Control structures – users viewpoint

Boolean values 0 – false
non-zero – true (-1)

Boolean operations

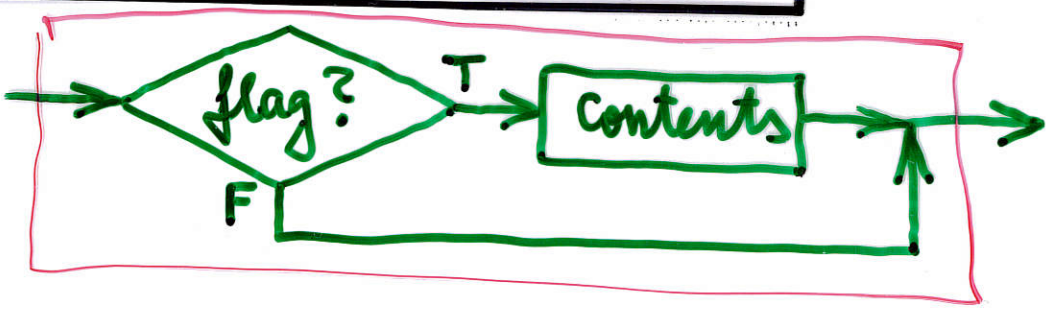
= (a b --- flag)
<
>
... e.t.c.
} binary

0= (a --- flag) unary 0= NOT
...

AND (f1 f2 --- and)
OR (f1 f2 --- or)
...
bitwise!

Control structures are allowed in colon-definitions only!

flag IF contents THEN

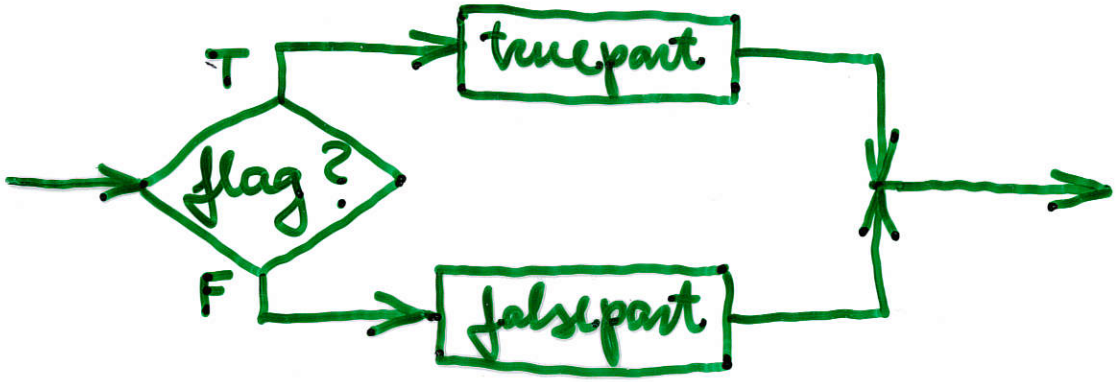



```
: ABS ( a --- |a| )
   DUP 0< IF NEGATE THEN ;

NEGATE ( a --- -a )
```

```
-5 ABS .
 5 ABS .
```

a
a a
a a<0
a

flag IF truepart ELSE falsepart THEN
ENDIF




```
: TEST ( a --- )
  0<  IF ." Negative"
      ELSE ." Non-negative"
      THEN ;
```

a
a<0

```
IF   ( flag --- )
THEN   (---)
ELSE   (---)
```

} stack-effects (run-time)

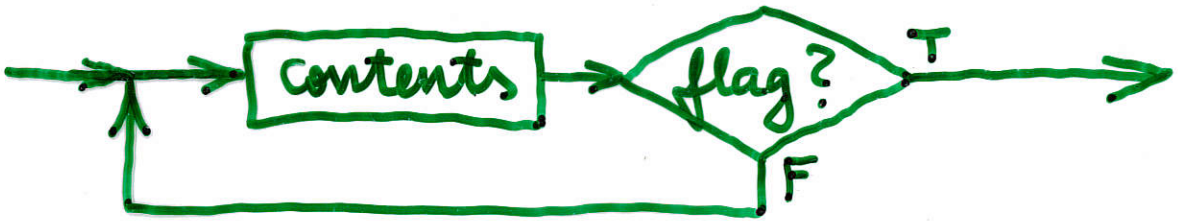
BEGIN contents AGAIN

EXIT
ABORT
QUIT

Endless loop (outer interpreter, for example).

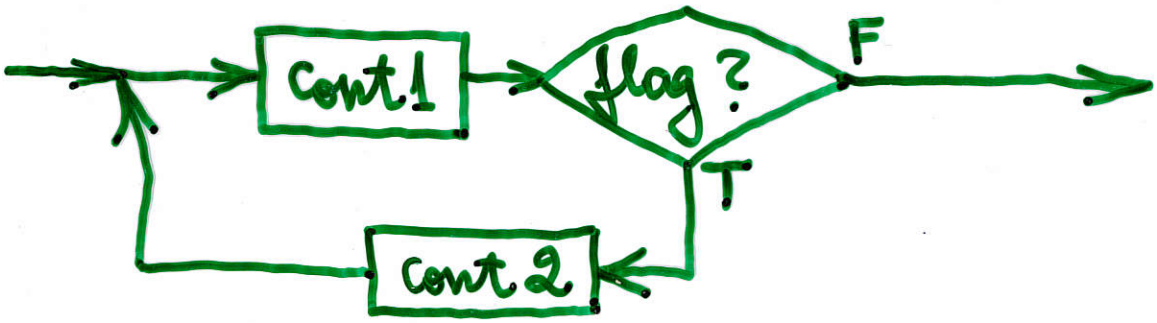
```
BEGIN  ( --- )
AGAIN  ( --- )
```

BEGIN contents flag UNTIL



UNTIL (flag ---)

BEGIN cont1 flag WHILE cont2 REPEAT



WHILE (flag ---)
REPEAT (---)

KEY (--- code)
EMIT (code ---)
CR (---)

```
: WAIT-SPACE ( --- )
  BEGIN
     CR ." ␣Press space-bar "
     KEY DUP EMIT 32 = 0=
  WHILE                  NOT
     CR ." ␣Try again "
  REPEAT
  CR ." ␣That's all right! " ;
```

Indefinite Loops

While DO loops are called definite loops, FORTH also supports "indefinite" loops. This type of loop will repeat indefinitely or until some event occurs. A standard form of indefinite loop is

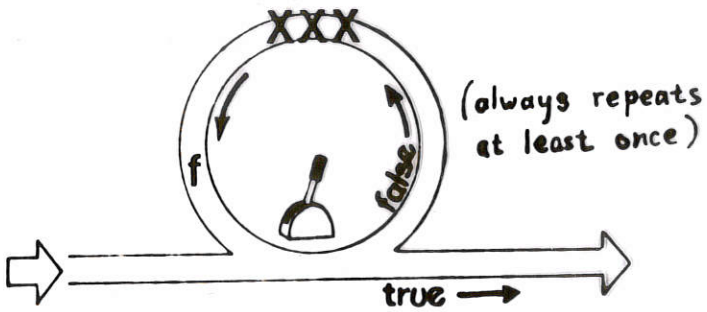
```
BEGIN ... UNTIL
```

The BEGIN...UNTIL loop repeats until a condition is "true."

The useage is

```
BEGIN xxx f UNTIL
```

where "xxx" stands for the words that you want to be repeated, and "f" stands for a flag. As long as the flag is zero (false), the loop will continue to loop, but when the flag becomes non-zero (true), the loop will end.



An example of a definition that uses a BEGIN...UNTIL statement is one we mentioned earlier, in our washing machine example:

```
: TILL-FULL   BEGIN  ?FULL UNTIL ;
```

which we used in the higher-level definition

```
: FILL   FAUCETS OPEN  TILL-FULL  FAUCETS CLOSE ;
```

?FULL will be defined to electronically check a switch in the washtub that indicates when the water reaches the correct level. It will return zero if the switch is not activated and a one if it is. TILL-FULL does nothing but repeatedly make this test over and over (thousands of times per second) until the switch is finally activated, at which time execution will come out of the loop. Then the ; in TILL-FULL will return the flow of execution to the remaining words in FILL, and the water faucets will be turned off.

Sometimes a programmer will deliberately want to create an infinite loop. In FORTH, the best way is with the form

```
BEGIN xxx 0 UNTIL
```

The zero supplies a "false" flag to the word UNTIL, so the loop will repeat eternally.

Beginners usually want to avoid infinite loops, because executing one means that they lose control of the computer (in the sense that only the words inside the loop are being executed). But infinite loops do have their uses. For instance, the text interpreter is part of an infinite loop called QUIT, which waits for input, interprets it, executes it, prints "ok," then waits for input once again. In most microprocessor-controlled machines, the highest-level definition contains an infinite loop that defines the machine's behavior.

Another form of indefinite loop is used in this format:

```
BEGIN xxx f WHILE yyy REPEAT
```

Here the test occurs halfway through the loop rather than at the end. As long as the test is true, the flow of execution continues with the rest of the loop, then returns to the beginning again. If the test is false, the loop ends.

Notice that the effect of the test is opposite that in the BEGIN...UNTIL construction. Here the loop repeats while something is true (rather than until it's true).

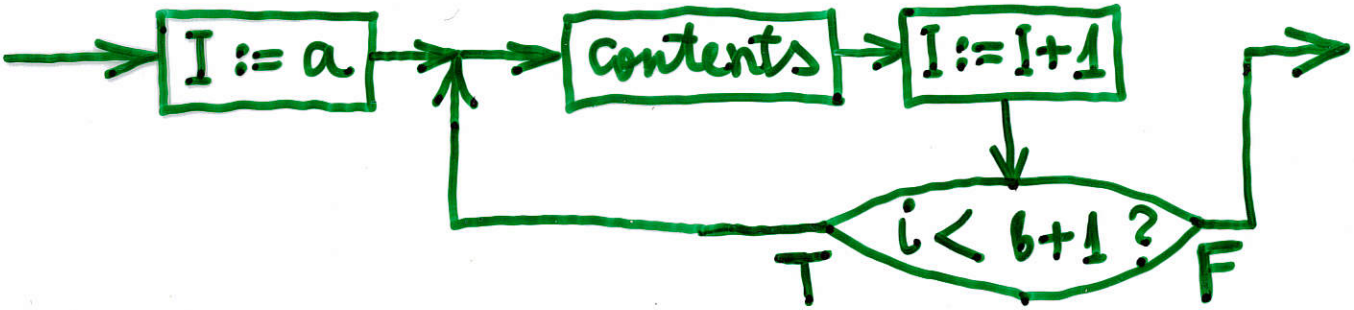
The indefinite loop structures lend themselves best to cases in which you're waiting for some external event to happen, such as the closing of a switch or thermostat, or the setting of a flag by another part of an application that is running simultaneously. So for now, instead of giving examples, we just want you to remember that the indefinite loop structures exist.

b+1 a DO contents LOOP

a ≤ b

10 0 DO...
0...9

a ... b



DO (n_1 n_2 ---)
LOOP (---)
I (--- i) index inside DO ... LOOP
LEAVE (---) exit DO...LOOP now
?DO (n_1 n_2 ---) no step, if $n_1 = n_2$ initially
+LOOP (step ---)

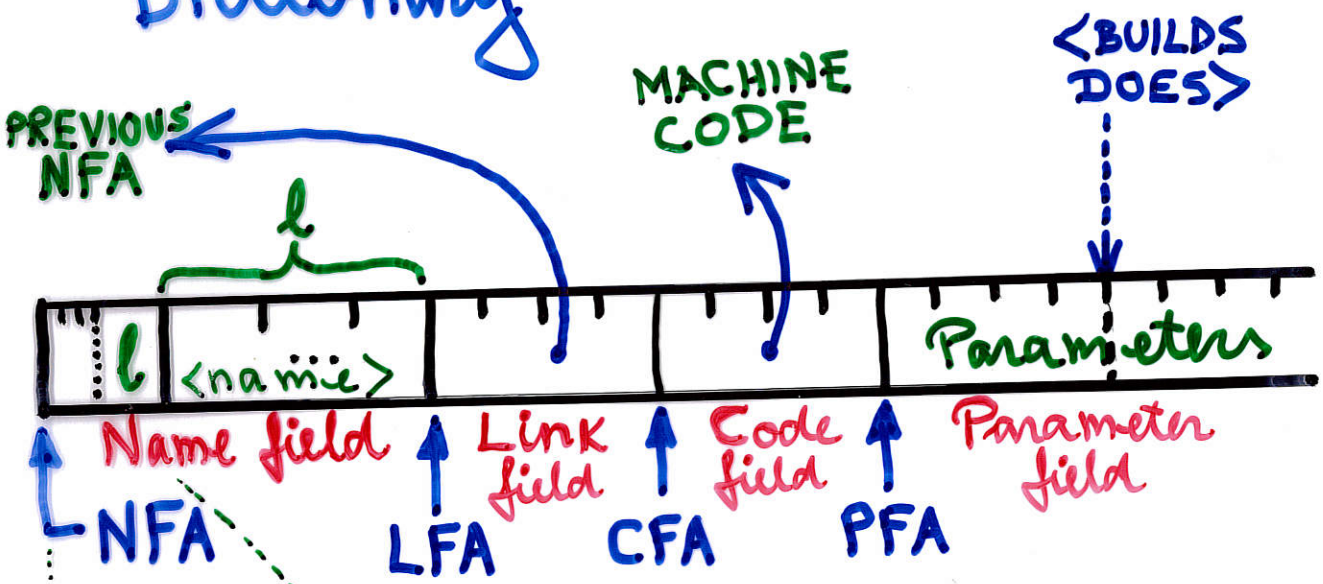
```
: SEQUENCE ( n --- )
  0 DO I . LOOP ;

5 SEQUENCE   0 1 2 3 4

: FACT ( n --- n! )
  DUP 1 DO I * LOOP ;
```

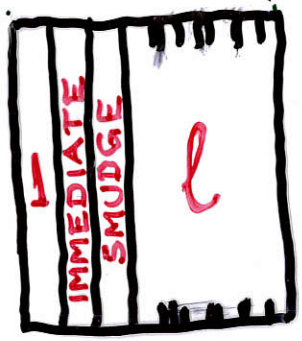
n n n n 1

n (1.. n-1
n n 1

n*1*2..
*(n-1)

Dictionary





$l \leq 31$ name length

IMMEDIATE execute this word during compilation.

SMUDGE name is invisible for dictionary search.

>NAME	(cfa --- nfa)	NAME>	(nfa --- cfa)
>LINK	(cfa --- lfa)	LINK>	(lfa --- cfa)
>BODY	(cfa --- pfa)	BODY>	(pfa --- cfa)

DP dictionary pointer, a Variable

```
: HERE ( --- firstfreelocation )
  DP @ ;

: ALLOT ( n --- )
  DP +! ;

: , ( n --- )
  HERE ! 4 ALLOT ;
```

EXECUTE (cfa ---) Executes a word.

' XXX (--- cfa) gives cfa of XXX during execution.

['] XXX (--- cfa) gives cfa of XXX during compilation and compiles it as a literal.

LATEST (--- nfa) nfa of topmost word (in current vocabulary).

More memory handling

CMOVE (addrfrom addrto len ---)
rewrite len bytes from addrfrom to addrto (from left to right!).

C, C@ C! ... with 8-bit arguments

FILL (addr len byte ---)
Fill len bytes starting at addr with byte.

```
: ERASE ( addr len --- )
  0 FILL ;

: BLANK ( addr len --- )
  32 FILL ;
```

DUMP (ad l ---)

Printing

TYPE (ad len ---)

.NAME (nfa ---)

.STACK (---)

Metaconstruction <BUILDS ... DOES> ...

When using existing defining word, we have two stages:

a) creation (we use defining word to create a new object),

b) use (we use our new word to do its work).

a) VARIABLE A
b) 5 A !

a) 5 CONSTANT V
b) V .

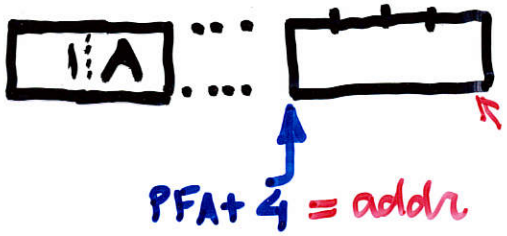
To create a <u>new</u> defining word we'll use <BUILDS ... DOES> ..., by which scenarios are needed for both stages.

```
: XXX   <BUILDS define-time actions
        DOES> use-time actions ;
```

We have define-time: ... XXX yyy a
and use-time: ... yyy b

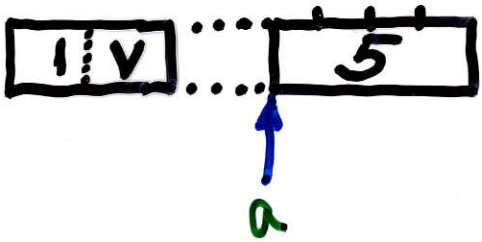
<BUILDS builds a header for yyy. Now the dictionary is free for define-time actions (HERE before these actions = pfa+4). DOES> terminates define-time actions. When yyy is executed, pfa+4 is on the stack and use-time actions are performed.

```
: VARIABLE
  <BUILDS  4 ALLOT  DOES> ;

VARIABLE A

A  ( --- addr )
```



```
: CONSTANT
  <BUILDS  ,  DOES> @ ;

5 CONSTANT V

V ( --- const )
```

a
5

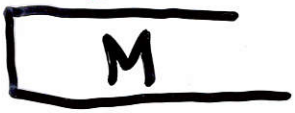


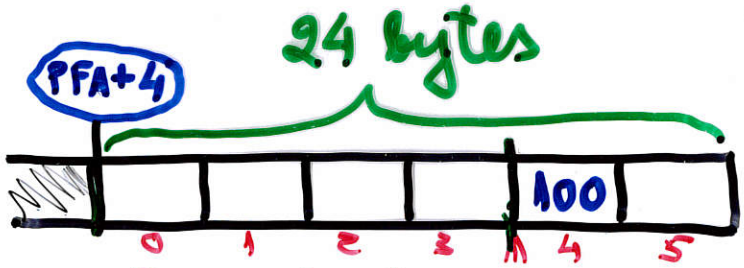
```
: ARRAY
  <BUILDS  4 * ALLOT
  DOES> SWAP  4 * + ;
```

def: #el ---
use: ind --- addr

i a
a i
a+i*4

```
6 ARRAY M
100 4 M !
```





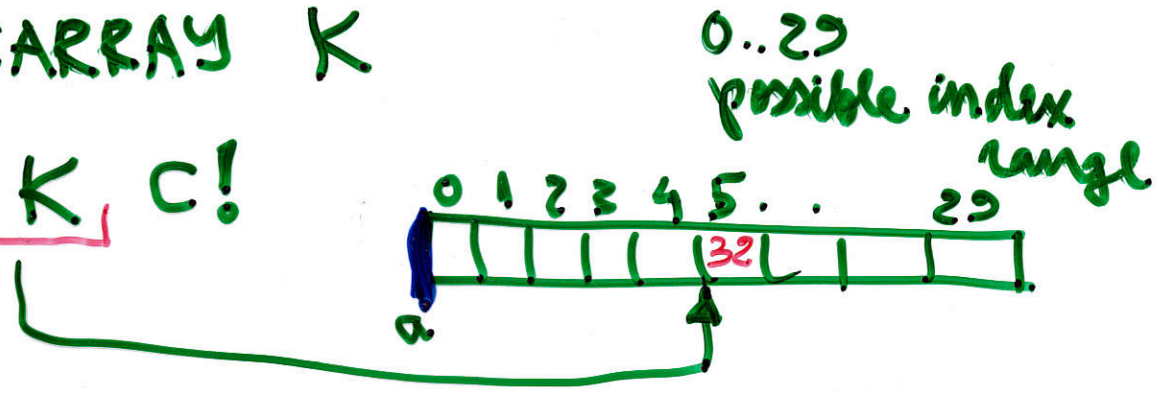

```
100
100 4
100 4 pfa+4
100 pfa+4  4
```

```
100 pfa+4  4  4
100 pfa+4  16
100 pfa+20
```


EXERCISE : Byte array.

30 CARRAY K

32 5 K C!

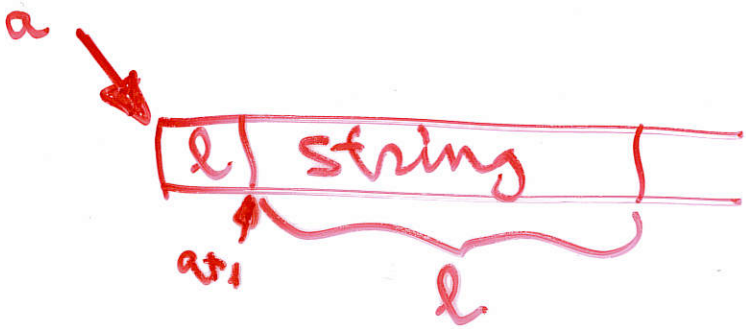


5 a

```
: CARRAY
<BUILDS ALLOT
DOES> + ;
```

32 5
32 5 a
32 5 +

!



CMOVE (afrom ato len ---)

COUNT (a --- a+1 l)

TYPE (a+1 l ---)

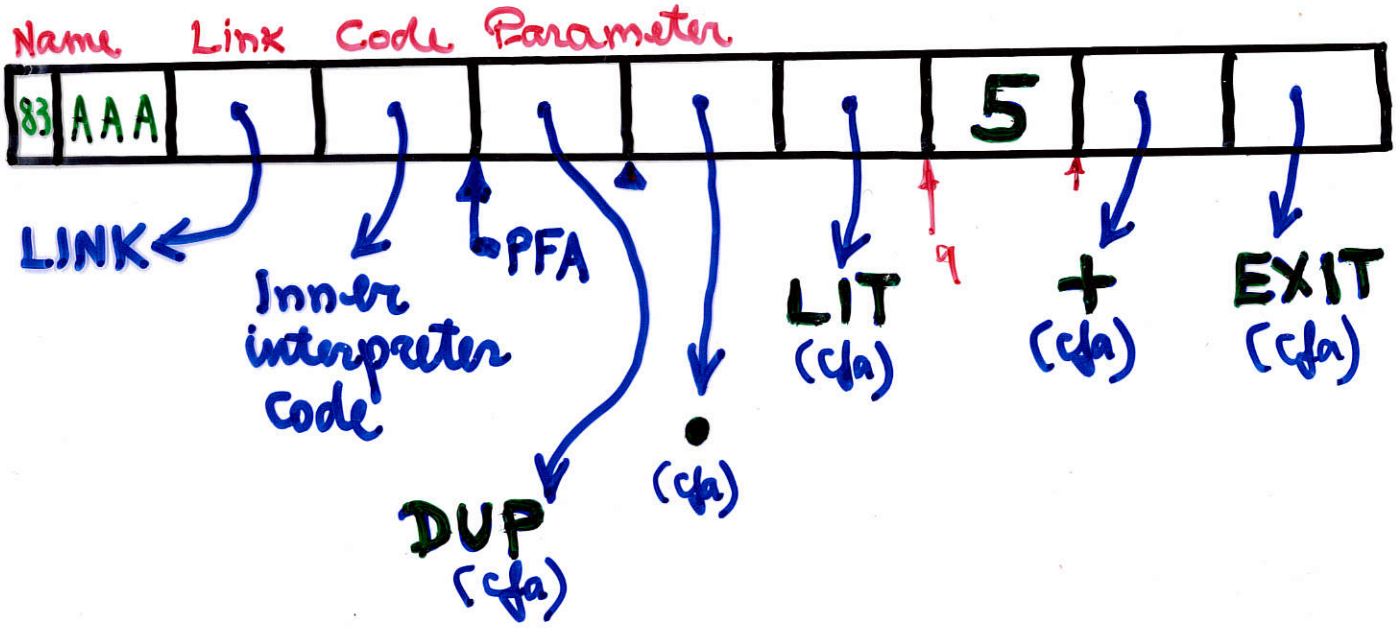
$A_{M \times N}$

def: m n ---

use: i j --- elad

Inner interpreter

: AAA DUP . 5 + ;



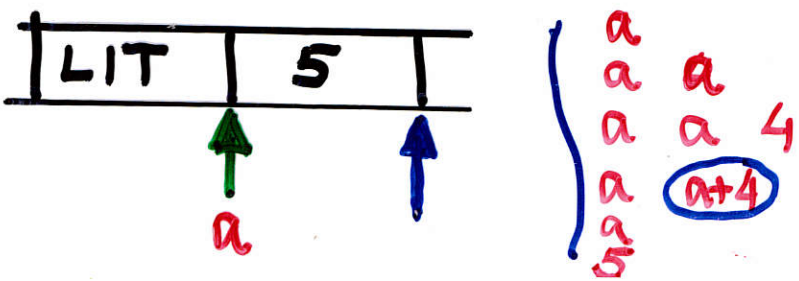
During the interpretation of colon definitions body, the top of the return stack points to the next cell in threaded list (continuation address of current level).

>R (n ---) push n to return stack.

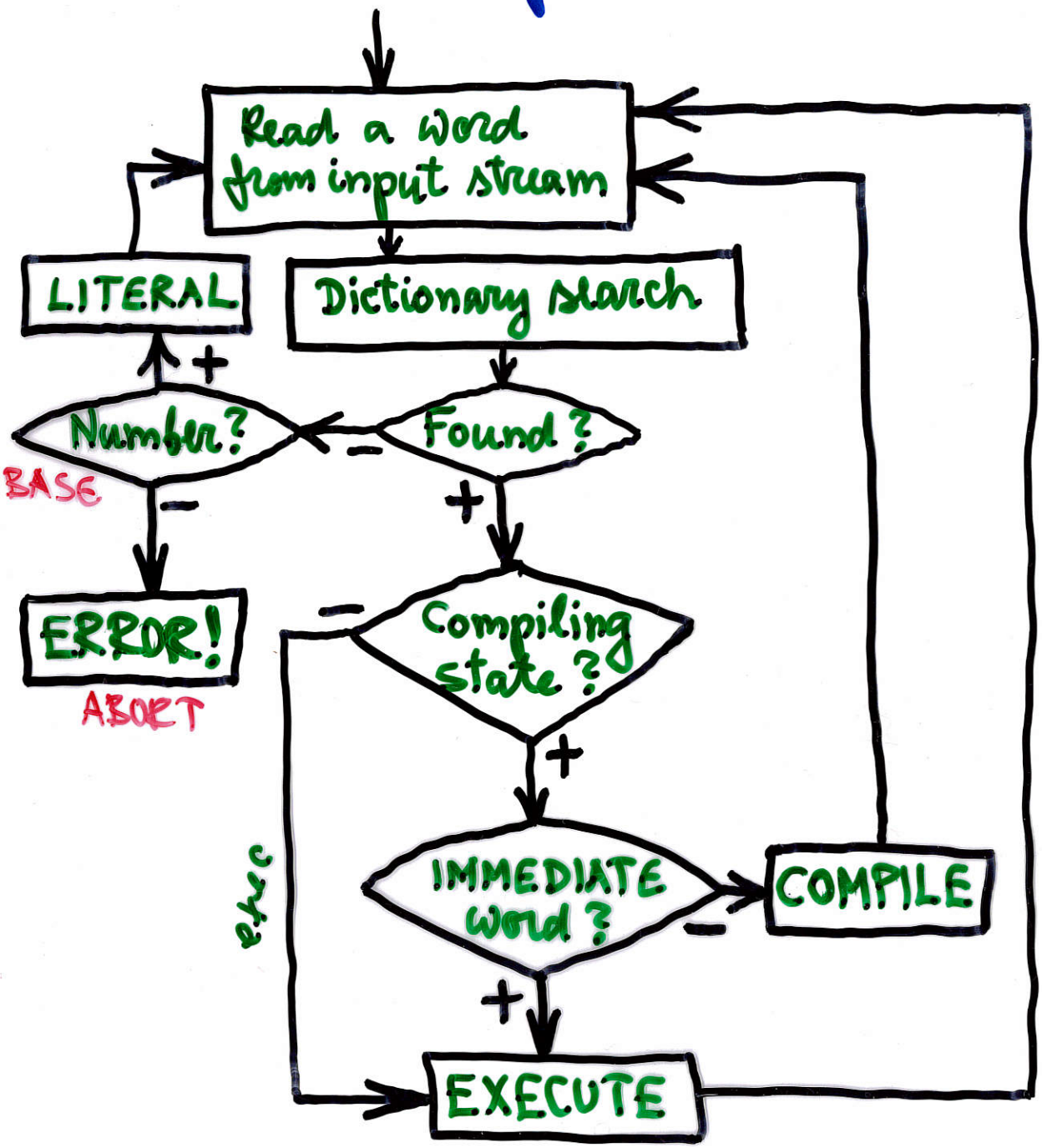
R> (--- n) pop n from return stack (and push it to data stack).

R@ (--- n) copy the top of return stack to data stack.

: LIT R> DUP 4 + >R @ ;



Text interpreter



STATE system variable : zero - execute
non-zero - compile.

IMMEDIATE (---) set IMMEDIATE bit for the latest defined word.

```
: COMPILE ( --- )
  R> DUP 4 + >R @ , ;
: LITERAL ( n --- ) compile time!
  STATE @ IF COMPILE LIT , THEN ; IMMEDIATE
```

Macro-level Forth

Example.

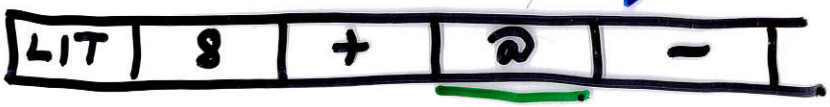
$$A@ = \begin{cases} C@ & , \text{ if BYTEFLAG = true} \\ @ & , \text{ if BYTEFLAG = false} \end{cases}$$

Compile-time problem.

```
VARIABLE BYTEFLAG
: A@
  BYTEFLAG @
  IF COMPILE C@
  ELSE COMPILE @
  THEN ; IMMEDIATE
```

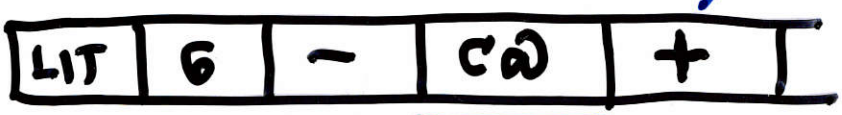
```
Ø BYTEFLAG !
: NNN  8 + A@ - ;
```

NNN




```
-1 BYTEFLAG !
: MMM  6 - A@ + ;
```

MMM



Search order and visibility. Vocabularies.

CURRENT system variable, which points to the "current vocabulary", where definitions are placed. LATEST definition is always in the current vocabulary.

CONTEXT system variable, which points to the "context vocabulary", where to search for the words. Searching always begins in the context vocabulary.

FORTH main vocabulary.

Execution of the vocabulary turns it to context vocabulary.

DEFINITIONS (---) turns context vocabulary to current vocabulary:

```
: DEFINITIONS
    CONTEXT @ CURRENT ! ;
```

VOCABULARY XXX IMMEDIATE
Define a users vocabulary XXX. CONTEXT and CURRENT remain the same.

Vocabulary system ≈ block structure.