Noémi Adorján: Forth step by step (1990)

Rector: László Csurgai Technical publisher. For z80 fig-Forth1.1g

Content

Introduction

The FORTH chained-code interpreted language, developed by Charles Moore in the early 1960s, in solving the management problems of the telescope of the Kitt Peak Observatory. Like other programming languages, FORTH also has several "dialects",

some of which are standard. One is FORTH 79, written by Leo Brodie in the great book "Starting Forth". A later version of the classic FIG-FORTH 1.1. (aligned with FORTH 79).
What does FIG mean? FORTH INTEREST GROUP, PO BOX 1105, SAN CARLOS, CA 94070. This is a company founded for the promotion, implementation and use of FORTH. Not only have standard FORTH recommendations been developed, but with several other useful publications, the FORTH interpreter source list has been published for 9 different processors (which is partly in the assembly language of the machine and partly in FORTH). The 8080 source list can be found in the download package Z80FORTH.Z80) and other important information about FORTH (practically free). With such a manual (FIG-FORTH INSTALLATION MANUAL) it is no longer difficult to write a FORTH interpreter, with little exaggeration that anyone can engage in it.

While playing with FORTH, we will see that everything in FORTH can be. But not everything is recommended!
Do not be surprised by the inexperienced programmer (the experienced person) when experiencing "wild" phenomena. He may have made a small mistake and accidentally palmed in the wrong place. If you do not have a better idea, you can always reset the machine, refill FORTH in the knowledge that this has happened to others. FORTH's most beloved fan can not say it's easy to learn. Its benefits are due in particular to the fact that it is a machine-friendly language (so it is necessary to understand the "soul" of the computer) so that it can be expanded, transformed (so we need to know how FORTH works), its small memory requirement (fig-FORTH here it's only 6656 bytes!) and that much of it is original, witty, but not necessarily easy to understand. Knowing FORTH does not go without any effort.

- FORTH is a tool that allows you to quickly, quickly store and run a reliable program in a short time, and even this work is enjoyable;
- you may want to know more about the computer than BASIC, FORTRAN etc. necessary for programming;
- along with FORTH, we get to know some very clever programming tricks, which can come in handy;
- FORTH is completely different from the "usual" (high level) programming languages □□(especially BASIC). We can teach him an important ability: to accept a different approach. (Those who work with computers need this more than others.)

Much of the FORTH itself was written in FORTH. The FORTH source texts of the FORTH basic words serve as an abundant example, from which the author has well deserved. For everyone who has decided to learn FORTH, there is plenty of success and fun!

1. Getting Started

Fill in and start FORTH. This can be done after running IS-DOS by running the Z80FORTH.COM file. Later, we will launch FORTH differently.
The FORTH interpreter starts running the version number (this is Z80 fig-FORTH 1.1g). The report "No file" will be discussed later . Then the interpreter waits for him to give him a command line. She waits until we finish her line. How do you know when we're done? From there, press ENTER to end the line.

Enter the ENTER line to FORTH; Until pressing ENTER, nothing else happens than the FORTH is waiting for us.

This is worth noting if you do not want to spend a lot of time waiting to implement our instructions without ENTER. If we have coated, we can fix it. Press the ERASE key to delete the last character from the line - of course just before ENTER is pressed.
Let's start by finishing it: give a blank command line: just press ENTER in an empty line. The OK received as a response means that FORTH has made the statements (in our case, the big one) completely. After OK, FORTH waits for our next wish for the beginning of the next line. A simple word you understand:

```
CR
```

OK this time it is shown one line down, FORTH has a carriage return and a line up character.
Those who received an error message instead of OK were ignored and not exactly the same

in FORTH, all the instructions to the interpreter must be capitalized

To make the effect more spectacular, write the same thing in a row several times. We need to know that

each word is separated from each other by one line (one or more)

So, if we say:

```
CR CR CR CR
```

FORTH prints the four empty lines and rewards the regular job statement with OK. But if we write that

```
CRCR CR CR
```

then FORTH will be violated by the incomprehensible CRCR, and without honoring us, it will stop it. The two "good" CRs are not read yet, only one error code is obtained. Sometimes we get something like this when we try to know the word FORTH. (Declining DTCs can be found in Appendix B.)
Let's play more "dumpers":

```
42 EMIT
```

The word EMIT prints the character whose code was given to it; 42 is the star code.
Define a word for astrologers!

```
: CS 42 EMIT;
```

- Here, the colon means that we define a new word. The colon is also a word !! so there is room for space.
- The post-colon CS (or whatever else) is a name; so the new word will be called. The name can contain any character except the space, carriage return, and backspace characters (these are used to divide the words, mark the end of the line, and repair).
- What comes next (42 EMIT) is the act; Here's how the new word will work.
- The semicolon (which is also the case, so it is not necessary to "write" the words "before it, nor" directly "afterwards" other words) concludes the definition.

We have written our first FORTH program. Now this is the same FORTH word as any other, so it can be run with the description of its name:

```
CS
```

In fact, we can use the creation of new words:

```
: PONT CR CS;

: VONAL CR CS CS CS;
```

In the example, we tried every word before we used them in other words. so it may be (and recommended) to "be sure". One of the most attractive attributes of FORTH is this: the building blocks from which the program finally compiles can be tested separately after they are written.
Most of the FORTH basic words are written in the same way in FORTH, using other basic words. For example, SPACE. which writes a space on the screen, so it is built up:

```
: SPACE 32 EMIT;
```

The already known CR means:

```
: CR 13 EMIT 10 EMIT;
```

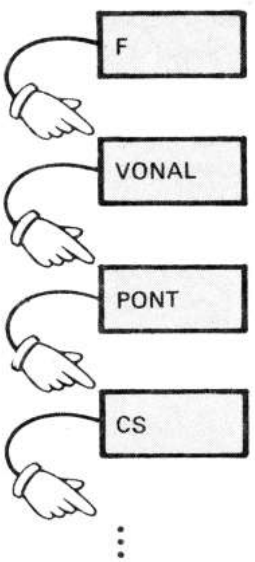
The first step is to mention the last step in using FORTH: from the FORTH interpreter to

```
BYE
```

so we can step out properly without losing data.

1.1. About the dictionary

What made CS, F, etc. executable word? What happens when we type such a "colon definition"?
FORTH keeps words that can be interpreted in a dictionary. After loading, the dictionary has the FORTH basic words. By creating new words, we will expand the dictionary - or, if you like, the FORTH language itself.
The names of dictionary words are written to the VLIST (Vocabulary List, the vocabulary, say: Vocabulary, meaning: dictionary). You can stop a "word" starting with VLIST by tapping any key. If we define our own words and then look at our vocabulary with VLIST, we see that the most recently defined words appear first; after the first operation, for example, the list of words begins as follows: F LINE POINT CS
The FORTH interpreter, when you want to interpret a word, first begins to search it in the dictionary. The last word defined; Here you will find information about where the last word written in front begins, so if you need to continue searching, and so on. If, then, in the example, after the definition of F, we write another word F:




```
: F70 EMIT;
```

then the word F is "redefined"; the FORTH warns you of an error message and writes the new word into the dictionary. Let's see what the result is

```
F
```

FORTH found this second F definition. (70 is the big F code.)
Let's stop for a moment! Good, it's good that CS, F etc. it can be executed by being included in the dictionary. We took it when we were defined. It may also be true that EMIT is in it. It's a basic word. But what do you know about 42 and 70? Are not all the songs in it? And if there is something in the dictionary, why does not the interpreter say why he pretends to be all right?
Principle. The principle is that what is not a dictionary word is a sure number, so the FORTH interpreter attempts to interpret the

resulting string after a failed search in the dictionary. If it does not go (there are "non-numeric" characters), then it's really a mistake. If it is a number, then this number will be in the stack.

1.2. What is a stack?

The stack (the English name stack) is an important part of FORTH. Each word is "mailed" to each other. For example, EMIT looks at the stack in the stack that the character code should be written on the screen; after knocking it down, it will destroy it from the stack. They call it a stack because there are more things (more than one number in our case) can be kept; We have only one access to one of them: the one that has been there for the last time, that is, "upside down". To get to know this, we learn a new FORTH basic word.

- Person . (ie a single point character).
- Function: prints the number at the top of the stack and places a space on the screen.
- Impact on the Stack: Clears the entered number from the top of the stack.

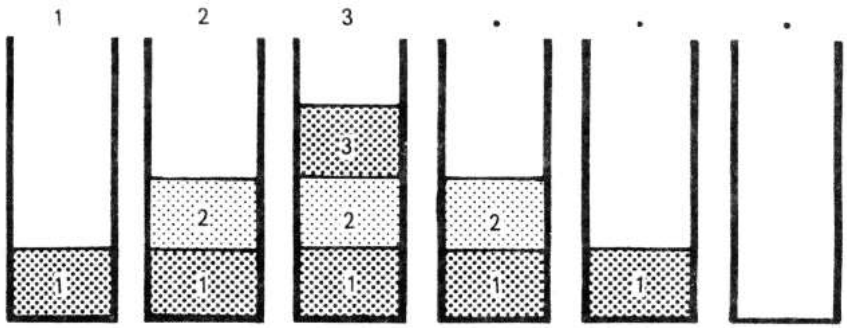
Let's try it!

```
65
.
```

We put 65 on the stack (first row), we asked "point" (second line). Back it up! He also deleted it. Let's make sure. Write another point! We get a bug signal, which means we wanted to use more batteries than we used to buy.
And if not? It's easy for the Experiment Reader to do a lot of things to get there. Maybe there was something in the stack. The simplest way to empty the stack is to type a word we know that FORTH does not know. The "angry" interpreter empties the vermet; if you then try some of the above, you will see that this is true. The method can be useful when we accidentally put the vermet full of "trash". (Let's say, in a cycle, forget to delete something unnecessary.

```
1 2 3. . .
```

Which number is to be written first? The one that is on top of the stack, the one that was last in the stack. It will also be deleted at the same time; the next item then writes and deletes the item below it. The answer is `3 2 1` . After each step the stack looks like the following figure.

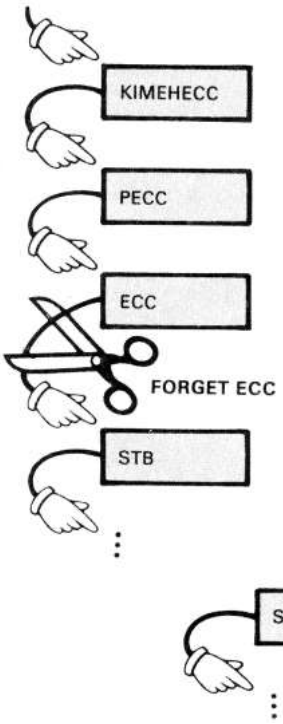


1.3. Back to dictionary

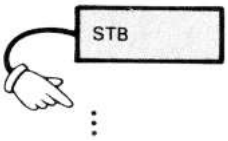
Something else. What if we focus on our definitions, do not we want to use them further? For example, we redefined a word, but we regret it.
The radical solution is the word COLD. It restores the dictionary, the vermet, and some other things that have not been reported so far to the original state of loading. The dictionary will therefore contain the FORTH basic vocabulary.
The delicious solution is FORGET. After the FORGET (still in the same line), enter the name of the word to be forgotten. For example, if we want to recall the second F word:

```
FORGET F
```

FORGET forgets the given word, plus the words that are defined afterwards (that is, words above "in the dictionary").

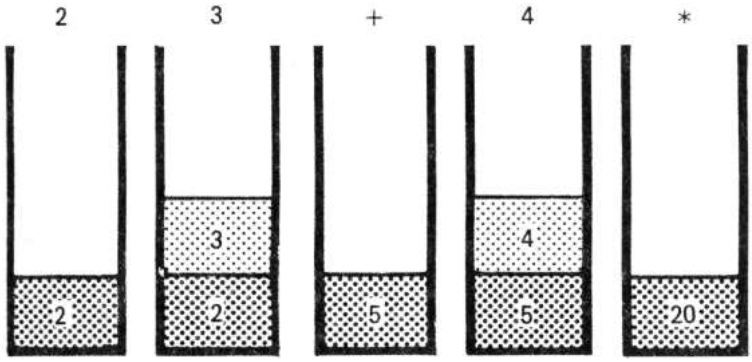


What??? Everything we have defined afterwards?
That's right. In principle, in any of the words written after the forgotten one, we could use this word you just want to delete. The words of the FORTH dictionary are built on one another (it can not be deleted from the middle only). (You can, however, keep the words of our words, how it will be, and just want to calm everyone: you will not have to type everything again because of a mistake!)
Which F word will oblige FORGET to forget if there are two? Almost unobtrusive can be said: from the "top", from the last defined. Finding words in the dictionary, for whatever purpose, is always from the top, in that direction you can quickly look at the vocabulary. In this example, we dig out the old, star F word,



1.4. We learn how to calculate - it's a bit unusual ...

What does FORTH arithmetic consist of? Of course FORTH words. Their names are so short that the more naive they may think of as an action signal. The four basic operations are: +, -, *, /. Each one is waiting for the two numbers at the end of the operation (ie two operands of the operation); they are also lifted from the stack and replaced by the result of the operation. Here is the following. See example.
(After that step, the data in the stack is drawn.) The `2 3 + 4 *` series described performs the same calculation as BASIC `(2 + 3) * 4` (say) .
The latter, more usual markup is called infix, as opposed to the FORTH (multiplying) postfix. The names reflect that the operation signal is in the infix spelling between the two operands, and the postfix in the postfix after the operands.



To become accustomed to postfix, the following can be crippled:

> The order of the operands in the postfix script is the same as in the infix, only the position of the operation signal varies.

infix	postfix
1 + 1	1 1 +
2 - 4	2 4 -
6/3	6 3 /

This means that, for example, in the case of subtraction, the word is awaiting the extraction on top of the stack, and below it the minor. This is commonly documented for FORTH programs:

```
(to be deducted --- difference to be deducted)
```

We are writing a lock signal so that the effect of the individual words on the stack can be indicated in the FORTH source text.
The word "FORTH" means the word "FORTH", its function is "enclosing" the text that is given to the interpreter, so that the interpreting between the opening and closing parentheses is not read by the interpreter, so that it does not execute it. so it can be documented in
FORTH .

```
( before after )
```

If you look at the order of the elements, you just have to imagine how to make the vermet right.
The base effect of the four basic operations:

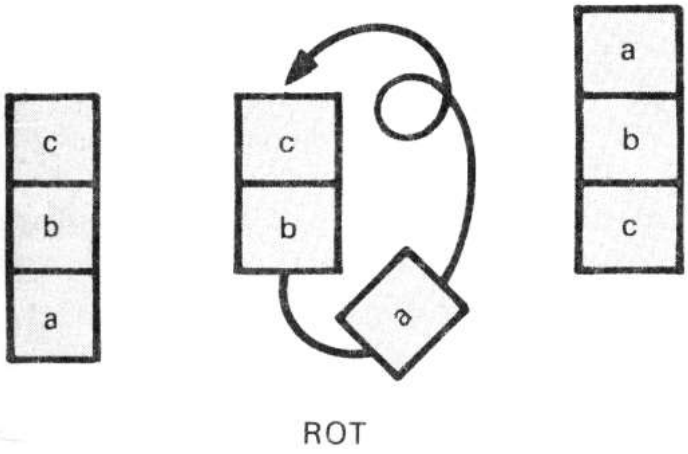
```
+ (sum up to add2 --- amount)
- (to be subtracted to extract --- difference)
* (to multiply1 to multiply2 --- product)
/ (to divide the dividing ratio)
```

This is how we implemented the basic operations. Even so: there are integers in the stack, FORTH arithmetic is a whole arithmetic. Accordingly, the division is also a division (that is, the whole quotient of the quotient).

1.5. Rearrange the stack

The FORTH words are expected to get the bugs in the correct order for the parameters needed for their operation. This is not always easy. At times, the parameters are generated in the wrong order in the stack, including unnecessary, but it may even be necessary for one. The following words are used to solve such problems:

SWAP	(ab-ba)	replaces the two top elements;
DUP	(a --- aa)	doubling the top element;
OVER	(the baby)	make a copy of the second item on top of the stack;
ROT	(abc --- bca)	removing the third element from the bottom and throwing it on the roof;
DROP	(the ---)	removes the top element.



ROT

For example, write a word whose effect on the stack:

(xy --- z); where z = xy- (x + y).

We can not start the thing with an arithmetic operation, since we would lose x and yt on the stack. We have to keep them in some way. Applying this OVER twice is a good catch. In addition to each step, we have indicated what will happen after the step in the stack; this way of writing is very useful until we become a rogue magician of the stack. (Do not be bothered by the fact that the definition is multi-line! FORTH allows this without further notice.)

```
: XY          (xy)
   OVER       (xyx)
   OVER       (xyxy)
   *          (xy product)
   ROT        (x product y)
   ROT        (product xy)
   +          (product amount)
   -          (z)
;
```

1.6. Useful, but non-standard words

There are some pile management FORTH words that are not included in the FIG baseline set **, but** many in FORTH are listed. The source text of the words is 7.3. section if anyone wants to use them.

DEPTH	(--- n)	(meaning: depth) puts the stack of stacks (before the DEPTH is executed) on the stack.
.STACK	(---)	the word STACK can be used to spell the stack. .STACK does not change the vermet.
PICK	(n1 --- n2)	copy the n1th element of the stack to the top of the stack. 2 PICK works the same way as OVER, 1 PICK like DUP.
ROLL	(n ---)	removes the nth element of the stack and puts it on top of the stack. 3 ROLL is ROT, 2 ROLL is equal to the SWAP word. With the standard marking of the stack, the ROLL function can only be written incorrectly.

1.7. One more word of the text of the notice

of. "Writes words on the screen after the specified text until the next mark. The closing quotation mark in ." must be in a row! For example, write a word that contains the two numbers found on the vermine, also print their amount on the screen, in plain text to clarify which number is what. The spin of the word is: (xy ---).

```
: LOCSI-FECSI                    (xy ---)
   OVER OVER                     (xyxy)
   CR
   . "It was up:". CR            (xyx)
   . "This was down:". CR        (xy)
   +                             (sum)
   . " amount: " . CR
;
```

Do not forget that you have to enter a space after ". special word. The space bar does not count in the text to be typed.

What was that about?

Summary of Chapter 1

The interpreter
works on a query-based basis, one line (to ENTER) takes a "question".
The line is interpreted as "word"; line, you will notice a word from the space character while it is over.
He is processing such a "formal" word. that

- he looks at the dictionary and tries to find it in the dictionary words;

- if found, it will execute and find the titles and data found in the dictionary and go to the next word in the line;
- if he did not find it, he would see if he could not interpret the number:
  - if that is the case then the number so obtained is placed in the stack and goes to the next word of the line;
  - if no dictionary word or number can be accepted, it will give an error message, stop reading the line, empty the vermet

About the dictionary:
There are words in it, chained together with "indicators"; the chain begins with the last word defined and the FORTH basic dictionary is drawn to the end.
There may be a name several times; by referencing the name, we call the "topmost", most recently defined word of such names. We can expand it (we have only learned the definition of the colon), but it can only be deleted with the word to be deleted all the definitions that are defined afterwards (no one eye can be collected from the chain, the entire upper end should be disconnected).

From the stack:
There are numbers, from which we always reach the one that was last reached.

There are two words that work with text: both (and. "
Both hold the text behind it to a
delimiter . The delimiter must be in a row with the word, and one
tends to forget about the space behind them, but do not.

The words learned:

:	(---)	Start a new word definition.
;	(---)	The double-point word definition is over.
FORGET	(---)	So we use: FORGET xxx where xxx is a word dictionary. This word will be deleted from the dictionary with the definitions that follow. FORTH basic word can not be deleted.
VLIST	(---)	Lists the dictionary words on the screen. You can interrupt the list by pressing any key.
(	(---)	Locks the string up to the closing parenthesis from the interpreter, and the text between the two brackets has no effect (can be used for documentation purposes). The opening and closing brackets must be in a row.
EMIT	(c ---)	Prints the character corresponding to the code found in the vermin on the screen.
SPACE	(---)	Writes a space on the screen.
CR	(---)	A carriage return and a line up character on the screen.
. "	(---)	Displays the following text on the screen for the closing "The word and the closing" should be in a row.
.	(n ---)	Prints the number at the top of the stack and a space on the screen. He takes the number from the stake.
+	(n1n2 --- n3)	It gives the sum of the two upper elements to the worm.
–	(n1n2 --- n3)	The n1-n2 gives a difference.
*	(n1n2 --- n3)	It gives the product of the two top elements.
/	(n1n2 --- n3)	Returns the n1 / n2 quotient.
DUP	(n --- nn)	Doubles the top element of the stack.
SWAP	(n1 n2 --- n2 n1)	Replaces the two tops in the stack.
DROP	(n ---)	Removes the top of the stack from the stack.
OVER	(n1 n2 --- n1 n2 n1)	Make a copy of the second item at the top of the stack.
ROT	(n1 n2 n3 --- n2 n3 n1)	He removes the third element of the stack from the bottom and throws it to the top.
COLD	(---)	"Cold Start". The dictionary, the vermet, and many more things will be restored to the original state after loading.

Words that have not been mentioned but are readily apparent to date:

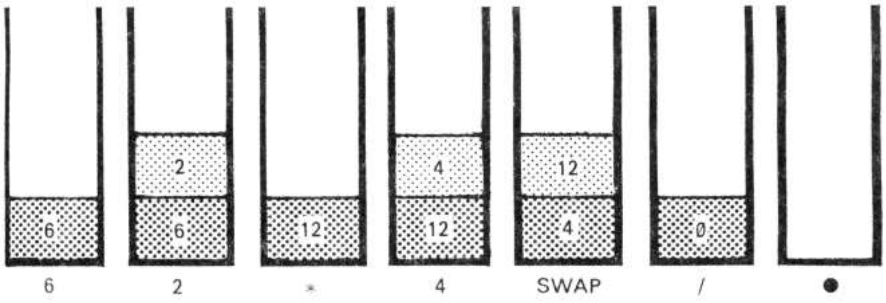
MIN	(n1 n2 --- min)	It gives the smaller of the two elements.
MAX	(n1 n2 --- max)	It gives the bigger of the two elements.
MODE	(n1 n2 --- m)	Returns the remainder of n1 / n2 division.
/MODE	(n1 n2 --- mh)	The remainder of the n1 / n2 division is also obtained.
ABS	(n --- n1)	Returns the absolute value of n.
MINUS	(n --- n1)	The result is -1 times.

Examples

1.1 What does the interpreter answer to the following lines?

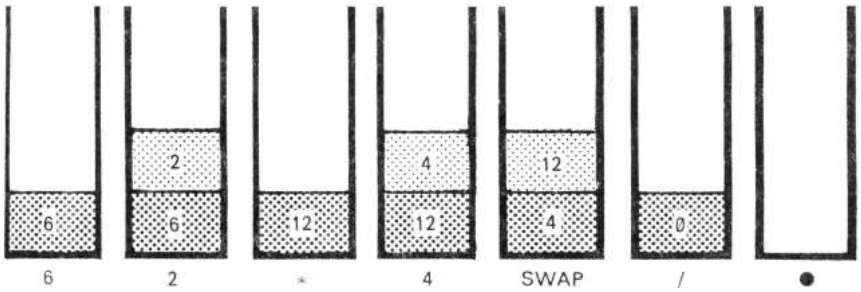
```
6 2 * 4 /.
```

Number displayed: 3



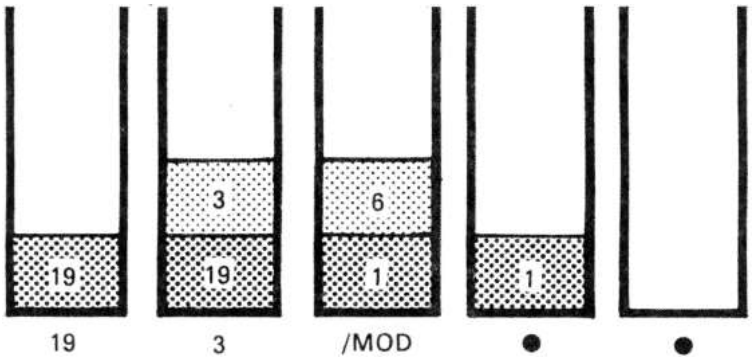

```
6 2 * 4 SWAP /.
```

The displayed number is 0




```
19 3 / MOD. .
```

Numbers appearing: 6 1



1.2. What is the stack for the following words?

```
: ALUL-DUP OVER SWAP;

    : ALUL-DUP        (xy)
       OVER           (xyx)
       SWAP           (xxy)
    ;

: DUPLA-DUP OVER OVER;

    : DUPLA-DUP       (xy)
       OVER           (xyx)
       OVER           (xy xy)
    ;

: 3CSERE ROT ROT SWAP;

    : 3CSERE          (xyz)
       ROT            (yzx)
       ROT            (zxy)
       SWAP           (zyx)
    ;
```

1.3 / a. Let us write a word having a vertex of y = 5x ^ 2 + 6x + 2

```
: A DUP DUP * 5 * SWAP 6 * + 2 +;
```

1.3 / b. Let us write a word with a vertex of y = 6x- (x ^ 2-1)

```
: B DUP DUP * 1 - SWAP 6 * SWAP -;
```

1.4. Write a word ** 5 that elevates the top of the stack to the fifth power. Thus, its curve effect is: (x --- x ^ 5)

an auxiliary word that raises the upper element of the stack to a square:

```
: ** 2 DUP *;
```

using this:

```
: ** 5 DUP ** 2 ** 2 *;
```

2. Comparative and logical operations

How do we compare two numbers in FORTH? Of course, the first one is placed on the stack (so since the first operands are given, the comparison mode is also a postfix). Then we call up a comparative operation. These are: <,>, =. It is important to keep it in mind

the order of the operands in the postfix script is the same as in the infix, only the position of the operation signal varies.

THE

```
2 3 <
```

For example, a result of an action will be that <a true value is placed on the stack.

2.1. The flag

The flag in English is flag to indicate something true or false. To illustrate these two options in general, such as FORTH, we use numbers. FORTH-in:

According to the agreement, the flag is false if it is 0, and true if anything else.

Comparative operations provide "well-fed" flags with a value of 0 or 1.
Write a word that tells what the user in the keyboard is thinking about. Reporting is done with a marker on the stack. In the spirit of the user, we have the following question: YES OR NO?
Now, wait until you press any key. The vermin is given a true value when the user pressed the large I letter. To do this, we need to learn the word that waits for one of the keys to be pressed on the keypad and puts the key code on the stack. This is the word a

```
KEY (--- code)
```

(The English word KEY means several things, probably the "key" translation is most likely.) After KEY everything stops until you press a key. On the screen, we do not see what we're typing (does not write it back than usual) except that the interpreter sends OK. The character code is in the stack - you can type the character with EMIT. with your code.
It's a little more comfortable to see you. what he writes. Here is a program that, like KEY, will press a key and put the correct code on the stack, and even type the character on the screen:

```
: ECHO (--- code)
   KEY DUP EMIT;
```

After that, the yes-no program (taking into account that code I is 73) is as follows:

```
: IVN              (the marker is
   "Yes or No?"    empty)
   ECHO            (the character on the stack)
   73 =            (then the desired indicator)
;
```

2.2. Data types seen so far

Two known words:

.	(number ---)	prints the number found on the screen on the screen;
EMIT	(code ---)	prints the character corresponding to the character code of the worm on the screen.

Both use one element from the stack. An element of the stack is a 16-bit machine word. (Machine word: 16-digit, 2-digit - that is, binary number, otherwise a 16-element series with 0 and 1 values). it assumes a 16 bit preset number (we will see how to work with longer numbers) and EMIT a character code that would otherwise fit 1 byte (8 bits). EMIT simply ignores one of the bytes of a machine word with 2 bytes!
For example, the bug is 42 (binary, since the stack has only machine numbers). How do you know which "which" is 42: a signed number, the * character code, or - we already know this is possible - is a "true" flag?

In FORTH, the type of data depends only on what kind of action is performed on them.

Thus, 42 is a character code if EMIT is used and a signed number if a . .
For example, + considers the top two elements of the stack as a signed number. If anyone still forgets the character code that has been given to KEY, it's the one to take.
When describing the spin of the word, the letters indicating the elements also indicate the type of elements. The types seen so far:

- c character character,
- n 16 bit, number,
- f flag.

Thus we document the comparative operations:

<	(n1 n2 --- f)	the flag is true if n1 <n2;
>	(n1 n2 --- f)	the signal is true if n1> n2;
=	(n1 n2 --- f)	the signal is true if n1 = n2;

2.3. Why should a marker be "well-educated"?

Write a word that tells you that the number found on the stack is between 0 and 9. The name of the word should be 1 TICKET and its stack is: (n --- f).
We can now examine whether a number is smaller than 10 (it is a whole number, it is the same as asking for a "no bigger than 9") and whether it is larger than -1 . From the two indicators, logical AND operation to see if the two responses are true at one time. Logical AND produces a third of the two logical values: if the two values □□were true then the result of the operation is true, otherwise it is false. The AND operation between the flags can be implemented with the AND FORTH key word. (AND in Hungarian: AND.)
Warning: AND performs the logical "and" operation with each bits of the binary form of the two operands! If, for example, the stack was 2 and 1, that is binary

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

and

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1,

then the logical AND result

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0,

that is, 0 will be, since one of the bits of each other is always 0. Whatever is the 2, the 1 is also considered a true value, so AND should have given a true value according to our logic. We need to know about this discomfort (which is another comfort) but at this moment it is unnecessary to worry about it; comparative actions are "well-behaved" with 0 or 1 flags that can not cause the above half-turn.

```
: 1JEGY                 (n-f)
   DUP -1>              (n f1)
   SWAP 10 <            (f1 f2)
   AND
;
```

Another important operation is the logical OR, which also gives a third of the two logical values. The result will be true if at least one of the two logical values □□is true. So then and then we get false only if both operaridus were fake. Obviously this OR does not correspond to the Hungarian word OR word. We say in Hungarian:

"Or flame at the old, wild county house,
or here our souls are sitting, submerged"

and that is to say that the two options exclude each other. We call the OR or EO to be distinguished from a negative OR compared to Hungarian OR. The negative OR gives a true result if one of the logical values □□obtained is true and the other is not. OR, we usually call the permissive OR if we think of the negative OR, let's say its name. Accordingly, the two words FORTH are OR (OR) and XOR (eXclusive OR, Exclude OR). These also work on bit as the AND, but it does not make any difference to the "well-raised" signals from the comparative operations.
Let's look at the counter NEM-1JEGY (n --- f) counterparty 1, which gives a real signal if the number obtained does not fall between 0 and 9 (i.e., less than 0 or greater than 9):

```
: NEM-1JEGY
   DUP 0 <
   SWAP 9>
   OR
;
```

Of course, the NEM-1JEGY, which runs counter to 1JEGY, is easier to write than using 1JEGY. An action must be added (negation, complement) that changes the meaning of the signal on the vermouth: it makes 0 from the true signal and 1 from the false, ie 0 value. This word is not included in the standard FIG-FORTH 1.1. but there are many FORTHs, and it's not hard to write:

```
: NOT 0 =;
```

So the

```
: NO-1JEGY 1JEGY NOT;
```

will work the same as the other NEM-1EGY defined above.

2.4. Quick Steps

Most computers have the machine operating instructions quickly, something to increase by 1 or multiply or distribute, examine the sign of Reduce, 2. In comparison, the series that 1 + (put on the stack 1, call + word) is slow and cumbersome. The so-called. "quick operations" cut off the unnecessary bends and, without any difficulty, start the right machine instructions. Quick Actions:

1+	(n --- n1)	one increases its n value;
1-	(n --- n1)	one decreases its n value;
2+	(n --- n1)	n doubled;
2 /	(n --- n1)	n;
0 =	(n --- f)	f is true if n = 0;
0 <	(n --- f)	f is true if n <0.

Obviously, the words that execute quick operations look the same as the same step-by-step commands, only one word for the operand with the operative sign; the word 1 + does the same thing as the 1 + series, only faster.
A faster version of NOT:

```
: NOT 0 =;
```

2.5. How do we store our programs?

In our attempts so far, it is annoying that the texts of the programs do not remain, can not be corrected or used again.
We can make the programs not directly handed over to the interpreter, but to some media, so-called. we write them in screens; can be repaired or read at any time with the interpreter. The screen in the screen means a screen in the Hungarian, which we will call the abbreviated shade.
The umbrella is the unit of storing text information. In a window, there are as many texts as you can handle at the same time; this is 16 rows, in a row with 64 characters (that is, 1 sq. can store exactly 1 kbyte information). All FORTHs are stored in their own way. The standard FORTH looks at a disk simply as a set of sectors that it allocates itself to the clouds (mainly based on older implementations that presume a small disk capacity). The fig-Forth so-called. it uses file folders to allow you to hold other files on the same disk (even multiple file files).
When we started the FORTH interpreter so far, the "No file" message warned that we did not select a file, so the interpreter can only be used in "question-answer" mode. If you want to load a file, you can do this when starting Forth by specifying the name of the popup file as a parameter. Ex .:

```
Z80FORTH SCREENS.FRT
```

Once you have loaded the desired file, you can start editing our program, which will of course require some kind of word processor. The fact that the fig-FORTH interpreter does not include a word editor (editors) is cited by computer science. The editor of William F. Ragsdale, attached to fig-FORTH, is also writing in FORTH, which must be completed in the same way as any other program we wish to run. Do not even dream about fullscreen editing; the command line editor needs to be used in the beginning; later it becomes quite useful. Its use is described in detail in the appendix .
If we have constructed an umbrella, then a

```
LOAD (n ---)
```

so we can pass it on to the interpreter. The worm must enter the number of the blind. As a result of LOAD, it is exactly the same as typing the text on a given number of umbrellas. If, as is usually the case, there are definite definitions, the words defined in the dictionary appear in the dictionary by loading, that is, LOAD.
If we want to load more successive umbrellas (for example, because our program does not fit into a window), then

```
->
```

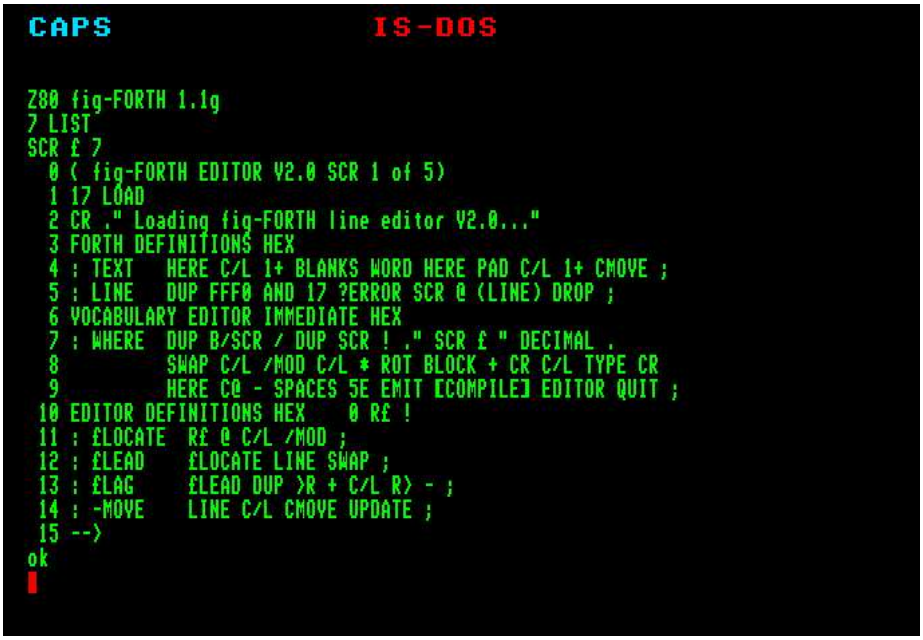
write the word to the end of the spectators. When loaded, this interpreting is stopped and the next begins to load. THE

```
S
```

the interpreting of the interpreter is interrupted and the interpreter continues where the LOAD was.
The interpreter almost does not have control over the keypad or receiver at that particular moment. The course stream interpreted by the interpreter, English-language literature as input stream, translates this into an influence.
The text of an umbrella on a

```
LIST (n ---)
```

so we can add it to the screen.



By agreement, the top row of each screen contains some reference to the content of the screen. This is used by

`INDEX` (from to ---)

which is not standard, but contains many FORTH basics (such as fig-FORTH) and can be written by anyone after reading Chapter 13 (see task 13/2). INDEX lists the top row of the number of the number of the two numbers that you entered, giving you a table of contents about our umbrellas.
In the first place we have to mention that a

`FLUSH` (---)

so you can write the modified blocks to disk.

What was that about?

Summary of Chapter 2

About comparative operations:
The worm is waiting for their operands; their order is the usual, only the location of the <,>, = "operation marks" (postfix markup mode) changes.
They make signs on the stack.

About Flags:
True or False: 0 is false, the other is true. They may be "well-educated" - this means that the true flag is always worth 1.
Comparative operations provide such.

About Types:
FORTH does not know what kind of data the stack is - this is the programmer's business.
The types and their markings so far:

- character code: c (character),
- signed number: n (number),
- flag: f (flag),

they all occupy one element in the verme (that is, a 16-bit word).

About logic operations:
AND is true if both operands are true.
OR is false if both operands are false.
Exclude OR is true if one of the two operands is true and the other is false.
Negation: It's a false one and vice versa.

And their FORTH implementation:
AND (AND) OR (OR) and XOR (negative OR) performs the corresponding logical operation bitwise. so if they are not applied to "well-raised" markers, they may lead you astray.
Negation can be accomplished by the word 0 =.

About Quick
Actions ; They are not a new operation, only a more time-consuming and more conservative version of some of the special cases of the old.

About the viewers:
Store text on a disc or tape.
You can load LOADs at any time, so the same happens as typing the text on the screen. Editing programs are used to write and maintain the umbrellas.

The words learned:

<	(n1 n2 --- f)	f is true if n1 <n2.

>	(n1 n2 --- f)	f is true if n1> n2.
=	(n1 n2 --- f)	f is true if n1 = n2.
AND	(n1 n2 --- f)	Bitwise logical AND.
GUARD	(n1 n2 --- f)	Bitwise Logical OR.
XOR	(n1 n2 --- f)	Bitwise logical exclusion OR.
1+	(n --- n1)	n increases by 1.
1-	(n --- n1)	n by 1.
2 *	(n --- n1)	n by 2.
2 /	(n --- n1)	n is divided by 2.
0 =	(n --- n1)	f is true if n = 0.
0 <	(n --- n1)	f is true if n <0.
KEY	(--- c)	Expect to press a key on the keypad and enter a keypad corresponding to the key.
LOAD	(n ---)	Load, "interpret" the specified number of umbrellas.
->	(---)	It comes from the interpretation of the given silhouette to the next.
LIST	(n ---)	List a screen on the screen.
S	(---)	Completing the interpretation of the shadow.

Examples

2.1. Write a 3: 0? (n --- f), which gives a true sign if n is divisible by 3.

```
: 3: 0?   (n --- f)
   3 MOD  (in the stack the remainder of the division)
   0 =    (then "true" is the answer if it is 0)
;
```

2.2. We have a box with a length of 100 cm, a width of 65 cm, a height of 10 cm. Write a BELE? , which tells the worm that the length, width, and height of the given length fit into the box? The spin of the word is: (width width height --- f). f is true if the length is <100, width <65 and height <10.

```
: BELE?
   10 <
   ROT
   100 <
   AND
   SWAP 65
   AND
;
```

2.3. Write a word 7-E (n --- f) that gives true to the vermin if the last digit 7 in the decimal number of n is the number 7!

```
: 7-E (n --- f)
   ABS (absolute value if negative)
   10 MOD (last digit)
   7 =
;
```

2.4. Write the words L-AND and L-OR (f1 f2 --- f), which do not perform the corresponding logical operations bitwise, but between the two flags. So, for example, 1 2 L-AND do not enter 0, but 1 (for two true tokens) 1. Both words are "well-fed".

A word by which we can make a marker "well-educated":

```
: 1EL 0 = 0 =;
```

For both words, the top 2 elements of the stack should be "well-raised"

```
: L-AND 1EL SWAP 1EL AND;
: L-OR 1EL SWAP 1EL OR;
```

3. Conditional Instructions

We already know how to get an indication of a true or false condition of a condition. Now we learn how to use the flags.

3.1. The IF ... ENDIF

Function of the IF ... ENDIF structure: the IF will emit the marker at the top of the stack. If the flag is true, the IF and ENDIF part will be executed if not, not. Before we give an example, let's quickly sketch it:

> All structures that contain control transmission (ie conditional and cycle-forming commands) can only be used in definition.

Write a word that finds a number between 0 and 9 on the worm, and prints it to the screen: COPY. Of course, we use the word 1JEGY (n --- f) as defined in the previous chapter . The font of the new word: (n ---)

```
: ONE SIZE IF "one-note" ENDIF CR;
```

The synonym for ENDIF is THEN. Other high-level programming languages ☐☐make the THEN completely different; before we let ourselves be mixed up, it's best to remember: this is different, THEN here is ENDIF!
IF and ENDIF can not be used without each other.

3.2. ELSE

If we do not only have to fulfill a given condition, but also in the opposite case, how can we build our program:

IF (here is what to do if the flag is true);
ELSE (here is what to do if not)
ENDIF

For example:

```
: ONE
   ONE "IF" ONE digit "
   ELSE" is not a single "
   ENDIF CR
;
```

Sometimes in the ELSE branch there is no other task than removing a duplicate copy of the marker. If you do not need to write an ELSE branch for that, there is a FORTH basic word that only doubles the upper element of the stack if it is not 0.

-DUP (n --- nn, if n <> 0) (n --- nm if n = 0)

3.3. Nested

be higher, depending on the condition of work to do IF or ELSE branch.
For example, write an EVES-OR (n ---) program that is

- n <10 for CHILDREN,
- 10 <= n for <20> KAMASZ,
- 20 <= n FELNOTT

answers. For example, the result of 3 EVES VOICES is CHILDREN.

```
: EVES I AM                (N ---)
   DUP                     (nn)
   10 <                    (n f 1)
   IF                      (where n <10)
      "child"              (work done)
      DROP                 (eyes but remains in the stack)
   ELSE                    (n => 10)
      20 <                 (f2)
      IF "" adolescent "   (if less than 20)
      ELSE" adult "        (where no )
      ENDIF
   ENDIF
   CR
;
```

Note that the "internal" IF structure is always entirely on the "outer" side. This is the key to which IF, ELSE and ENDIF belong in a FORTH text. If we see a program item like this:

```
IF A IF B ELSE C IF D ENDIF ENDIF ELSE AND ENDIF
```

then in order to get it right, look for the innermost IF and we already know that the ENDIFs will soon be his. (Similar to the second innermost IF.) Apparently, the second IF has an ELSE branch:

and the whole second IF is on the very branch of the first one. It is best to write such a series in a bit more detail; something more obvious is the same as the following:

```
IF
   A
   IF B
   ELSE
      C
      IF D
      ENDIF
   ENDIF
ELSE
AND ENDIF
```

that is, if the IF, ELSE and ENDIF joins each other, they will be executed on the given branch a little bit further.

What was that about?

Summary of Chapter 3

On the IF ... ELSE ... ENDIF structure: Use
only in definition.
IF is the only one of the three words that will change the vermen (use a marker).
ELSE may be omitted.
If the IF is a true marker on the vermin, IF and ELSE (if there is no ELSE, IF and ENDIF) are executed, if false, then between ELSE and ENDIF (or, in the absence of ELSE). Execution in both cases continues after ENDIF.
ENDIF is synonymous with THEN.
These structures can be embedded in any depth.

Even from a stacker word about -DUP:
If the worm is 0, it does not do anything, otherwise it is the same as the DUP.

Examples

3.1. Write a word SN (n1 --- n2) that gives -1 on the vermin if n1 is negative, + 1 if positive and zero if null.

```
: SN         (n1 n2)
  DUP        (n1 n1)
  IF         (if n1 <> 0)
  DUP ABS /  (dividing the number with its own absolute value)
  ENDIF      (if n1 was 0, then 0 is in the worm)
;
```

3.2. Write a SZOVEGEL (n ---) word that specifies any of the following messages, depending on the value of n: NULL, ONE, MINUS, KETTO, MUST, ONE.

```
: Text
   DUP
   IF
      DUP
      ABS 2>
      IF "OTHER" DROP.
      ELSE
         DUP
         0 <IF "minus" ENDIF.
         ABS
         1 = IF "a."
         ELSE "two".
         ENDIF
      ENDIF
   ELSE "zero" DROP.
   ENDIF
   SPACE
;
```

3.3. Write an ALPHA (---) word that waits for a character from the keyboard and

- if he has a digit, he writes: SZAM,
- if letter: BETU,
  if nothing else, it does not write anything.

The numbers in the numbers are 48 and 57 in capital letters between 65 and 90.

We have to examine twice whether our character is between something and something else. A serious FORTH programmer does not write anything twice, rather write a word.

```
: ELEMENT
   ROT SWAP OVER
```

```
        <ROT ROT
        >
        OR 0 =
    ;
```

So it's easy (we also use the word ECHO that was previously made):

```
    : ALPHA
        ECHO CR
        DUP 48 57 ELEMENTS
        IF "DROP
        ELSE 65 90 ELEMENTS
            IF" grandma "
            ENDIF
        ENDIF CR
    ;
```

3.4. Let's say, which decides from the year that the leap year is? Leap years are: all four are divisible by year, with the exception of one hundred. Leap years, however, are 400 years old. That is, from the turn of the century, only leap years, which can be divided by 400. For the simplest solution, we use the word EXIT as described in 5.1. We'll get to know this chapter .

```
: LEAP-YEAR? (Ev --- f) (leap year?)
    DUP 400 MOD = 0 EXIT ENDIF IF 1 DROP
    DUP 100 MOD = 0 EXIT ENDIF IF DROP 0
          4 MOD = 0
;
```

Let's try to interpret the next, more concise solution:

```
: LEAP-YEAR? (ev --- f) (SPEED)
    DUP 4 MOD 0 =
    OVER 16 MOD 0 =
    ROT 25 MOD 0 =
    NOT OR AND
;
```

3.5. Using the lessons learned so far, we can now calculate with Christian Zeller's algorithm that the given date is the seven-day day.

```
: WEEKDAY (day
    1 to 2 ) (1 week, 2 days, ..., 7 days)
    OVER 3 <IF
        1 SWAP 12 + SWAP
    ENDIF
    100 / MOD
    DUP 4 / SWAP 2 * -
    SWAP DUP 4 / + +
    SWAP 1+ 13 5 * / + +
    2- 7 MOD 1+
;
```

Using the word WEEKDAY, you can say the day of the week on either day:

```
24 12 2000 WEEKDAY.
```

4. Indexed Cycles

4.1. The return stack

The FORTH interpreter uses two holes. One is already known: this is a computation stack or datum, which we mean when we are talking about a stack. The other one is mainly used by the interpreter itself, most often to note it: to run a word (jump to the appropriate address, execute the code found there) after returning. It is therefore called a return stack, briefly called virem.
Our vibration programs can be used to temporarily store stack items when you keep in mind

the elements of the vire for each word (which does not deliberately and slyly use the virus to modify the control) should be left in the same way as found; the state of the vibe may change only within one word.

The word handling words (here as well, like everywhere, we document the stack of stakes for the calculation stack):

> R	(n ---)	moves the top of the stack to the vire.
R>	(--- n)	the top element of the wine is moved to the stack.
R	(--- n)	Copies the top element of the wine into the stack; the vial remains unchanged.

A task that's good for you: Write the largest of the top 4 of the stack on the screen! The stack is ultimately unchanged.

```
: .MAX              (n1 n2 n3 n4)
   DUP> R           (a viremen: n4)
   OVER MAX         (m1 m2 m3 max3,4)
   SWAP> R          (a viremen: n4 n3)
   > OVER R>        (n1 n2 n1 max3,4)
   MAX              (n1 n2 max1,3,4 )
   OVER MAX         (n1 n2 max)
   . R> R>          (n1 n2 n3 n4)
;
```

4.2. Staging One by

One DO ... LOOP is another structure that can only be used in word definition. Cycle organization; to repeat the program part (cycle core) between DO and LOOP.
For example, write 10 times: DO NOT ENABLE.

```
: HAZI-FEL 10 0 DO CR. "No time to waste" LOOP;
```

The DO ... LOOP so called. indexed cycle. This means there is a cycle index somewhere - a cindex or cycle counter that counts how many times the cycle nucleus is performed. The starting value of the cindex and the end value called the index limit are given to DO. The DO of the DO: (index limit start value ---). DO does these two values ☐☐for virem. During the run of the cycle core, the virgin is at the top of the cindex current value, and below it the cycle limit. So, if you want to use the cindex in the cycle core, you can simply pull it out of the vire:

```
: ABC                                  (
  CR
  . "The ASCII code for the big bet:"
  91 65                                cycle)
  DO CR                                (cycle)
     R                                 (we put the cindex)
                                       (the current value)
     DUP EMIT                          (we write the character)
     SPACE                             (followed by a space)
     .                                 (and then the code itself)
  LOOP CR                              (end of the cycle)
;
```

At the first run of the loop core, the cindex value is the given starting value (in our example, 65, the letter A). The LOOP will always increase the cindex one by one and see if it has not reached the cycle limit. When it is reached, the cycle is completed (it clears the two upper values). so that the cycle core lasts when the cindex is less than the index limit. The last letter of the alphabet is ABC, the code of which is 90.

4.3. With IFs

The ABC code table would be nice to fit the screen if not just a code, but 10, say, in a row. How can you only get CR in the loop core when you have already written 10 items? We will investigate whether the cindex 10 is divided into 5 residues. This will be true for the first time (if the start value is 65) and then for each tenth round.

```
: ABC
   CR. "The ASCII code for the big bet:"
   91 65 DO
      R 10 MOD 5 =
      IF CR ENDIF
      R DUP EMIT SPACE. 2 SPACES
   LOOP
   CR
;
```

We know that some of the FORTH basic words are written in FORTH, using the words "more basic". This is SPACES (n ---), which writes a number of spaces on the screen. SPACES is essentially a SPACE for DO ... LOOP.

```
:       SPACE    (n ---)
   0 MAX         (so that the starting value can not be)
                 (greater than the cycle limit)
   -DUP          (should not be done 0 times?)
   IF            (if n is not 0)
0 DO SPACE LOOP
   ENDIF
;
```

Prior to implementation, it is necessary to examine whether there is zero on the vermin because the DO. It follows from the operation of LOOP that

The DO ... LOOP cycle core will always be executed at least once before the LOOP conducts the first test.

4.4. Cycles inside each other

Write a multiplication table! The board will have 10 rows and 10 columns. For example, in the 3rd place in row 3, write the result of 3 x 4 multiplication.

The . it is not suitable for writing a table because it has multiple long and one-digit numbers for a long time. THE

. R (nm ---)

literally n numbers in a width width box, right-aligned (space for spaces to enter the number of characters you are typing). The elements of our table are maximum 3 digits, so if 4. Write them out with R, each with two spaces (except 100), will not "stick together".

Version 1: The nth row of the table is constructed so that 1, 2,. . . , Multiply 10 numbers by n and write the products:

```
: 1SOR              (--- n)
   CR 11 1 DO       (n)
      R             (n cindex)
      OVER *        (to burn product)
      4 .R          (notice)
   LOOP
   DROP             (leaving no garbage)
;
```

The table itself is enough to repeat 1SOR with numbers 1, ... 10:

```
:
    TABLE 11 1 DO R 1SOR LOOP CR;
```

Version 2: Resolve the same in a word with nested DO ... LOOP cycles! (In the example, the cycle boundary is chatted with the outside with k and the inside with b.)

```
: TABLA
   11 1 DO CR     (A Virma: chat k cindex's)
      11 1 DO
                  (elõássuk cindex-kt the viremrõl)
        R> R> R   (the stack: cindex-b chat b cindex b)
        ROT ROT   (the stack: cindex's cindex-b chat b)
        > R> R    (downgraded to virmet)
        R * 4 .R  (we print the product)
      LOOP
   LOOP CR
;
```

Version 3: The same solution, so that we can take the external cindy in time:

```
: TABLE
   11 1 DO        (here is only the outer cycle)
      CR R        (things are in the virgin)
      11 1 DO     (cinder-binds)
        R         (cindexs-b)
        OVER *    (beginners tend to spoil it)
        4 .R      (to compress them and to do)
      LOOP        (the cycle core would run second )
                  (cindexs are lost)
      DROP        (cindexs do not have to)
   LOOP CR
;
```

In many FORTH, the cindex is I, the outer cindex (the third element of the stack) is J, the even more external cindex (the fifth element of the stack) with K. The stack of all three words (--- n). If I and J are in our FORTH, we can write the TABLA program more easily. In the fig-Forth dictionary, there is only the word I, the pair of the word J is missing, but this is only temporary in the next paragraph of our book so let's look at this solution too!

Version 4: Using the words I and J:

```
: TABLE
   11 1 DO
      CR 11 1 DO
         IJ *
         4 .R
      LOOP
   LOOP CR
;
```

4.5. What is easy to ruin

Let's see how word I works, and write the missing word J in the fig-Forth dictionary! With I, we seem to have a simple task, as doing nothing more than the R word: copy the top of the vire into the stack. And yet, the obvious

```
: IR;
```

definition is not good for this. When the word I enters the interpreter, it puts the address to which I will return. In addition to the definition of the former I, we would get this title on the vermin instead of the cindex. A good solution puts the second element of wine on the stack:

```
: I R> R SWAP> R;
```

Likewise, in the J word, we have to move the third rather than the third element of the vi:

```
: J             (
   R> R> R> R   stack: vc n2 n1 n)    (Viread n)
   SWAP> R      (stack: vc n2 n)      (Viread n n1)
   SWAP> R      (stack: vc n)         (Viread: n1 n n2)
   SWAP> R      (stack: n)            (Viread n n1 n2 vc)
;
```

Anyone tempted not to write the SWAP R> series three times, but to write a word or a cycle, but try - just be careful not to call the new word or the start of the cycle any longer over the viremes!

4.6. Get out of the cycle

of a DO ... LOOP cycle can be interrupted at any time so that cindexet and limit cycles in the Virma "összeigazítjuk". That's what LEAVE does. With LEAVE, the nearest LOOP will find that the cycle must be completed.
For example, write a word BETU (--- n) that waits for characters up to ENTER (ENTER code 13), but no more than 20 characters. BETU returns the number of characters received, except for spaces. The characters are compressed and compared in a DO ... LOOP cycle. During the cycle, there will be a count on the vermin, giving 1 for each "real" character.

```
Alphabet            (if
   0                this is the counter)
   20 0 DO
      KEY EMIT DUP  (counter)
      DUP 13 =
      IF LEAVE      (if it was ENTER, the nearest one)
         DROP       (LOOP)
      ELSE
         32 -       (was a space?)
         IF 1+      (if not, increase the counter)
         ENDIF
      ENDIF
   LOOP
;
```

4.7. A Different Walkthrough

Agatha Christie's short abstract of a novel:

10 small indians
9 small indians
small indians
7 small indians
6 small indians
5 small indians
4 small indians
3 small indians
2 small indians
1 small indians
0 small indians

How can this be printed on the screen?

```
: MONTH
   11 0 DO
      10 R -
      CR 4. R 2 SPACES "small indian"
   LOOP CR
;
```

The same goes even easier if the cindex is not moved by 1, but by -1. This is the DO ... + LOOP structure. The + LOOP from the LOOP differs from that

- the cindex is not increased by 1, but by the number found in the vermin; the stack of + LOOP: (n ---);
- does not finish the cycle when the cindex is equal to the cycle limit, but exceeds it, ie (if n> 0) is greater, or (if n <0) is smaller.

The above program can be written using DO ... + LOOP as follows:

```
: MONTH
   CR 0 10 DO
      R
      CR 4. R 2 SPACES "small indian"
   -1 + LOOP
   CR
;
```

With the word + LOOP, you can of course walk with any step.

```
: MESE2
   CR 42 2 DO
       R DUP
       CR 3 .R SPACE "man"
       2/3 .R SPACE. "Par"
   2 + LOOP
   CR
;
```

What was that about?

Summary of Chapter 4

From the return stack, that is, from the vibe:
This will return the interpreter if he skips a word to be executed.
So when a word is over, the vire must be ok.
Viruses are considered to be DO ... LOOP and DO ... + LOOP cycles of the cindex and the cycle boundary.
When handling vira, you should also pay attention to adding a word to a new item.

DO ... LOOP, DO ... + LOOP cycles:
Both indexed cycles: a cyclic counter (cycle index, cindex) monitors how many cycles have run down.
Cindex is a virgin, it can be accessed at any time.
The DO gives the start value of the cycle limit (cindex end value) and the cindex initial value for the worm.
The LOOP increases the cindex one by one (leaving the vermet in peace), + adds LOOP an as the worm is given.
You can get out of the loops with the LEAVE word out of the way; LEAVE aligns the cindex and the cycle boundary with the viruses so that the nearest LOOP or LOOP will "feel" that the cycle is over.

DO ... LOOP, DO ... + LOOP, IF ... ELSE ... ENDIF structures:
Optionally, I can nested.
They can only be used in word definition.
The same is true of the other structures we are still learning.

The words learned:

> R	(n ---)	The top element of the stack is placed on the vire.
R>	(--- n)	It puts the top element of the wine on the stack.
R	(--- n)	The top element of the wine is copied to the stack, the vial remains unchanged.
DO	(n1 n2 ---)	The beginning of the index cycle. n1 is the cycle limit, n2 is the starting value. Use with LOOP or + LOOP.
LOOP	(---)	End of the cycle of the index cycle. Increases the cindex by one and verifies whether you have reached the cycle limit. If not, he will go back to DO. If so, he gets out of the cycle.
LOOP +	(n ---)	End of the cycle of the index cycle. For cindex, it gives n and looks for greater (if n> 0) or less (if n <0) than the cycle limit. If not, he will go back to DO. If so, he gets out of the cycle.
I	(--- n)	DO ... is used for loop cycles. Copy cindex to stack.
LEAVE	(---)	DO cycle. Corrects the cindex and the cycle boundary to the viruses; this closest LOOP desire + LOOP gets out of the cycle.
SPACES	(n ---)	n writes space on the screen.
.R	(nm ---)	n is written in a m width field right aligned.

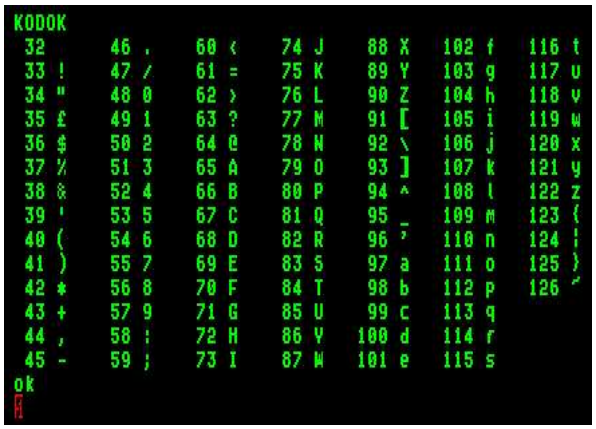
Examples

4.1. Write a word TEGLA (nm ---), which writes m. Each of them must be n stars!

```
: TABLE            (nm ---)
    0 DO           (m line is written)
      DUP          (n will be needed for every line)
      CR
      0 TO         (a n is to be executed)
         42 EMIT   (the star is written in a cycle)
      LOOP
   LOOP
```

```
      DROP CR
   ;
```

4.2. Write a "normal" ASCII code table:



The elements are written in 7 columns, the sequential elements are under one another. The first code to be displayed is 32, the last is 126.

```
   : KODOK
      14 0 DO           (Will be 14 lines)
         CR R           (TARUN Which row)
         7 0 DO         (item 7 in a row)
          32            (initial element of the table)
          OVER +        (the first element of the row)
          R 14 * +      (actual element of the row)
          DUP 127 <IF   (greater than 126)
            DUP 3 .R    (codes not described in)
            SPACE EMIT
            3 SPACE
          ELSE DROP
          ENDIF
         LOOP
         DROP
      LOOP CR
   ;
```

4.3. Factorial computation: n! = 1 * 2 * 3 *. * n. Write a factorial count F (n --- n!). If a number smaller than 1 is received in the worm, 0 is returned.

```
   : F              (n --- n!)
      DUP 0> IF
         1          (this will be the series)
         SWAP 1+    (vermen 1 and n + 1)
         1 DO       (the product is the product)
            R *
         LOOP
      ELSE DROP 0
      ENDIF
   ;
```

4.4. The so-called. Elements of a Fibonacci Line: 1, 2, 3, 5, 8, ...,
From the third element, each element is the sum of the previous two. Write a word FIB (n1 --- n2) that puts the n1th element of the Fibonacci line on the stack! It is assumed that the number of vermin is not less than 1. Try FIB to get the first 16 elements of the line.

```
   : FIB (n1 --- n2)
      DUP 3 <IF (if n1 = 1 or 2 then)
              (the desired result is equal to n1)
      ELSE
         1 2 ROT 2 DO
            SWAP OVER +
         LOOP
         SWAP DROP
      ENDIF
   ;

   : FIBTEST CR 16 0 DO R FIB 5 .R LOOP CR;
```

4.5. Write a PRIM? (n --- f), which gives a true value if n is the prime number, ie, dc and itself no divisor. +1, -1 is not a prime. Write the prime numbers between 1 and 2000.

```
: PRIM?                 (n-f)
   ABS                  (we do not deal separately)
   DUP 2> IF            (with negative numbers)
      1
      OVER 2 / 2+ 2 DO
         OVER R
         MOD 0 =        (if R divisor, rebound)
         IF 0 = LEFT    (the flag)
         ENDIF
      LOOP
      SWAP DROP
   ELSE 2 =             (unexamined cases)
   ENDIF                (only 2 prim)
;

: PRIMTEST
   CR 2000 1 DO
      I PRIM? IF
         I 5 .R
      ENDIF
   LOOP CR
;
```

4.7. Display the first 13 lines of the Pacal triangle. (The Pascal triangle in mathematics is the arrangement of binomial coefficients in triangular form.)

```
: PASCTRIANGLE (n ---)
   CR DUP 0
   DO
      1 OVER 1- I - 2 *
      SPACES
      I 1+ 0
      DO
         DUP 4 .R
         JI
         - * I 1+ /
      LOOP
      CR DROP
   LOOP
   DROP
;

13 PASCTRIANGLE
```

5. Indexless cycles

5.1. The endless cycle (BEGIN ... AGAIN)

With BEGIN ... AGAIN you can repeat the **cycle nuclei** BEGIN and AGAIN indefinitely. In such a BEGIN ... AGAIN cycle, the FORTH interpreter itself runs the FORTH language, the source of which we will get acquainted with. (Something like this: BEGIN Read a line, do it! AGAIN.)
The endless cycle can also be ended. Let's look at how the following word works:

```
: EXIT R> DROP;
```

for example when it is used:

```
: BETUK (---)
   BEGIN
     KEY DUP EMIT
     13 = IF CR EXIT ENDIF
   AGAIN
;
```

When the word BETUK starts to execute, the title on the top of the vibe is where the interpreter continues to run after the BETUK has been executed. When you enter EXIT, the address of the return from EXIT (to BETUK) is at the top of it, but you will not sit there for a long time because the act of EXIT is just about to drop you from there. EXIT does not return to BETUK, but the place

where BETUK should be, that is, the word BETUK, so it can force completion of a word.
EXIT is a basic word in many FORTH versions, but not FIG-FORTH 1.1.

5.2. Departure at end of cycle (BEGIN ... UNTIL)

BEGIN ... UNTIL repeats the cycle nucleus between two words; after each run of the cycle core, UNTIL will eat a marker from the stack, deciding whether to go back to BEGIN or go on.

UNTIL (f ---) will continue the cycle if you find a false flag.

For example, write prime numbers smaller than 200. We use the 4.7. task PRIM? (n --- f), which tells us whether the number in the vermin is prime.

```
: PRIMEK
   CR 2                (first prime number)
   BEGIN
      DUP 5 .R         (print)
      BEGIN            (we are looking for further)
         1+ DUP PRIM?
      UNTIL            (if prime, exit)
      DUP 199>         (if it is too big)
   UNTIL               (exit)
   DROP                (throw away trash)
;
```

5.3. Outbound in the middle of the cycle (BEGIN ... WHILE ... REPEAT)

WHILE (f ---) checks the end of the cycle for a stack.

WHILE (f ---) will continue the cycle if you find a true marker.

For the true signal, WHILE will translate the program: the WHILE and REPEAT part of the program will be executed and REPEAT will return to BEGIN (unconditionally). If WHILE finds a false flag, the program continues with REPEAT words. For example:

```
: TURELMES
   BEGIN
      CR. "Spray Spenoth (I or N)"
      KEY DUP EMIT       (the answer to the bug)
      73 -               (code I?)
   WHILE
         (here we get if I did not have a letter I)
      CR. "Incorrect answer, try again!"
         (the control goes back to BEGIN)
   REPEAT
         (here we get if I was a letter)
      CR: "I'm really kidding!" CR
;
```

What was that about?

Summary of Chapter 5

The words learned:

BEGIN	(---)	Selects the start of a cycle. Use BEGIN ... UNTIL, BEGIN ... WHILE ... REPEAT and BEGIN ... AGAIN
AGAIN	(---)	Returns BEGIN without a condition.
UNTIL	(f ---)	If you receive a false flag, you will return to BEGIN, if not, the program will walk away from the cycle.
WHILE	(f ---)	If you receive a true signal, the program goes down to REPEAT, then goes back to BEGIN unconditionally. If not, the program will exit REPEAT from the cycle.
REPEAT	(---)	Returns BEGIN without a condition.
EXIT	(---)	Returning from one word to the word he calls.

Structures (IF, Indexed and Indexed Cycles) can be embedded at any depth. Matching the keywords correctly is controlled by the interpreter and does not translate the word if something is wrong.

Examples

5.1. What's the difference between the TURELMES word in chapter 5 and the next version?

```
: TURELMES
   BEGIN
      CR. "Spray Spanning? (I or N)"
      KEY DUP EMIT
      73 = IF EXIT ENDIF
      CR. "Incorrectly, try again!"
```

```
    AGAIN
       CR. "That's right!" CR
;
```

We also used EXIT as defined in Chapter 5 on this topic . The cycle here remains with the letter I. But EXIT will not only get out of the loop but also from the word itself: the revelatory post after AGAIN will not appear.

5.2. Write a LOG2 (n1 --- n2) word. If n1 is positive, n2 is a number for which 2 ^ n2 <= n1. If not, n2 be 0.

```
: LOG2                (n1 --- n2)
   0 MAX DUP
   IF                 (if a positive number is obtained)
      0> R            (the exponent is generated in
      1               virem ) (the current power will be released ) (vermen
      BEGIN           n1 and power)
        2 *           (next power)
        R> 1+> R      (next exponent)
        OVER OVER <   ("exaggerated" n1- et?)
      UNTIL           (if so, the end of the cycle)
      DROP            (not in the power of need)
      DROP            (n1 not be)
      R>              (this is the first exponent from whom)
      1-              ( "outgrown" the last good)
   ENDIF              (if we did not get a positive number)
;                     (the 0 in the grove is just fine)
```

See also: