP REPEAT ;
```

Leo: Solange `DEPTH` größer als null bleibt, wird gedropt, `BS` säubert also den Stack.

Theo: O.K. Nun Puzzle Nr 1. Was macht:

```
BS 1 ' DUP CATCH .S
```

Ich gebe dir den Hinweis, dass `CATCH` hier beinahe dieselbe Wirkung hat wie `EXECUTE`. Der Unterschied besteht darin, dass `CATCH` nach Ablauf noch eine Null auf den Stack legt. Überleg dir die Antwort, bevor du auf `[rtn]` drückst.

Leo: `BS` leert den Stack, dann kommt eine `1` und die wird gedupt.

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

Oh ja, und die Extra-Null. Warum übrigens die Null?

Theo: Wirst du gleich sehen.

```
BS 2 ' ; CATCH .S [rtn] ( 2 -14 ) ok
```

Wenn die Extra-Zahl von `CATCH` eine Null ist, bedeutet das, dass der Vorgang geglückt ist. Wenn der Vorgang nicht glückt, dann liefert er anstelle der Null eine Fehlernummer.

Leo: Was ist denn dann in diesem Fall nicht geglückt?

Theo: Probier mal dasselbe mit `EXECUTE`.

```
BS 2 ' ; EXECUTE .S [rtn]
 Compile-only word (message # -14)
```

Leo: Hm!
(2 Minuten Stille)
Das `.S` hinter `EXECUTE` wird nicht ausgeführt, das hinter `CATCH` aber sehr wohl.

Theo: Richtig. `EXECUTE` wirft hier das Handtuch. `CATCH` fängt eventuelle Fehler auf und sorgt dafür, dass der Stack wiederhergestellt wird, d.h., dass er nach der misslungenen Befehlsausführung genauso tief ist wie vorher. `CATCH` ersetzt das Token des missglückten Vorgangs durch die Fehlernummer. Nach gelungenen Befehlsausführungen kommt die Erfolg vermeldende Null naturgemäß oberhalb des erreichten Ergebnisses zu liegen. Sieh dir mal `BS 1` an.
Und das folgende Beispiel:

```
BS 3 S" DUP" ' EVALUATE CATCH .S
```

Leo: Das hat denselben Effekt wie Beispiel 1.

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

Theo: Richtig. Aber jetzt das:

```
BS 4 S" vijg" ' EVALUATE CATCH .S
```

Leo: Dieses "`vijg`" kann nicht ausgewertet ("evaluiert") werden, also wird nach der `4` eine bestimmte Fehlernummer auf den Stack gelegt.

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

Wo kommen die zwei Nullen her?

Theo: Bedenke, dass nach einem Fehler der Stack vor und nach `CATCH` gleich tief ist.

Leo: Ah ja, natürlich, der `vijg`-String lag ja auch auf dem Stack. Aber warum hat der sich in `0 0` verwandelt?

Theo: `EVALUATE` macht sich zunächst an den String, verwendet dabei den Stack, und im Moment des Fehlers stand da unverkennbar `0 0`. Nun kommt was zum Nachdenken:

```
BS 5 ' CHAR CATCH 8 .S
```

Leo: `CATCH` führt `CHAR` aus, das den ASCII-Code von `8` aufsucht, und hängt noch eine Null dran.

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

Theo: Richtig. Für das folgende Beispiel musst du wissen, dass `CATCH` vor Erledigung seiner Aufgabe den momentanen Input-Strom aufbewahrt. Wenn etwas schief geht, stellt es den Stack (die Stacks) und den Input-Strom wieder her. Warum? Denk doch mal kurz darüber nach, was passieren würde, wenn man nach einem Fehler beim Laden einer Datei den Input-Strom auf sich beruhen lassen würde.

```
BS 6 ' ' CATCH 666 .S [rtn] ( 6 -13 666 ) ok
```

`CATCH` führt den Tick auf `666` aus. Das gelingt natürlich nicht. Es legt die Fehlernummer -13 auf den Stack und stellt den Input-Strom wieder her. Von dem wird auf die Stelle unmittelbar nach `CATCH` verwiesen. `666` wird erneut gelesen und ausgeführt, und schließlich ist `.S` an der Reihe.

Es tut mir Leid, aber ich muss jetzt weiter. Ich gebe dir einen kleinen Puzzle-Ausschnitt, den du mit den Kenntnissen, die du jetzt hast, lösen können müsstest.

```
BS 7  ' S" CATCH Vijgeblaadje" .S
BS 8  ' ' CATCH vijg .S
BS 9  ' TO CATCH BL .S
BS 10 ' DUP ' CATCH EXECUTE .S
BS 11 ' DUP ' CATCH CATCH .S
BS 12 ' DUP ' CATCH ' CATCH CATCH .S
BS ( 13 ) ' DROP CATCH .S
BS 14 ' THROW CATCH .S
```

Tschüss!

Leo: Tschüss!

[wird fortgesetzt]

CATCH und THROW II

Theo: Man verwendet `CATCH` und `THROW` in komplexen Programmen, die Teile enthalten, deren Ablauf nicht immer vorhergesehen werden kann. Um hier kein allzu ausgedehntes Beispiel geben zu müssen, nehme ich meine Zuflucht zu einem Kunstgriff:
Ich betrachte das gesamte Forthsystem als ein komplexes Programm, in welchem der Anwender viele unvorhergesehene Dinge tun kann und in welchem folglich viele `THROWs` vorkommen. Das gesamte Forth setze ich in ein `CATCH` und da baue ich ein kleines Programm drum herum, so dass du sehen kannst, was passiert.
Ziel des Ganzen: Demonstration der Funktion von `THROW` und `CATCH`.

```
\ CATCH and THROW
: .DEPTH ( -- ) DEPTH 0 .R ." : "
  DEPTH 0<                      \ Wenn noetig:
  IF BEGIN 0 DEPTH 0= UNTIL     \ Stackreparatur
  ." Stack underflow "          \ mit Meldung.
  THEN ;

: .TOP2 ( -- )          \ Zeige, wenn vorhanden,
  DEPTH 1 >             \ die obersten zwei
  IF OVER . THEN        \ Zahlen, die sich auf
  DEPTH IF DUP . THEN ; \ dem Stack befinden.

: .TOP4 ( -- )
  DEPTH 4 > IF ." ~ " THEN
  DEPTH 2 > IF 2>R .TOP2 2R> THEN
  .TOP2 ;

: STATE-MERKMAL ( -- char )
  STATE @ IF [CHAR] ] ELSE [CHAR] [ THEN ;

: .SITUATION ( -- )
  CR .DEPTH .TOP4 STATE-MERKMAL EMIT SPACE ;

CREATE EINGABEPUFFER 80 ALLOT

: EINGABE ( -- adr len )
  EINGABEPUFFER DUP 80 ACCEPT SPACE ;

: RISIKO-PROGRAMM ( -- )
  .SITUATION EINGABE EVALUATE ;

: BS ( -- )
  BEGIN ['] RISIKO-PROGRAMM CATCH
     DUP
     IF DUP . ." THROW ist ausgefuehrt "
     THEN DROP
  AGAIN ;
```

----- Beispiel einer Sitzung -----

Fett gedruckter Text ist Eingabe, der Rest ist Reaktion des Programms.
`[rtn]` ist die Returntaste.

Achtung: die sonst üblichen Reaktionen von Forth auf Fehler unterbleiben nun.

```
BS [rtn]
0: [ DROP [rtn] -4 THROW ist ausgefuehrt
0: [ 12 [rtn]
1: 12 [ 1999 [rtn]
2: 12 1999 [ : ZWAP [rtn]
2: 12 1999 ] QWERTY [rtn]  -61 THROW ist ausgefuehrt
```

Mit `QWERTY` kann `EVALUATE` nichts anfangen. Es ist kein Forth-Wort und auch keine Zahl, aber der Compile-Zustand bleibt aufrechterhalten, der Stack bleibt intakt.

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

Trotz der Störung scheint `ZWAP` doch compiliert worden zu sein.

```
2: 12 1999 [ ZWAP [rtn]
2: 1999 12 [ ' ASDF [rtn]  -13 THROW ist ausgefuehrt
```

TICK kann das Wort `ASDF` nicht finden.

```
2: 1999 12 [ -56 THROW [rtn]
[rtn]  ok
.S [rtn] ( 1999 12 )  ok
```

Mit `BS` gerät man in eine unendliche Schleife, aus der man jedoch mit Worten wie `QUIT` oder `BYE` herauskommen kann. Das Letztere ist natürlich nicht beabsichtigt. Bei `-56 THROW` haben wir es mit einem speziellen `THROW` zu tun, bei welchem vereinbart wurde, dass es `QUIT` ausführt.
Probier auch `ABORT` und `TRUE ABORT" Hallo!"` aus.
`ABORT` hat `-1 THROW` auszuführen und `ABORT"` den Befehl `-2 THROW` .

Die Puzzle-Zeilen aus Kapitel 208

```
: BS BEGIN DEPTH 0> WHILE DROP REPEAT ;
```

```
BS 7 ' S" CATCH Vijgeblaadje" .S [rtn] ( 7 adr len 0 )
```

`adr len` sind Adresse und Länge des Strings "Vijgeblaadje".

```
BS 8 ' ' CATCH vijg .S [rtn]
```

Forth gibt eine Fehlermeldung aus, da es nach dem `CATCH` noch einmal probiert, `vijg` zu finden.

```
BS 9 ' TO CATCH BL .S [rtn] ( 9 -32 32 )
```

`TO` kann nicht auf `BL` wirken: Fehlernummer -32, sodann `BL` (=32).

```
BS 10 ' DUP ' CATCH EXECUTE .S [rtn] ( 10 10 0 )
```

Dup-Aktion und erfolgbestätigende Null

```
BS 11 ' DUP ' CATCH CATCH .S [rtn] ( 11 11 0 0 )
```

Dup-Aktion und 2 erfolgbestätigende Nullen

```
BS 12 ' DUP ' CATCH ' CATCH CATCH .S [rtn] ( 12 12 0 0 0 )
```

```
BS ( 13 ) ' DROP CATCH .S [rtn] ( )
```

Drop-Aktion auf einen leeren Stack, danach eine erfolgbestätigende Null. Mein Forth protestiert nicht.

```
BS 14 ' THROW CATCH .S [rtn] ( 14? 14 )
```

Die erste 14 kann überschrieben sein.

Über das Manipulieren des Input-Stroms.

MANY TIMES

Forth am Werk

Wenn der Benutzer eine Zeile eintippt und auf `[rtn]` drückt, rückt Forth dieser Zeile wie folgt zu Leibe:

Es sucht nach dem ersten Nicht-Zwischenraum (hier beginnt ganz bestimmt ein Forth-Wort), dann nach dem ersten darauf folgenden Zwischenraum (da endet das Wort). Hierbei fungiert die Systemvariable `>IN` als Orientierungshilfe.
Forth hält nämlich in `>IN` eifrig fest, welche Stelle in der Zeile es sich gerade anschaut.

Danach sucht es in seinem Wörterbuch nach dem gerade herausgeschälten Wort und führt es, falls die Suche erfolgreich war, aus (executiert es). Findet Forth das Wort nicht, dann hört es in seinem Bemühen auf und kümmert sich nicht mehr weiter um den Rest der Zeile.

Solange es gelingt, wiederholt Forth diese Prozedur: Das nächste Wort in der Reihe wird ermittelt, wobei stets `>IN` ins Spiel kommt, und ausgeführt. Bis zum Ende der Zeile.

Da Forth keine Syntax hat, braucht es von vornherein nicht zu untersuchen, ob die Zeile fehlerfrei ist.
Eventuelle Tippfehler kommen von selbst ans Licht, denn Forth kann verkehrt geschriebene Worte sowieso nicht finden.
Mit Programmierfehlern ist das natürlich eine andere Sache. Aber das ist keine Syntaxfrage.

```
) >IN @ . [rtn] ?
 1234567... Positionen
```

Der Inhalt von `>IN` wird über `@` hereingeholt. Forth hat die Zeile bis `@` einschließlich des Zwischenraums danach dann schon gelesen.

```
)       >IN @ . [rtn] ?
 1234567890123... Positionen

)       >IN @ . >IN @ . [rtn] ?
 123456789012345678901... Positionen
```

Forth vertraut darauf, dass der Anwender kein Wort eintippt, das den Inhalt von `>IN` heimlich verändert. Aber sag sowas nie in Anwesenheit eines Forth-Programmierers ...

Forthprogrammierer ans Werk

```
: MANY ( -- ) >IN @ STOP? AND >IN ! ;
```

Halt, warte noch einen Moment! `STOP?` ist kein Standard-Forthwort. Darauf müssen wir erst noch schnell eingehen.

```
: 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] ?
```

liefert den Wert null. Aber gib dieselbe Zeile noch einmal ein und drück unmittelbar nach `[rtn]` auf irgendeine Taste. Drei Möglichkeiten.

1. Du drückst auf `[Esc]`: Forth protestiert und reagiert mit einer Fehlermeldung.

2. Du drückst auf eine andere Taste, kein `[Esc]`, aber auch kein `[Zwischenraum]`: Forth zeigt `-1` (TRUE) an.

3. Du drückst auf `[Zwischenraum]`: Forth macht Pause, hält mit allen Aktionen inne und wartet auf eine weitere Taste. Wird das wieder ein `[Zwischenraum]`, dann ist das Ergebnis `0` (FALSE). Nach `[Esc]` kommt wieder Protest und die übrigen Tasten liefern `-1` (TRUE).

Das klingt schrecklich kompliziert, aber wenn du es ein paar Mal ausprobiert hast, wird es ganz schnell logisch. `STOP?` (kein Standard-Forthwort also) sitzt häufig in Worten wie `WORDS SEE DUMP`, Worte, die Text erzeugen, welcher am Bildschirm durchlaufen kann. Mit `STOP?` hast du die Möglichkeit, das Durchlaufende mal kurz anzuhalten oder auch ganz abzubrechen.

```
: MANY ( -- ) >IN @ STOP? AND >IN ! ;
```

Solange `STOP?` hier null liefert, wird `>IN` auf null gesetzt: Forth wird hinters Licht geführt, 'denkt' fortwährend, noch nichts von der Zeile gelesen zu haben, und fängt die Zeile stets wieder von vorn an.

Mit `MANY` kannst du interaktive Schleifen herstellen:

```
) 888 . MANY DROP [rtn] ?
```

Ziemlich sinnlos, aber:

```
) BL [rtn]
) DUP EMIT 1+ MANY DROP [rtn] ?
```

TIMES ist eine differenzierte Variante von MANY, bei der du angibst, wie oft Forth das erste Stück der Zeile ausführen soll.

```
0 VALUE #TIMES          \ Zaehler
: TIMES ( n -- )
  #TIMES 1+ TUCK        ( #times+1 n #times+1 )
  0 TO #TIMES           \ Sicherheit über alles.
  = STOP? OR            \ n-tes Mal oder Eingriff von Stop?
  IF DROP EXIT THEN     \ Dann klar.
  TO #TIMES 0 >IN ! ; \ Zaehler erhoehen
                        \ und noch 'ne Runde.

) BL [rtn]
) DUP EMIT 1+ 96 TIMES DROP [rtn] ?
```

Die Anzahl der Wiederholungen, wie oft Forth die Zeile wieder von vorn beginnen soll, gibst du dem Wort TIMES über den Stack mit. Der Value #TIMES ist für den internen Gebrauch durch TIMES bestimmt. Wenn du den trotzdem von vornherein auf -100, oder schlimmer noch, auf 100 setzt ...

BASE für die Dauer eines einzelnen Wortes verändern

HX DM BN

Leo: In Forth kann man mit verschiedenen Zahlensystemen arbeiten, indem man beispielsweise `HEX` oder `DECIMAL` eintippt. Nun habe ich aber auch schon gesehen, dass man mit einem Dollarzeichen, das vorn an eine Zahl geheftet wird, bewirken kann, dass diese Zahl als Hexadezimalzahl interpretiert wird, selbst wenn sich Forth gerade im dezimalen Zustand befindet. In deinem Forth geht das nicht, ich würde es aber für gar nicht so schlecht halten. Und eigentlich hätte ich diese Idee gern noch um eine Möglichkeit erweitert gesehen, so'n Dollarzeichen nicht nur Zahlen, sondern auch Worten, die Zahlen ausgeben, als führendes Zeichen voranzustellen. Aus `$.` und `$U.` und `$.S` würden dann, um nur ein paar Beispiele zu nennen, Varianten der ihre Zahlen dezimal ausgebenden Worte `.` und `U.` und `.S` werden. Ist das möglich, ohne dass man für alle derartigen Worte Extra-Definitionen anfertigen muss?

Theo: Tja, kaum dass du das Arbeiten in verschiedenen Zahlensystemen normal findest, und das ist es in Forth, schraubst du deine Ansprüche höher und strebst eigentlich eine Methode an, die die Eintipparbeit noch weiter reduziert. Lang lebe die Benutzerfreundlichkeit.

Leo: Du klingst nicht sehr enthusiastisch. Hast du was gegen die Idee?

Theo: Nein, eigentlich nicht. Ich denke schon, dass man das machen kann. Ein gewöhnlicher Forth-Interpreter liest ein Wort und sucht nach ihm in seinem Wörterbuch. Wenn er es findet, ist alles gut. Wenn er es aber nicht finden kann, schaut er nach, ob es nicht vielleicht eine Zahl ist. Wenn auch das nicht fruchtet, gibt er auf. Die Routine, die dafür verantwortlich ist, muss natürlich auch mit der Situation "Dollarzeichen vorn an einer Zahl" umgehen können. Wenn sie aber schon dieser Aufgabe gewachsen ist, kann man auch gleich dafür sorgen, dass sie, wenn sie ein solches `$` antrifft, wieder erst nachschaut, ob ein bestehendes Wort daran festhängt, und dann, ob es sich um eine Zahl handelt.
Weißt du, ich bin nicht so sehr dafür, dass der Forth-Interpreter, der sehr einfach sein kann, durch solcherart nebensächlichen Krimskrams verdorben wird. Außerdem ist es ein Eingriff in ein bestehendes Forthsystem, wenn man eine solche Möglichkeit hinzufügen will. Dazu muss man die einzelnen Systemabhängigkeiten gut verstehen.

Leo: Geht es nicht einfacher, ohne Eingriff in den Interpreter?

Theo: Ja, das geht schon, wenn du mit einem separaten Wort anstelle eines an das nächste Wort angehefteten führenden Zeichens einverstanden bist. `HX` zum Beispiel:

```
HX 10  \ Lies die Eingabe als hex.
HX .S  \ Liefere hex als Ausgabe.
```

Leo: Und Forth selbst? Bleibt das dann im Dezimalzustand? Wie wird denn so'n `HX` dann aussehen?

Theo:
```
: HX
  HEX                       \ Zeitweilige Basiszahl
  BL WORD COUNT EVALUATE    \ Aktion!
  DECIMAL ; IMMEDIATE       \ Basiszahl wiederherstellen
```

Nur zu, probier's mal aus.

Leo:
```
HX 10 . [rtn] 16  ok
10 HX . [rtn] A  ok
```

Jawoll, mein Kompliment, das ist Zauberei.

Theo: Jubel nicht zu früh. Ein paar Schwachpunkte sitzen noch drin. Erstens habe ich es für selbstverständlich gehalten, dass sich das System im Dezimalzustand befindet, denn am Ende setze ich es ganz normal (?) wieder auf dezimal. Das ist nicht besonders schön. Wir rufen den `>R` ... ettungsdienst:

```
: HX
  BASE @ >R                 \ Rette Basiszahl
  HEX                       \ Zeitweilige Basiszahl
  BL WORD COUNT EVALUATE    \ Aktion!
  R> BASE ! ; IMMEDIATE     \ Stell Basiszahl wieder her
```

Leo: Das kapier ich. Gerettet werden kann nicht auf den gewöhnlichen Stack, da das nächste Wort alles Mögliche sein kann. Man weiß also nicht, was es mit dem Stack anrichten wird. Und zweitens?

Theo: Zweitens steht da knallhart `HEX` drin, ich hätte aber gern eine Routine, die dasselbe auch mit anderen Basiszahlen fertig bringt. Das könnte man mit einer Variablen oder einem Value lösen, aber ich betrachte das als eine ausgezeichnete Gelegenheit, dich noch einmal mit `CREATE` und `DOES>` zu konfrontieren. Sieh dir das mal an:

```
: FFBASE ( zeitweilige-basiszahl ccc -- )
  CREATE C, ALIGN IMMEDIATE
  DOES> C@                  ( tempbase )
  BASE @ >R                 \ Save Basiszahl
  BASE !                    \ Zeitweilige Basiszahl
  BL WORD COUNT EVALUATE    \ Aktion!
  R> BASE ! ;               \ Stell Basiszahl wieder her
DECIMAL
16 FFBASE HX   2 FFBASE BN   10 FFBASE DM
```

Theo: Nun verfügen wir mit `FFBASE` über ein Wort, mit welchem wir ganz einfach und kurz zu jeder gewünschten Basiszahl so'n HX-artiges Wort definieren können. Die (allgemeine) Aktion dieses Wortes steht hinter `DOES>` und das `C@` am Anfang der Aktion liest die (betreffende) momentane Basiszahl aus dem Body des Wortes aus.
Es ist jammerschade, aber in bestimmten Situationen funktioniert das nicht zufrieden stellend. So geht es einem immer wieder, wenn man sich ans Programmieren macht. Versuch mal herauszubekommen, wann es nicht geht. Ich gebe dir einen Hinweis.

(Theo:)

Der Interpreter verarbeitet drei Sorten von Worten, gewöhnliche Worte, Immediate-Worte und Zahlen. Der Interpreter kann sich in zwei Zuständen befinden, interaktiv und compilierend.
Das sind 6 Möglichkeiten und mit einer davon stimmt was nicht. Finde das mal heraus.

Zwei Tage später

Leo: Ich bin gestern dahinter gekommen, warum du das Wort FFBASE genannt hast! Und ich habe entdeckt, was daran nicht stimmt. Wir haben uns noch nicht über das Compilieren unterhalten, aber in einer Definition liefert

```
:  ..  HX .  ..  ;
```

nix. Es geht dabei nichts schief, aber es passiert auch nichts. HX zeigt keine Wirkung und ich begreife auch, warum. Es ist nämlich völlig ohne Bedeutung, welche Basiszahl beim Compilieren des Punktes eingeschaltet ist. Mit Zahlen geht alles gut. Du musst dir also noch etwas ausdenken, damit `HX` auch solchen Forth-Worten beikommt, die compiliert werden. Das gelingt dir sicher auch noch.

Theo: Natürlich geht das. Der Code wächst damit gut.

```
: (BASE) ( token tempbase -- )
  BASE @ >R                \ Bewahre Basiszahl auf
  BASE !                   \ Temporaere Basiszahl
  EXECUTE                  \ Aktion
  R> BASE ! ;              \ Stelle Basis wieder her

: FFBASE ( tempbase ccc -- )
  CREATE C, ALIGN IMMEDIATE
  DOES> C@                 ( tempbase )
  BL WORD DUP FIND
  S>D STATE @ AND          \ Compilieren?
  IF DROP NIP SWAP         ( xt tempbase )
     POSTPONE 2LITERAL     ( tempbase x )
     POSTPONE (BASE)
     EXIT
  THEN
  IF NIP SWAP (BASE) EXIT  \ Ausfuehren
  THEN DROP SWAP           ( word tempbase )
  BASE @ >R                \ Bewahre Basiszahl auf
  BASE !                   \ Temporaere Basiszahl
  COUNT EVALUATE           \ Zahleneingabe
  R> BASE ! ;              \ Stelle Basis wieder her

DECIMAL
16 FFBASE HX   2 FFBASE BN   10 FFBASE DM
```

HX beeinflusst nur das nächste Wort. Wenn Forth sich an diesem Wort verschluckt, gibt es eine Fehlermeldung aus und steigt aus. `HX` kommt dadurch nicht mehr dazu, die Basiszahl wiederherzustellen. Dem kann dadurch abgeholfen werden, dass man das fehlerempfindliche `EVALUATE` in ein `CATCH` setzt. Die letzten beiden Zeilen von `FFBASE` werden dann:

```
COUNT ['] EVALUATE CATCH
R> BASE ! THROW ;
```

Solch eine Sicherheitsmaßnahme heißt natürlich mit Kanonen auf Spatzen schießen, sie funktioniert aber ausgezeichnet.

Die sechs Möglichkeiten von Theo

1. `HX 10 . [rtn] 16 ok`
   `10` wird als Hexzahl aufgefasst.

2. `: DUTZEND HX C . ;`
   `HX` sieht `C` als eine Hexzahl an und compiliert es. Der Definition kann man es nicht mehr ansehen, dass `HX` eine Rolle gespielt hat.
   ? Was wird `BN DUTZEND` bewirken?

3. `HX SEE FFBASE [rtn] ...`
   `SEE` decompiliert `FFBASE` in hex. Natürlich nur dann, wenn die Basiszahl nicht innerhalb `SEE` explizit manipuliert wird.

4. `: .BITS BN U. ; -1 .BITS [rtn] 11...11 ok`
   `.BITS` gibt immer binär aus. Allein in diesem Zusammenhang wird die Veränderung von `BASE` auch wirklich compiliert.
   ? Was wird `HX .BITS` bewirken?

5. `: [.S] .S ; IMMEDIATE BN [.S] [rtn] ...`

6. `: TEST HX [.S] 0<> ;`
   `HX [.S]` gibt während des Compilierens von `TEST` den Stackzustand in hex wieder und lässt keinerlei Spuren davon in `TEST` zurück.

Das letzte Wort

```
BN #16 . [rtn] 16  ok
```

Wie schon gesagt, kann man in manchen Forthsystemen mit einem einer Zahl vorangestellten speziellen Zeichen die Basiszahl außer Kraft setzen (`#` für dezimal, `$` für hexadezimal und `%` für binär). Nachdem `BN` seinen Einfluss geltend gemacht hat, kommt der Gartenzaun an die Reihe. Der Gartenzaun hat das letzte Wort ... und gewinnt.

D

300

Index

Alle Forth-Worte in alphabetischer Reihenfolge

!

" # $ % & '

( ) * + , - . /

Ø 1 2 3 4 5 6 7 8 9

: ; < = > ? @

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

[\ ]

~

	Kapitel	Word Set
!		
!	6, 103	core
#		
#	111, 203	core
#>	111, 202	core
#S	111, 202	core
#TIB	123	core
'		
' (Tick)	8, 124, 129	core
(		
(	5, 125	core, file
(LOCAL)	-	locals
*****		
*	3, 121	core
*/	19, 121	core
*/MOD	20, 121	core
+		
+	3, 121	core
+!	6, 103	core
+LOOP	114, 205	core
,		
, (Komma)	106	core
-		
-	3, 121	core
-ROT	101	nicht standard
-TRAILING	-	string

.

. (Punkt)	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

Ø

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, 211	double
2OVER	28, 101	core
2R>	102	core ext
2R@	102	core ext
2RDROP	102	nicht standard
2ROT	28, 101	double ext
2SWAP	28, 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
>NUMBER	108	core
>R	24, 102	core

?

?	-	toolkit ext
?DO	33, 114	core ext
?DUP	101	core

@

@	6, 104	core
@+	104	nicht standard

A

ABORT	133	core, error ext
ABORT"	30, 125, 133	core, error ext
ABS	20, 120	core
ACCEPT	16, 103, 107	core
AGAIN	113, 204	core ext
AHEAD	115, 204	toolkit ext
ALIGN	106, 211	core
ALIGNED	105	core
ALLOCATE	-	memory
ALLOT	15, 106	core
ALSO	130	search ext
AND	20, 116	core
ASSEMBLER	-	toolkit ext
AT-XY	109	facility ext

B

BASE	11, 108	core
BEGIN	12, 113, 204	core
BIN	-	file
BINARY	11, 108	nicht standard
BL	5, 132	core
BLANK	132	string
BLK	-	block
BLOCK	-	block, file
BODY>	129	nicht standard
BUFFER	-	block, file
BYE	-	toolkit ext

C

C!	17, 103	core
C"	-	core ext
C,	106	core
C@	17, 104	core
C@+	104	nicht standard
CASE	115	core ext
CATCH	208, 209, 211	error
CELL+	16, 105	core
CELL-	105	nicht standard
CELLS	14, 105	core
CHAR	32, 124	core
CHAR+	16, 105	core
CHAR-	105	nicht standard
CHARS	16, 105	core

...

```
 ...
CLOSE-FILE      -                 file
CMOVE           -                 string
CMOVE>          -                 string
CODE            -                 toolkit ext
COMPARE         -                 string
COMPILE,        127               core ext
CONSTANT        6, 126            core
CONVERT         -                 core ext
COUNT           17, 104           core
CR              2, 109            core
CREATE          15, 126, 211      core
CREATE-FILE     -                 file
CS-PICK         115               toolkit ext
CS-ROLL         115, 204          toolkit ext
```

D

```
D!              -                 double ext
D+              28, 122           double
D-              28, 122           double
D.              28, 110, 202      double
D.R             110, 202          double
D0<             118               double
D0=             118               double
D2*             28, 120           double
D2/             28, 120           double
D<              28, 118           double
D=              28, 118           double
D>F             -                 float
D>S             28, 132           double
D@              -                 double ext
DABS            120               double
DECIMAL         11, 108           core
DEFINITIONS     130               search
DELETE-FILE     -                 file
DEPTH           101               core
DF!             -                 float ext
DF@             -                 float ext
DFALIGN         -                 float ext
DFALIGNED       -                 float ext
DFLOAT+         -                 float ext
DFLOATS         -                 float ext
DMAX            119               double
DMIN            119               double
DNEGATE         28, 120           double
DO              11, 114, 205      core
DOES>           128, 211          core
DROP            5, 101            core
DU.             110, 202          nicht standard
 ...
```

```
 ...
DU.R            110, 202          nicht standard
DU<             -                 double ext
DUMP            16                toolkit ext
DUP             3, 101            core
```

E

```
EDITOR          9                 toolkit ext
EKEY            107               facility ext
EKEY?           107               facility ext
EKEY>CHAR       107               facility ext
ELSE            10, 112, 204      core
EMIT            1, 109            core
EMIT?           -                 facility ext
EMPTY-BUFFERS   -                 block ext
END-CODE        -                 toolkit ext
ENDCASE         115               core ext
ENDOF           115               core ext
ENVIRONMENT?     -                core
ERASE           132               core ext
EVALUATE        131, 211          core
EXECUTE         8, 129            core
EXIT            127               core
EXPECT          -                 core ext
```

F

```
F!              -                 float
F*              -                 float
F**             -                 float ext
F+              -                 float
F-              -                 float
F.              -                 float ext
F/              -                 float
F0<             -                 float
F0=             -                 float
F<              -                 float
F>D             -                 float
F@              -                 float
FABS            -                 float ext
FACOS           -                 float ext
FACOSH          -                 float ext
FALIGN          -                 float
FALIGNED        -                 float
FALOG           -                 float ext
FALSE           10, 117           core ext
FASIN           -                 float ext
 ...
```

```
 ...
FASINH          -               float ext
FATAN           -               float ext
FATAN2          -               float ext
FATANH          -               float ext
FCONSTANT       -               float
FCOS            -               float ext
FCOSH           -               float ext
FDEPTH          -               float
FDROP           -               float
FDUP            -               float
FE.             -               float ext
FEXP            -               float ext
FEXPM1          -               float ext
FILE-POSITION   -               file
FILE-SIZE       -               file
FILE-STATUS     -               file ext
FILL            103             core
FIND            129, 211        core
FLITERAL        -               float
FLN             -               float ext
FLNP1           -               float ext
FLOAT+          -               float
FLOATS          -               float
FLOG            -               float ext
FLOOR           -               float
FLUSH           -               block
FLUSH           -               file
FLUSH-FILE      -               file ext
FM/MOD          122             core
FMAX            -               float
FMIN            -               float
FNEGATE         -               float
FORGET          4, 124          toolkit ext
FORTH           130             search ext
FORTH-WORDLIST  -               search
FOVER           -               float
FREE            -               memory
FROT            -               float
FROUND          -               float
FS.             -               float ext
FSIN            -               float ext
FSINCOS         -               float ext
FSINH           -               float ext
FSQRT           -               float ext
FTAN            -               float ext
FTANH           -               float ext
FVARIABLE       -               float
FSWAP           -               float
F~              -               float ext
```

G

GET-CURRENT	-	search
GET-ORDER	-	search

H

HERE	15, 106	core
HEX	11, 108	core ext
HOLD	111, 203	core

I

I	11, 114	core
IF	10, 112, 204	core
IMMEDIATE	33, 131	core
INCLUDE-FILE	-	file
INCLUDED	-	file
INVERT	116	core

J

J	114	core ext

K

KEY	2, 107	core
KEY?	107	facility ext

L

LEAVE	33, 114, 205	core
LIST	-	block ext
LITERAL	127	core
LOAD	-	block
LOCALS|	-	local ext
LOOP	11, 114, 205	core
LSHIFT	116	core

M

M*	30, 122	core
M*/	-	double
M+	-	double
MANY	210	nicht standard
MARKER	-	core ext
MAX	119, 201	core
MIN	119	core
MOD	19, 121	core
MOVE	103	core
MS	132	facility ext

N

NEGATE	26, 120	core
NIP	22, 101	core ext

O

OF	115	core ext
ONLY	130	search ext
OPEN-FILE	-	file
OR	20, 116	core
ORDER	130	search ext
OVER	5, 101	core

P

PAD	132	core ext
PAGE	36, 109	facility ext
PARSE	123	core ext
PICK	101	core ext
POSTPONE	124, 211	core
PRECISION	-	float ext
PREVIOUS	130	search ext

Q

QUERY	131	core ext
QUIT	131	core

R

R/O	-	file
R/W	-	file
R>	24, 102	core
R@	24, 102	core
RDROP	102	nicht standard
READ-FILE	-	file
READ-LINE	-	file
RECURSE	127	core
REFILL	-	core ext, block ext, file ext
RENAME-FILE	-	file ext
REPEAT	12, 113, 205	core
REPOSITION-FILE	-	file
REPRESENT	-	float
RESIZE	-	memory
RESIZE-FILE	-	file
RESTORE-INPUT	-	core ext
ROLL	101	core ext
ROT	35, 101	core
RSHIFT	116	core

S

S"	125	core, file
S>D	28, 132	core
SAVE-BUFFERS	-	block, file
SAVE-INPUT	-	core ext
SCR	-	block ext
SEARCH	-	string
SEARCH-WORDLIST	-	search
SEE	22, 124	toolkit ext
SET-CURRENT	-	search
SET-ORDER	-	search
SET-PRECISION	-	float ext
SF!	-	float ext
SF@	-	float ext
SFALIGN	-	float ext
SFALIGNED	-	float ext
SFLOAT+	-	float ext
SFLOATS	-	float ext
SIGN	111, 202	core
SM/REM	122	core
SOURCE-ID	-	core ext, file
SPACE	34, 109	core
SPACES	34, 109	core
SPAN	-	core ext
STATE	-	core, toolkit ext
SWAP	3, 101	core

T

THEN	10, 112, 204	core
THROW	208, 209, 211	error
THRU	-	block ext
TIB	123	core
TIME&DATE	-	facility ext
TIMES	210	nicht standard
TO	22, 124	local, core ext
TUCK	22, 101	core ext
TYPE	17, 109	core

U

U.	14, 110, 201, 202	core
U.R	110, 201, 202	core ext
U<	14, 118	core
U>	118	core ext
UM*	122	core
UM/MOD	122	core
UMAX	119	nicht standard
UMIN	119	nicht standard
UNLOOP	115, 205	core
UNTIL	12, 113, 204	core
UNUSED	131	core ext
UPDATE	-	block

V

VALUE	22, 126	core ext
VARIABLE	6, 126	core
VOCABULARY	130	nicht standard

W

W/O	file	core
WHILE	12, 113, 205	core
WITHIN	118	core ext
WORD	123, 211	core
WORDLIST	-	search
WORDS	4, 130	toolkit ext
WRITE-FILE	-	file
WRITE-LINE	-	file

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