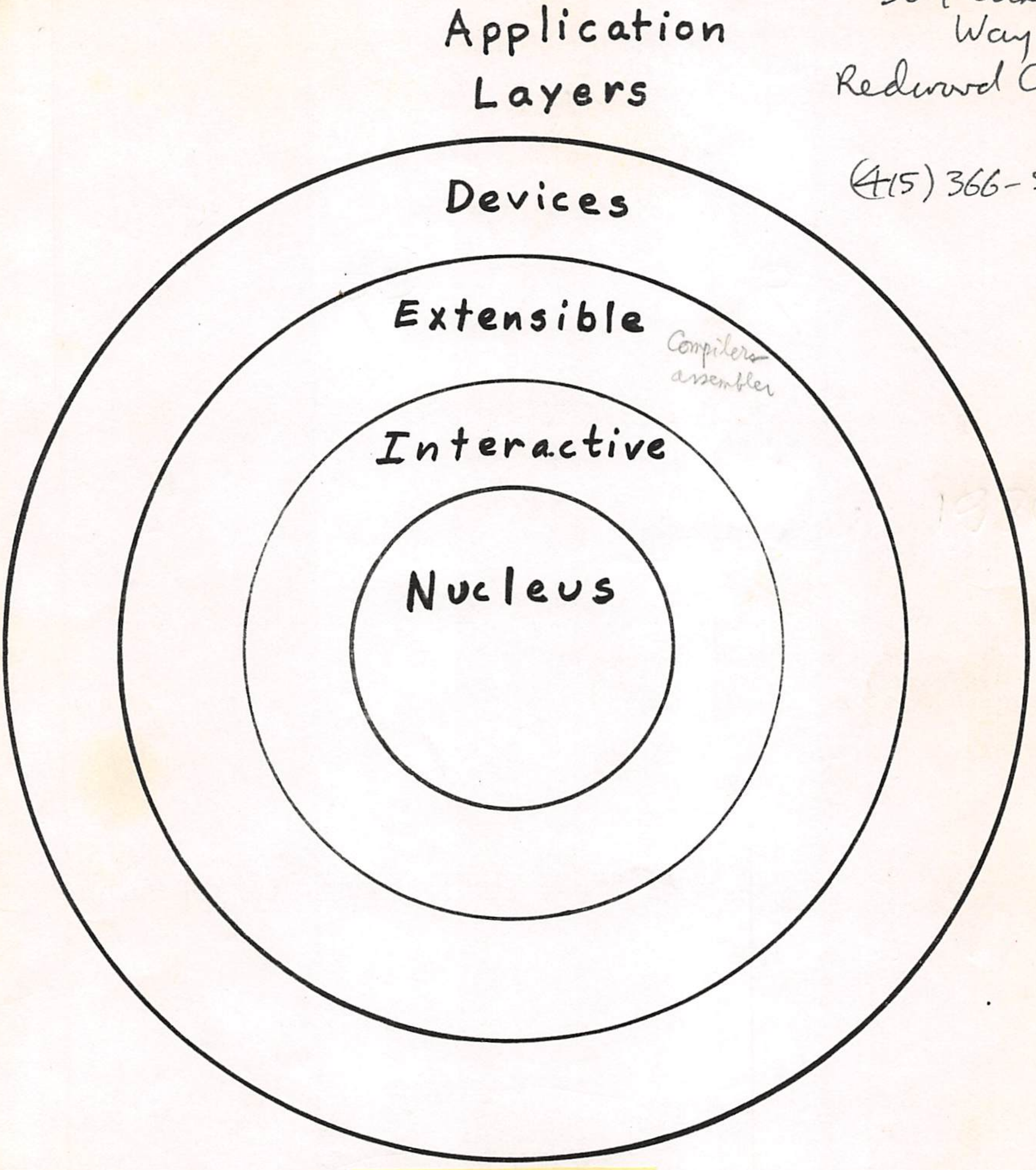
NOTES FROM THE KIM HARRIS COURSE

GIVEN AT SLAC

in June ~~this year~~ 1980

Andrew Korsak, Ph.D.
Consultant
504 Lakemead
Way
Redwood City CA
94062
(415) 366-5228

FORTH'S LAYERS



FORTH LANGUAGE

WORD — a sound or a combination of sounds, or its representation in writing, that symbolizes and communicates a meaning ... a command or an order.
(from the American Heritage Dictionary)

FORTH SYNTAX:

a sequence of Words,
separated by spaces
possibly terminated by a Carriage Return

CR

A word may contain any ASCII characters except spaces, Carriage Return, or Back Space (which erases previous character entered, except Carriage Return).

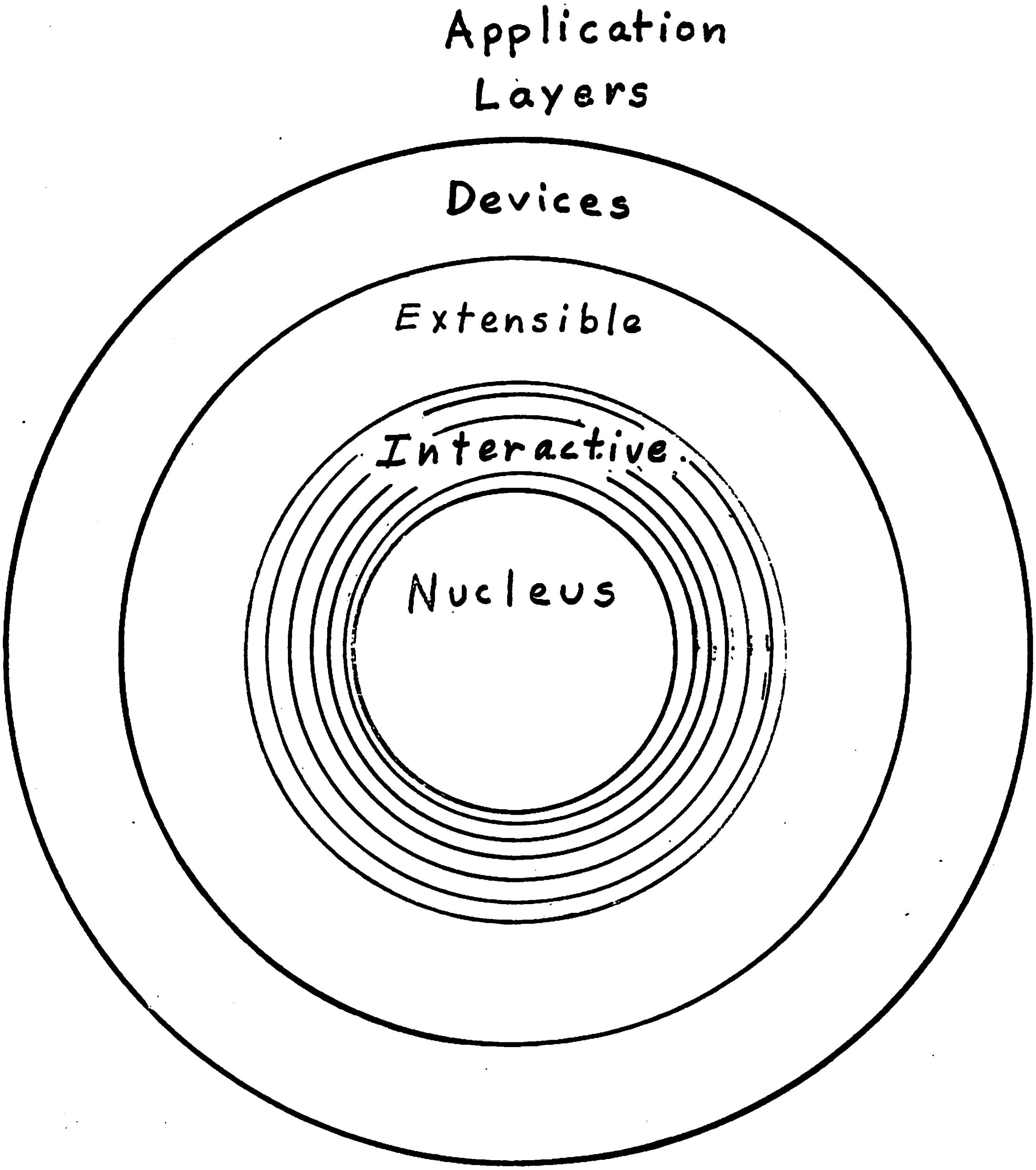
normally 32

variable
WIDTH
determine
no. chars/word
saved in dictionary

Uniqueness:

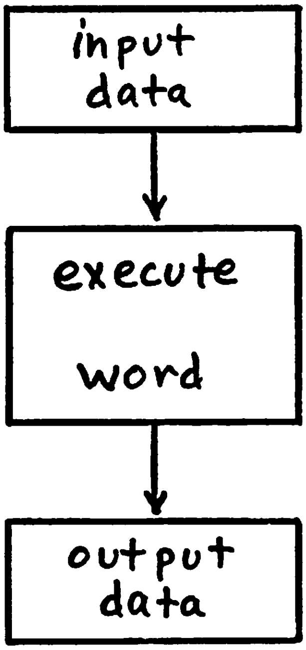
Words are distinguished from all others by length (ie, number of characters) and first 31 characters.

"EXECUTION"


Application
Layers
Devices
Extensible
Interactive
Nucleus

L1.2 (a' G2 a)

EXECUTING a FORTH WORD



examples of words:

FORTH	123	CR	OK
word executed	word converted to binary and stored		reply indicates all words successfully processed

. CR 123 OK

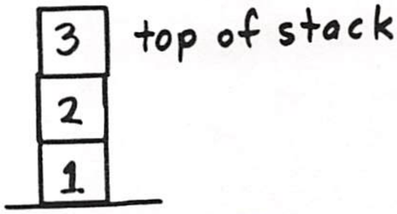
.: command to print out and discard most available number

FORTH has no equivalent to a program "statement"

Stack useage:

numbers entered are pushed onto a stack.

1 2 3 CR OK



. CR 3 OK
. CR 2 OK
. CR 1 OK
. CR STACK EMPTY

SIGNED INTEGERS

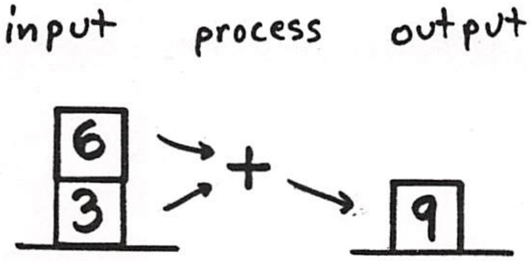
16 bit, range: signed -32768 to 32767
unsigned 0 to 65535

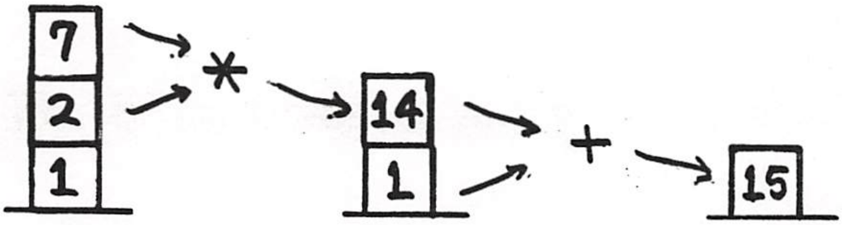
```
-100   .  CR  -100   OK
  -1  U.  CR  65535  OK
32767  .  CR  32767  OK
32768  .  CR  -32768 OK
32768 U.  CR  32768  OK
65535  .  CR  -1     OK
65535 U.  CR  65535  OK
```

Operators & Operands

Operands (data) are on the stack.

Operators take their inputs from the stack and leave their outputs on the stack.





Arithmetic:

stack useage leads to postfix order of operands and operators (RPN, Reverse Polish Notation).

1 2 + . CR 3 OK

7 2 * 1 + . CR 15 OK

3 2 − . CR 1 OK

Notational convention:

word   input stack parameters --- output stack parameters

top of stack listed last →

top of stack listed last →

↑ indicates the execution of word

example:

\+   n2 n1 --- sum

L4
fig

ARITHMETIC OPERATORS:

16 bit signed integer (range -32,768 to 32,767)

+			
-	n1 n2	---	(n1-n2)
*			
/	numerator denominator	---	quotient
MOD	numerator denominator	---	remainder
/MOD	numerator denominator	---	rem. quot.
MINUS	n	---	-n
ABS	n	---	|n|
1+ 1- 2+ 2-			
MAX	n1 n2	---	greater
MIN	n1 n2	---	smaller
*/	n1 ↑ n2 n3	---	quotient of $\frac{n1 * n2}{n3}$
	(32bit intermediate product)		
*/MOD	n1 ↓ n2 n3	---	rem. quot.

16 bit unsigned integer (range 0 to 65,535)

AND
OR
XOR

Examples of arithmetic operators:

```
3   MINUS  .  CR  -3  OK

-3  ABS    .  CR   3  OK

17  4  MAX  .  CR  17  OK

0  -1  MIN  .  CR  -1  OK
```

The composite operators */ and */MOD are useful for scaled, fixed point calculations:

5% of 20000

```
20000  5  100  */  .  CR  1000  OK
```

5 $\frac{1}{2}$ % of 20000

```
20000  55  1000  */  .  CR  1100  OK
```

25.1
fig

32 bit signed integers

Each takes 2 stack cells

d --- nlow nhigh

Entering: digits

punctuation characters:

.

Display: D.

examples

12.3 D. (CR) 123 OK

32 bit signed integer
(range -2,147,483,648 to
2,147,483,647)

← extended arithmetic operators
(chopped off)

D+ d1 d2 --- d(d1+d2)

DMINUS d --- -d

not in FIG FORTH

M+ d n --- d sum

M/ d n --- ($\frac{d}{n}$ quot.)

M/MOD d n --- d($\frac{d}{n}$ quot.) rem.

← unsigned

M* n1 n2 --- 32 bit prod.

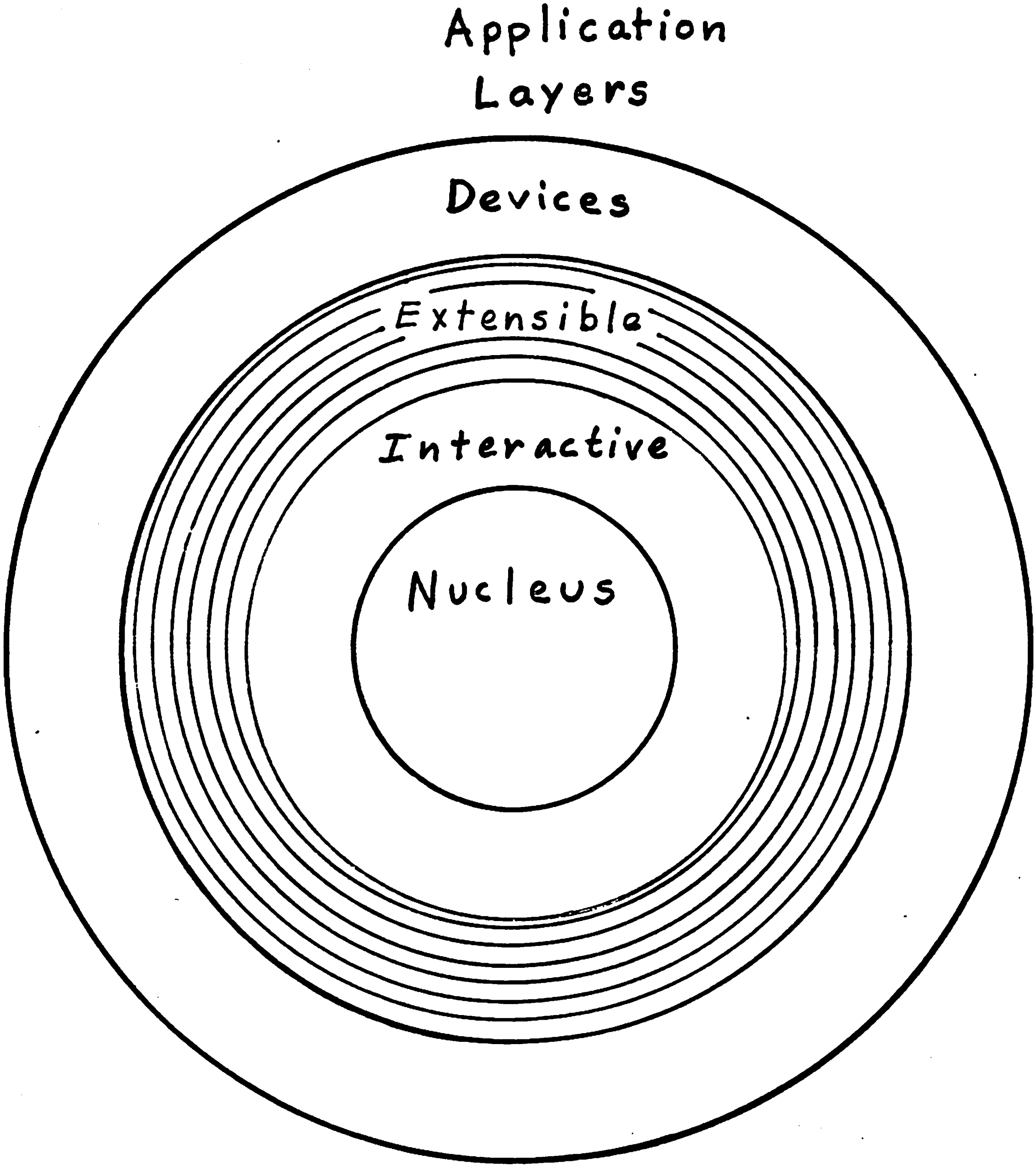
16 bits each

M stands for "mixed"

POLYFORTH has also

M*/ d n_1 n_2 --- $\frac{dn_1}{n_2}$

FORTH COMPILER



The collection of defined FORTH words is called a dictionary.

(from the American Heritage Dictionary:)

DICTIONARY — a listing of words ... with specialized information about them.

FORTH's dictionary contains words' names and a compiled form of their definitions in the order they were defined.

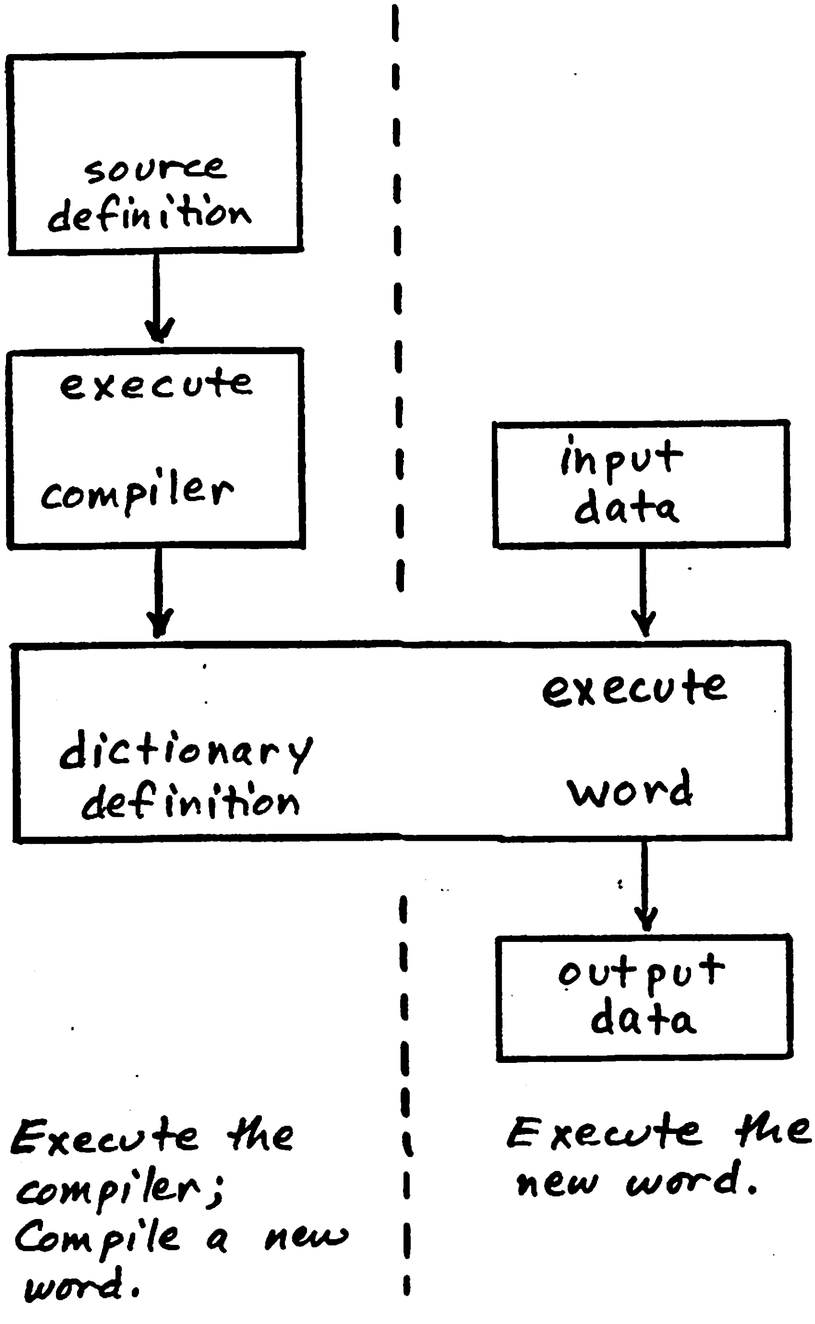
Defining a new word

$$: \ \text{new\_word} \qquad \underbrace{\textbf{definition}}_{\text{previously defined words or numbers}} \ ;$$

examples

definition	**useage**
`: 8* 2* 2* 2* ;`	`7 8* . (CR) 56 OK`
`: % 100 */ ;`	`200 5 % . (CR) 10 OK`

USING A FORTH COMPILER



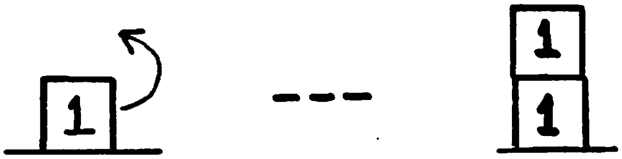
Execute the compiler; Compile a new word.

Execute the new word.

16 bit STACK MANIPULATION OPERATORS

DROP

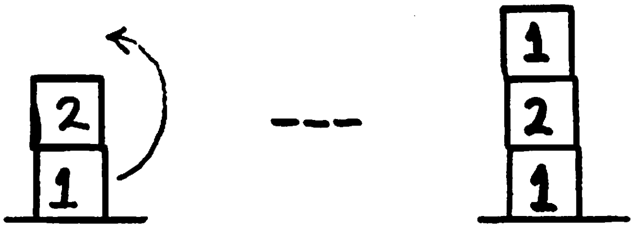
DUP



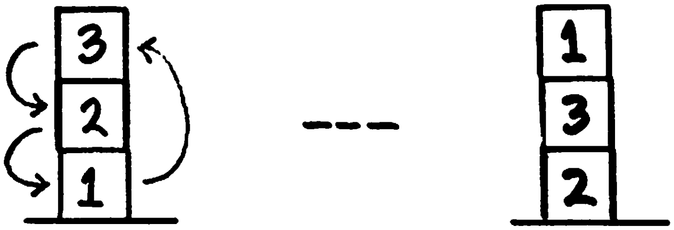
should never need more than 2 of these between other words

SWAP

OVER



ROT



examples of stack manipulation

```
3 DUP * . (CR) 9 OK

: SQUARE   DUP * ; (CR)   OK

3 SQUARE . (CR) 9 OK
```

SYMBOLIC CONSTANTS

Defining a 16 bit constant

number CONSTANT name ← a kind of compiler - distinct from ":"

examples

definitions	**useage**
10 CONSTANT TEN	TEN . (CR) 10 OK
9430 CONSTANT MY-ZIP	MY-ZIP . (CR) 9430 OK

A CONSTANT's name may be used anywhere a number (literal) can be used.

MY-ZIP TEN 3 * + . (CR) 9460 OK

4th
fig

VARIABLES

– a symbol whose value can be changed.

defining a variable

$\underbrace{\text{value}}_{\text{initial value}}$ VARIABLE name

examples

```
   0   VARIABLE  X
9876   VARIABLE  ZIP
```

operations on variables

fetch the value

variable_name @
(pronounced fetch)

change the value

new_value variable_name !
(pronounced store)

examples of using variables

```
1 X ! (CR)  OK
X @ . (CR)  1  OK
MY-ZIP  ZIP ! (CR)  OK
ZIP @ . (CR)  9430  OK
TEN 3 + X ! (CR)  OK
X @ . (CR)  13  OK
```

define additional, useful operators

fetch and display   useage

```
: ?  @ . ;                        X ? (CR) 13 OK
```

increment contents of a variable

```
:  +!    ( increment variable_name --- )
     DUP  @  ROT  +  SWAP  !  ;
```

```
2  X  +! (CR)  OK
   X  ?  (CR)  15  OK

-5 X  +! (CR)  OK
   X  ?  (CR)  10  OK
```

Base conversion of numbers

the conversion to and from the internal (binary) value and the external, displayed form can be performed according to any base (radix).

examples

```
   16   HEX     .  CR   10    OK
 7FFF   DECIMAL .  CR   32767   OK

 40 3 * 7 + DUP . HEX . CR 127 7F OK
```

this conversion is controlled by the contents of variable BASE

```
: HEX       16 BASE ! ;
: DECIMAL   10 BASE ! ;
```

could also define

<u>definition</u>	<u>useage</u>
`: OCTAL   8 BASE ! ;`	`TEN OCTAL . CR 12 OK`
`: BINARY  2 BASE ! ;`	`BINARY 100111 OCTAL . CR 47 OK`

What is

```
BASE  ?

OCTAL  BASE  ?

BINARY  BASE  ?
```

Define a word to display the value of BASE in decimal, regardless of BASE's current value.

```
:  ?BASE    BASE @
            DUP  DECIMAL .
            BASE !  ;
```

useage

```
DECIMAL    ?BASE  (CR)  10   OK

HEX        ?BASE  (CR)  16   OK

BINARY     ?BASE  (CR)  2    OK
```

How VARIABLEs work

```
variable_name   ---   address
```

examples

```
BASE  .  (CR)  10294  OK
X     .  (CR)  7920   OK
ZIP   .  (CR)  7930   OK
```

address operators

```
      address  @  ---  contents
value address  !  ---
```

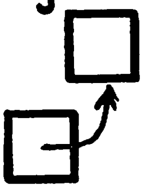
examples

```
BASE . (CR) 10294 OK
10294 @ . (CR) 10 OK
8 10294 ! (CR) OK
TEN . (CR) 12 OK
```

This is a very simple and general capability.

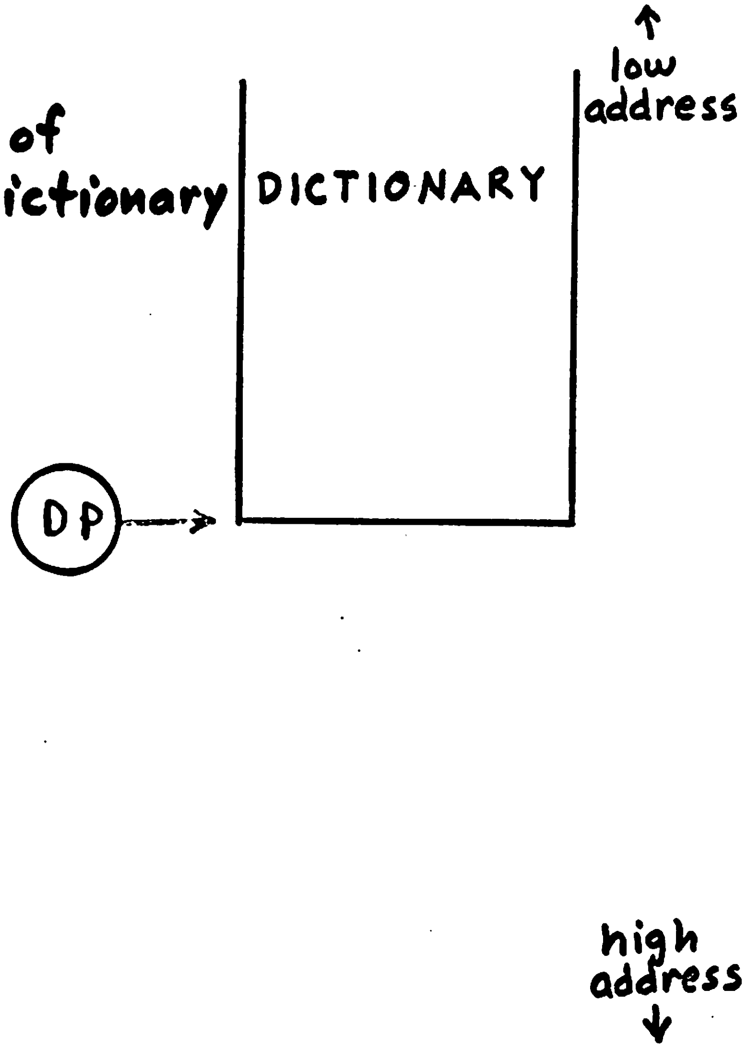
example (indirect addressing)

```
  0 VARIABLE VALUE
  0 VARIABLE POINTER
MY-ZIP VALUE !
VALUE POINTER !
 POINTER @ @ . (CR) 9430 OK
```



DICTIONARY ALLOCATION

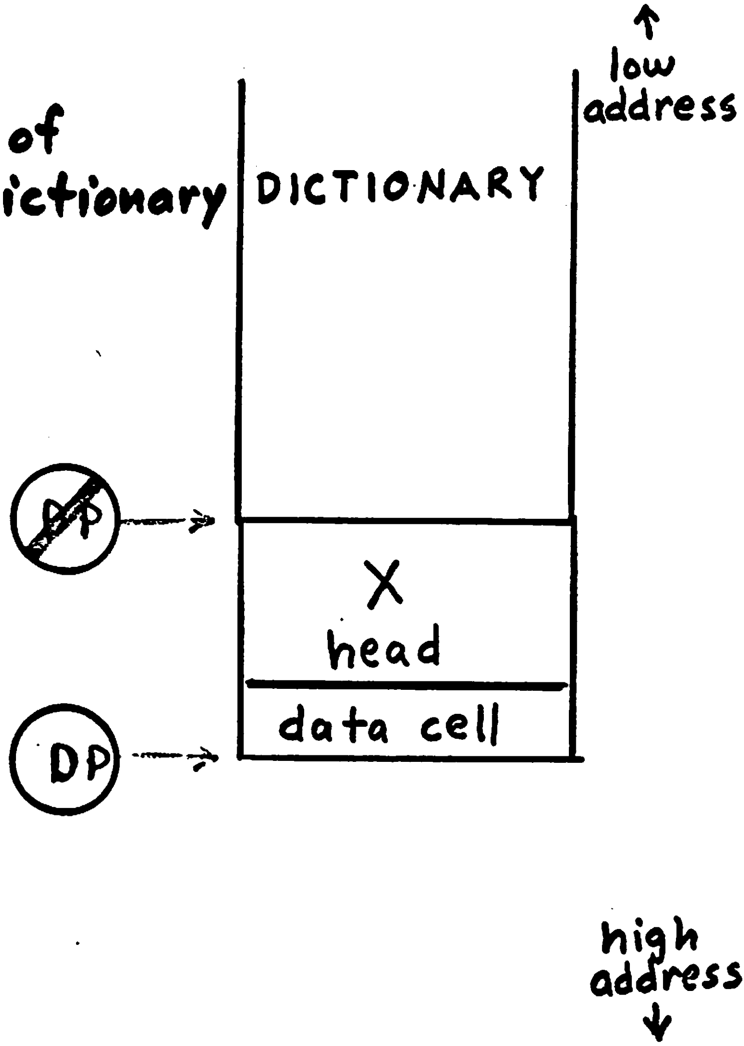
HERE --- address of top of dictionary stack



DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack

VARIABLE X

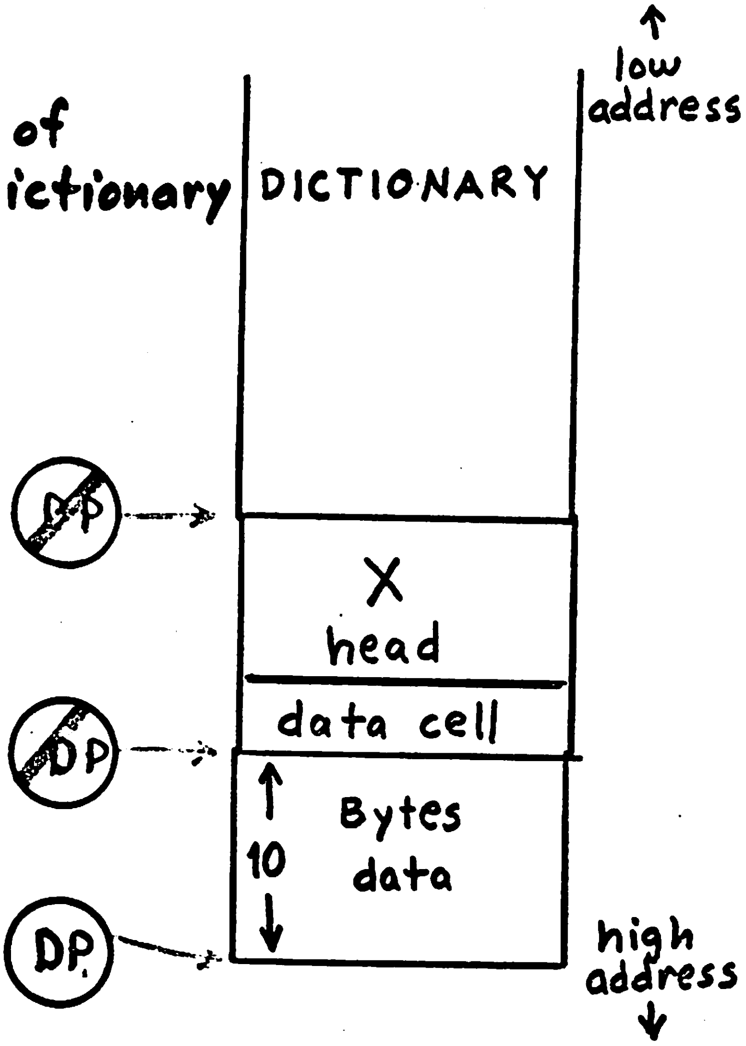


DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack

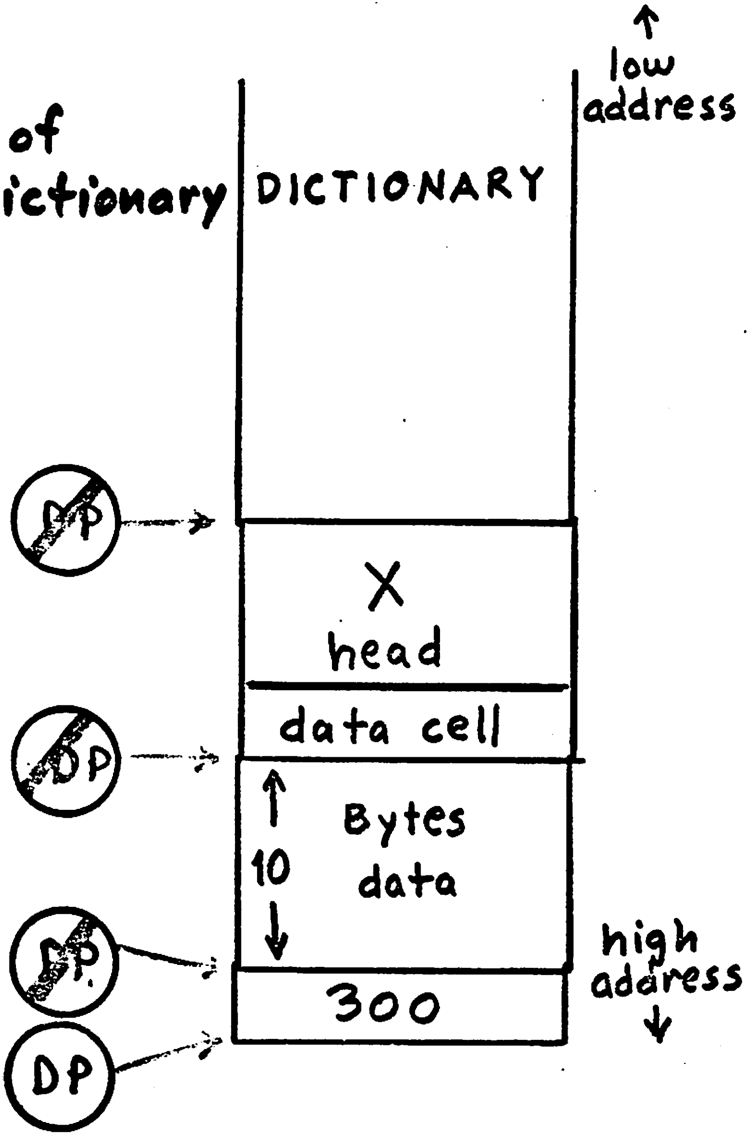
VARIABLE X

10 ALLOT



DICTIONARY ALLOCATION

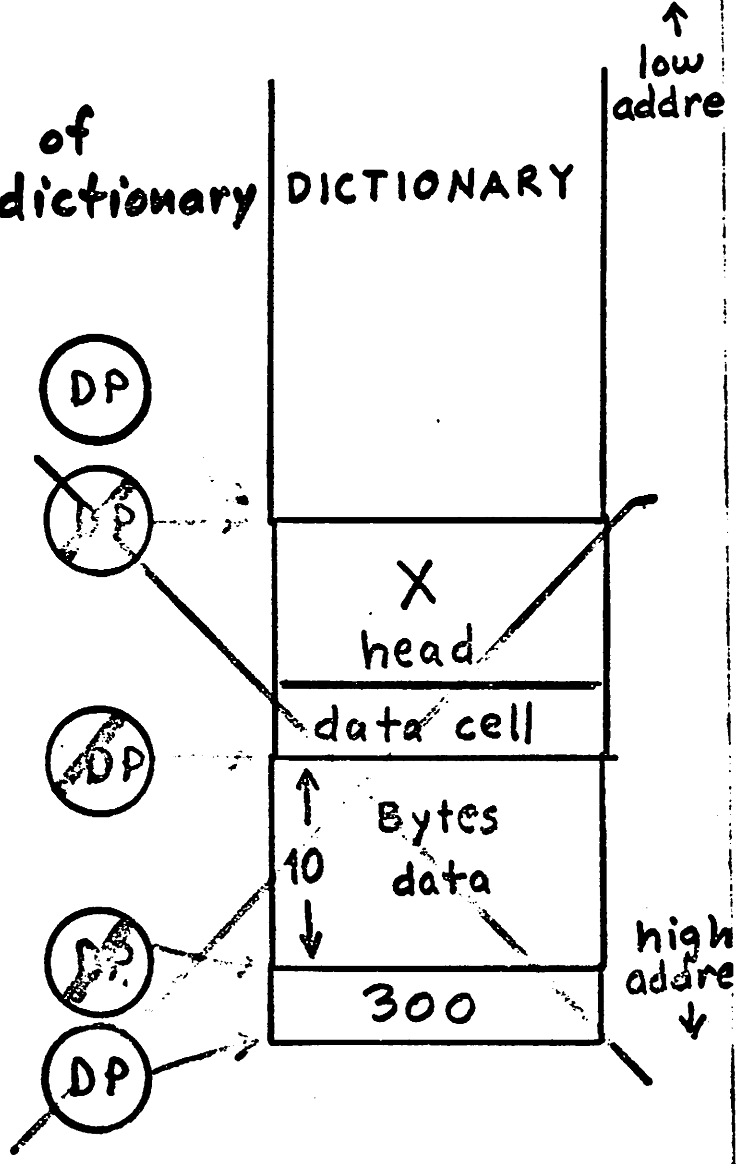
HERE --- address of top of dictionary stack



```
VARIABLE X

10 ALLOT

300 ,
```

DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack



VARIABLE X

10 ALLOT

300 ,

FORGET X

L16
fig

Example of address manipulation: pseudo variable arrays

Defining a variable array

```
0 VARIABLE 'TABLE 6 ALLOT     (size 4 cells)
```

Initializing the array

```
1 'TABLE       !     ( 1st cell)
2 'TABLE 2 +   !     ( 2nd  "  )
3 'TABLE 4 +   !     ( 3rd  "  )
4 'TABLE 6 +   !     ( 4th  "  )
```

Accessing cells of the array

```
'TABLE @ . CR 1 OK            ( 1st cell)
'TABLE 4 + @ . CR 3 OK        ( 3rd cell )
```

To simplify cell selection and to improve readability, define

```
: TABLE   ( subscript --- addr-of-cell )
     2* 'TABLE + ;
```

then

```
0 TABLE @ . CR 1 OK           ( 1st cell)
2 TABLE @ . CR 3 OK           ( 3rd cell)
```

or if you prefer subscripts to start at 1, define

```
   : TABLE ( subscript --- addr-of-cell )
        1- 2* 'TABLE +  ;
```

then

```
1   TABLE      @ . (CR) 1 OK    ( 1st cell)
2   TABLE      @ . (CR) 2 OK    ( 2nd cell)
```

Another way to create an initialized variable array

```
1  VARIABLE 'TABLE   2 ,   3 ,  4 ,
     ( size is 4 cells )   ↑
```

"compiles" top stack value into dictionary

Access is the same as before

```
 2 TABLE       @ . (CR) 2  OK      ( 2nd cell)

-15    2 TABLE      ! (CR) OK      ( 2nd cell)

'TABLE 2 +   ? (CR) -15  OK        ( 2nd cell)
```

Searching the dictionary:

PFA

' name --- { if found, returns the address of this name in the dictionary
else, name ? (abort) }

(pronounced "tick")

useful for

determining if a word is in the dictionary without executing it,

determining if a new name "collides" with an existing word,

obtaining the dictionary address of a word.

Examples:

```
' FORTH . (CR)   7534  ok

' SCRUB . (CR)   SCRUB ?
```

Executing a word in the dictionary:

name in interpret state, searches the dictionary and executes the word

or, can execute a word given its dictionary address:

```
dictionary_address  CFA  EXECUTE
```

causes the word at that address to be executed.

Example: deferred execution

```
: GREET  ." How are you? " ;
0 VARIABLE DEFER

' GREET  DEFER  !

DEFER @  CFA  EXECUTE (CR)  How are you? ok
```

This idea is used for 'computed GO TO'

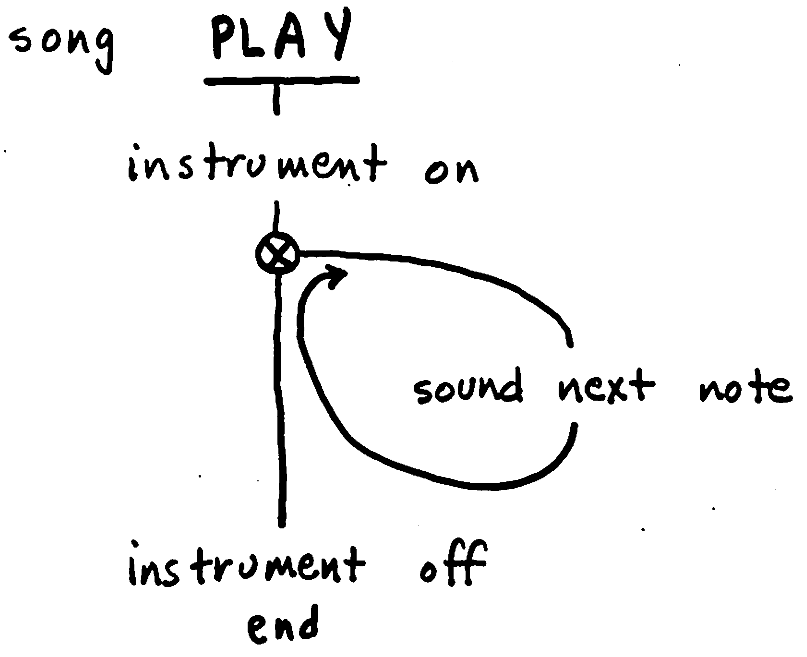
STRUCTURED PROGRAMMING

successive refinement

hierarchical decomposition of a problem

top-down	start at entire application's function
bottom-up	start at primitive, fundamental operations

example: music playing program

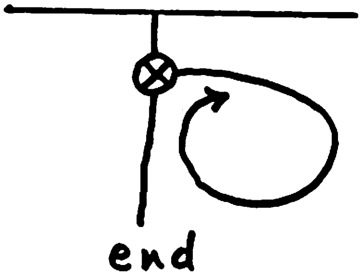


instrument on
set tempo
set scale
end

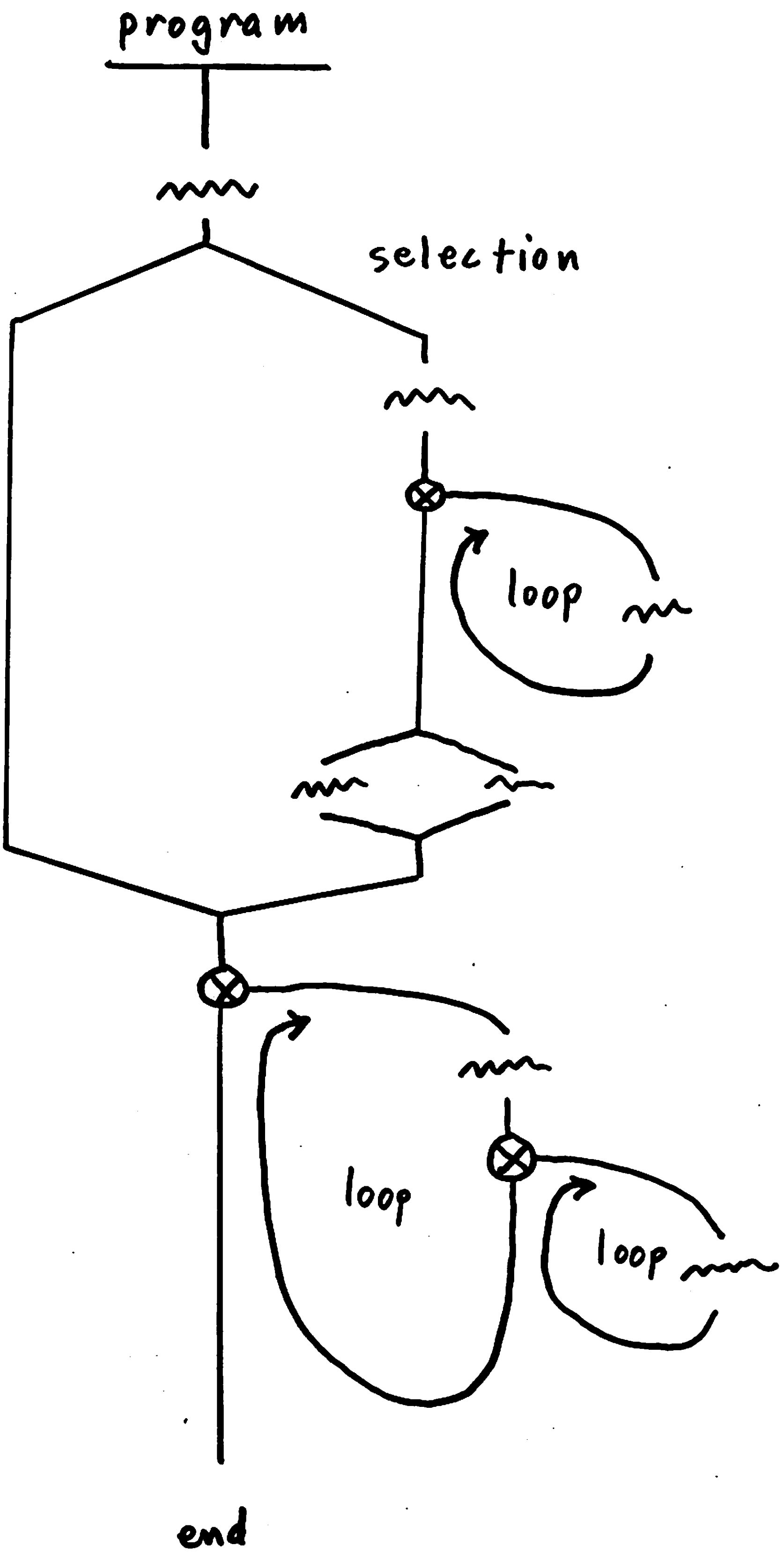
instrument off
quiet
end

sound next note
set frequency
start sound
wait for note's duration
stop sound
end

wait for note's duration

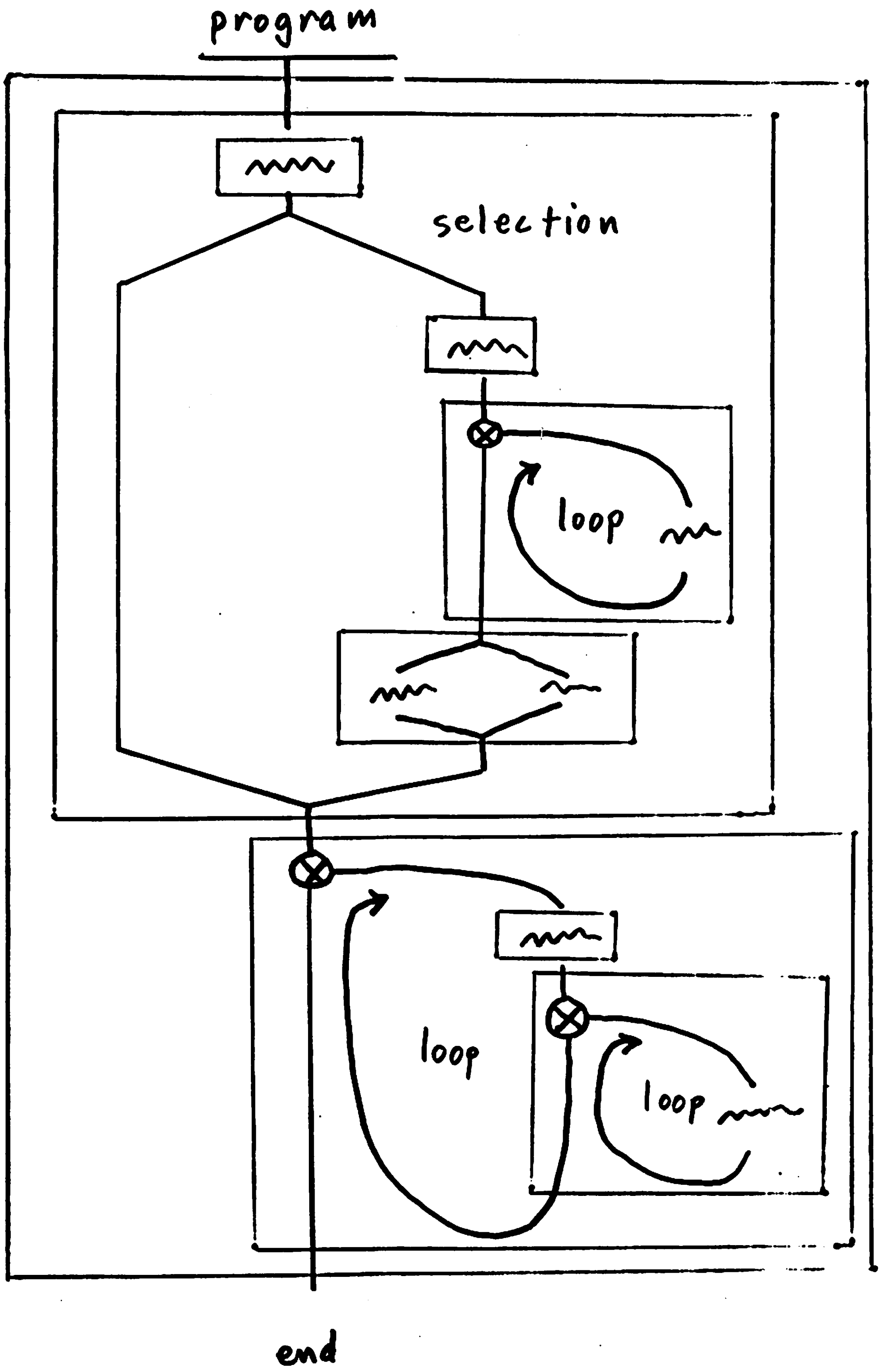


Structured programming provides a uniform way to break a complicated structure into simple parts.



RULE: 1 control path in ; 1 control path out
data data

Structured programming provides a uniform way to break a complicated structure into simple parts.



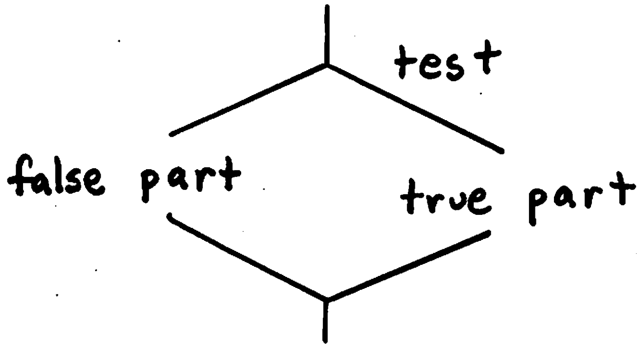
RULE: 1 control path in ; 1 control path out
data data

D-CHARTS

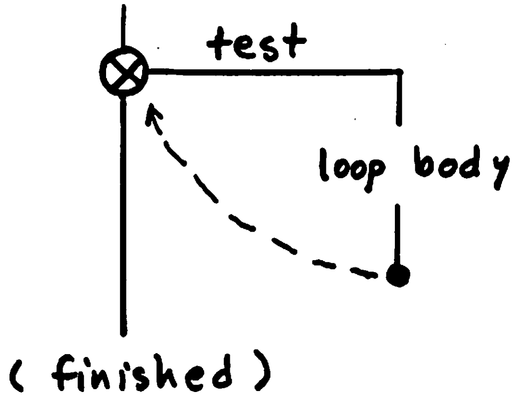
sequential operations:

step one
step two
step three
⋮

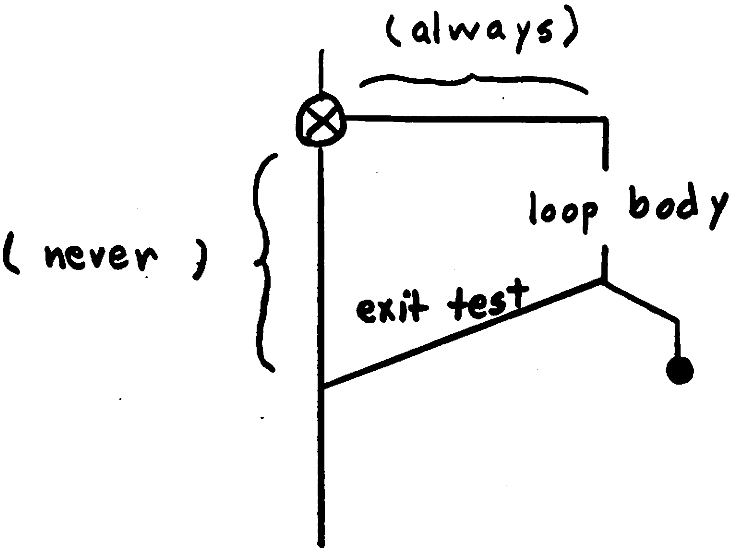
selection:



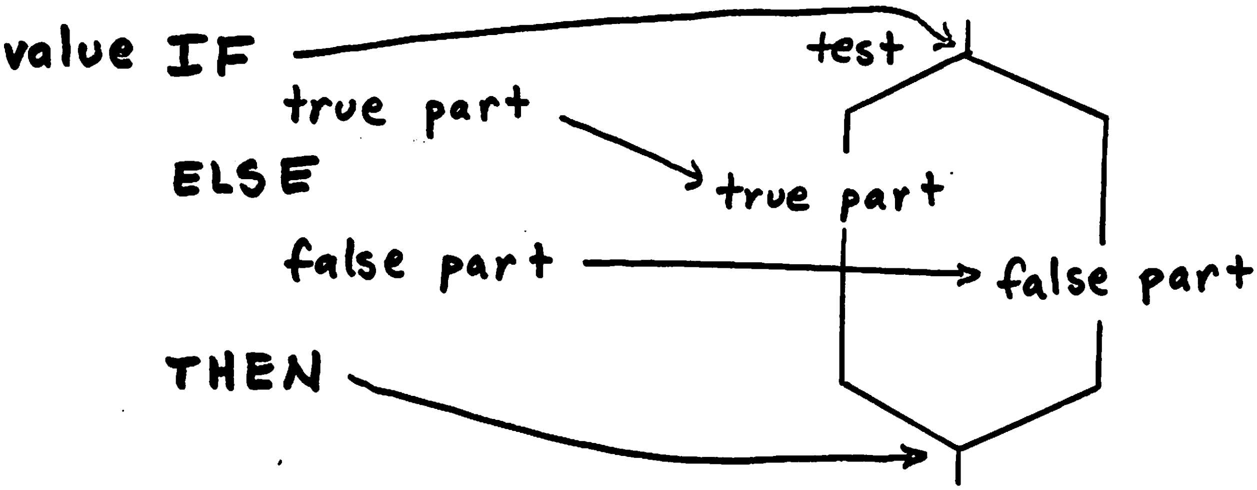
loop:



or



FORTH compiler control structure for SELECTION



value = 0 means false

value ≠ 0 means true

example:

<u>definition</u>

```
: TEST  IF ." TRUE "  ELSE ." FALSE "
        THEN ;
```

<u>useage</u>

```
1   TEST (CR)  TRUE  OK
0   TEST (CR)  FALSE OK
-15 TEST (CR)  TRUE  OK
```

NOTE: IF, ELSE and THEN can be used **only** within a : definition.

COMPARISON OPERATORS

16 bit signed integer:

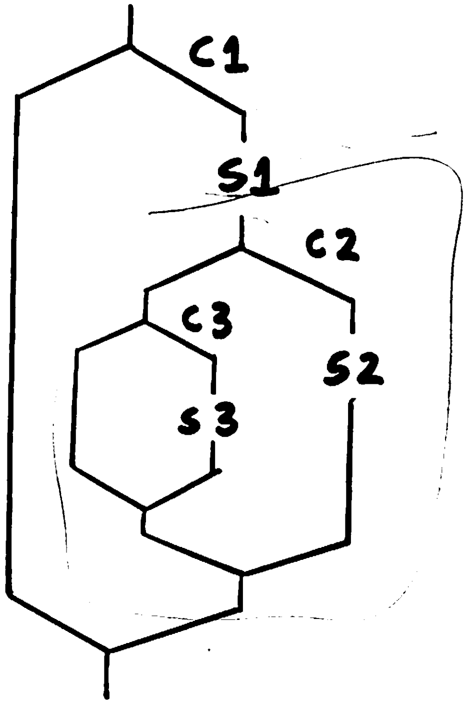
0=	n ---	0 if $n \neq 0$ 1 if $n = 0$
0<	n ---	0 if $n \geq 0$ 1 if $n < 0$
=	n1 n2 ---	0 if $n1 \neq n2$ 1 if $n1 = n2$
−	n1 n2 ---	0 if $n1 = n2$ $\neq 0$ if $n1 \neq n2$
<	n1 n2 ---	0 if $n1 \geq n2$ 1 if $n1 < n2$
>	n1 n2 ---	0 if $n1 \leq n2$ 1 if $n1 > n2$

comparison examples

```
 0   0=  TEST  CR  TRUE   OK
 1   0=  TEST  CR  FALSE  OK
-1   0<  TEST  CR  TRUE   OK

 4   3   =   TEST  CR  FALSE  OK
-4  -3   <   TEST  CR  TRUE   OK
 1  10   >   TEST  CR  FALSE  OK
```

Nesting IF structures

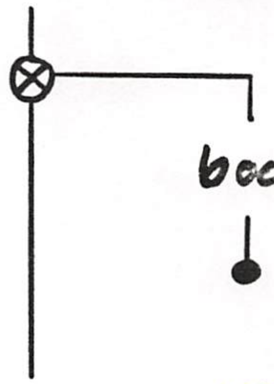
```
c1  IF
        s1
        c2  IF
                s2
            ELSE
                c3  IF
                      s3
                    THEN
            THEN
     THEN
```



L26
a
fig

DO LOOPS

final initial DO
loop body
{ LOOP
or
inc +LOOP }



function:

stored on top of return stack & limit beneath that

DO removes 2 parameters, sets index ← initial

loop body always executed once

LOOP adds 1 to index,
exits loop if index ≥ final
otherwise, branches back to DO

+LOOP removes inc, adds it to index
exit condition: inc > 0 exit if index ≥ final
inc < 0 exit if index ≤ final

within loop body

I --- current loop index

so I remains valid

LEAVE sets limit ← current loop index
so will exit next time at
LOOP or +LOOP

NOTE:

DO, LOOP, +LOOP, & LEAVE
can be used only within : definitions.

L27
fig

examples of DO loops:

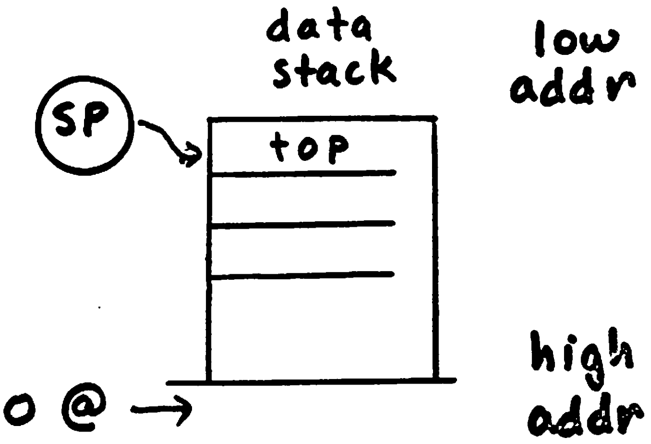
```
: COUNT  DO  I .  LOOP ;
   4 0 COUNT CR  0 1 2 3  OK
   0 4 COUNT CR  4  OK
 -16 -20 COUNT      -20 -19 -18 -17  OK
```

```
: 2+COUNT  DO  I .  2  +LOOP ;
 10 0  2+COUNT CR  0 2 4 6 8  OK
  9 0  2+COUNT CR  0 2 4 6 8  OK

: 10-COUNT  DO  I .  -10  +LOOP ;
 50 100  10-COUNT CR  100 90 80 70 60   OK

: INC-COUNT  DO
                I .  DUP
             +LOOP
             DROP ;
  1 5 0  INC-COUNT CR  0 1 2 3 4  OK
  2 5 0  INC-COUNT CR  0 2 4  OK
 -3 -10 5  INC-COUNT CR  5 2 -1 -4 -7  OK
```

```
: +COUNT   DO
              I .   I 0= IF
                             LEAVE
                         THEN
           LOOP   ;

5  1   +COUNT  CR   1  2  3  4    OK
5  -3  +COUNT  CR   -3  -2  -1  0  OK
```

non-destructive stack print with top to the right for figFORTH

SP@ --- value of SP
= address of top stack cell



number of cells on the stack:

```
: DEPTH   SO @   SP@  -   2 /  1 - ;
```

stack dump :

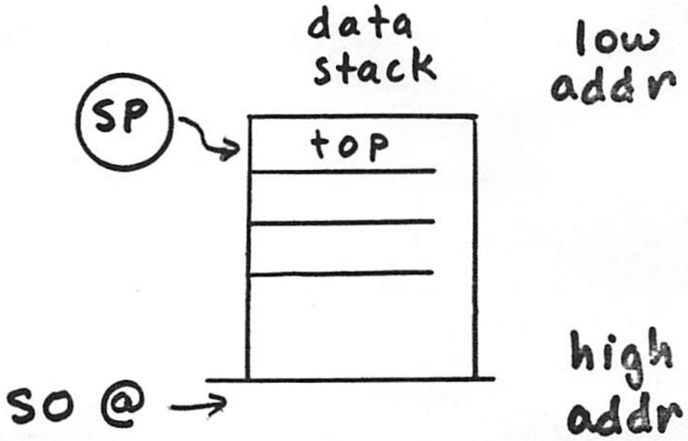
```
:  .S
          SP@  2 -    SO @  2 -
          DO   I  @  .     -2   +LOOP

;
```

__usage__

```
1 2 3  .S CR  1 2 3  OK
```

non-destructive stack print with top to the right for figFORTH

SP@ --- value of SP
= address of top stack cell



number of cells on the stack:

```
: DEPTH   SO @   SP@  -   2 /  1 - ;
```

stack dump :

```
:  .S   DEPTH IF
        SP@ 2 -   SO @  2 -
        DO   I @ .    -2   +LOOP
        ELSE  ." Empty " THEN
;
```

<u>usage</u>

```
1 2 3  .S (CR)  1  2  3   OK
```

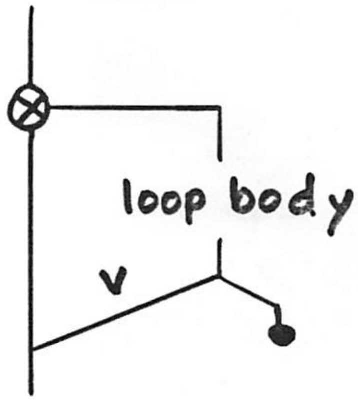
```
: PICK   ( n --- n-th item )
  2 *  SP@  +  @  ;
  DUP +
```

L30
r.1
fig

conditional loops

loop UNTIL a condition becomes true

```
  BEGIN
     loop body
v UNTIL
```



function:

loop body is always executed once

UNTIL removes v
- exit loop if $v \neq 0$ (true)
- branch to BEGIN if $v = 0$ (false)

NOTE: BEGIN & UNTIL can be used <u>only</u> within : definitions

examples:

```
: COUNT-DOWN   BEGIN
                  DUP . 1- DUP 0=
               UNTIL DROP ;
5 COUNT-DOWN CR  5 4 3 2 1     OK

: HALVES  BEGIN
                DUP . 2/ DUP 0=
             UNTIL  DROP ;
16 HALVES CR  16 8 4 2 1      OK
```

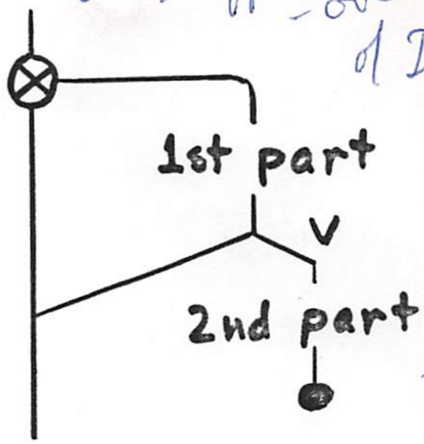
Could use -DUP which doesn't dup when have 0

L31
fig

conditional loops

loop WHILE a condition remains true

```
BEGIN
    1st part loop body
v WHILE
    2nd part loop body
REPEAT
```



Very efficient - overhead of DO LOOP is 20 times this

function:

1st part loop body always executed once

WHILE removes v
exit loop if v = 0 (false)
(ie, branch to REPEAT)
otherwise, execute 2nd part
then branch to BEGIN

NOTE: BEGIN, WHILE, & REPEAT
can be used <u>only</u> within : definitions

This structure is very general.
Either loop body part may be omitted.
If the 1st part is omitted, then this is a loop with a pre-test.

FORTH EDITOR

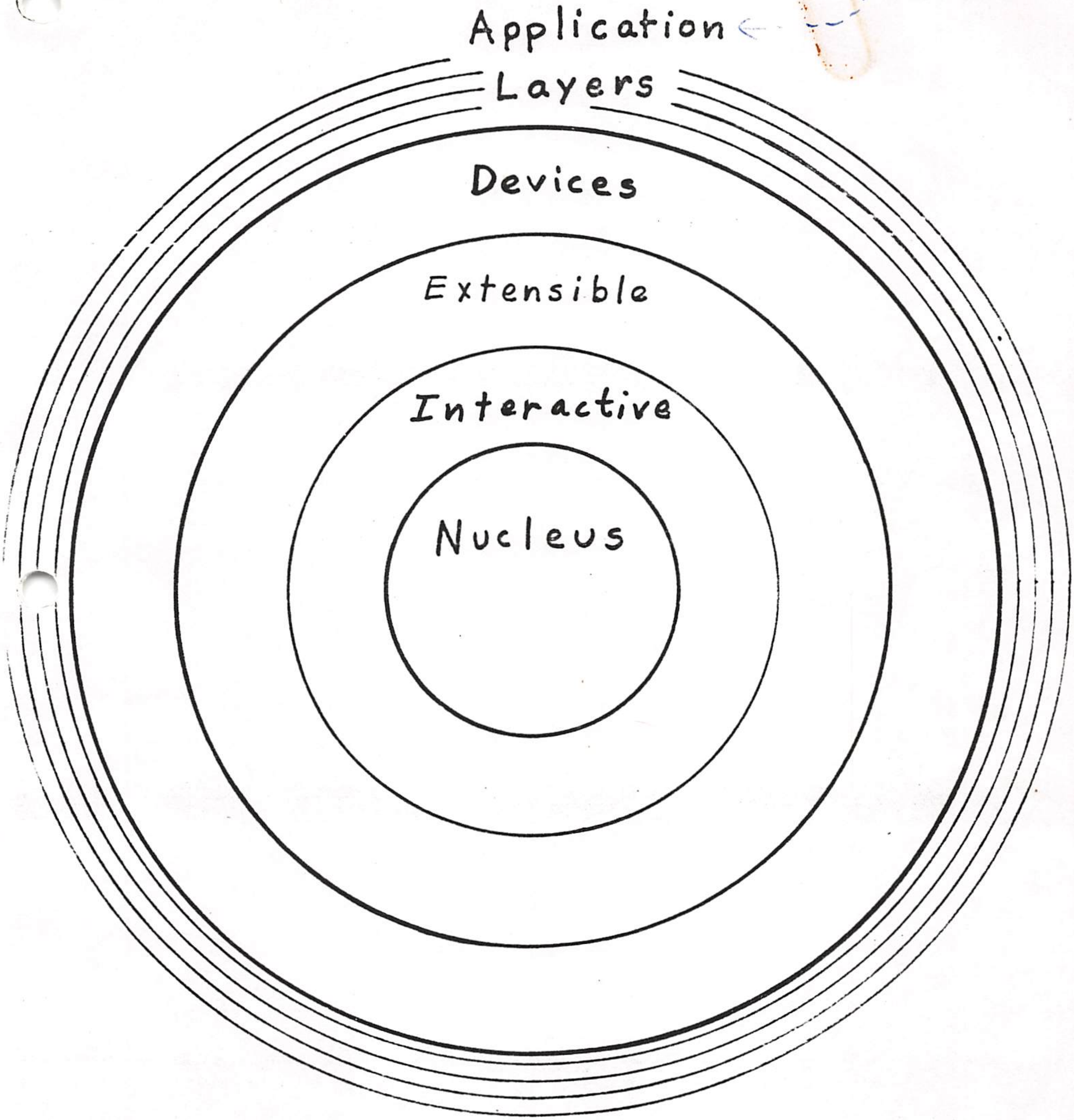


fig FORTH Editor: Bootstrap & Extensions

```
HEX
:  TEXT HERE C/L 1+ BLANKS WORD HERE PAD C/L 1+ CMOVE  ;
:  LINE DUP FFF0 AND 17 ?ERROR SCR @ (LINE) DROP ;
:  -MOVE LINE C/L CMOVE UPDATE ;
:  P 1 TEXT PAD 1+ SWAP -MOVE ;
DECIMAL
```

These words define the elementary editing command "P" which places a line of text on a screen. Blanks are significant. FORTH should respond "OK" after each line is entered. The syntax for its use is:

```
line-number P text-to-be-entered-on-the-line
```

For example, to enter line one of screen 87 type:

```
1 P FORTH DEFINITIONS HEX
```

and type return. FORTH should respond "OK". If you then type:

```
screen-number LIST
```

you should see that text at line number 1.

```
16 LIST
SCR # 16
  0  ( Screen Editor... CLEAR  COPY                                   )
  1  : CLEAR              ( CLEAR screen by number-1 *)
  2      SCR ! 10 0 DO FORTH I EDITOR E LOOP ;
  3
  4  : COPY           ( duplicate screen-2 onto screen-1 *)
  5      B/SCR * UFFSET @ + SWAP B/SCR * B/SCR OVER + SWAP
  6      DO DUP FORTH I BLOCK 2 - ! 1+ UPDATE LOOP
  7      DROP FLUSH ;
  8    EDITOR
  9 : WIPE    ( 1stScr# lastScr# ---      blanks range of screens )
 10     1+ SWAP DO    FORTH I    EDITOR CLEAR    LOOP ;
 11
 12 : RIGHT  ( 1stScr# lastScr# --- )
 13     ( copies range of screens from DR0 to DR1 )
 14     1+ SWAP DO    FORTH I    I FA +    EDITOR COPY    LOOP ;
 15
OK
```

Hex

```
17 LIST
SCR # 17
  0 (    EDITOR:    NEW )
  1 DECIMAL
  2 : NEW    ( line# ---    replaces text from line# until null line)
  3    FORTH    16 0 DO    CR I 3 .R SPACE    I OVER =
  4       IF    QUERY    1 TEXT    PAD 1+    C@
  5          IF ( not null line )   I    EDITOR R    FORTH 1+
  6          ELSE    08 EMIT ( BS )    I    SCR @    .LINE
  7          THEN
  8       ELSE    I    SCR @    .LINE
 9      THEN    LOOP    DROP    ;
 10
 11
```

```
SCR # 148
  0 (  double number support                            WFR-80APR24 )
  1 (  operates on 32 bit double numbers    or two 16-bit integers )
  2
  3 : 2DROP    DROP  DROP  ;  ( drop double number )
  4
  5 : 2DUP    OVER  OVER  ;   ( duplicate a double number )
  6
  7 : 2SWAP    ROT  >R  ROT  R>  ;
  8        ( bring second double to top of stack )
  9 ;S
 10
 11
 12  XXXXX
 13
 14
 15
```

```
SCR # 149
  0 (  String MATCH for editor                        PM-WFR-80APR25 )
  1 : (MATCH)                    ( address-3, address-2, count-1 ---- )
  2                         ( leave boolean matched=non-zero, nope=zero )
  3    -DUP  IF  OVER  +  SWAP    ( neither address may be zero ! )
  4              DO  DUP  C@  FORTH  I  C@  -
  5                  IF  0=  LEAVE  ELSE  1+  THEN    LOOP
  6             ELSE  DROP  0=  THEN  ;
  7 : MATCH   ( cursor address-4, bytes left-3, string address-2, )
  8 ( string count-1, ---  boolean-2, cursor movement-1 )
  9   >R  >R  2DUP  R>  R>  2SWAP  OVER  + SWAP
 10   ( caddr-6, bleft-5, $addr-4, $len-3, caddr+bleft-2, caddr-1 )
 11   DO  2DUP  FORTH  I  SWAP  (MATCH)
 12    IF ?R 2DROP  ~~DROP~~  FORTH  I  SWAP  -  0  SWAP  0  0  LEAVE
 13    ( caddr  bleft  $addr  $len  or else  0  offset  0  0  )
 14      THEN  LOOP    2DROP  ( caddr-2, bleft-1, or 0-2, offset-1 )
 15    SWAP  0=  SWAP  ;
 OK
```

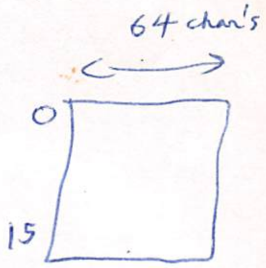
MATCH finds ok but cursor advancement must step over the found string. Parameter return must be incremented by string length!

This patch is untested!

R> - ← ~~Same~~ All as in NAUTILUS ~~Editor~~ Fig Source

scr 71

Scr# LOAD results in executions & compilations at end of screen (or block, if sooner)



figFORTH EDITOR GLOSSARY

#LAG --- addr n 88
Leave address of start of current line in a disk buffer. Also leave n, the # characters following the current cursor position.

#LEAD --- addr offset 88
Leave the address of the start of the current line and the offset to the current cursor position.

#LOCATE --- offset line# 88
Leave the current cursor offset relative to start of line and current line#. Uses contents of R# .

Use LOAD (as & text word) to stuff a lot of

-MOVE addr line# --- 88
Move C/L characters from addr to line# of the current screen on the disk.

— **CLEAR** screen# ---
Erase the designated screen with blanks.

— **COPY** source# dest# ---
Copy contents of screen from source# to dest# .

— **D** line# --- 89
Copy line# of current screen to PAD. Delete it by copying lower lines up one line and erase line 15.

— **E** line# --- 89
Erase line# of current screen with blanks.

H line# --- 89 (non destructive)
Copy line# of current screen to PAD.

— **I** line# --- 91
Insert the contents of PAD after line# of current screen. Lines below line# are moved down one line; the contents on line 15 is lost.

— **L** --- 90
Relist the current screen then the current line followed by the current line number. Uses the contents of SCR .

LINE line# --- addr 87
Leave the address of line# of the current screen.

— **M** n --- 90
Move the cursor by the signed number or characters, n . Print the current line followed by its line number.

— **NEW** line# ---
Print the current screen down to line# . Replace lines with entered text until a null line is entered (ie, (CR) only) then print the remainder of the screen.

— **P** line# --- 91
Put text following P in line# of current screen. Previous contents of this line is lost. — throws away any excess over 64 char's

Note: any overhang is lost

C Copy text following the space after C at the cursor pos'n, pushing the remaining line contents to the right ← "insert" (same as MVP FORTH's I)

TRICK:

X Extract text following the space after X, sliding the remainder of the line to the left (same as MVP FORTH's D)

357 CONSTANT APPLICATION

Note: If right end of line has a char, and left end of next line has a char, then the "word" overlapping the line boundary will get "sucked" into the current line and the next line will be slid to the left accordingly

APPLICATION LOAD

E2
Fig

R line# --- 91

Copy line at PAD to current screen at line# .

S line# --- 89

Spread lines of the current screen. line# becomes blank, the previous contents of this line is moved down one line as are lower lines. Line 15 is lost.

T line# --- 90

Make line# the current line of the current screen. Move the cursor to the beginning of this line, copy it to PAD , then print the line followed by its line number.

TEXT delim --- 87

Move text from TIB to PAD until delim character is encountered.

TOP --- 91

Move cursor to the beginning of the current screen.

WHERE in blk# --- 88

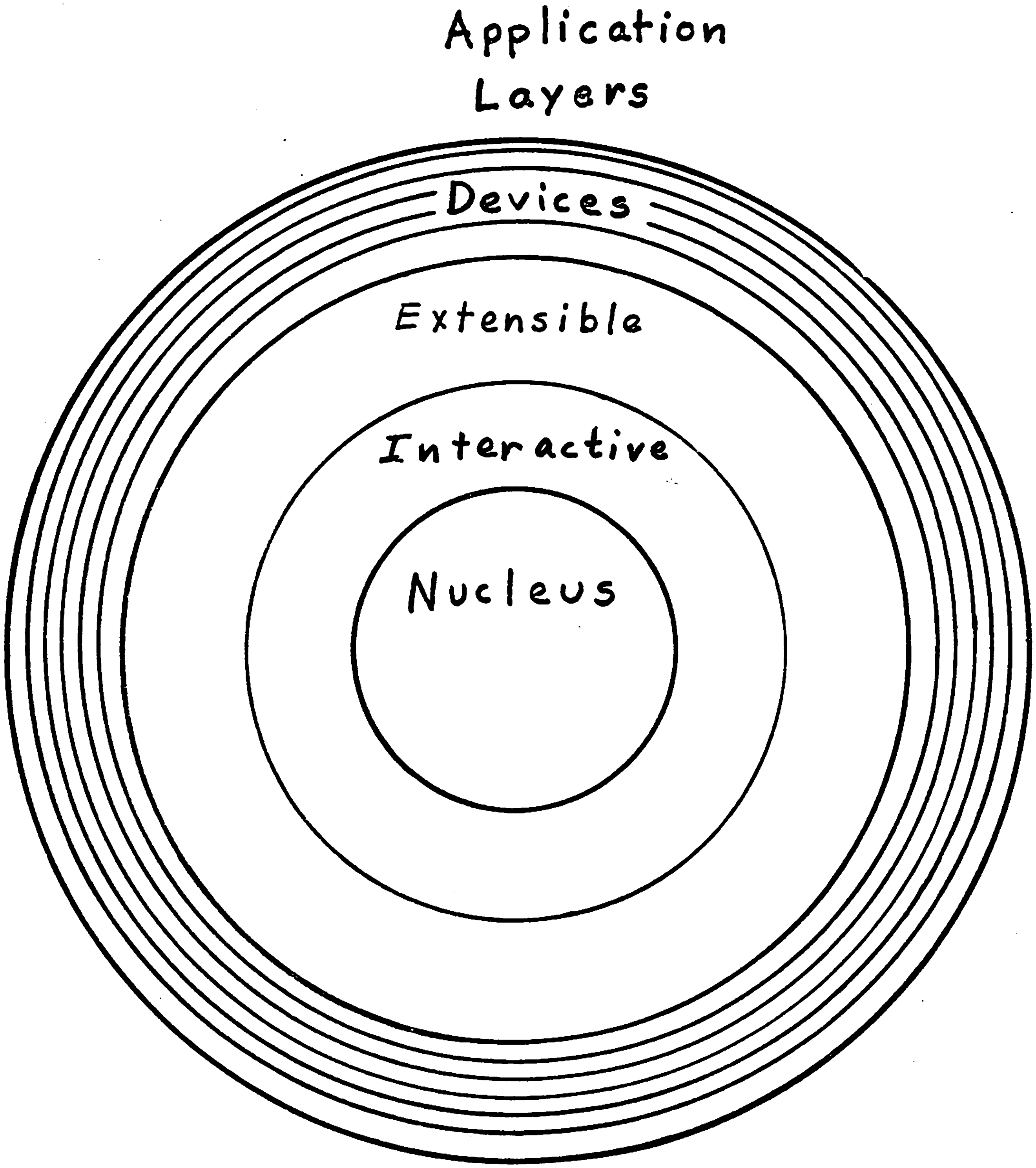
List the screen corresponding to blk# and type the line where in is pointing (the contents of IN) inside the screen. Used after a compilation error from mass storage (ie, a LOAD).

;S forces immediate termination of loading of the screen

on line 0, add a screen description & date, author I.D.

B.O.1 (G3 f

MASS STORAGE



MASS STORAGE
a generalized interface

	single density 8" floppy	double density 8" floppy	hard winchester	tape
	256K bytes	512K bytes	10M bytes	22.8M bytes
blocks	0 .. 249	250 .. 749	750 .. 10 500	10 501 .. 32 767

example only

example

block # mass storage address of 1024 byte "block" ← directs you to appropriate device

A 16 bit block# addresses 33.5 M bytes
A 32 bit block# addresses 2.2 tera bytes

to a program — to a device

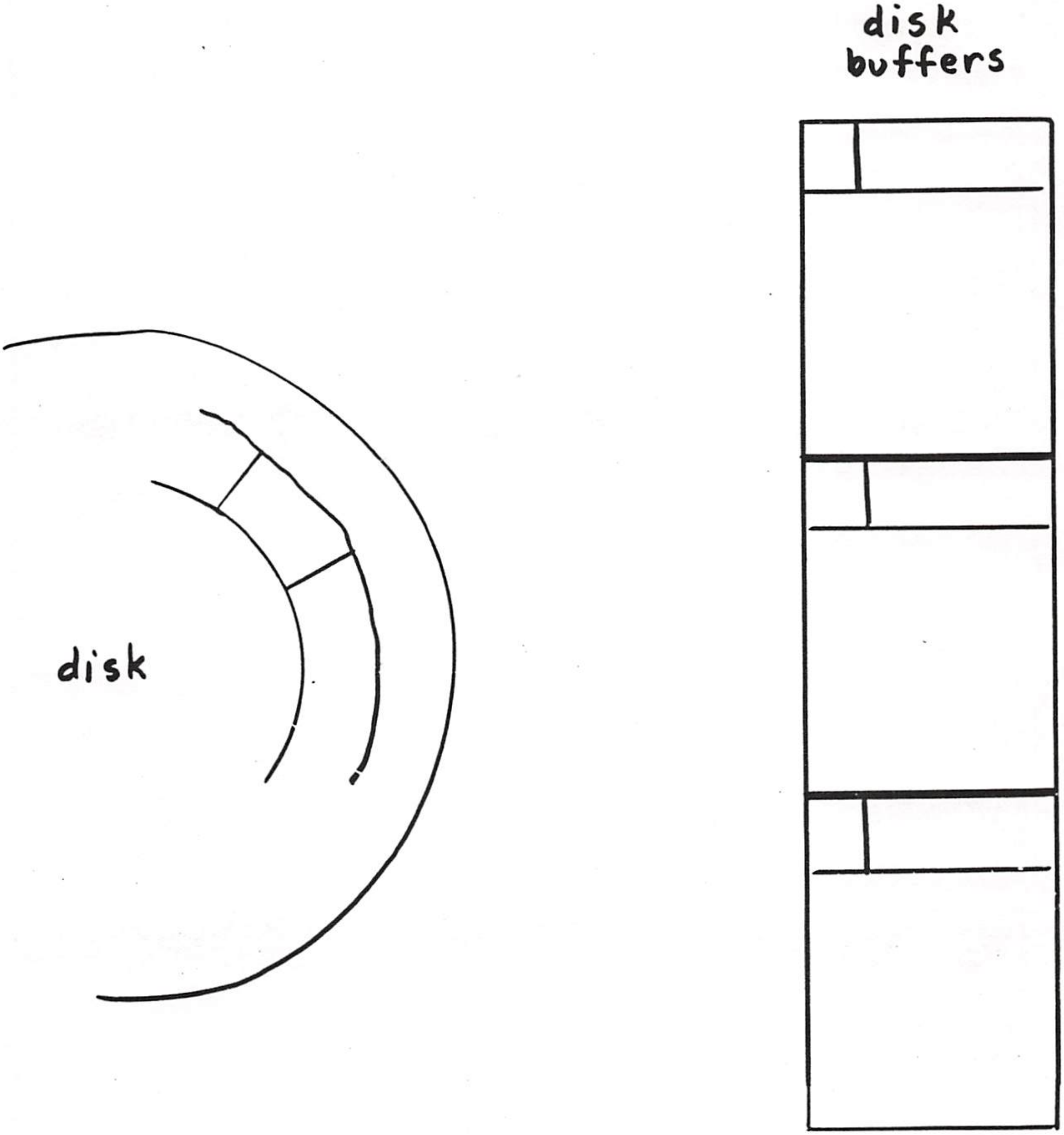
logical block # — mapping process → physical block #

$\underbrace{\text{mapping process}}_{\text{OFFSET @ +}}$

Add B/BUF (const.) bytes/buffer

B1

DISK ACCESS

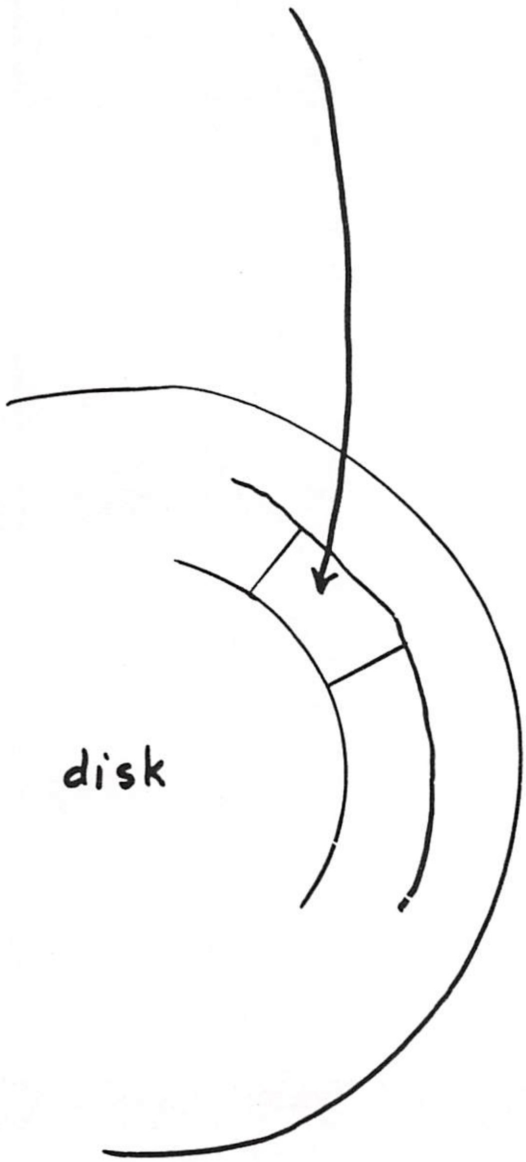


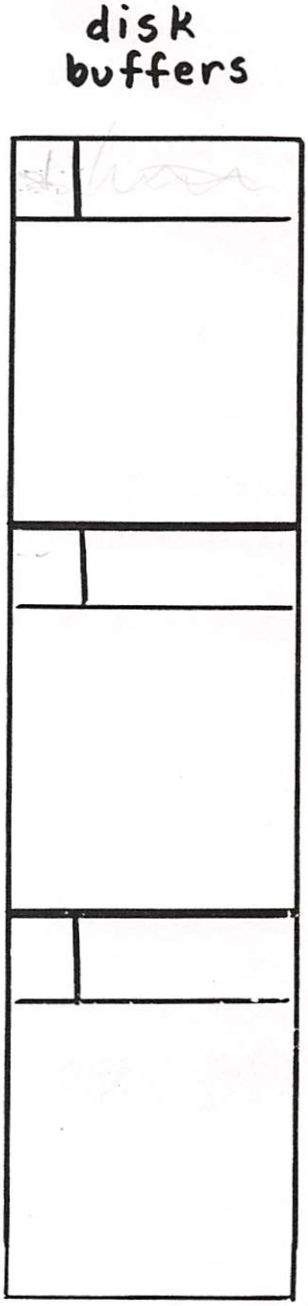
no. buffers & size defined by assembler equates

128 bytes in 8080 version

DISK ACCESS

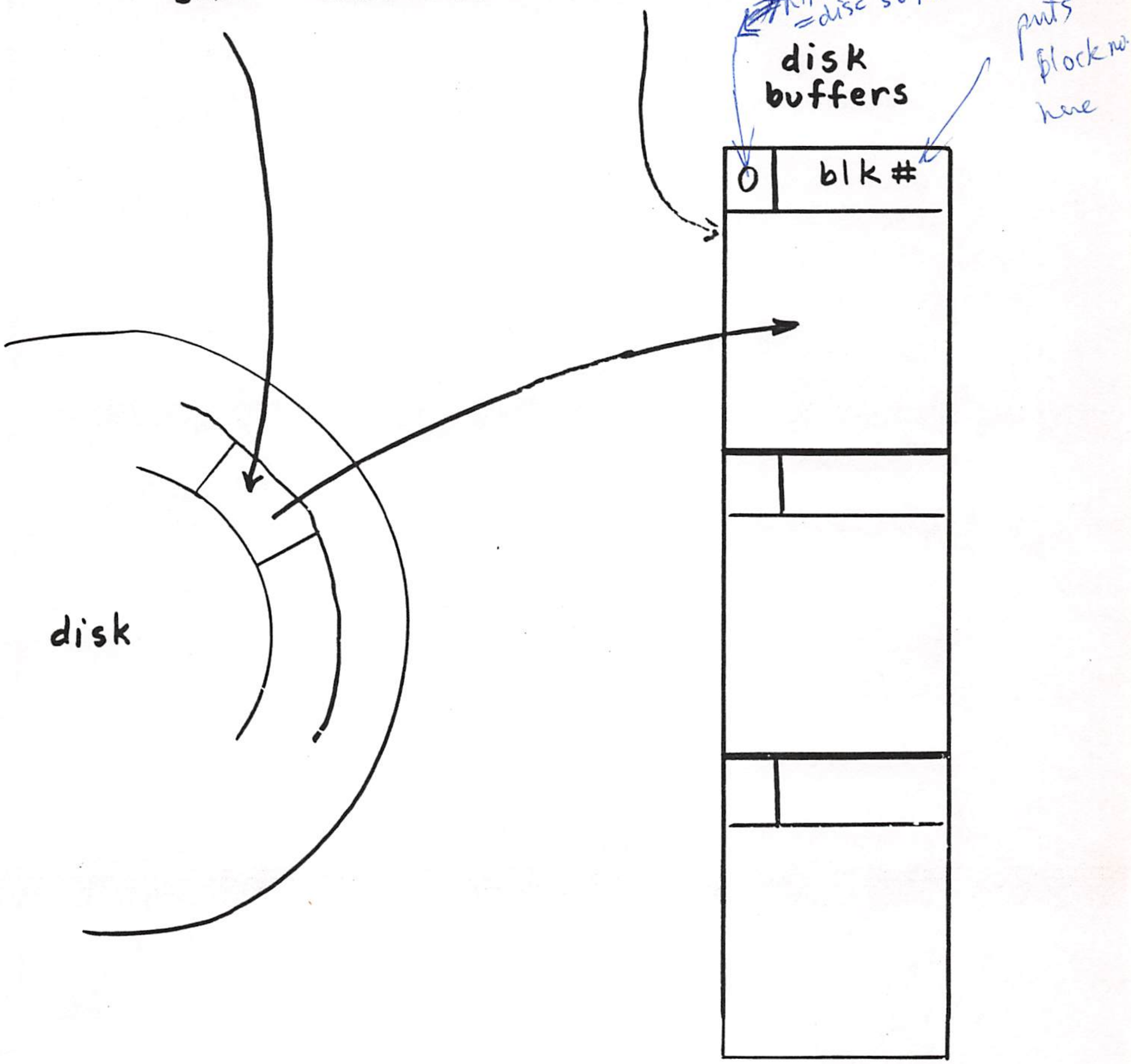
blk# BLOCK (takes care of I/O operation)





DISK ACCESS

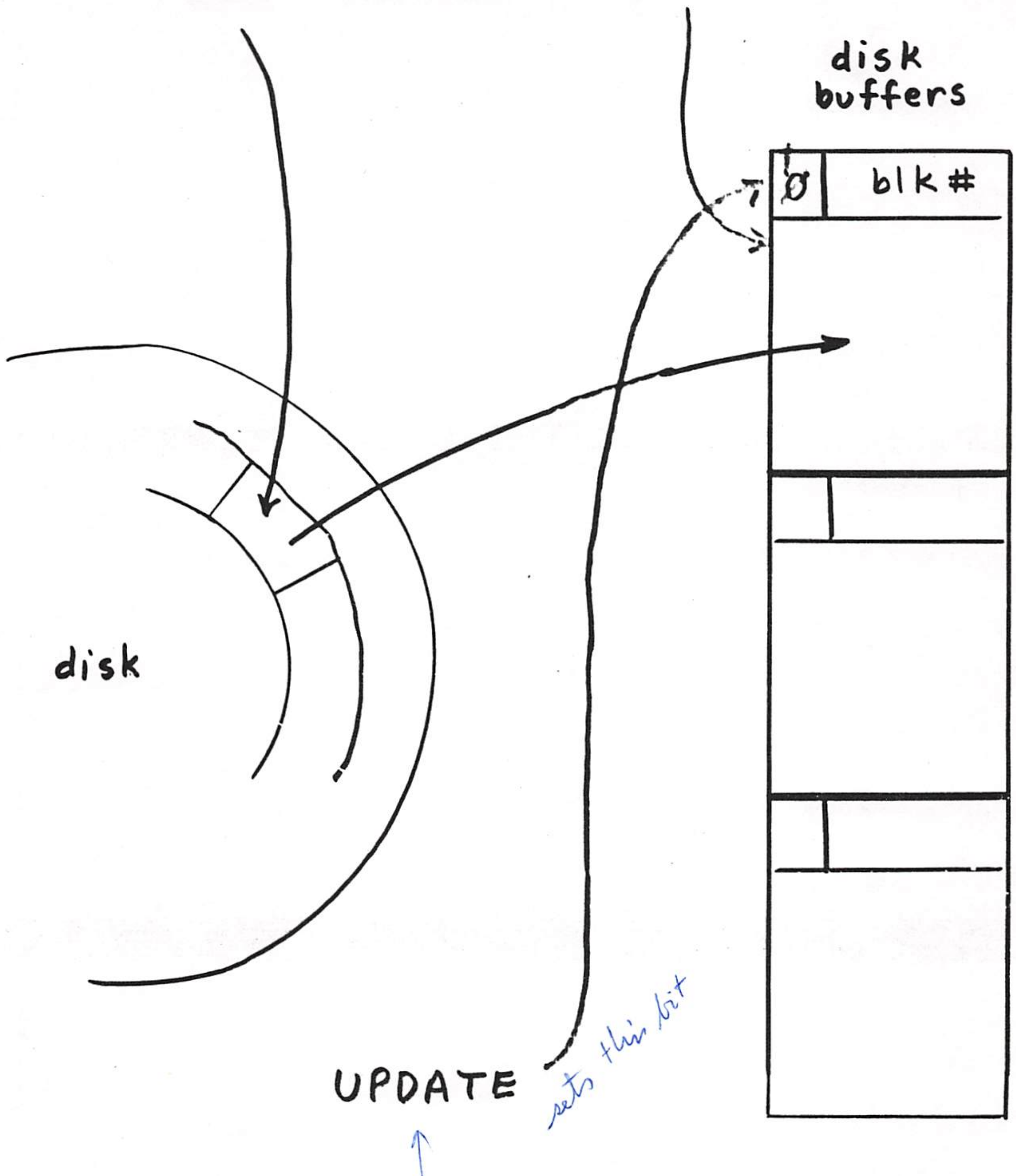
blk# BLOCK --- addr



B2
C

DISK ACCESS

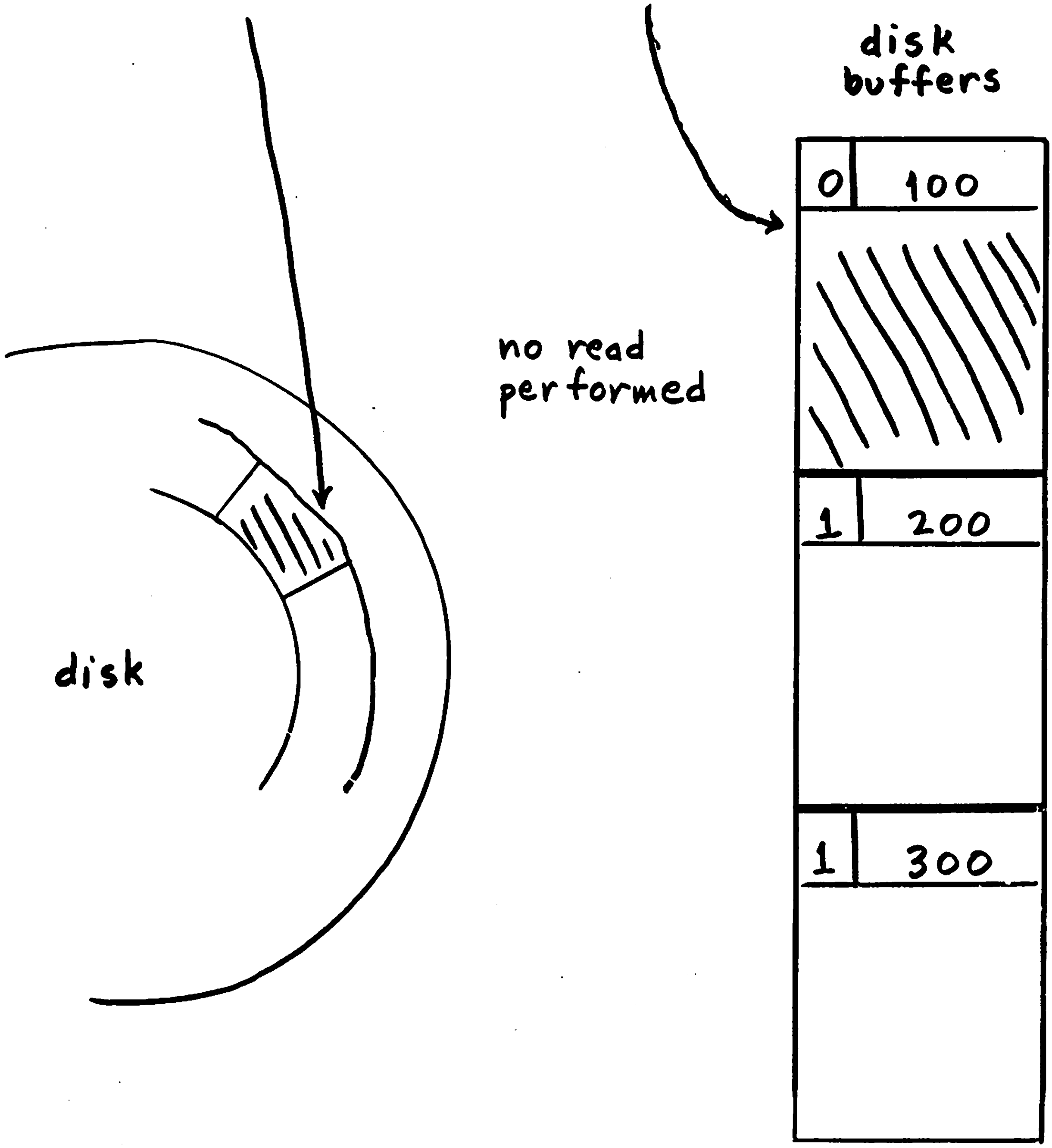
blk# BLOCK --- addr



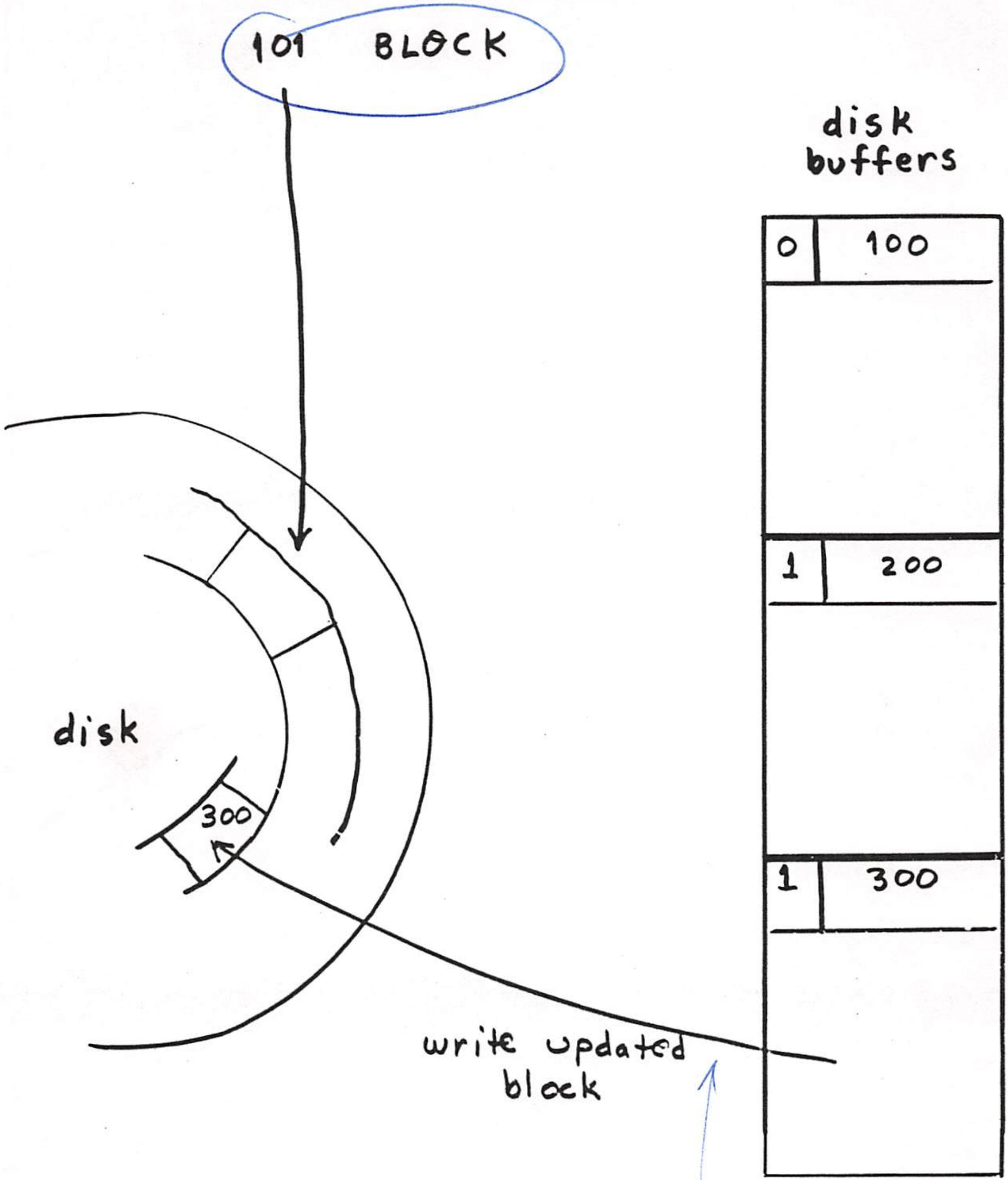
must execute to tell system buffer
is "dirty"

DISK ACCESS

100 BLOCK --- addr



DISK ACCESS

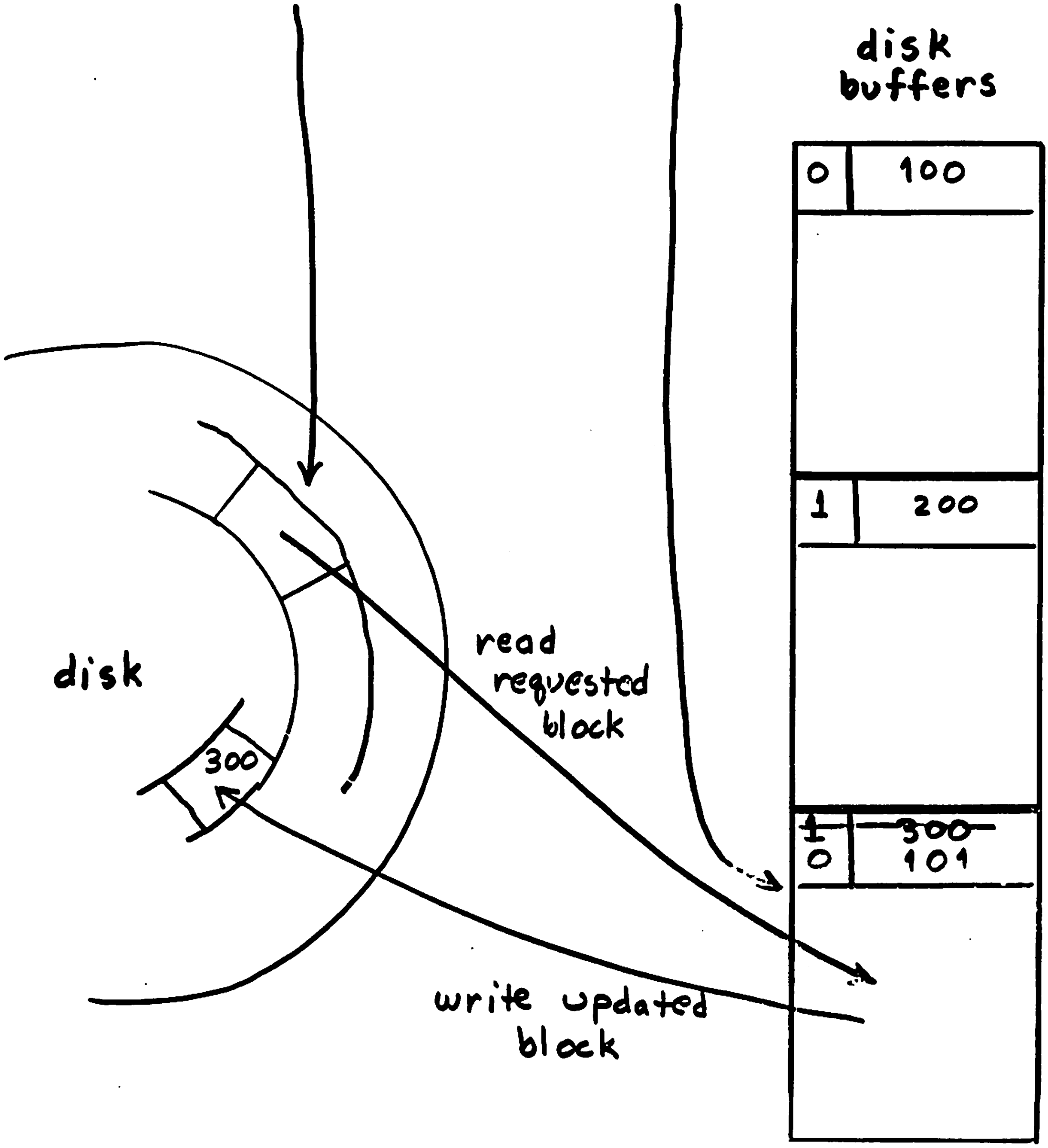


variable?
PREV points to "oldest"

∃ an algorithm to
decide which buffer
to clobber & reuse

DISK ACCESS

101 BLOCK ··· addr



disk operations:

FLUSH forces all UPDATEd buffers to be written to mass storage

MUST be executed:

before changing disks

before powering down

before restarting system

can be executed:

after editing

EMPTY-BUFFERS writes 0's into <u>all</u> disk buffers without writing any UPDATEd buffers to disk

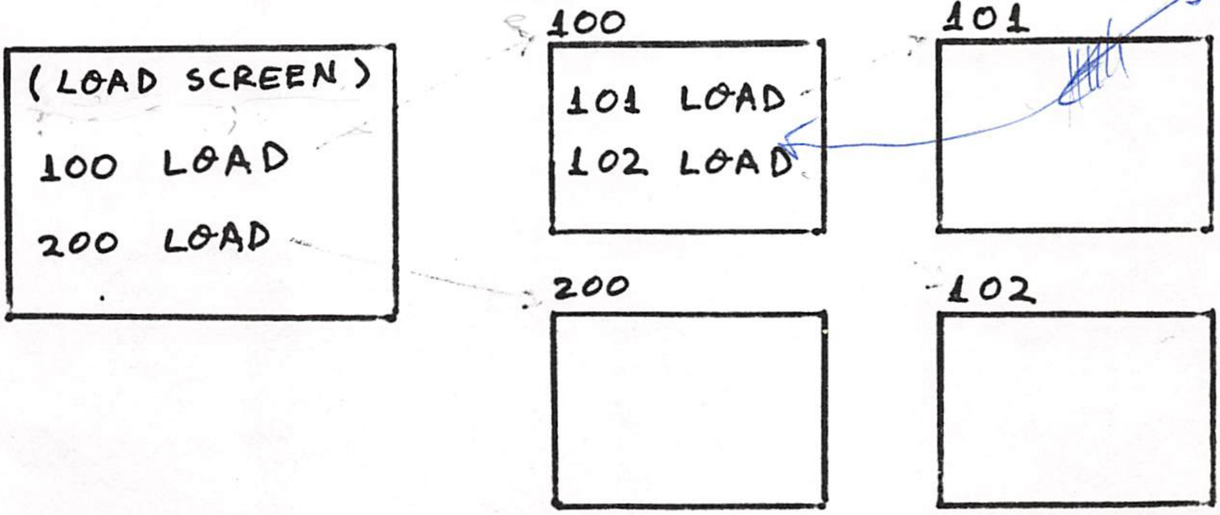
Buffers are shared by all users.

screen# **LIST** displays screen at terminal (or other device)

screen# **LOAD** interprets & compiles screen

nestable:

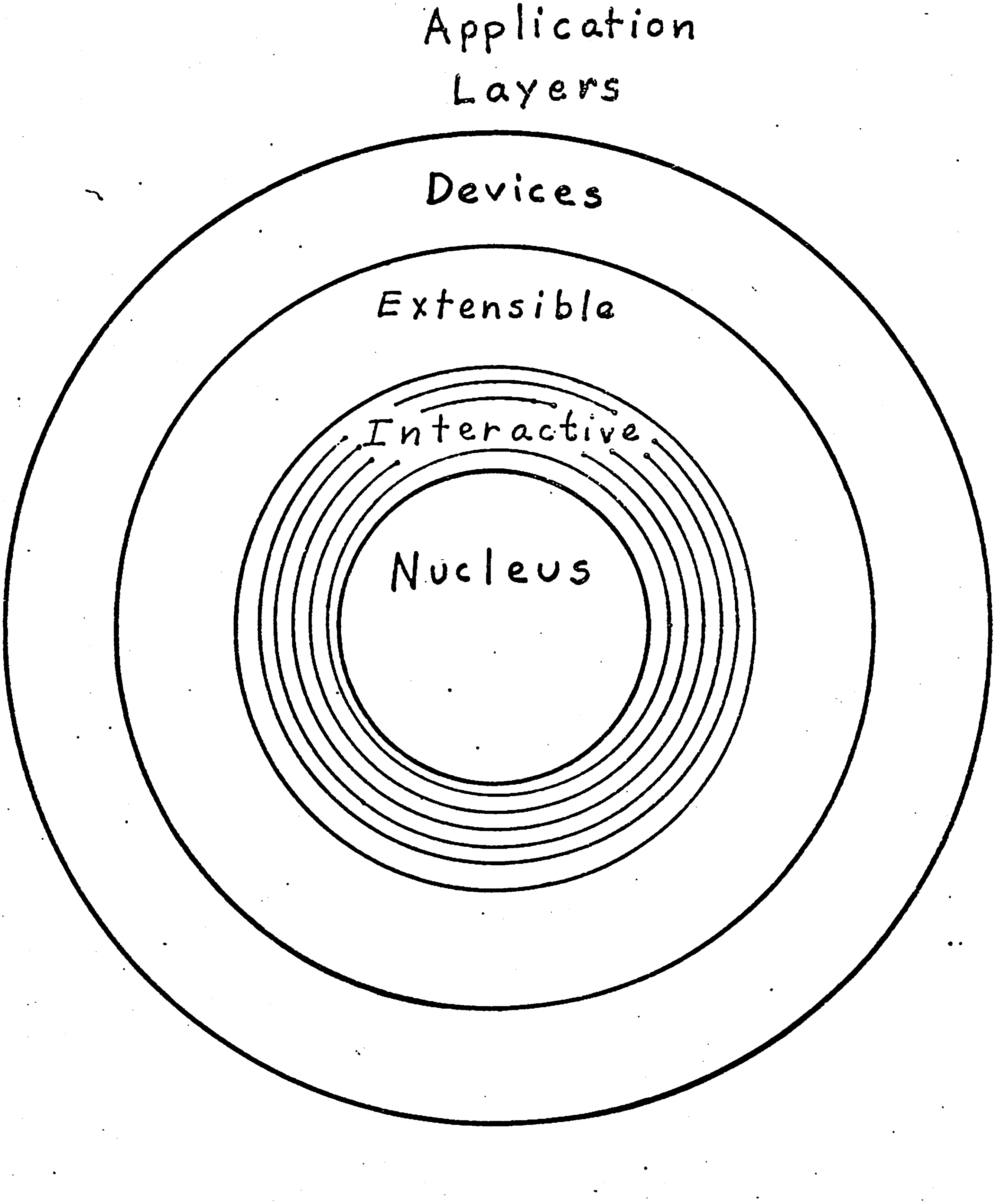
stick in DUP . LOAD all over the place to debug



within a screen **;S** terminates **LOAD.**

Start screen, end screen, INDEX -types out lines 0

STRING OPERATIONS



STRING HANDLING

Input

Conversion

- Numeric string ↔ binary
- Copying
- Formatting

Comparison

Output

STRING INPUT

Read a string of characters from
your terminal
(or a communications channel)

dest_addr max#chars **EXPECT**

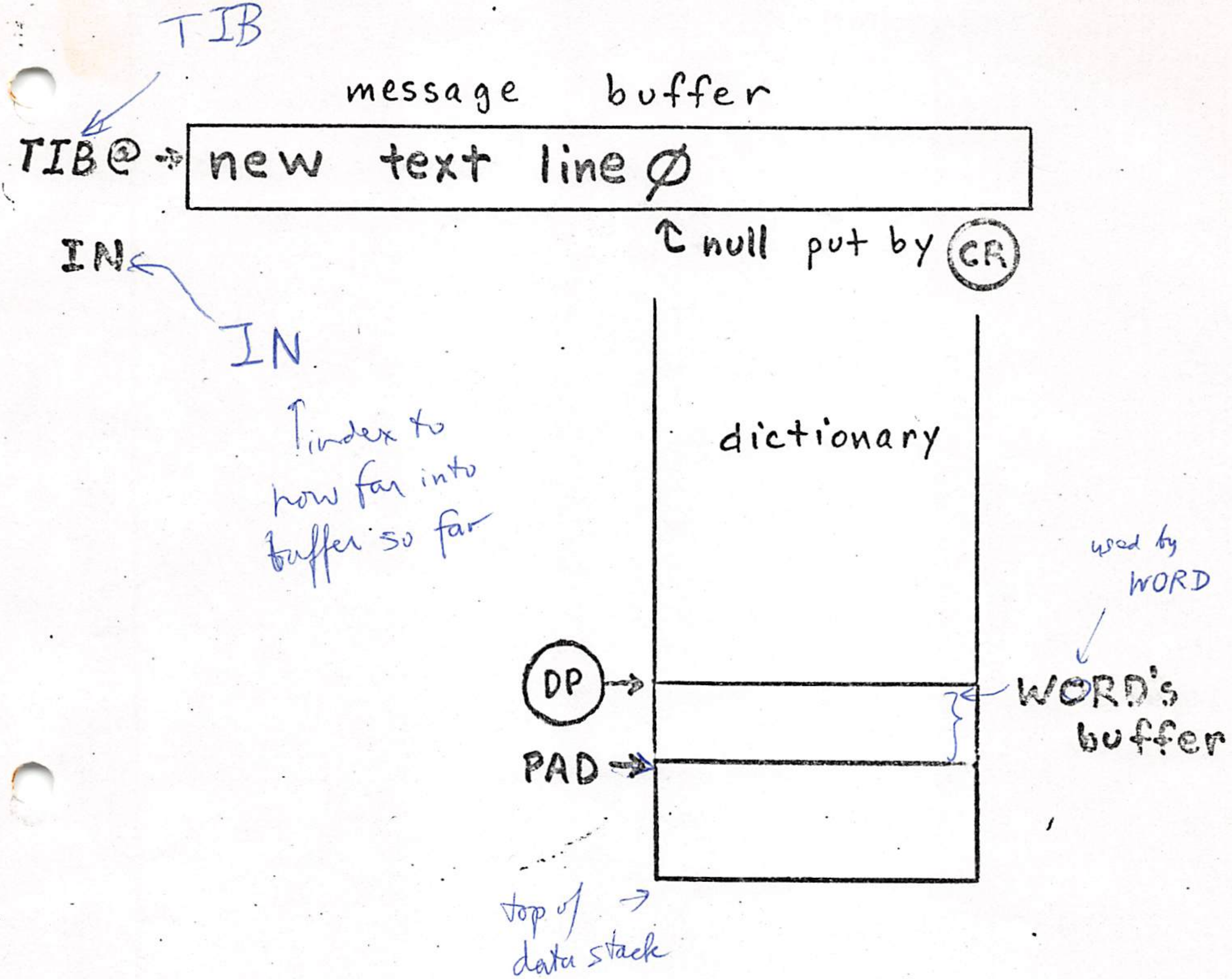
in FIG, built as loop on KEY

in FORTH INC, based on interrupts & KEY is 1 EXPECT

performs BackSpace editing

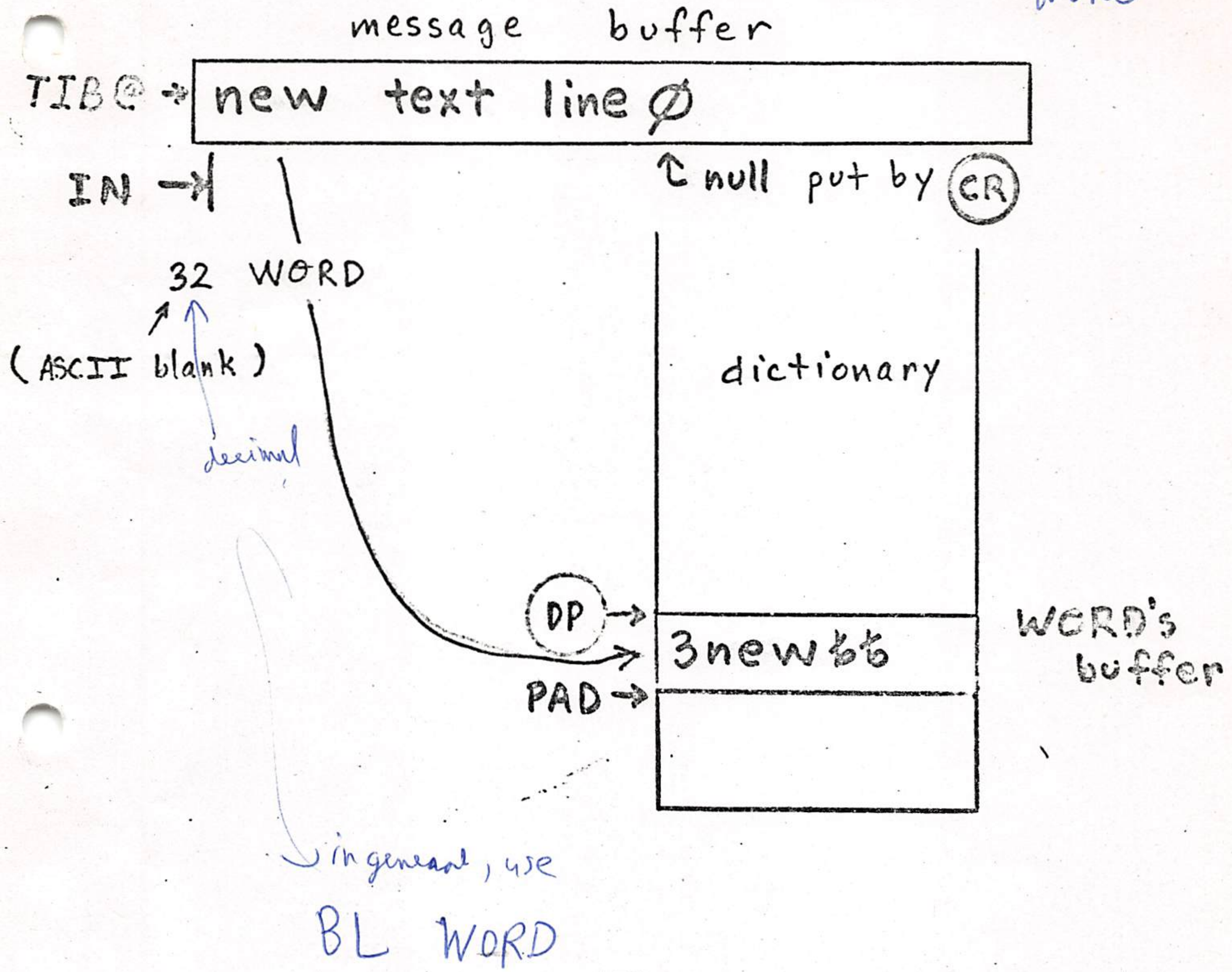
terminates when { max# chars entered
or
(CR) entered }

52
b
fig


TIB
message buffer
TIB@ →
new text line 0
null put by CR
IN
IN
index to how far into buffer so far
dictionary
used by WORD
DP →
WORD's buffer
PAD →
top of data stack

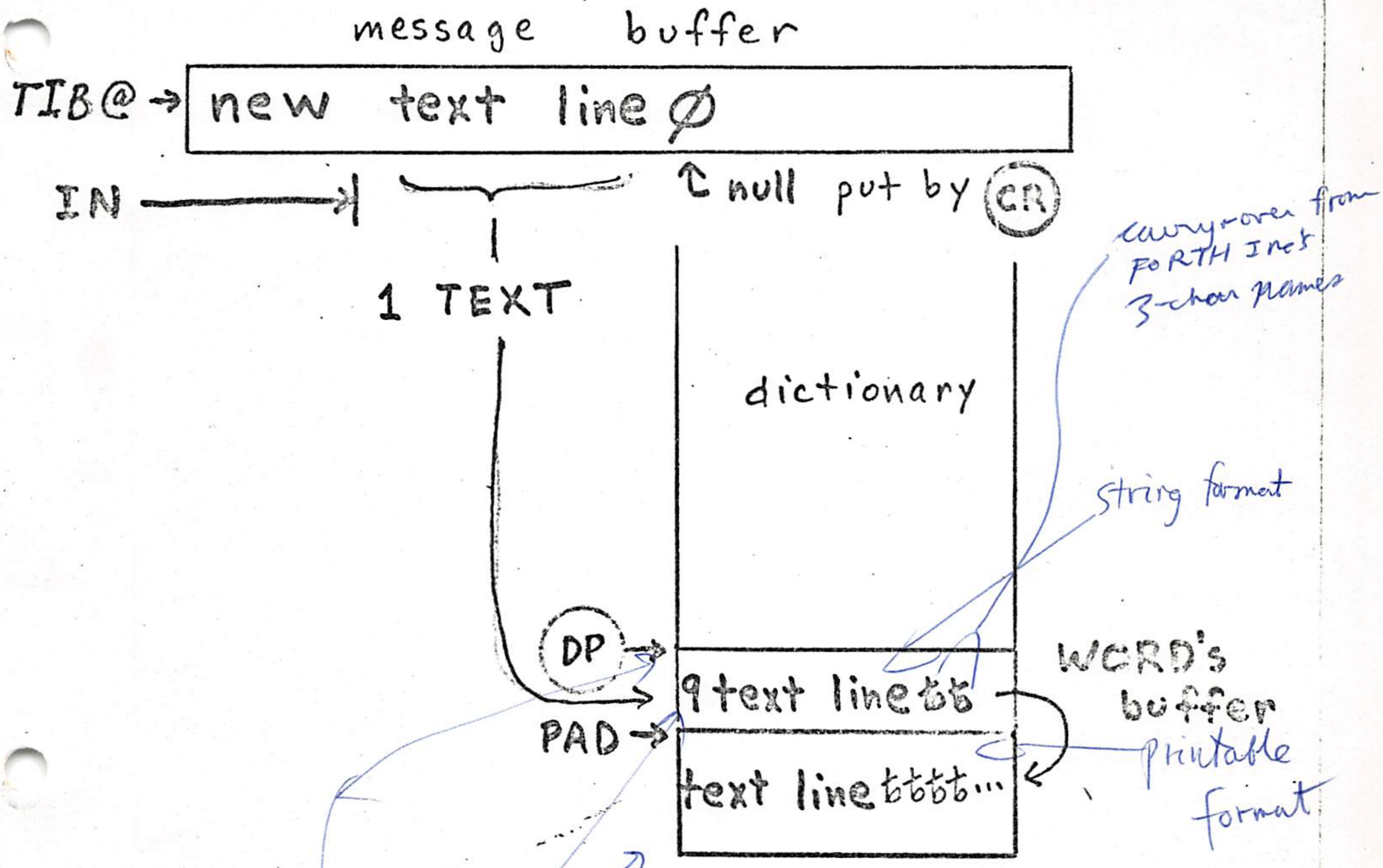
52
c
fig

Action of WORD



s2
d e
fig

Action of TEXT

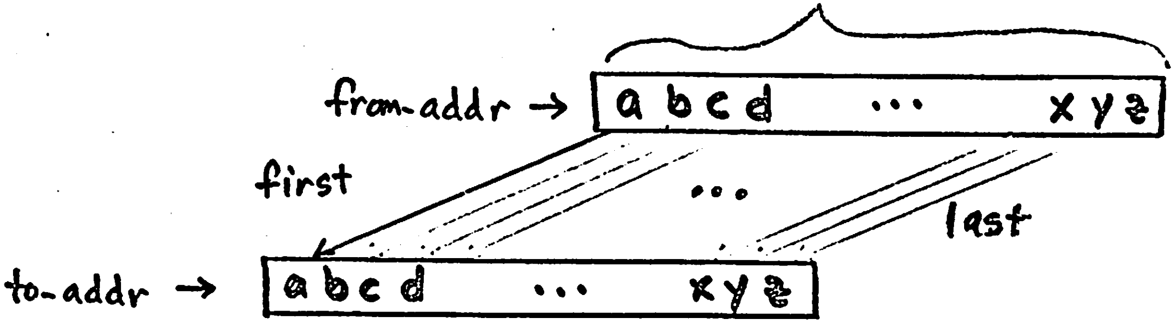


gives both forms (of formats)

Warning!
Gets ~~erased when~~ written on by next keyboard entry
or by anything that uses WORD

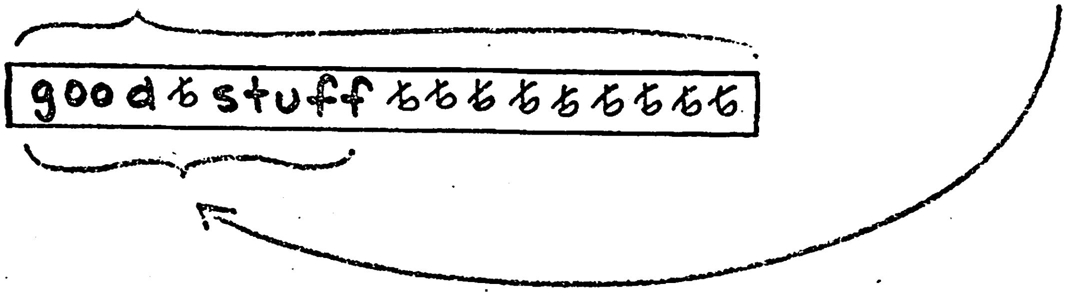
Copying strings

from-addr to-addr #bytes **MOVE**



Remove trailing blanks

addr before#bytes **-TRAILING** --- addr after#bytes



Initialize strings (or arrays)

addr #bytes **BLANKS** stores blanks

addr #bytes **ERASE** stores zeros

addr #bytes character **FILL** stores "character"
starting at "addr"
for "#bytes"

SS
rd
fig

STRING OUTPUT

Write characters to
your terminal
(or a communications line)

FORTH INC. : Interrupt driven routine

addr #bytes **TYPE** write string — FIG : TYPE is loop on EMIT

FORTH INC: 1 TYPE

chr-value **EMIT** write single character

CR write Carriage Return (Line Feed)

SPACE write single blank

#spaces **SPACES** write several blanks

." wow **"** write string

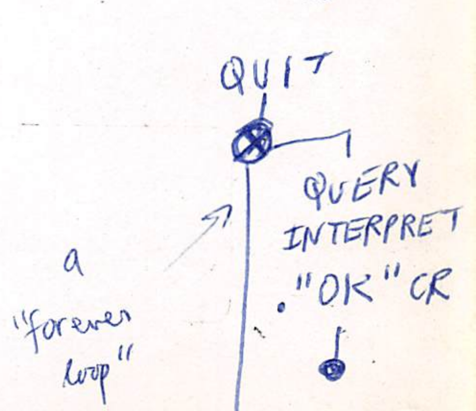
can call QUERY

Example

BEGIN (optional)

: ECHO TIB @ 80 EXPECT
0 IN !
1 ~~TEXT~~ better WORD
HERE COUNT
TYPE CR 0 END

can also use



STRING CONVERSION

ASCII character to numeric value:

use:

ASCII chr --- { chr_value if interpreting
else compiling, then
chr_value is compiled
as a 16 bit literal

?

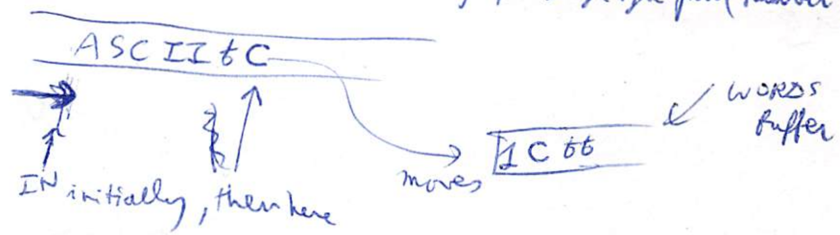
definition:

```
: ASCII    BL WORD    HERE 1+ C@
           LITERAL  ;   IMMEDIATE
```

~~makes~~ wait till we learn about compilers

↖ Compiles into dictionary the ~~single~~ byte pair (number)

On execution,



Numeric string to binary conversion:

addr-string **NUMBER** --- dvalue

and following conversion,

variable **DPL** contains

-1 if numeric string was not punctuated (converted value is in the 16 bit signed integer range)

else the number of digit characters following the last punctuation (32 bit converted value)

see p.60 USING FORTH (old version) for 4 ways to print 32 bit, 31 bit
16 15 bit no's

use of HOLD to stick in special char's in format
e.g. print 400.0 deg °C in our temperature conversion example

: .DEG SWAP OVER DABS
<# # ASCII . HOLD #S SIGN #> TYPE SPACE

(loop around #

Note: (can put any other "action" here
e.g. $ XX, XXX, XXX.XX (without leading ','s

Building the word " . "

e.g.

ABS ⊙

DABS

SIGN

will print a leading sign

: . DUP S->D <# #S #> TYPE ;

need copy of signed value

see p.51 USING FORTH

So if you want financial figures specially formatted

Pictured numeric output conversion:

Convert 123_{10} to a numeric string:

$$\frac{123}{10} = 12 \text{ remainder } \boxed{3}$$

$$\downarrow$$

$$\frac{12}{10} = 1 \text{ remainder } \boxed{2}$$

$$\downarrow$$

$$\frac{1}{10} = 0 \text{ remainder } \boxed{1}$$

Convert unsigned 32-bit value to 10 digits:

```
: UD.10  <#  # # # # # # # # # #  #>
  TYPE ;
```

← uses PAD

<#: starts numeric conversion

#: converts one digit

#>: ends conversion — leaves what TYPE wants

only works on pos no's

```
3.1415928   UD.10 (CR)   0031415928 OK
```

#S will do a complete conversion of the number string on the stack

eg. <# #S #>

```
: #S BEGIN # DUP 0. = UNTIL )
```

16 bit version

see Scr. 75 for 32 bit version

USER allows "private" variables

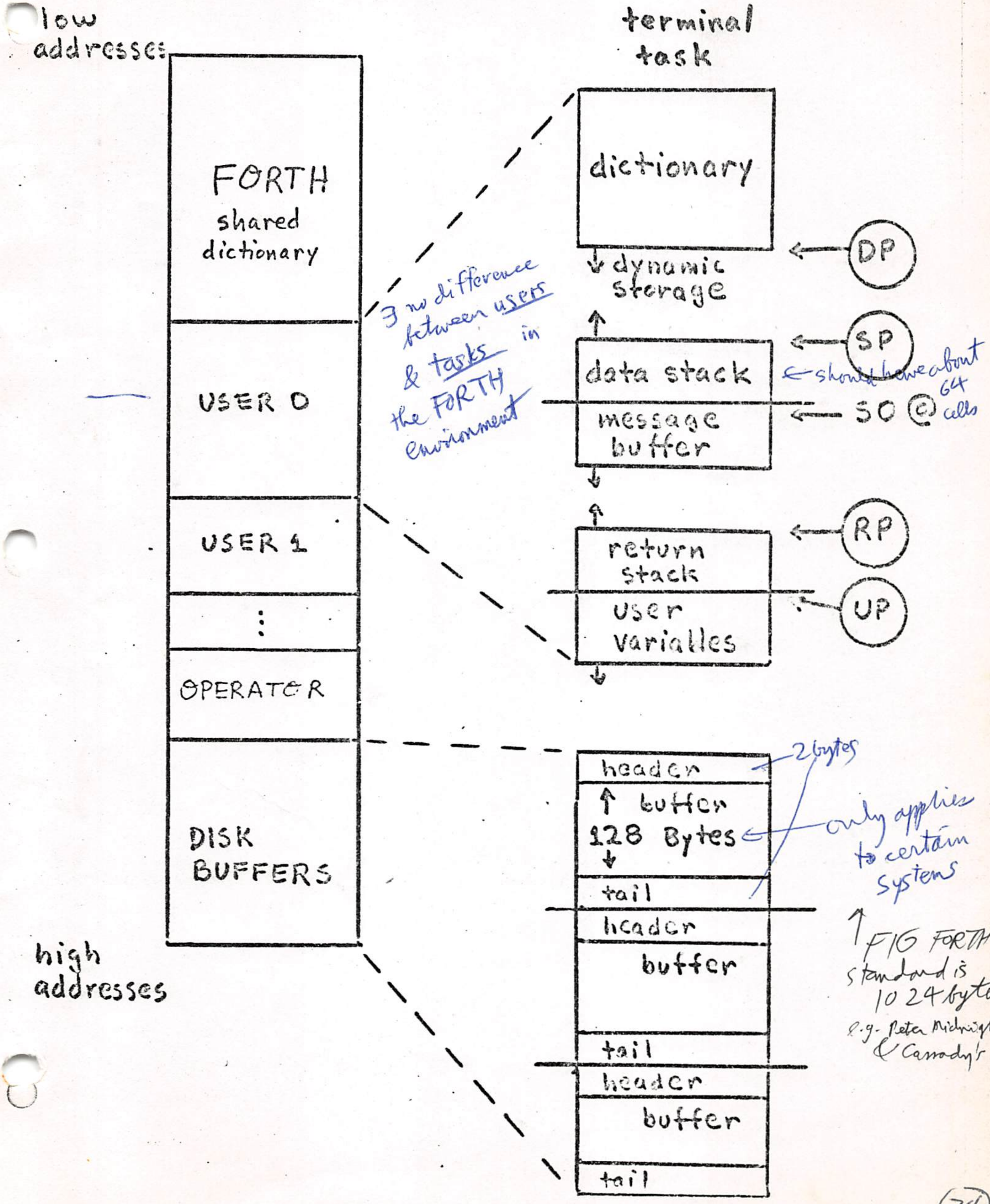
e.g. offset USER BASE

- creates a single head in dictionary

& execution time procedure which adds the offset of the current user

M2b
fig

MEMORY ALLOCATION



∃ no difference between users & tasks in the FORTH environment

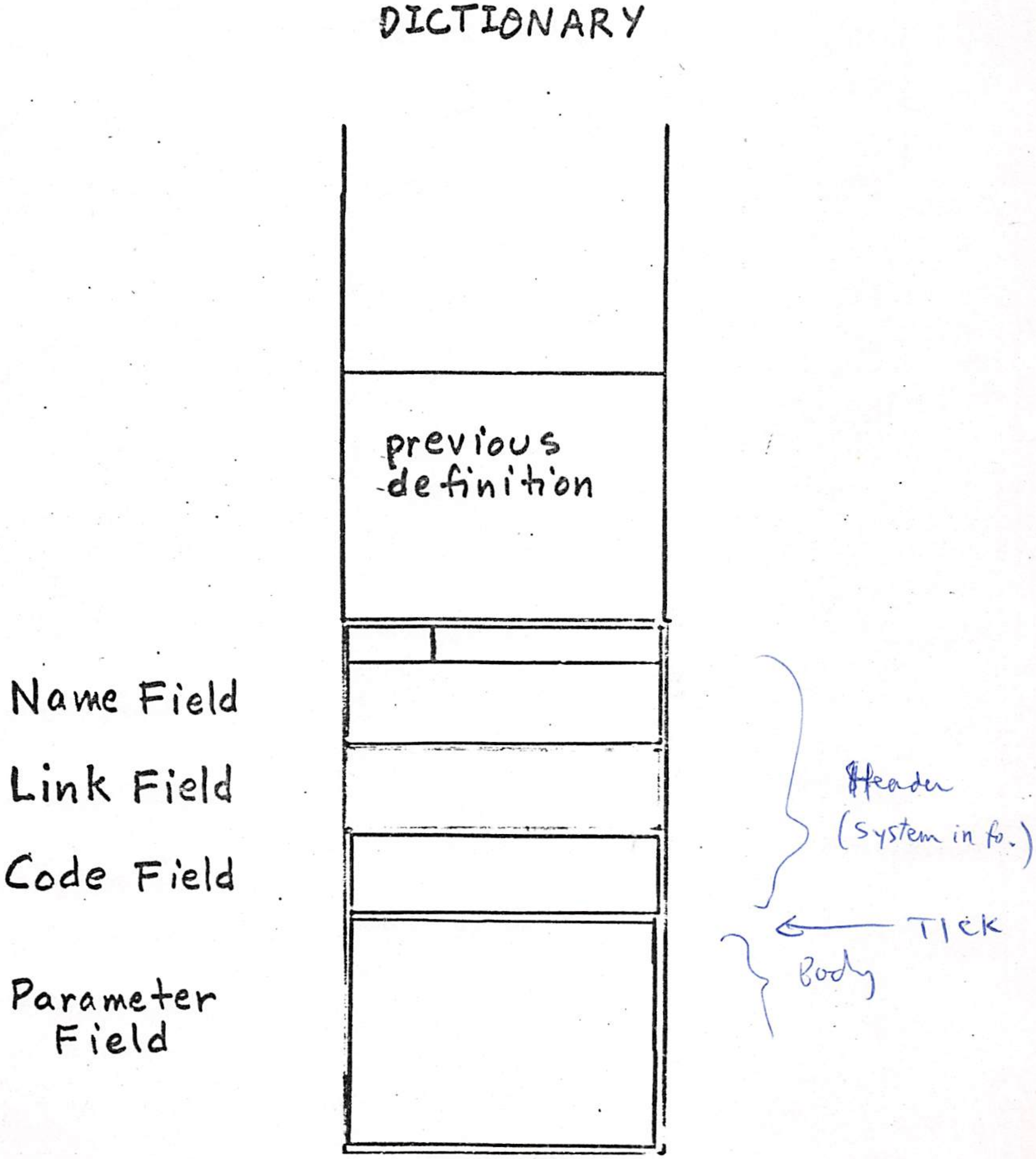
should have about 64 cells

2 bytes

only applies to certain systems

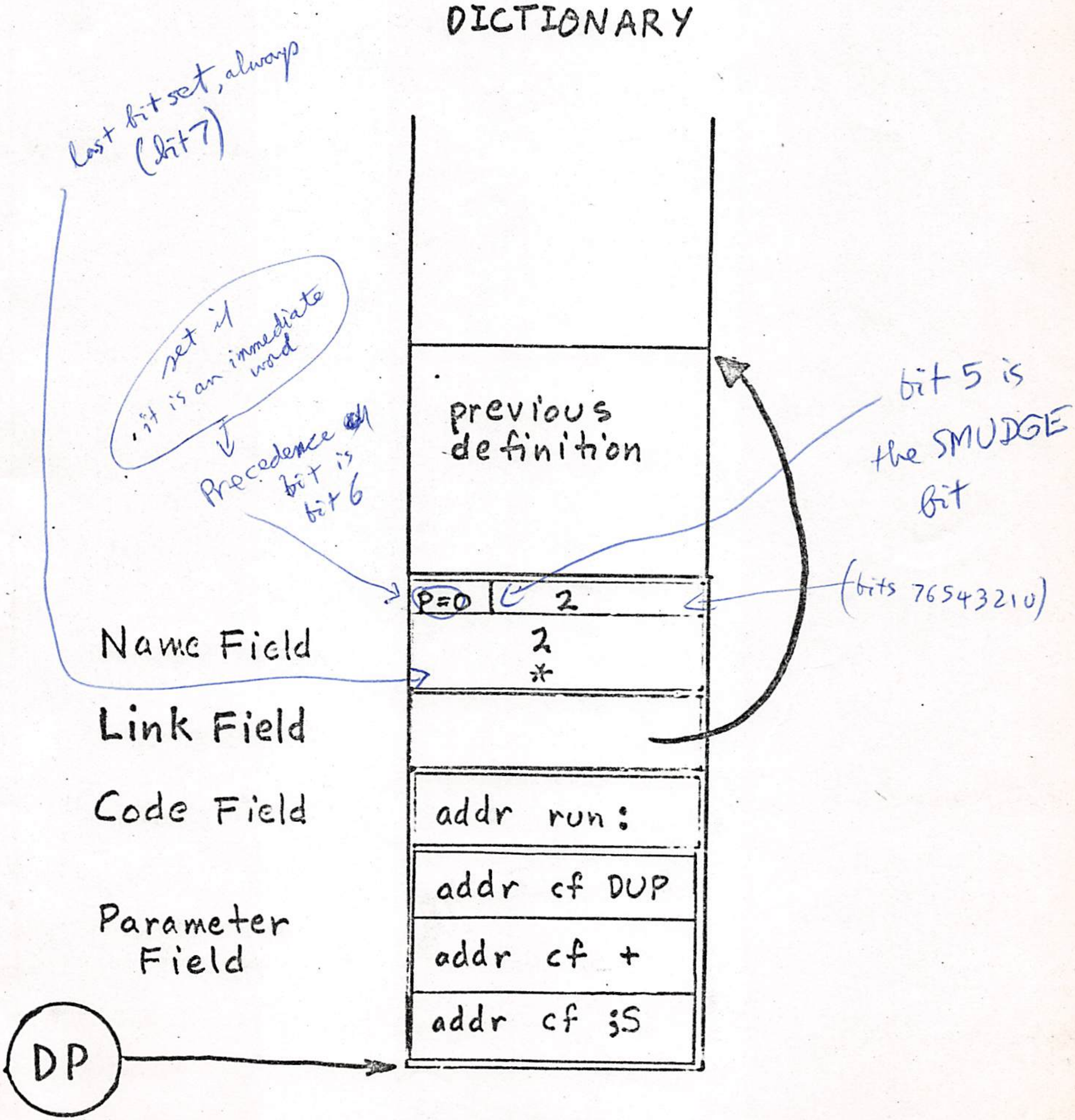
FIG FORTH standard is 1024 bytes
e.g. Peter Midnight's & Cassady's

Dictionary Definition Format



Dictionary Definition
Format

: 2* DUP + ;



TEXT INTERPRETATION

Application Layers

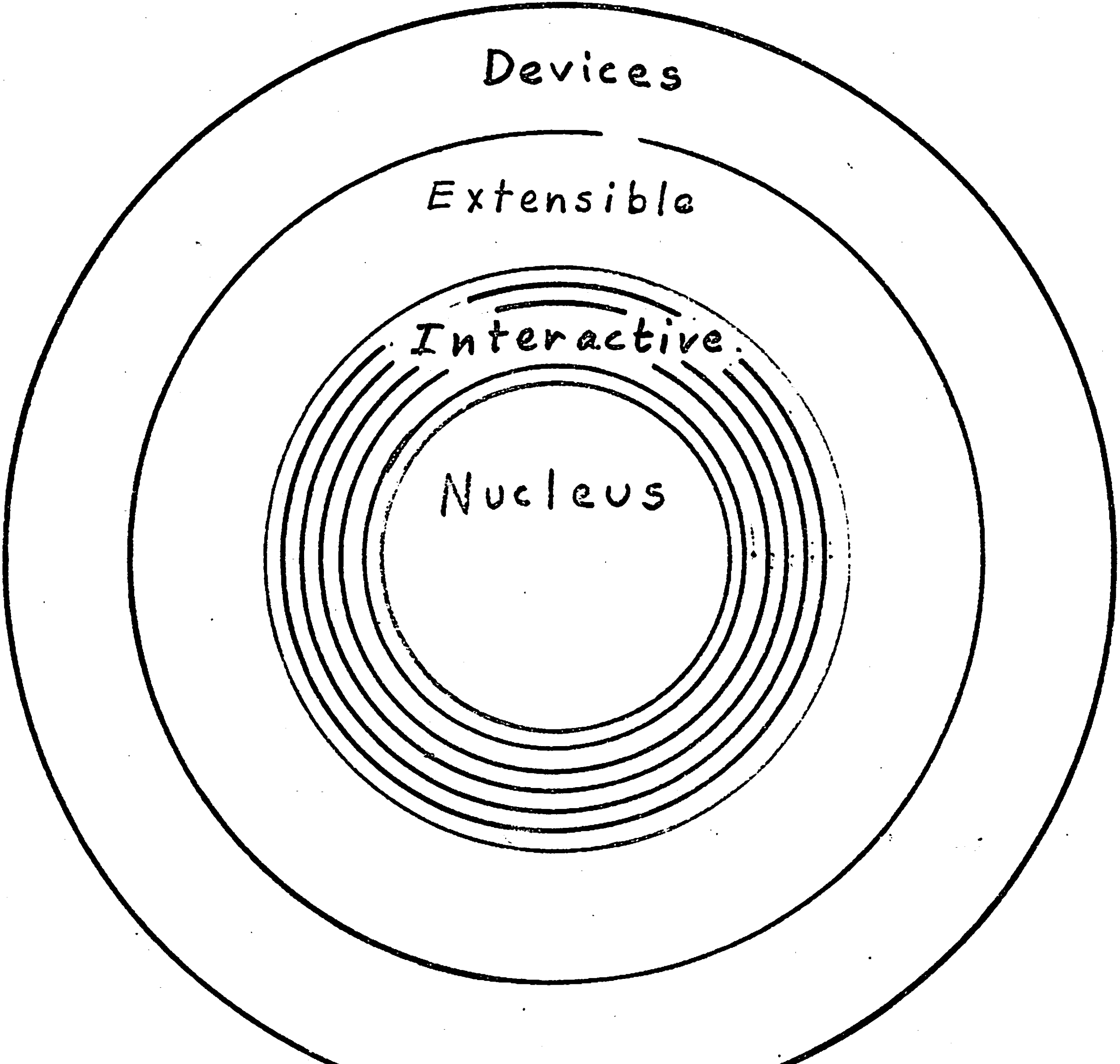
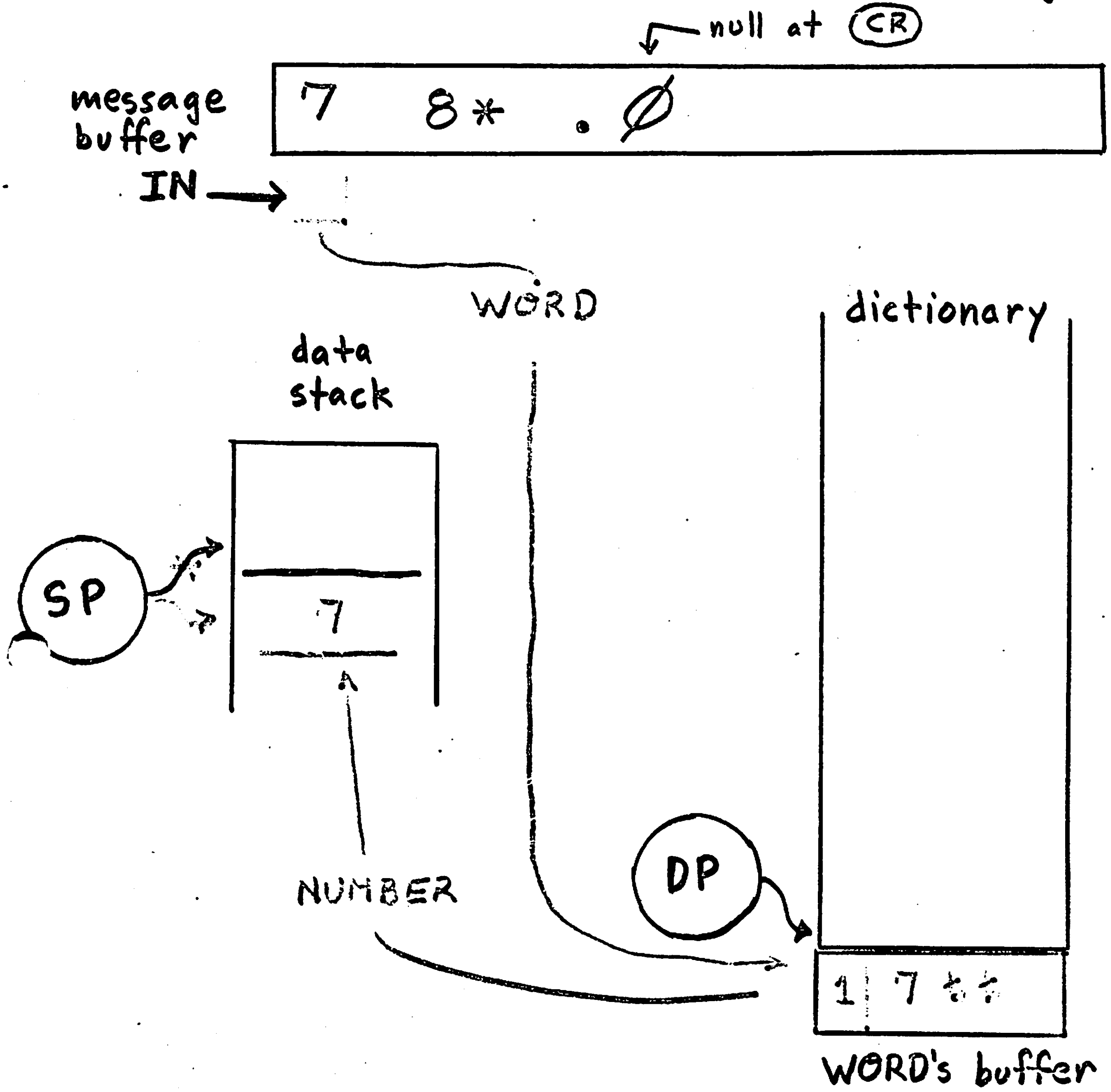


Fig 119

TEXT INTERPRETATION

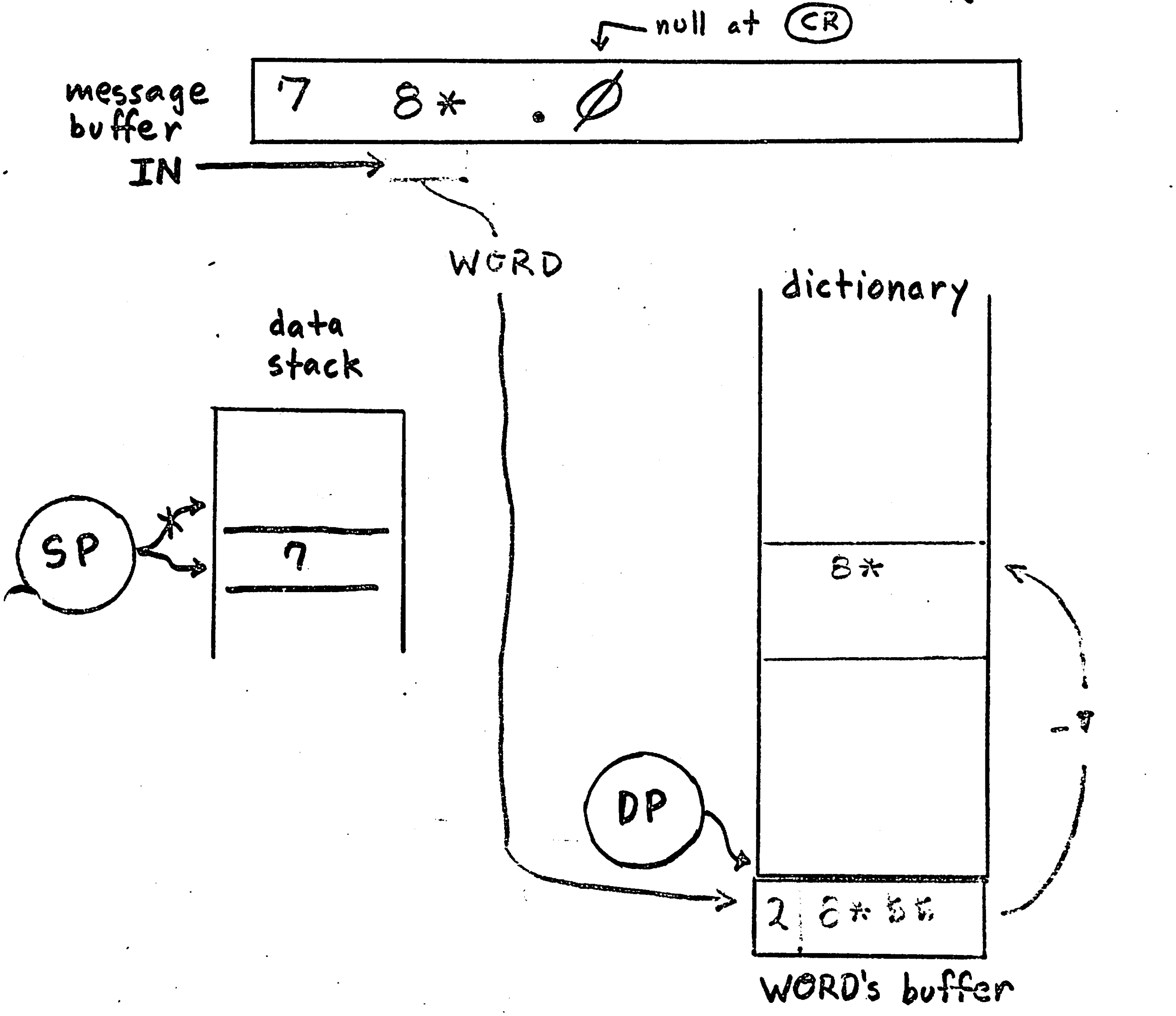
This & next page: typical interpreting sequence

#chrs EXPECT (reads line into message buffer)

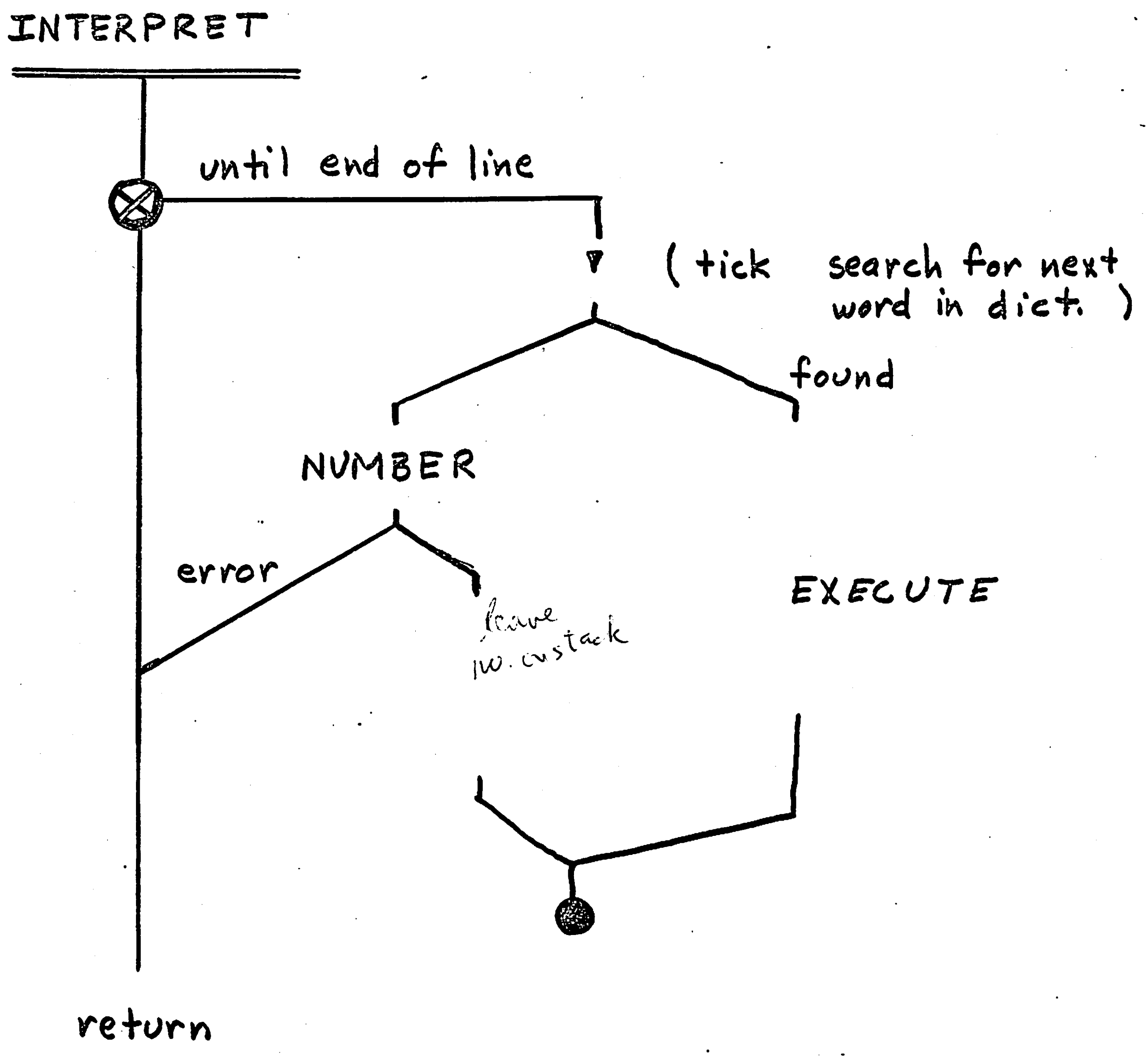


TEXT INTERPRETATION

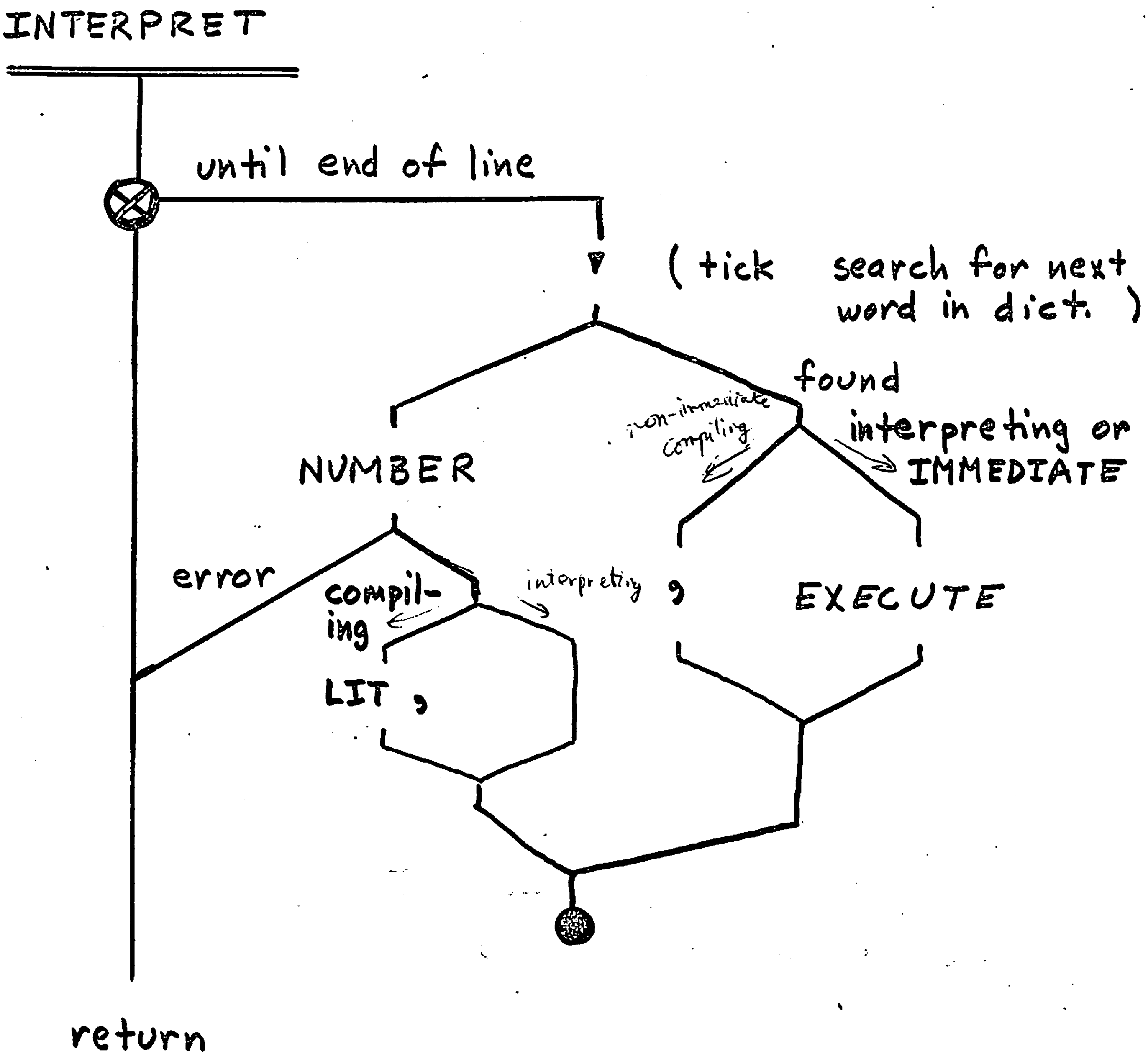
#chrs EXPECT (reads line into message buffer)



TEXT INTERPRETER



TEXT INTERPRETER
and COMPILER



I3
fig

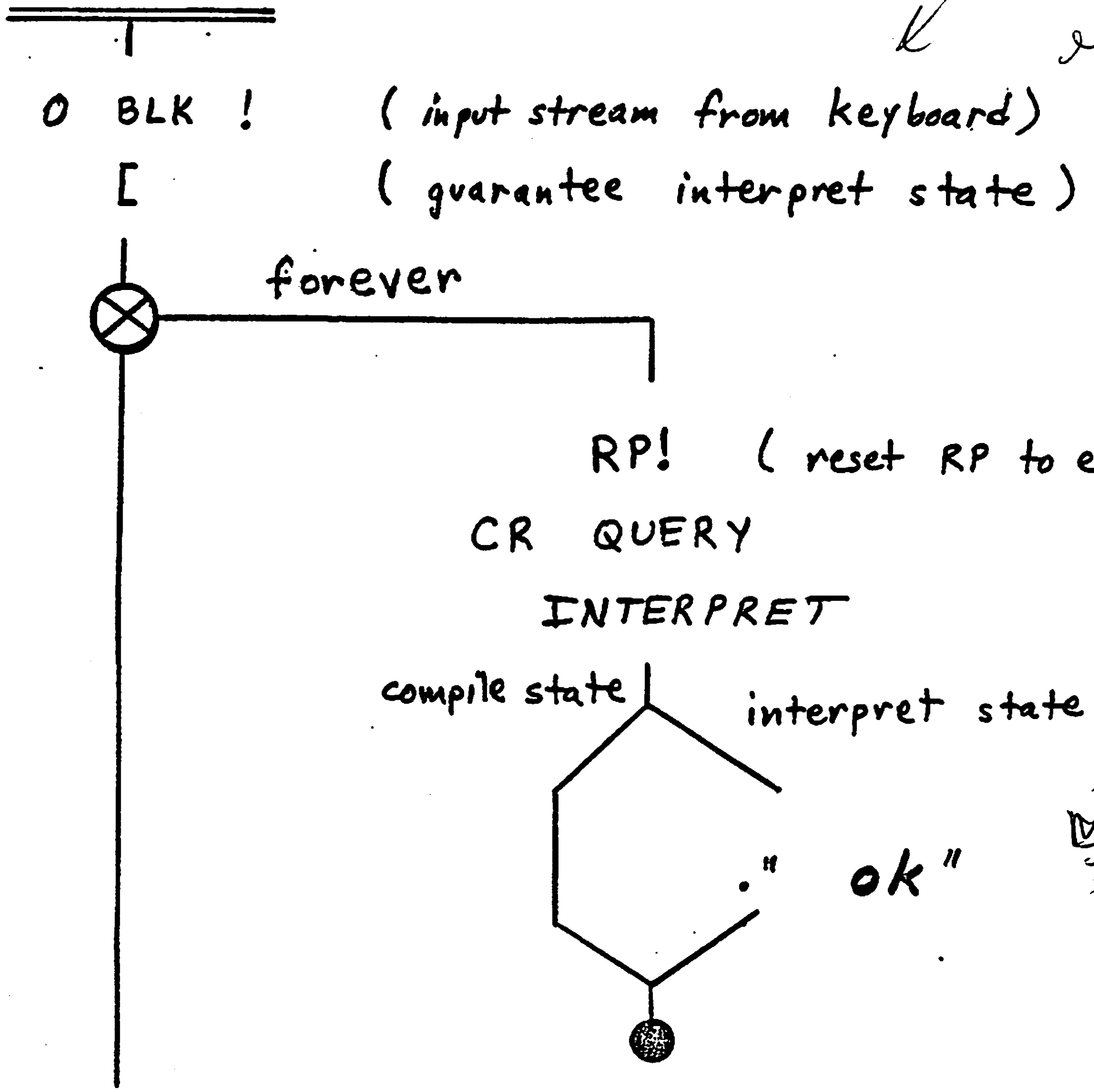
USER'S EXECUTIVE

HEX

```
: QUERY   TIB @  50  EXPECT  ( read line
                               from
                               terminal )
          0  IN !  ;
```

QUIT

The "ultimate" executive loop, for a single user system

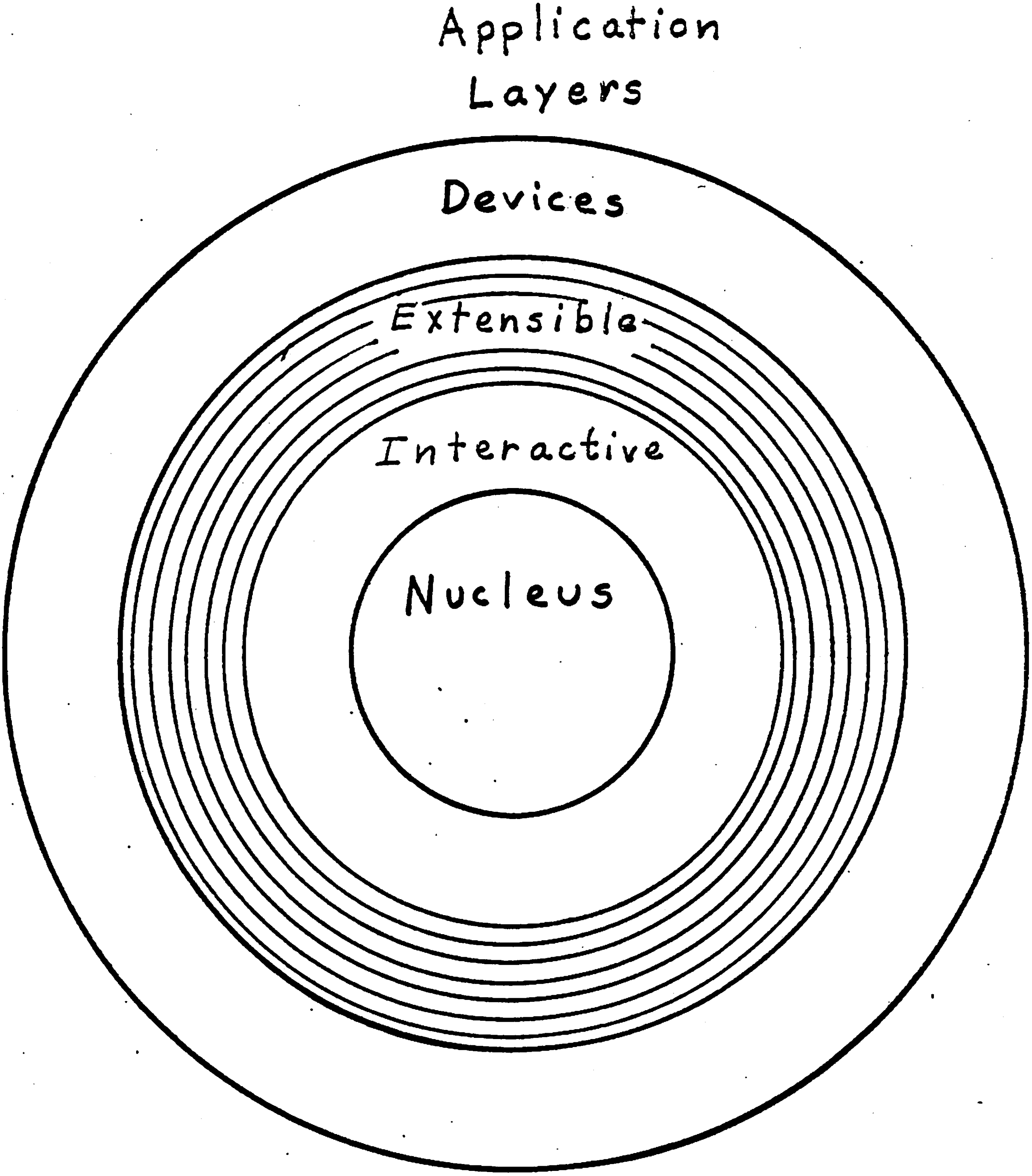


```
0 BLK !     ( input stream from keyboard)
[           ( guarantee interpret state )
            RP!   ( reset RP to empty)
       CR  QUERY
         INTERPRET
             ." ok"
```

note: when in the assembler, you're in the interpret state

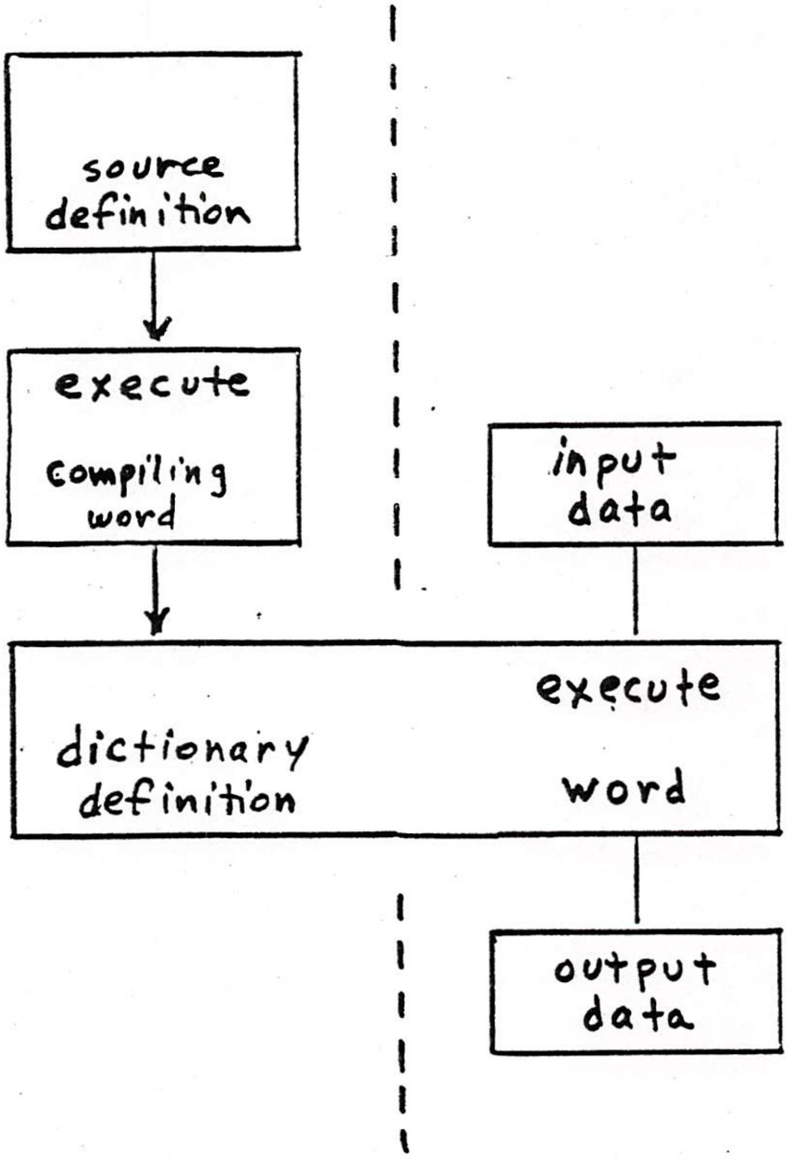
(82)

FORTH COMPILER

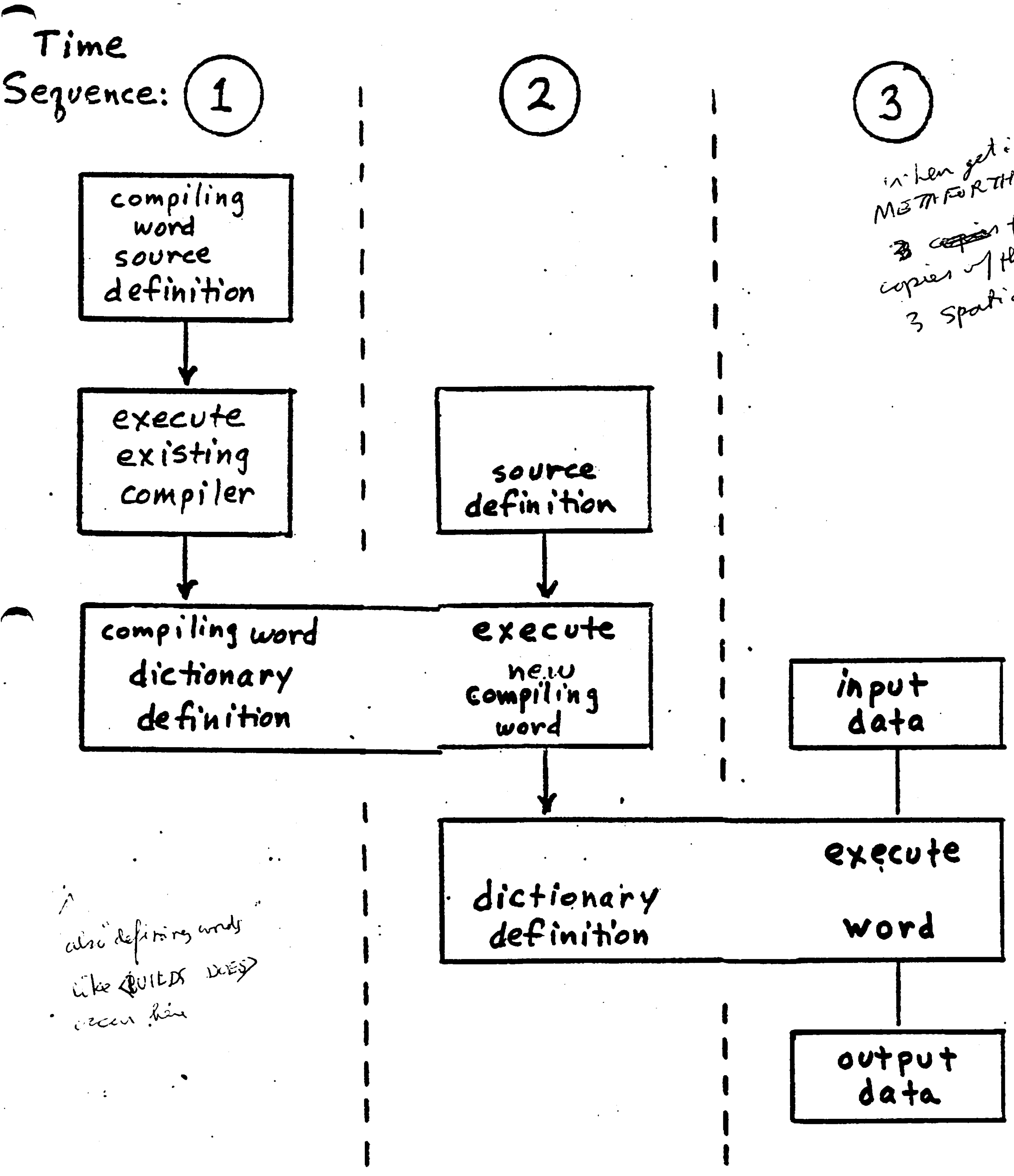


C.2
a

USING COMPILING WORDS



USING COMPILING WORDS



when get into METAFORTH, get 3 ~~copies~~ temporal copies of this, & 3 spatial "periods"

also "defining words" like <BUILDS DOES> occur here

Compile a new compiling word.

e.g. IF, "comma"

Execute the new compiling word; Compile a new word.

Execute the new word.

During compilation,

"normal words" are compiled CFA

by storing each code field address

in the next cell of the dictionary.

"compiling words" are executed

at compile-time. The contents of

the dictionary may or may not be

affected.

Compiling words are defined using

any defining word (eg, : VOCABULARY)

then use the word **IMMEDIATE**

following the definition. This sets the

Precedence bit of the previously

defined word in its dictionary definition.

Some compiling words may be used

only within : definitions; others

may be used either inside or out.

c2
r1
fig

Example:

VOCABULARY FILES

defines a non-immediate word.

Using FILES outside of a : definition, causes it to be executed, switching the accessible vocabulary.

Using it inside a : definition, as in

: ENTER FILES get put ;

causes FILES to be compiled in the definition of ENTER. No vocabulary access is affected.

When ENTER is executed, FILES will be executed, switching vocabularies.

VOCABULARY FILES IMMEDIATE

defines a compiling word.

Using FILES outside a : definition
is the same as the non-immediate version.

However, using FILES inside a : definition
causes vocabularies to be switched during
compilation.

```
: ENTER FILES get put ;
```

The words get and put must be
in the FILES vocabulary.

This version of FILES is an example of an
IMMEDIATE word which has a valid use
both inside and outside a : definition.

C4
r1
fig

Selecting compilation or text interpretation:

[terminates compilation
begins text interpretation — STATE @ = 0

] terminates interpretation
begins compilation — STATE @ ≠ 0

Used internally within : and ; to start and stop compiling.

May also be used for compile-time arithmetic and other operations within a : definition.

Note ; this had to be an IMMEDIATE word!

Is not an immediate ; word

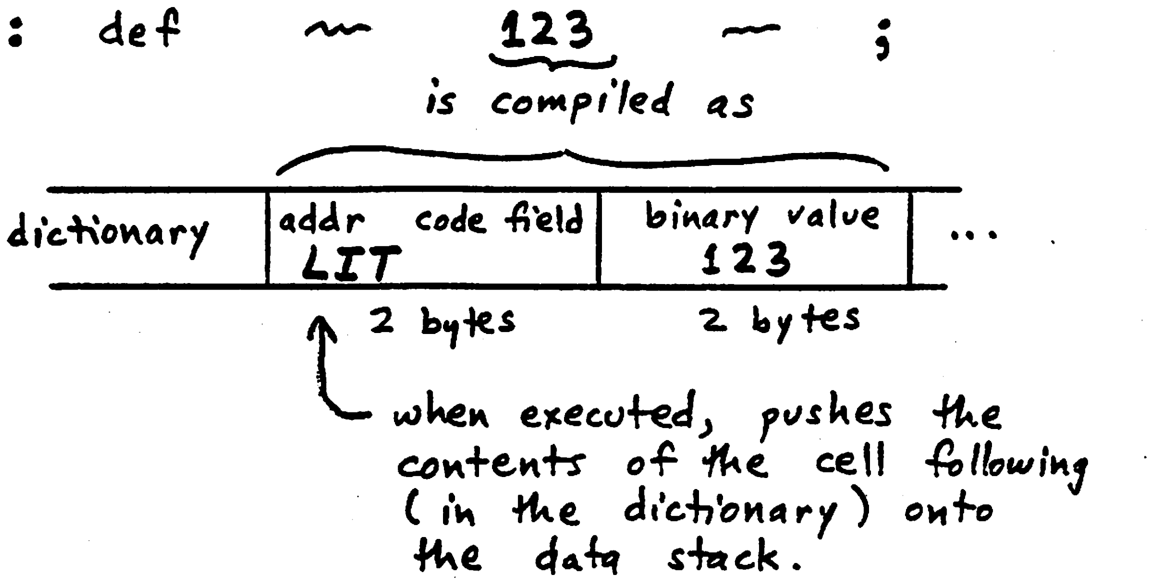
The compilation of literal values:

a literal is a numeric character string

example: 123

While interpreting, a literal is converted to binary value and pushed onto the data stack.

When encountered inside a : definition, a literal may be converted to its binary value, but the pushing of the value onto the stack must be defered until the definition is executed.



when executed, pushes the contents of the cell following (in the dictionary) onto the data stack.

Performing compile-time arithmetic (and other compile-time operations):

The expression `1024 16 /` has a constant value. The definition

```
: slow  ~~  1024  16  /  ~~  ;
```

will perform the divide when the definition is executed and will take up 10 bytes of dictionary space.

If instead, the following definition is used,

```
: fast  ~  [ 1024 16 / ]  LITERAL
        ~~  ;
```

the divide is done when 'fast' is compiled, and only 6 bytes of space is used.

Dictionary definition of 'fast' :

...	addr cf LIT	binary value 64	...

C7
r1
fig

At interpretation-time,

▼ DUP

pushes the parameter field address of DUP onto the stack.

Using the same phrase in a : definition,

```
: ADR-DUP ▼ DUP ;
```

results in the address of DUP being compiled. When ADR-DUP is executed, the parameter field address of DUP is pushed onto the stack.

In fig-FORTH ▼ is an IMMEDIATE word.

TICK is an "intelligent" word — not in POLYFORTH

there's a controversy —

(idea came from Europeans

c8
r1
fig
a

Deferred compilation:

A non-immediate word in a : definition is compiled when it is encountered (ie, not deferred).

② `: not-deferred ~ , ~ ;`

dictionary . . .	address code field ,	. . .

③ When this definition is executed, , is executed resulting in the top of the stack to the dictionary.

A compiling word may need to force a word to be compiled when the compiling word is executed.

① `: deferred ~ COMPILE , ~ ;`

dictionary	addr cf COMPILE	addr cf ,	...

When this definition is executed, COMPILE is executed. This takes the 16 bit value which follows in the definition being executed and compiles that value into the dictionary.

This technique <u>cannot be used to</u> compile an IMMEDIATE word.

Because the 'immediate' word would execute anyway

Examples:

when bit is ON word is smudged

turns off compiler

```
: ;     ?CSP     COMPILE ;S     SMUDGE [ ;
                                     IMMEDIATE
```

None of the words within the definition of ; are IMMEDIATE, so each is compiled normally.

When ; is executed, the compile-time stack size is checked by ?CSP ,

;S is compiled into the definition which is being compiled when ; is executed (at sequence 2)

SMUDGE makes the sequence 2 word name findable, and

[terminates compilation .

```
: LITERAL     STATE @ IF
        COMPILE   LIT   ,     THEN ; IMMEDIATE
```

None of the words within the definition are IMMEDIATE, so each is compiled normally.

When LITERAL is executed from within a : definition, the code field address of LIT is compiled into the sequence 2 definition,

then the top of the stack (at sequence 2 compile-time) is compiled following LIT.

When LITERAL is executed outside of a : definition, it does nothing.

Compiling IMMEDIATE words:

Compiling words sometimes need to force the compilation of IMMEDIATE words.

For example, the word ' is IMMEDIATE in fig-FORTH. Words like FORGET must perform a dictionary search at interpret-time.

This could be done by switching to interpret state within the definition of FORGET, as in

CFA (in FIG FORTH)

: FORGET ~ [' '] LITERAL ~ ;

exit compiling

get PFA of 'TICK'

reenter compiling

This function is performed by [COMPILE] which forces the compilation of the word following it in a : definition, even if that word is IMMEDIATE.

: FORGET ~ $\underbrace{\text{[COMPILE] '}}$ ~ ;

↓

dictionary ...	addr code field '	...

—forces immediate compilation

CONTROL STRUCTURES:

The control structures IF THEN, BEGIN UNTIL, and all others are built from two branch primitives:

Unconditional branch:

dictionary ...	addr cf BRANCH	branch address	...

When executed, BRANCH causes the next word to be executed to be the word in the dictionary at the branch address.

Depending on the implementation of the address interpreter, the branch may be

absolute	then the branch addr is a 2 byte absolute machine address. When the branch is executed, this address is stored in FORTH's Interpreter Pointer.
or	
relative	then the branch addr is either a 1 or 2 byte signed value which is added to the contents of the Interpreter Pointer when the branch is executed.

c11
r1
fig

Conditional branch:

dictionary	addr cf OBRANCH	branch address	...

when executed, OBRANCH
pops the top of the data stack,
if it is ~~≠0 (true)~~ (false) then performs
the branch (same as unconditional
branch)
otherwise ~~(false)~~ (true) skips over
branch addr and executes the
word following in the dictionary.

Calculating branch addresses:

The : compiler uses the data stack during compile-time to compute the branch addresses. This permits indefinite nesting of control structures.

HERE returns the address of the next available location in the dictionary.

C13
r1
fig
a

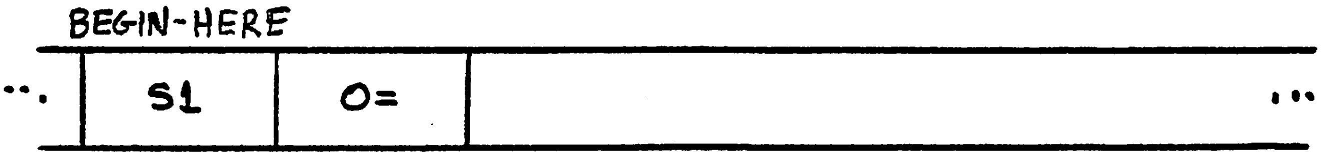
Example: 2 byte relative branch addresses

: BEGIN HERE ; IMMEDIATE

: UNTIL COMPILE OBRANCH HERE - , ;
IMMEDIATE

calculate backward branch

... BEGIN S1 0= UNTIL S2 ...

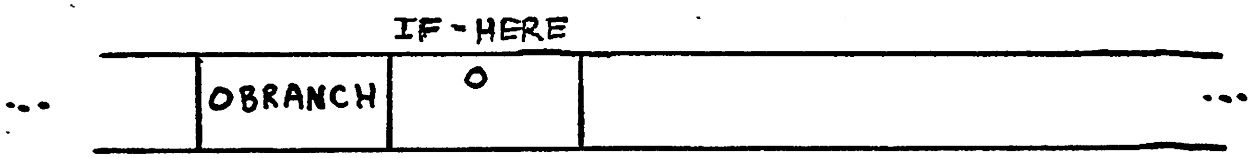


: IF COMPILE OBRANCH HERE 0 , ; IMMEDIATE

: THEN HERE OVER - SWAP ! ; IMMEDIATE

calculate forward branch

... IF S1 S2 THEN S3 ...



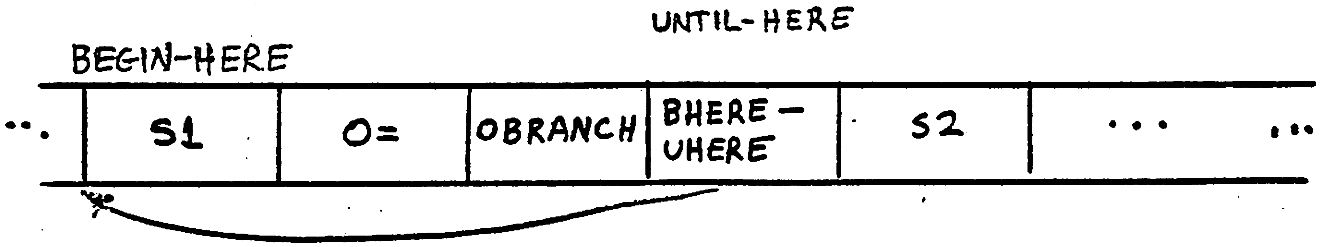
c13
r1
fig
ab

Example: 2 byte relative branch addresses

```
: BEGIN   HERE  ;  IMMEDIATE

: UNTIL   COMPILE  OBRANCH  HERE  -  ,  ;
  IMMEDIATE
```

calculate backward branch

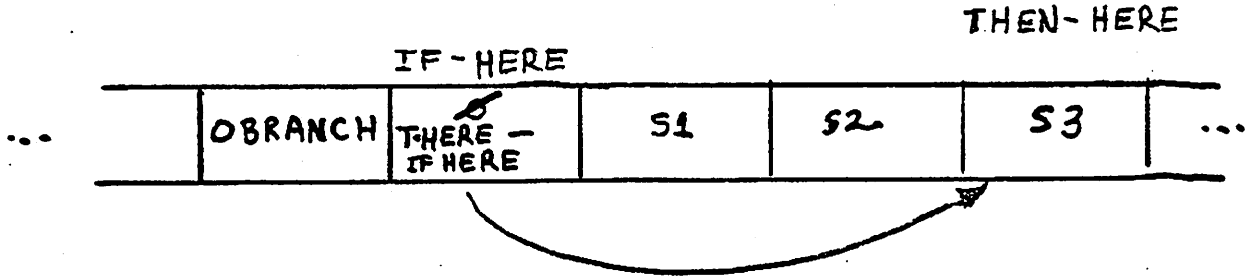
```
... BEGIN   S1   O=  UNTIL   S2  ...
```



```
: IF   COMPILE  OBRANCH   HERE   0  ,  ;  IMMEDIATE

: THEN   HERE  OVER  -  SWAP  !  ;  IMMEDIATE
```

calculate forward branch

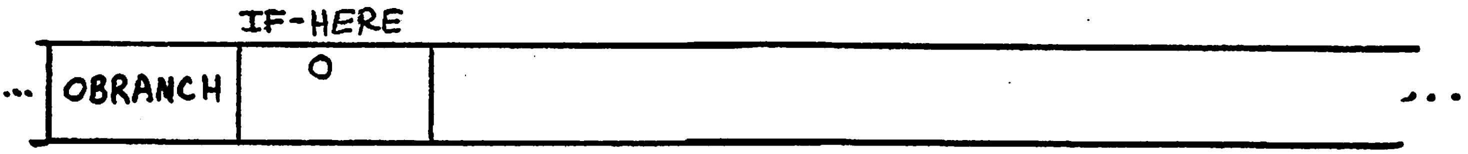
```
... IF        S1    S2    THEN    S3  ...
```



C14
r1
fig
a

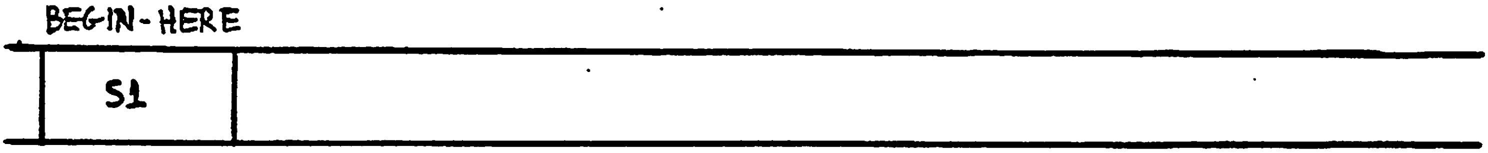
```
: ELSE   COMPILE BRANCH   HERE 0 ,
         SWAP  [COMPILE] THEN ;  IMMEDIATE
```

```
... IF     S1     ELSE    S2    THEN  S3 ...
```



```
: WHILE   [COMPILE] IF  ;  IMMEDIATE

: REPEAT  >R  COMPILE BRANCH   HERE - ,
              R>  [COMPILE] THEN ;  IMMEDIATE
```

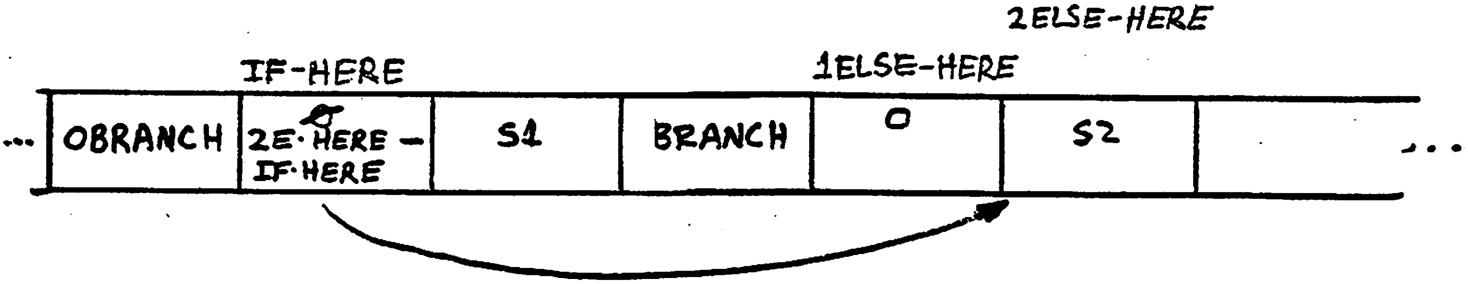
```
... BEGIN   S1   WHILE   S2   REPEAT   S3 ...
```



C14
r1
fig
ab

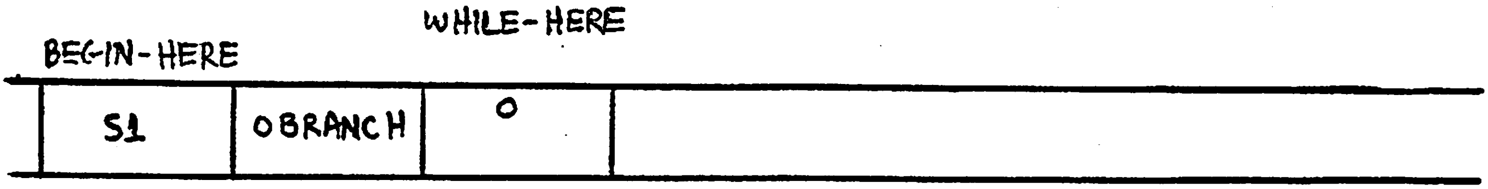
```
: ELSE   COMPILE  BRANCH    HERE  0  ,
              SWAP  [COMPILE]  THEN  ;  IMMEDIATE
```

```
... IF      S1          ELSE      S2       THEN  S3 ...
```



```
: WHILE   [COMPILE]  IF   ;  IMMEDIATE

: REPEAT   >R   COMPILE  BRANCH    HERE  -   ,
                R>  [COMPILE]  THEN  ;  IMMEDIATE
```

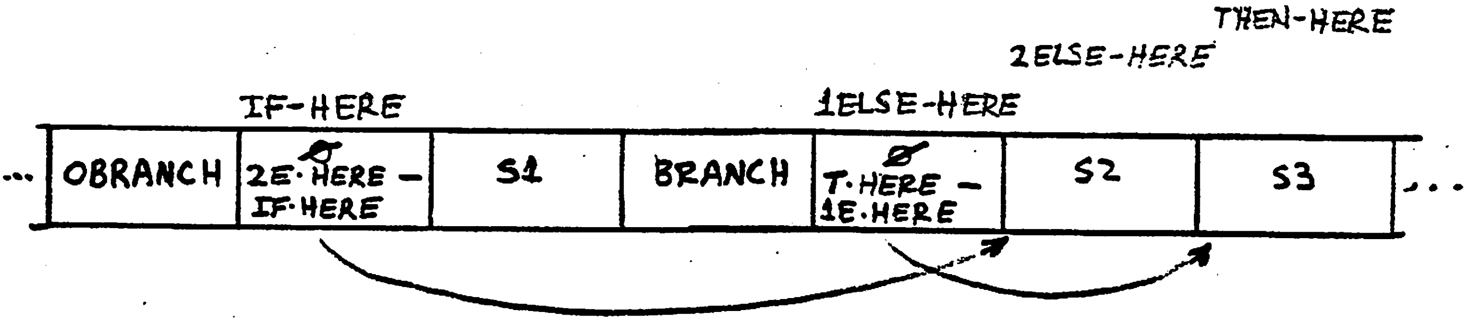
```
... BEGIN   S1    WHILE    S2    REPEAT    S3  ...
```



C14
r1
fig
abc

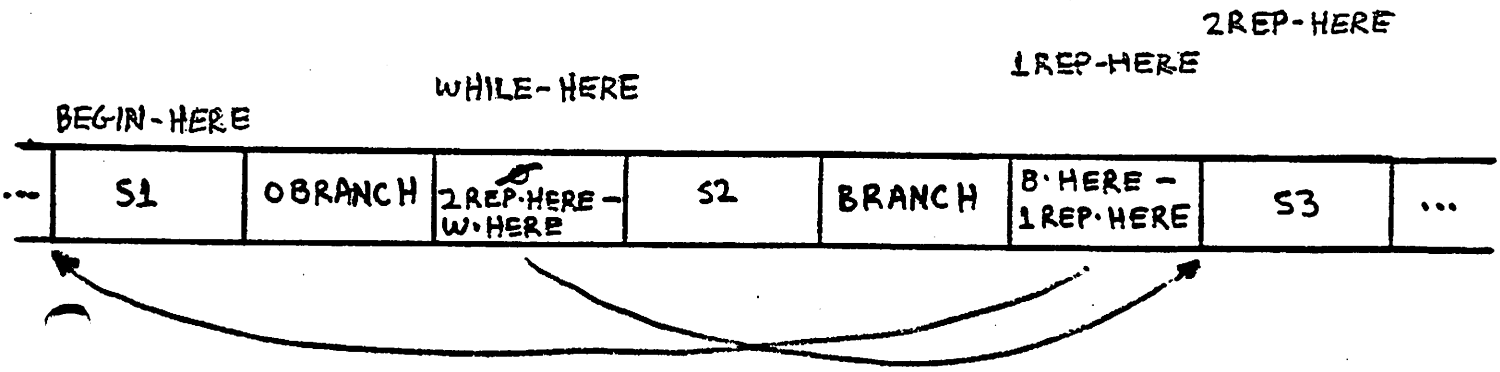
```
: ELSE   COMPILE  BRANCH   HERE  0 ,
         SWAP  [COMPILE]  THEN  ;  IMMEDIATE
```

```
... IF     S1      ELSE     S2     THEN  S3 ...
```



```
: WHILE   [COMPILE]  IF  ;  IMMEDIATE

: REPEAT   >R   COMPILE  BRANCH   HERE  -  ,
                R>  [COMPILE]  THEN  ;  IMMEDIATE
```

```
... BEGIN   S1    WHILE    S2    REPEAT    S3 ...
```



c15
fig

```
 0 ( figFORTH control structure compiling word definitions )
 1 ( no compiler security )
 2 : <-BRANCH   HERE - ,    ;    ( BACK in Installation Manual )
 3 : ->BRANCH   HERE OVER -   SWAP !   ;
 4
 5 : IF   COMPILE OBRANCH   HERE 0 ,    ;    IMMEDIATE
 6 : THEN    ->BRANCH   ;   IMMEDIATE
 7 : ELSE   COMPILE BRANCH   HERE 0 ,
 8    SWAP   [COMPILE] THEN   ;   IMMEDIATE
 9
10 : BEGIN   HERE   ;   IMMEDIATE
11 : UNTIL   COMPILE OBRANCH   <-BRANCH   ;   IMMEDIATE
12 : AGAIN   COMPILE  BRANCH   <-BRANCH   ;   IMMEDIATE
13 : WHILE   [COMPILE] IF   ;   IMMEDIATE
14 : REPEAT   >R   COMPILE BRANCH   <-BRANCH
15    R>   [COMPILE] THEN   ;   IMMEDIATE
OK
```

```
 0 ( figFORTH compiling words, part 2 )
 1
 2 : DO   COMPILE (DO)   HERE   ;   IMMEDIATE
 3 : LOOP   COMPILE (LOOP)   <-BRANCH   ;   IMMEDIATE
 4 : +LOOP   COMPILE (+LOOP)   <-BRANCH   ;   IMMEDIATE
 5
```

```
 0 ( figFORTH control structure compiling words, part 3 )
 1 ( redefinitions to add compiler security )
 2 : IF   ?COMP   [COMPILE] IF   2   ;   IMMEDIATE
 3 : THEN   ?COMP   2 ?PAIRS   [COMPILE] THEN   ;   IMMEDIATE
 4 : ELSE   ?COMP   2 ?PAIRS   COMPILE BRANCH   HERE 0 ,
 5    SWAP   2   [COMPILE] THEN   2   ;   IMMEDIATE
 6 : BEGIN   ?COMP   [COMPILE] BEGIN   1   ;   IMMEDIATE
 7 : UNTIL   ?COMP   1 ?PAIRS   [COMPILE] UNTIL   ;   IMMEDIATE
 8 : AGAIN   ?COMP   1 ?PAIRS   [COMPILE] AGAIN   ;   IMMEDIATE
 9 : WHILE   ?COMP   [COMPILE] IF   2+   ;   IMMEDIATE
10 : REPEAT   ?COMP   >R >R   [COMPILE] AGAIN
11                   R> R> 2 -   [COMPILE] THEN   ;   IMMEDIATE
12 : DO   ?COMP   [COMPILE] DO   3   ;   IMMEDIATE
13 : LOOP   ?COMP   3 ?PAIRS   [COMPILE] LOOP   ;   IMMEDIATE
14 : +LOOP   ?COMP   3 ?PAIRS   [COMPILE] +LOOP   ;   IMMEDIATE
15
```

fig-FORTH Compiler Security

detects and aborts on most errors involving control structures:

missing parts of a control structure,
incorrect nesting,
use of compiling words outside a : def.

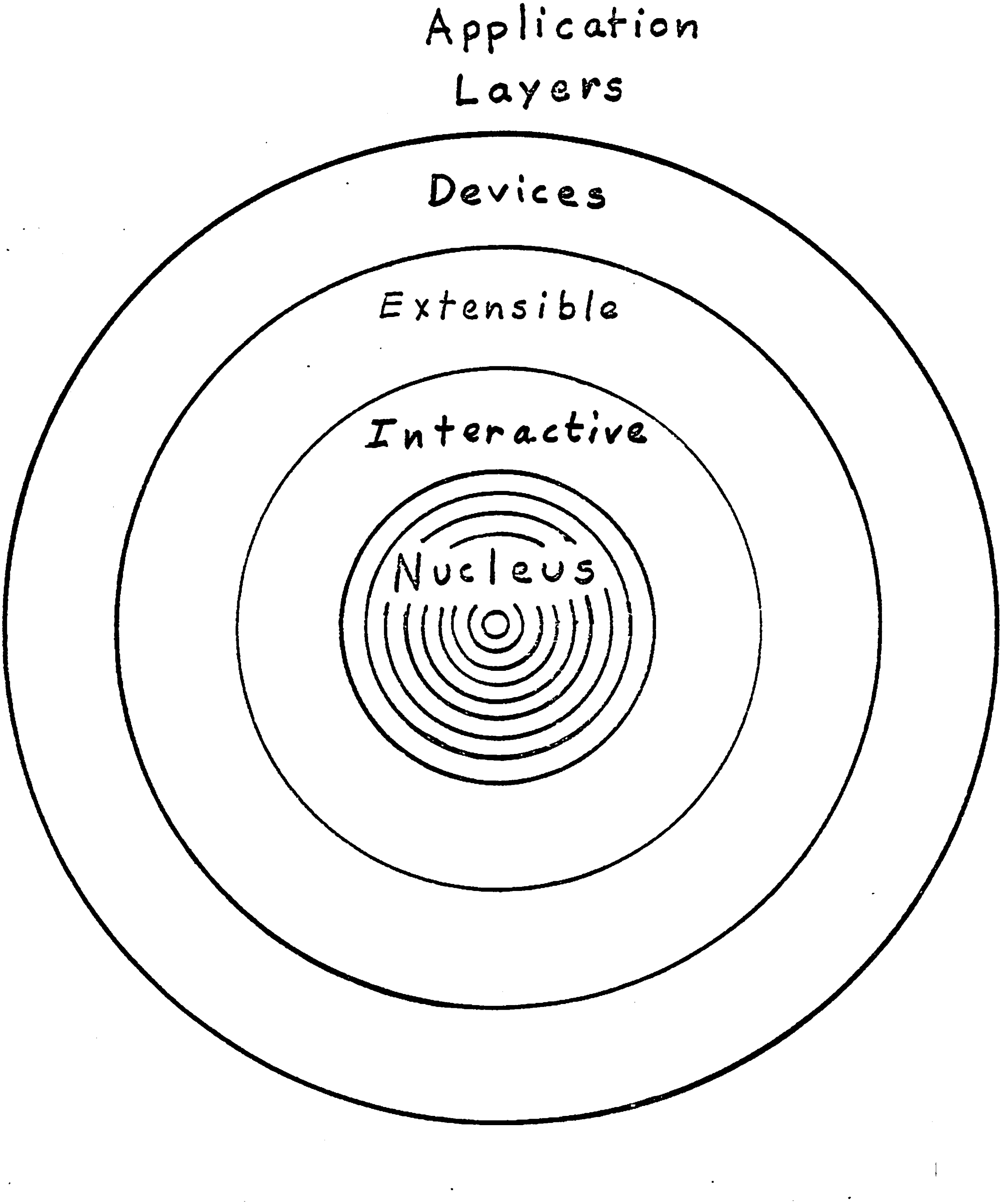
Security words:

?EXEC	if executed in EXECution state (ie, text interpretation state) then does nothing otherwise, an ABORT is executed.
?COMP	opposite above, aborts if not executed while compiling.
!CSP	stores contents of SP in user variable CSP
?CSP	aborts if contents of SP ≠ contents of CSP
?PAIRS	aborts if top two stack values are NOT equal

Use of security words in compiling words:

compiling word	security action
:	?EXEC !CSP
;	?CSP
BEGIN	1
UNTIL	1 ?PAIRS
IF	2
ELSE	2 ?PAIRS 2
THEN	2 ?PAIRS
DO	3
LOOP +LOOP	3 ?PAIRS
BEGIN	1
WHILE	4
REPEAT	1 ?PAIRS 2 - 2 ?PAIRS

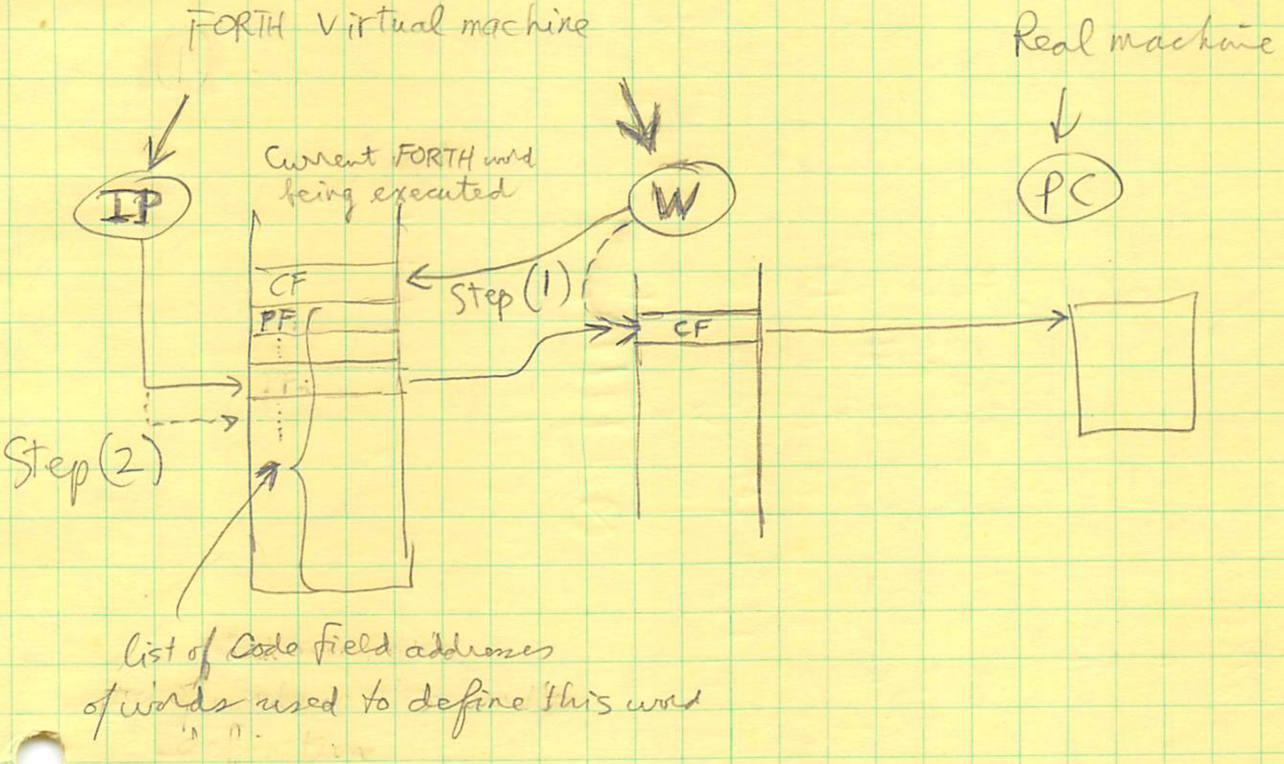
ADDRESS INTERPRETER



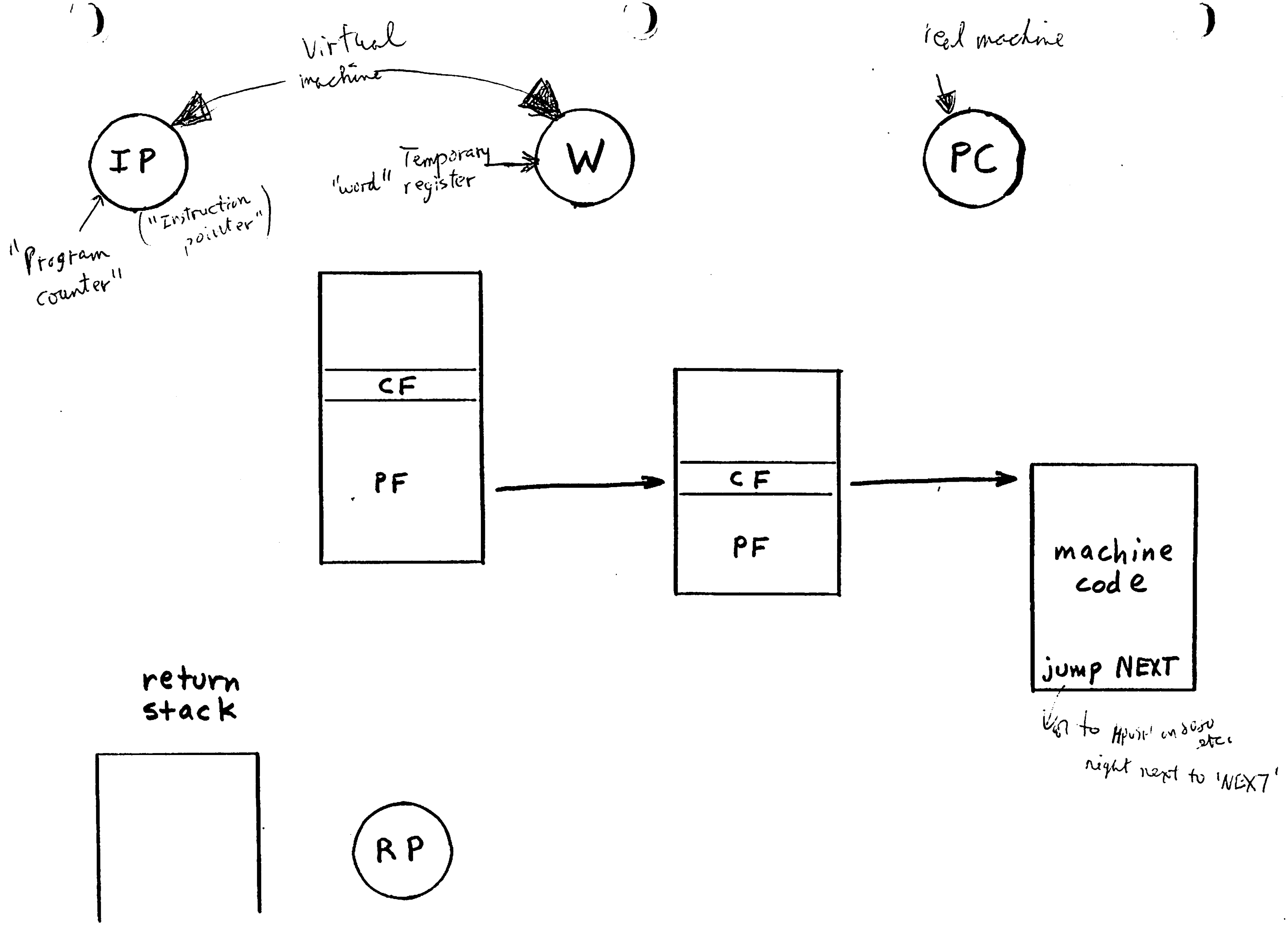
1-12-81

FORTH's address interpreter

(see Kim Harris course notes p. 108)

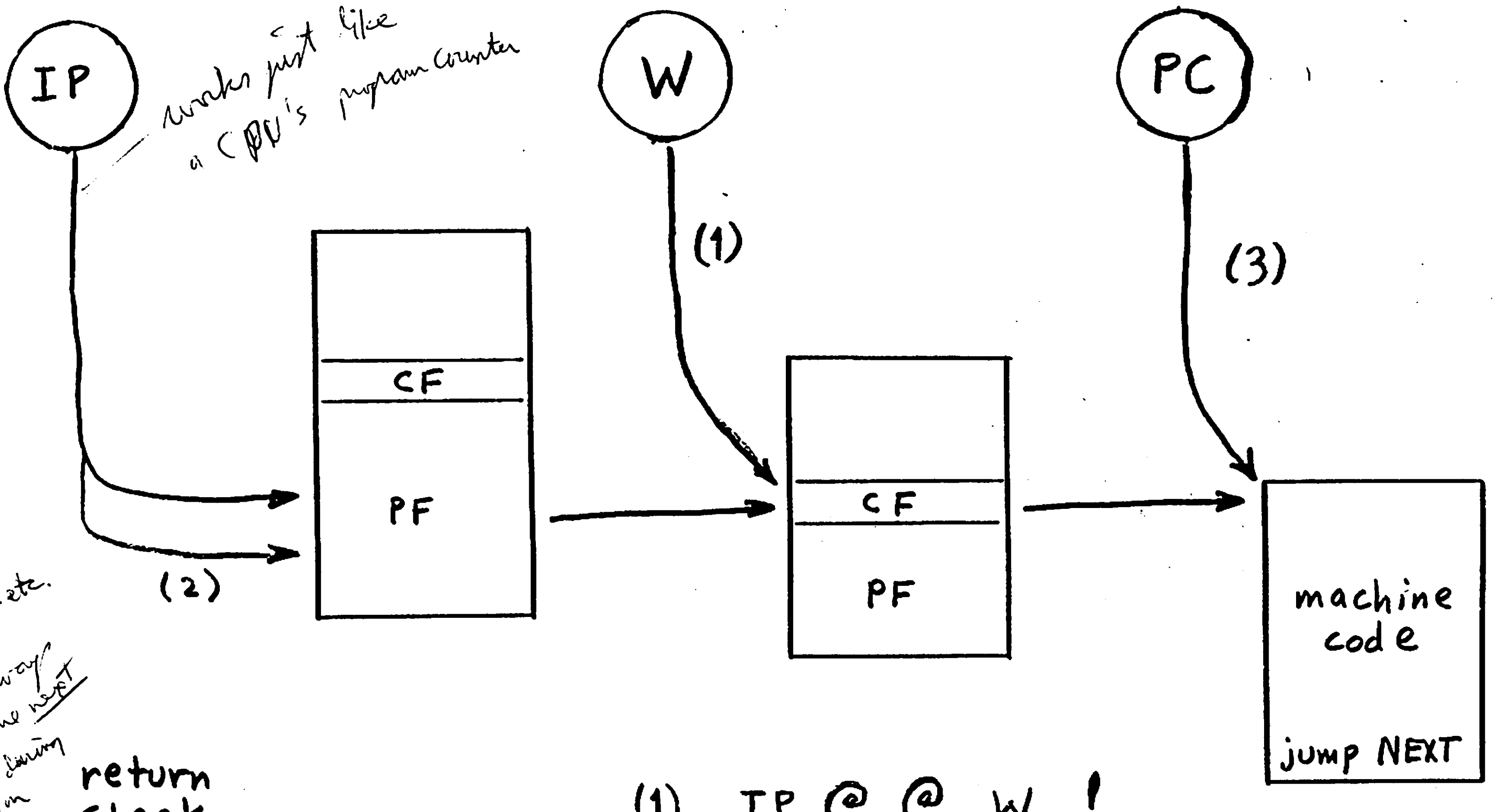


I4
a
fig



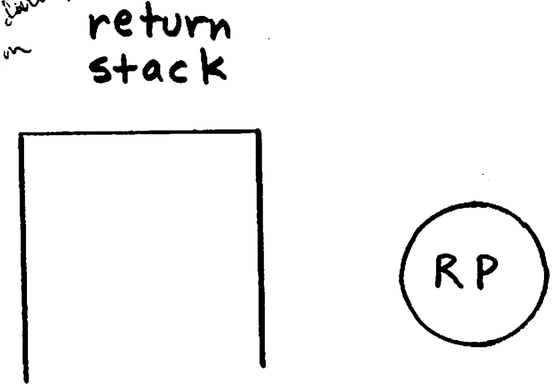
Or to 'Hpush' on 8080 etc.
right next to 'NEXT'

NEXT FORTH's Address Interpreter



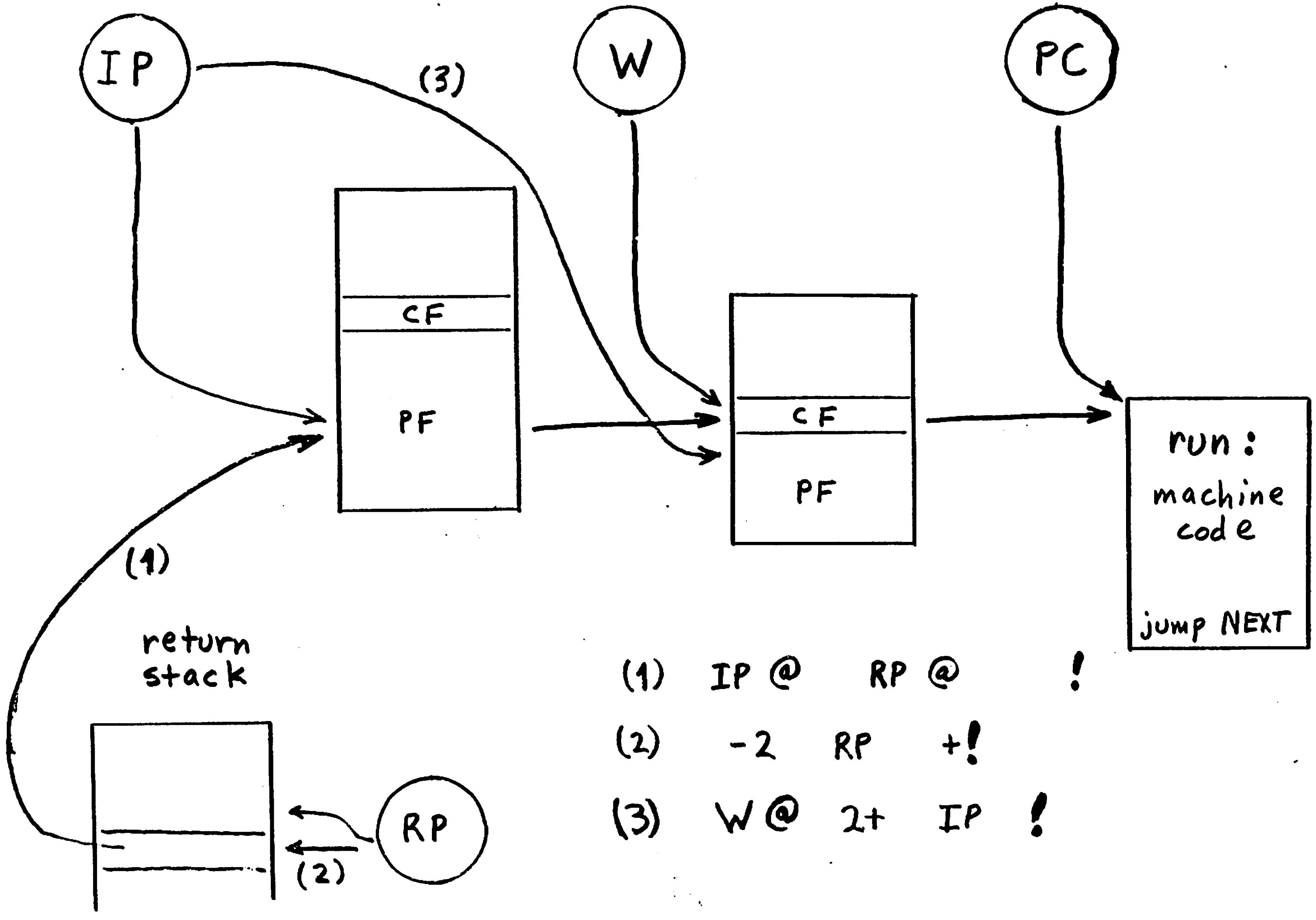
works just like a CPU's program counter

8080, 6502 etc. PACE, PC & IP always point to the next instruction during execution



```
(1)  IP @ @ W !
(2)  2 IP +!
(3)  W @ @ PC !
```

run: "subroutine" nest

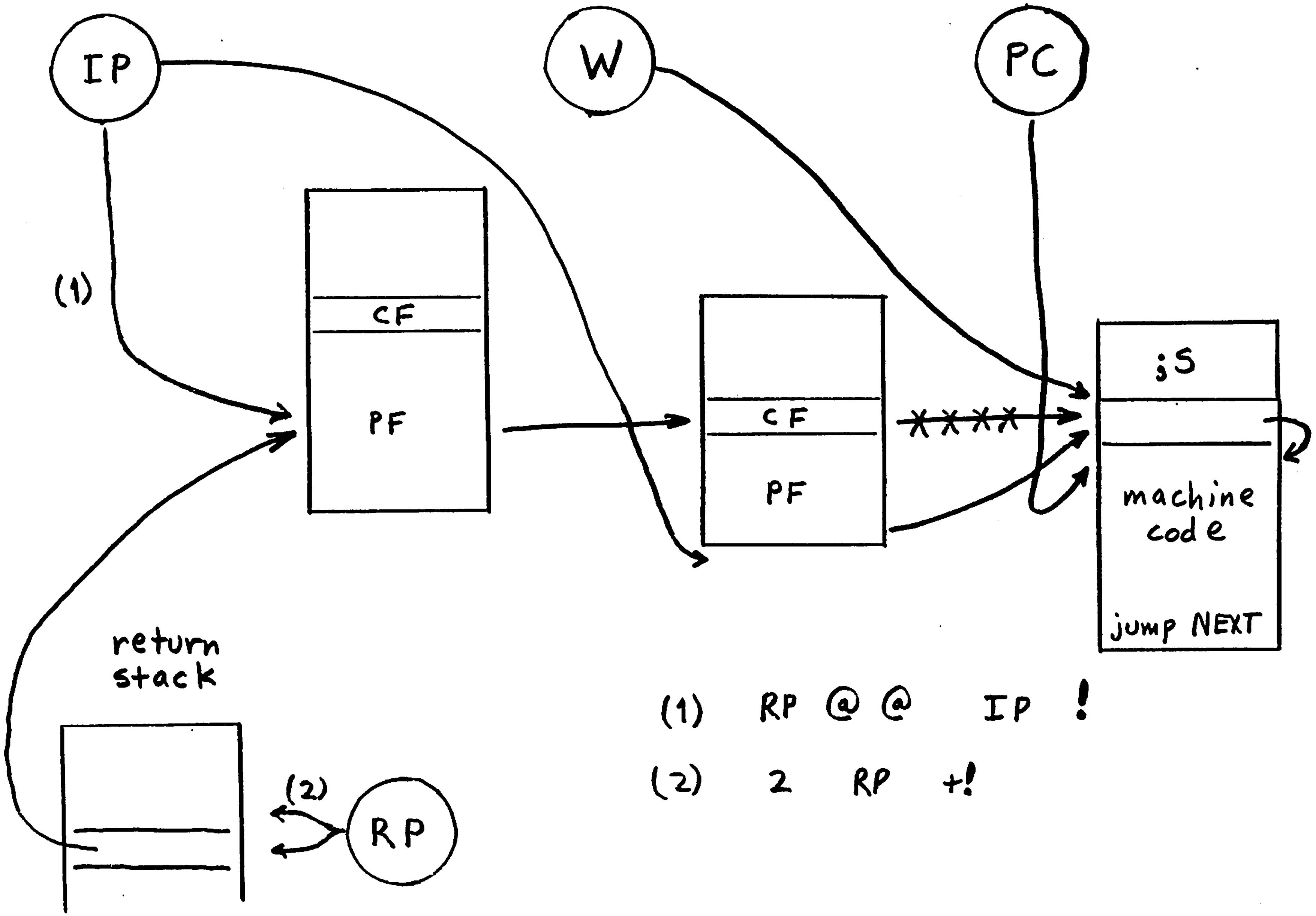


(1) IP @ RP @ !

(2) -2 RP +!

(3) W@ 2+ IP !

;S "subroutine" unnest



(1) RP @ @ IP !

(2) 2 RP +!

I4 d fig

EXECUTION of 8*

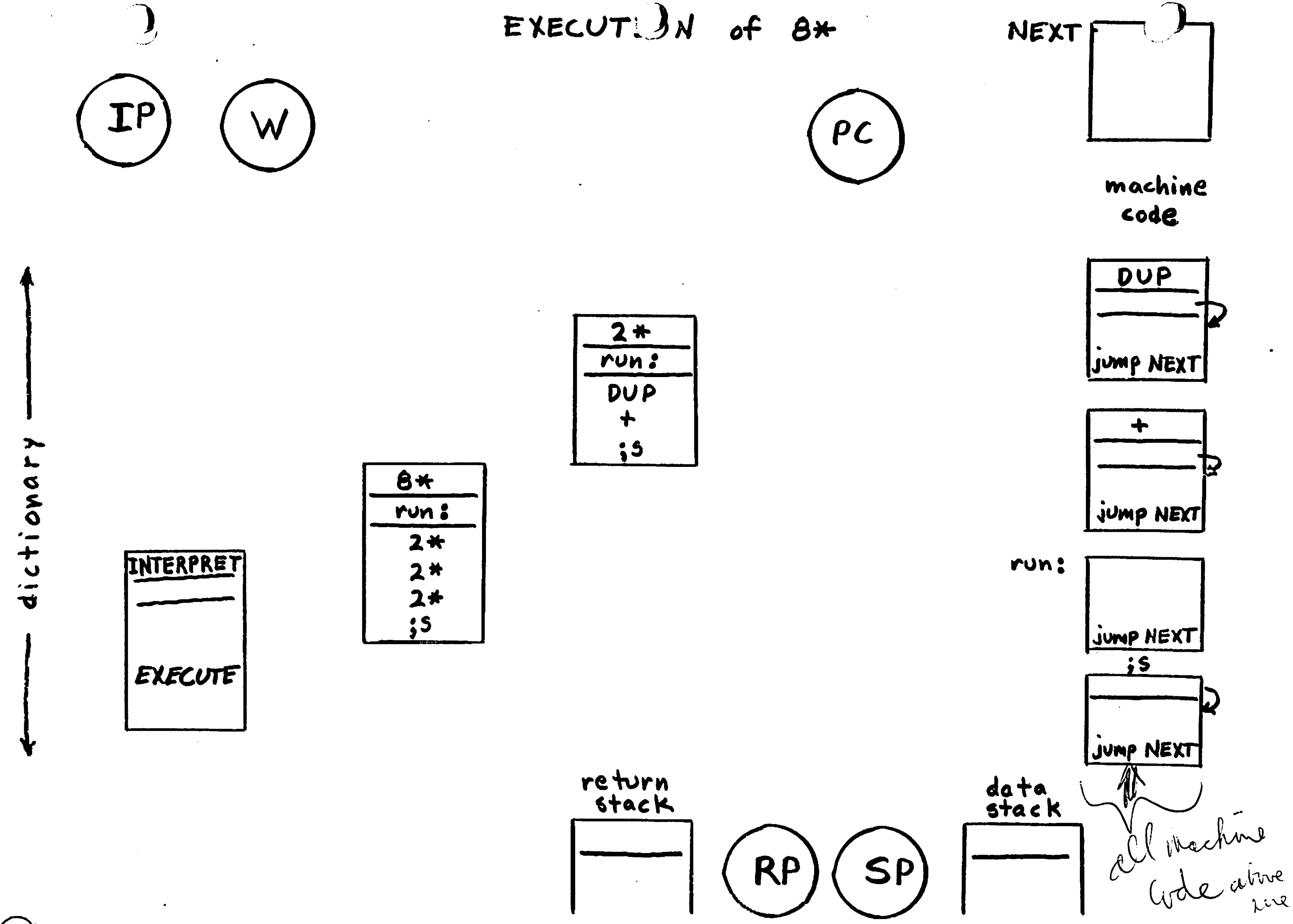
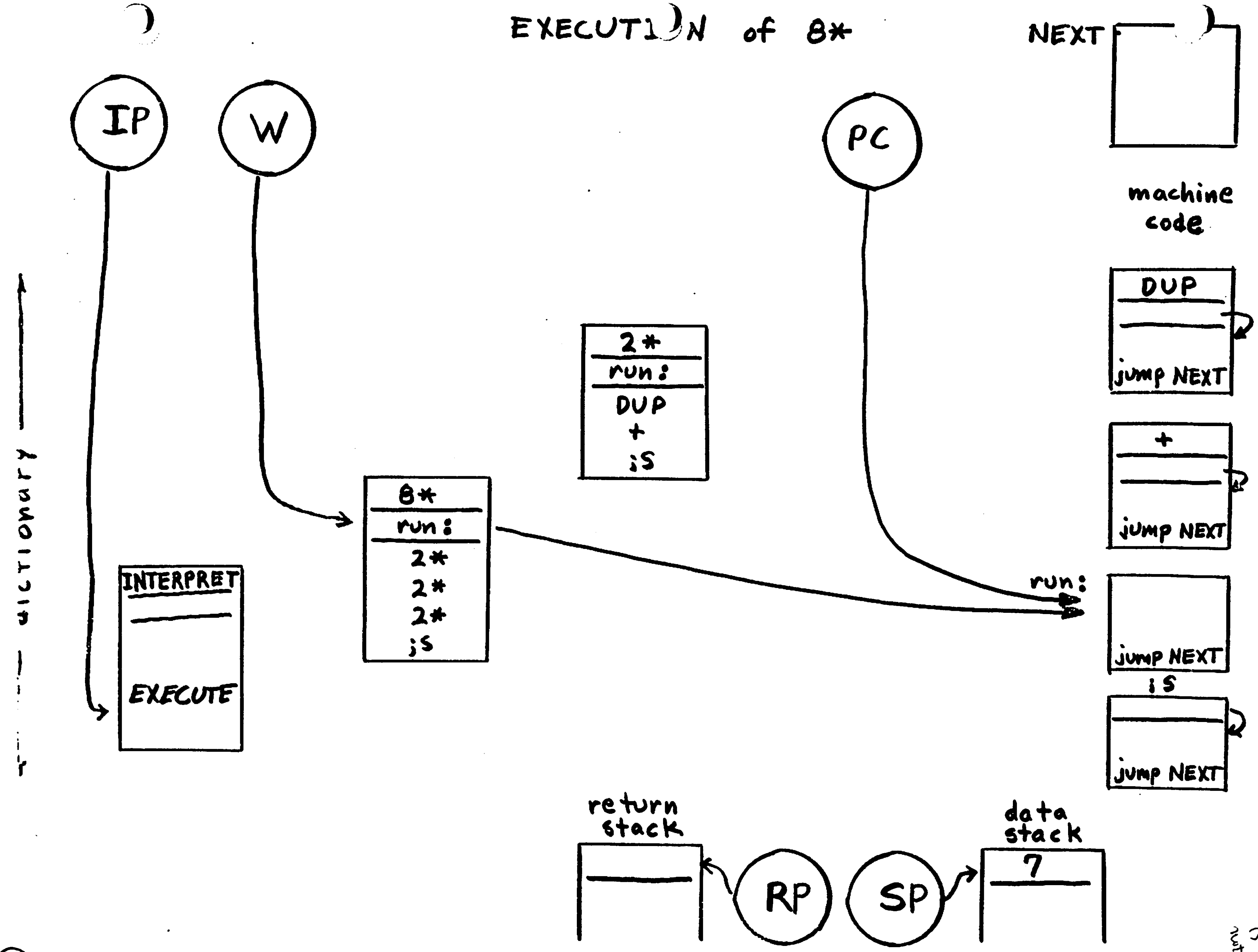


fig I5a

EXECUTION of 8*



(note: there is no p. "6")

I5 c fig

EXECUTION of 8*

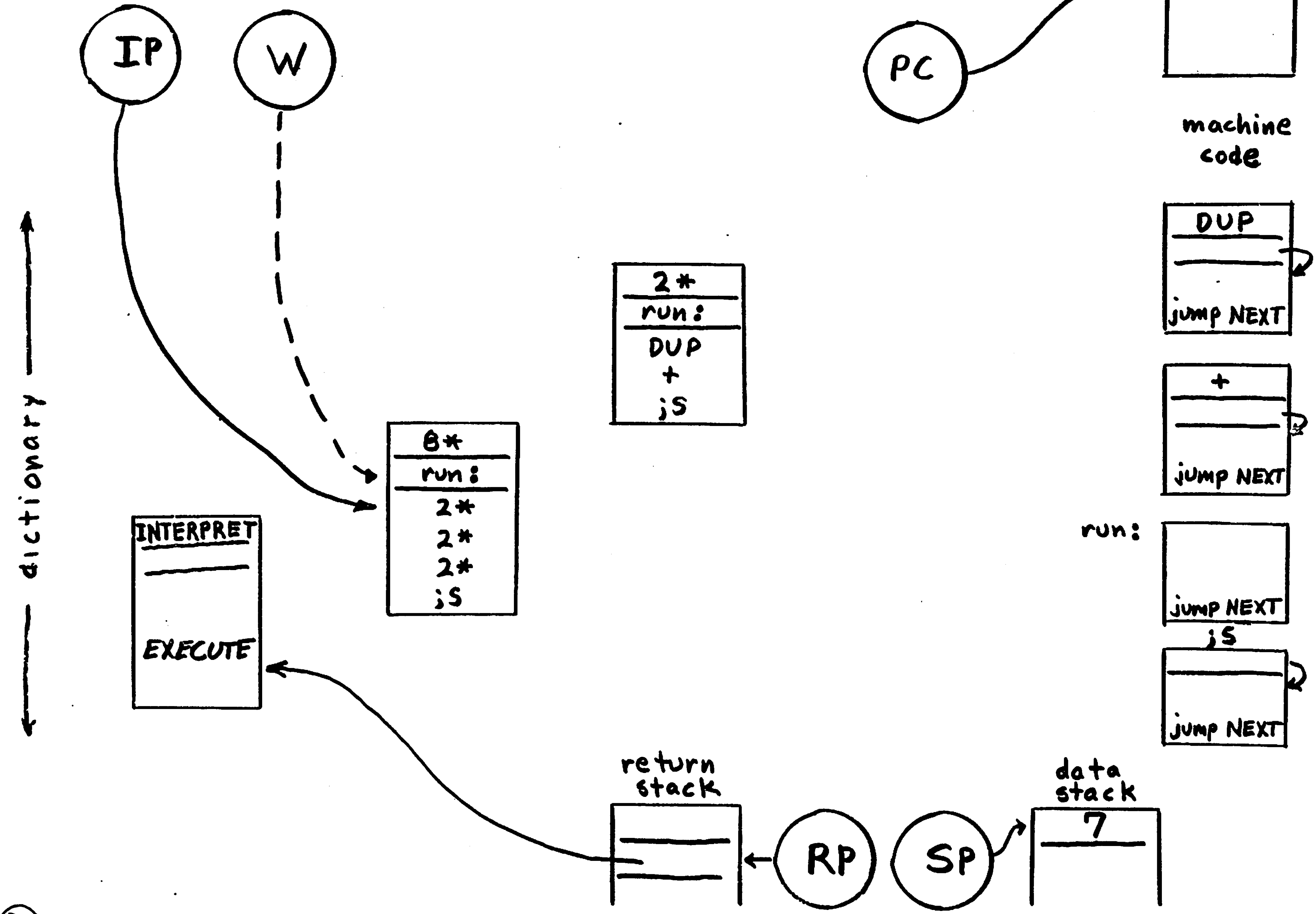
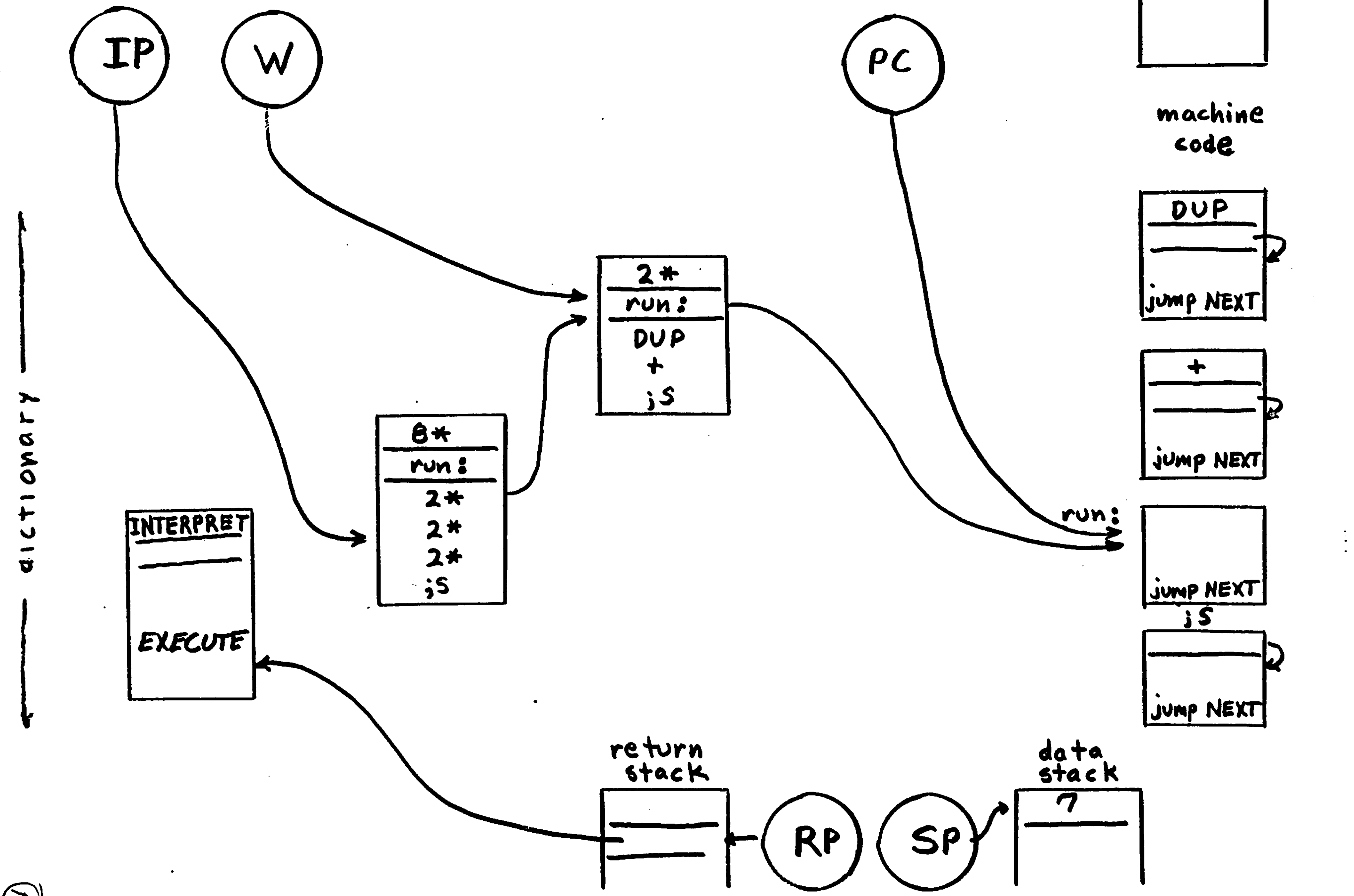


fig 15a

EXECUTION of 8*



EXECUTION of 8*

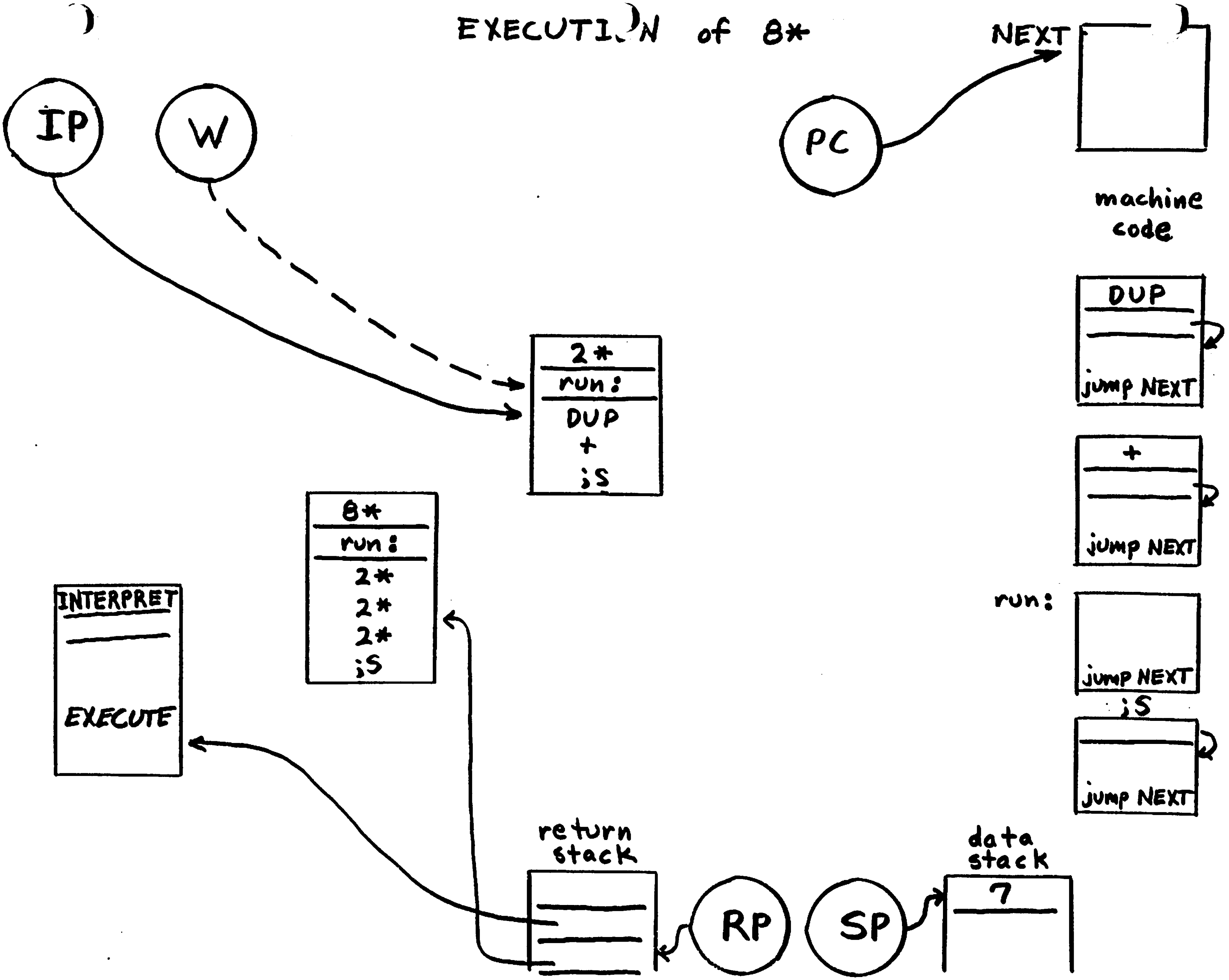


fig 15

EXECUTION of 8*

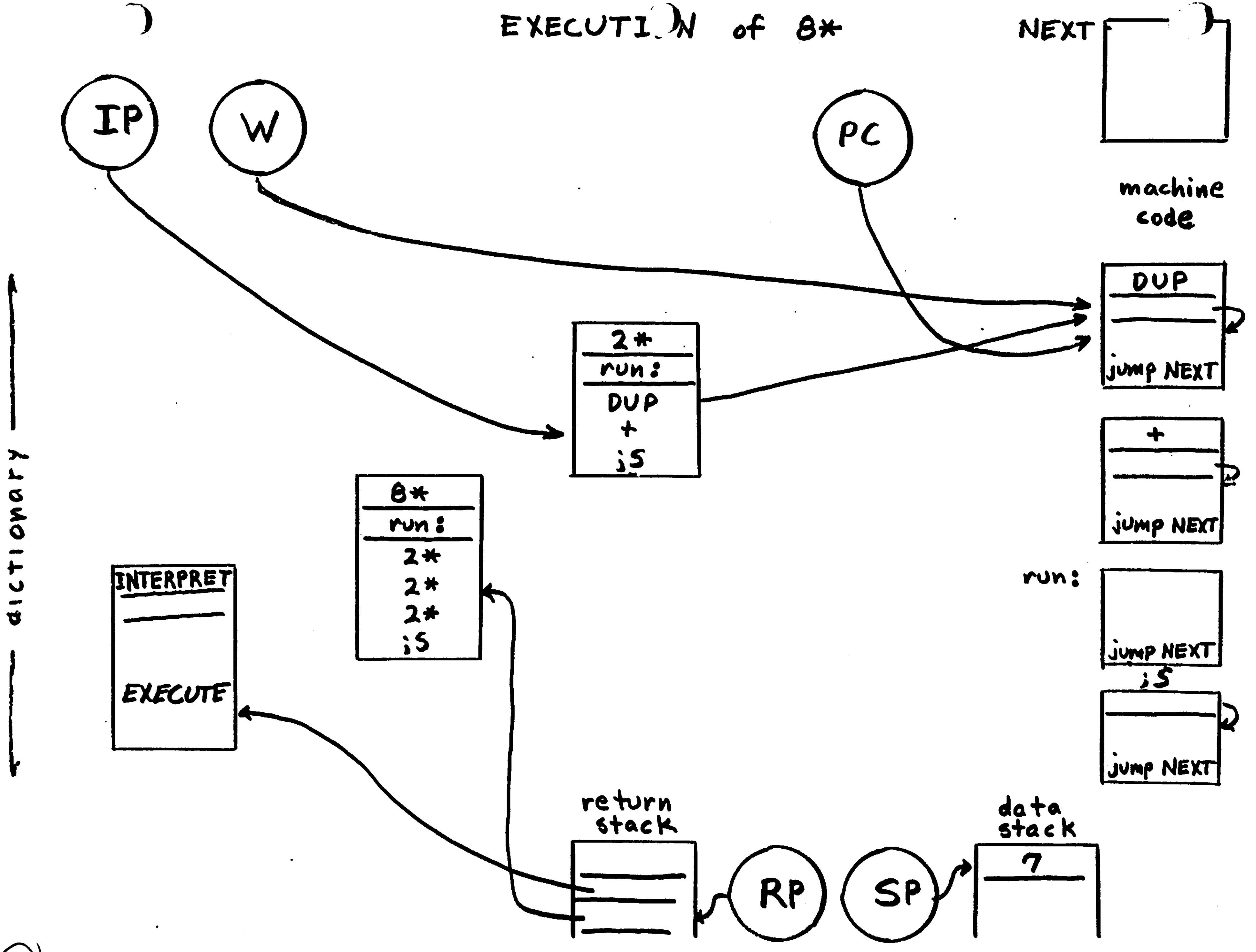
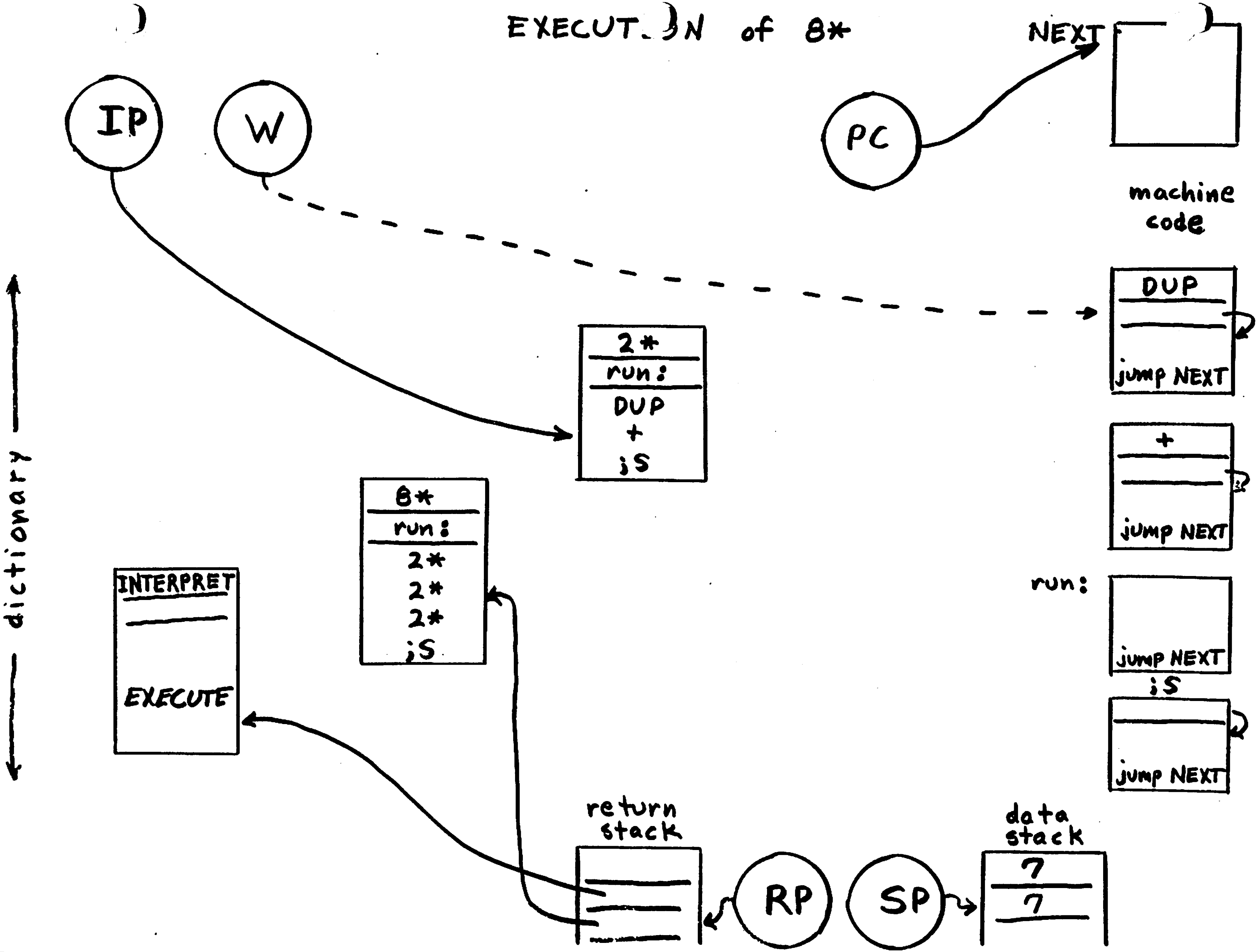


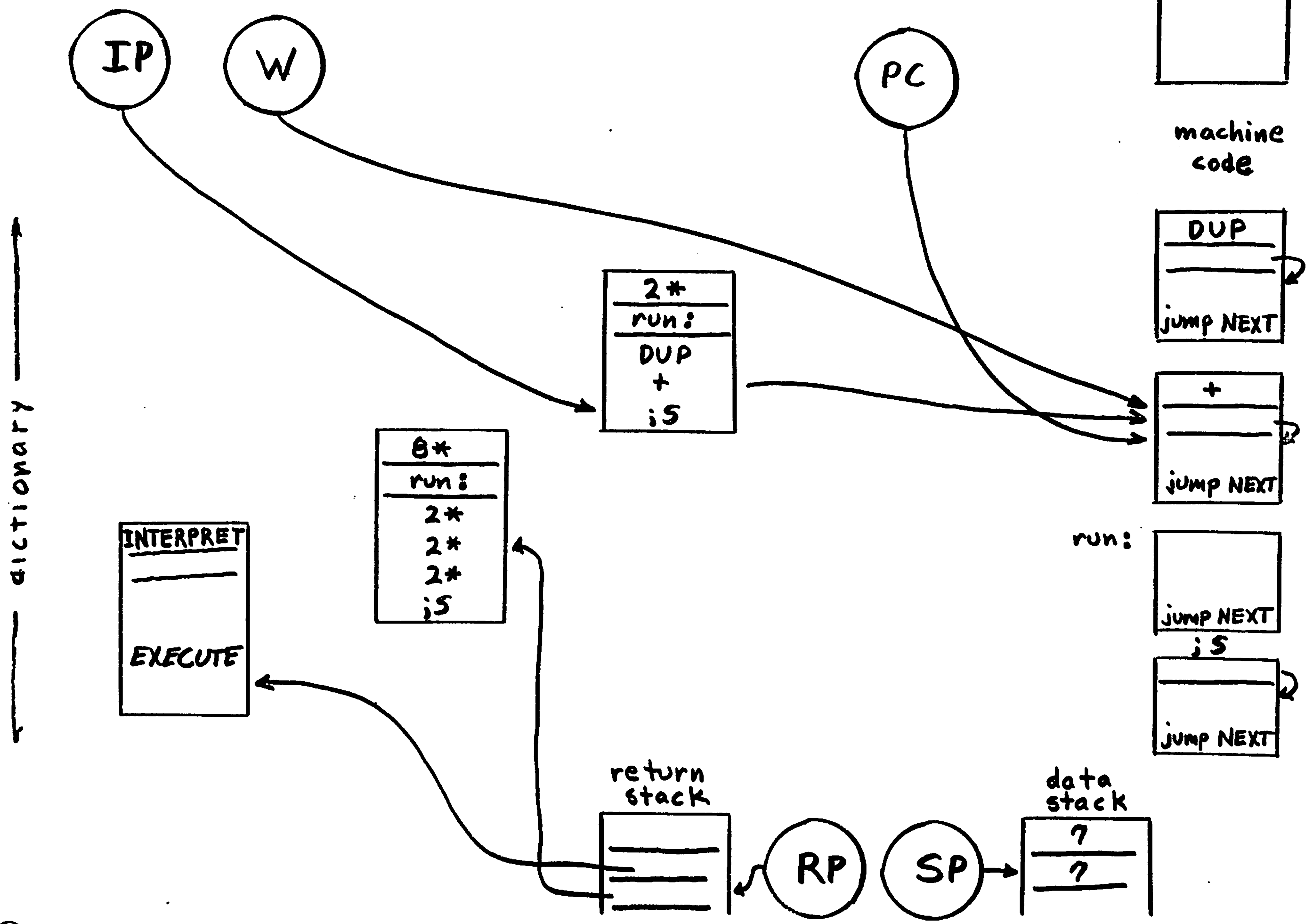
fig 15

EXECUTION of 8*



15
5
fig

EXECUTION of 8*



EXECUTION of 8*

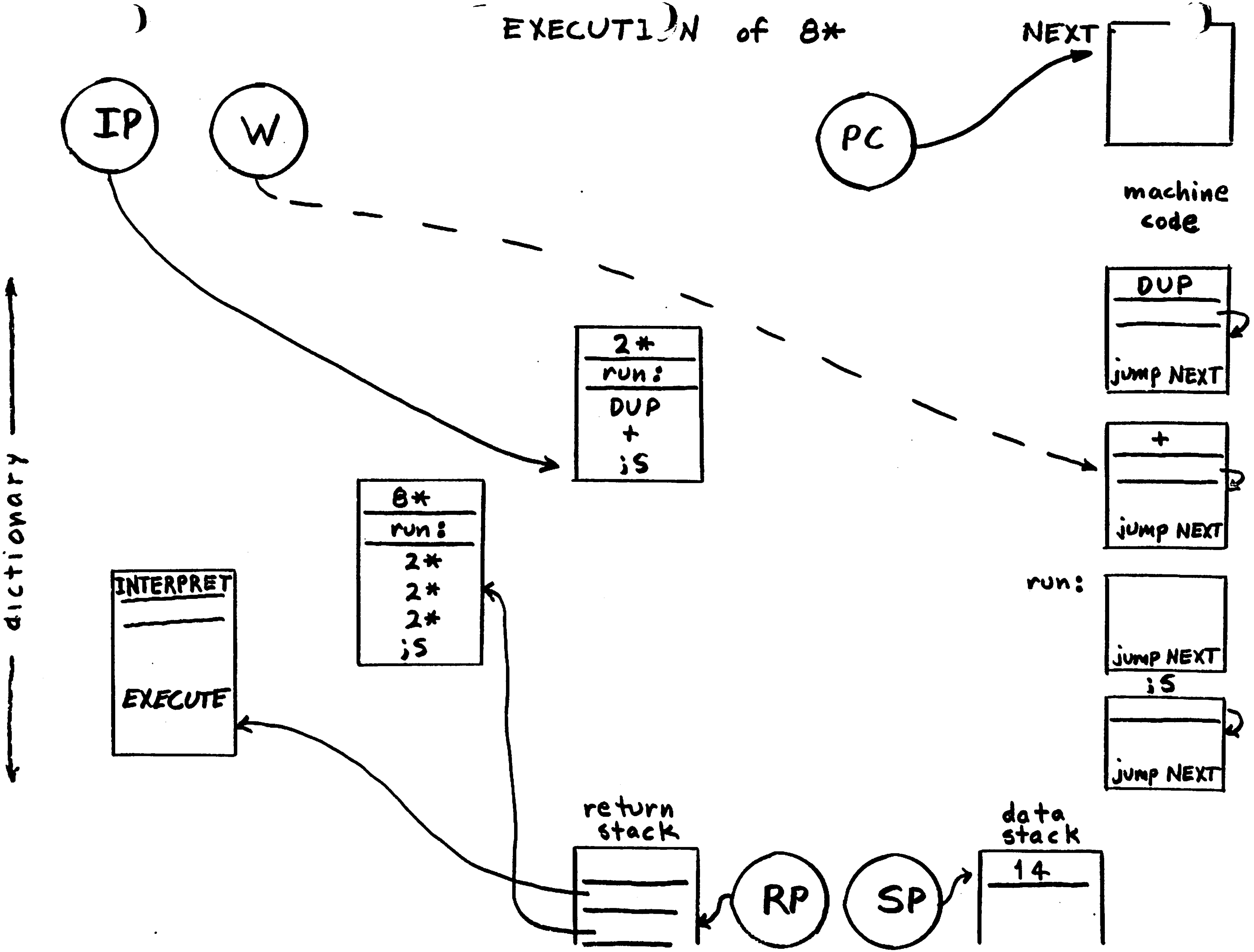
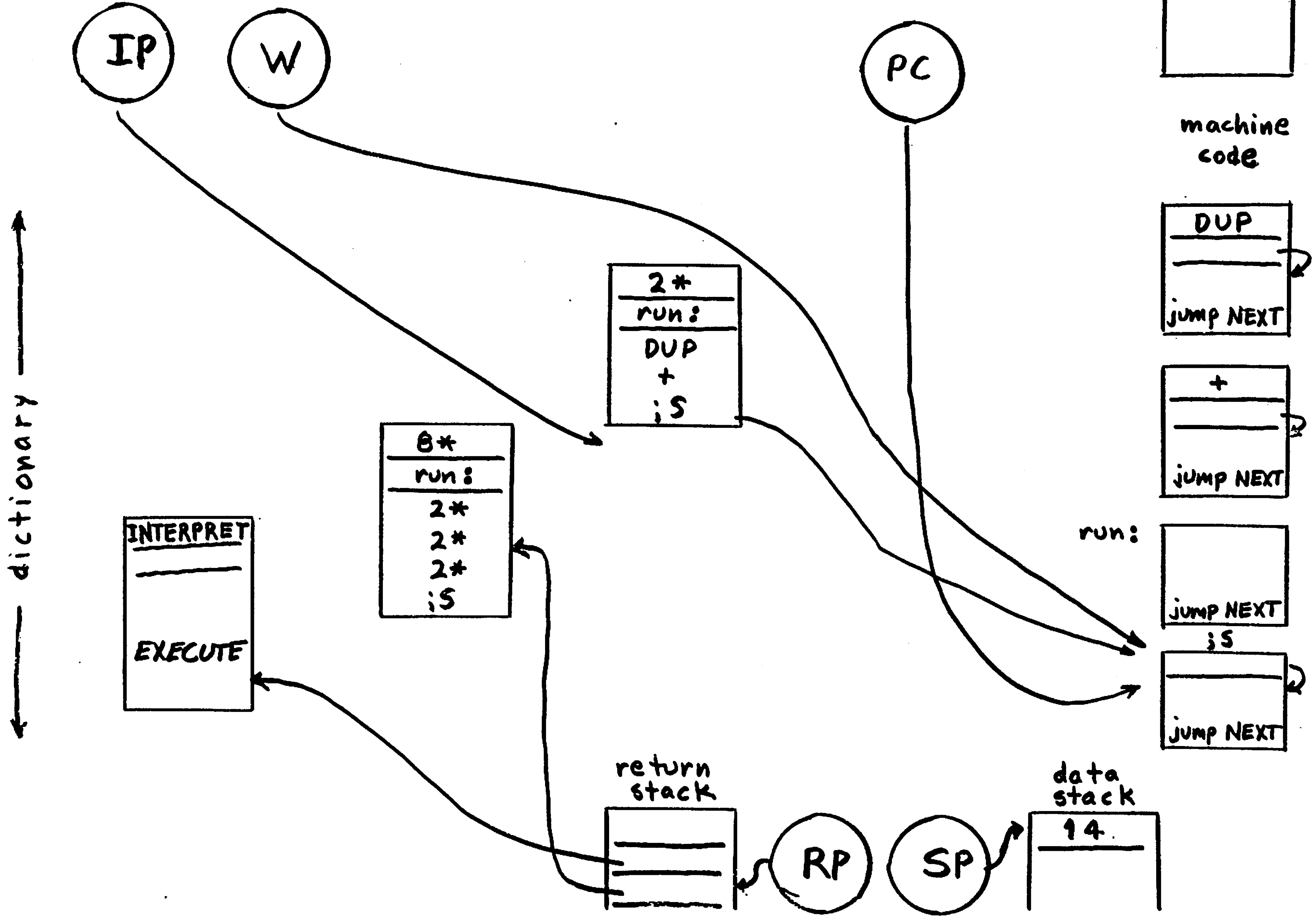


fig. 4.5

EXECUTION of 8*



I5 k fig

EXECUTION of 8*

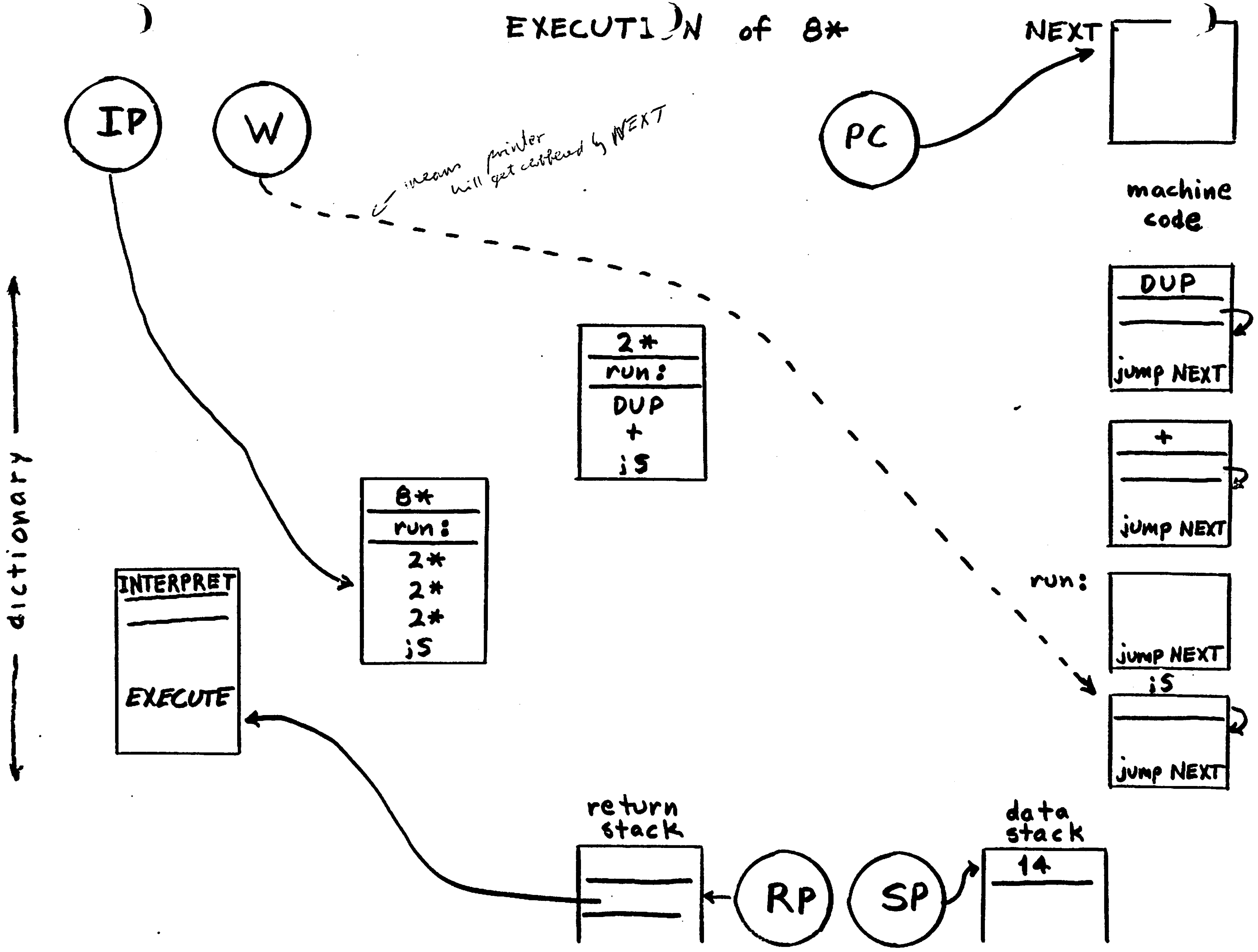


fig. 15

Why is this address interpreter fast inspite of the above overhead ?   NEXT in 2 instructions on PDP-11

Answer: all we do is fetch addresses

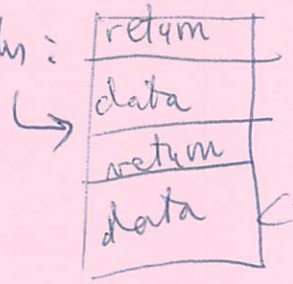
E.g. P-code interpreter of PASCAL has to a detailed set of case statements

microcoded versions of FORTH are running on HP 2100
2109

— may be coming on LSI-11

Why FORTH has 2 stacks

Burroughs:



to access this data, requires multiple indirect fetches

"frame activation & deactivation"

- turns out to be less efficient

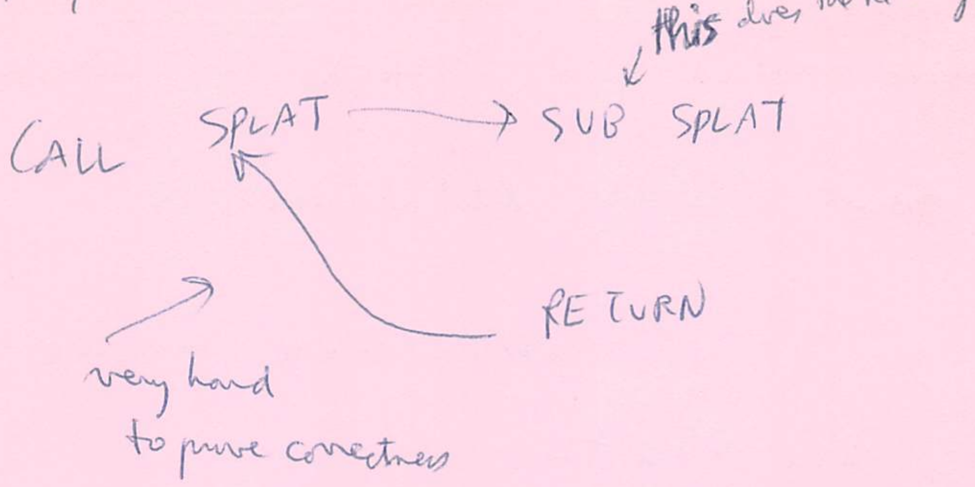
one major improvement over ALGOL

Use of W & indirect threaded code allows the <u>nesting</u> operation at the <u>caller</u>

& a structured nesting & unnesting

(over)

e.g in BASIC, FORTRAN



in FORTH, we don't have "CALL" → so we can test SPLAT independently

Forth class 6/29

1

240 B/SCR

```
* VARIABLE #START

: ELEMENT (subscript
           index--- addr) 2 * B/BUF /MOD #START
     @ + BLOCK + UPDATE;
```

(Test virtual array)

```
: INIT-ARRAY 500 0 DO I I ELEMENT ! LOOP;

: .ARRAY 0 DO CR I . SPACE I ELEMENT ? LOOP ;
```

(Make the virtual array into a file)

```
: AVAILABLE ~~#~~ (--- adr) #START @ BLOCK
     UPDATE ;
```

(e.g.)

```
0 AVAILABLE !
```

(Storing numbers)

```
: PUT 1 AVAILABLE +!
     AVAILABLE @ ELEMENT ! ;
```

eg. 3 PUT

(Inspecting file entries)

```
: SEE AVAILABLE @ 1+ 1 DO CR I . I ELEMENT ? LOOP ;
```

Forth Class 6/29/80

2

p. I26

see screen #52

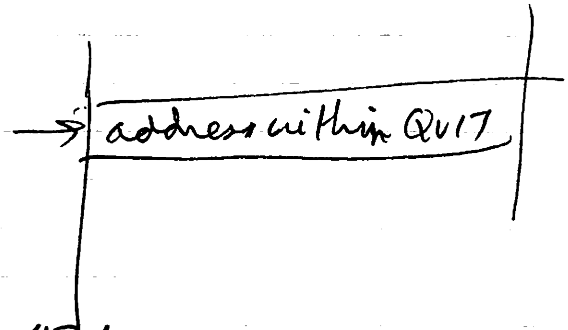
: INTERPRET -FIND

IF STATE @ <

- how to get out (at end of line):

see QUIT

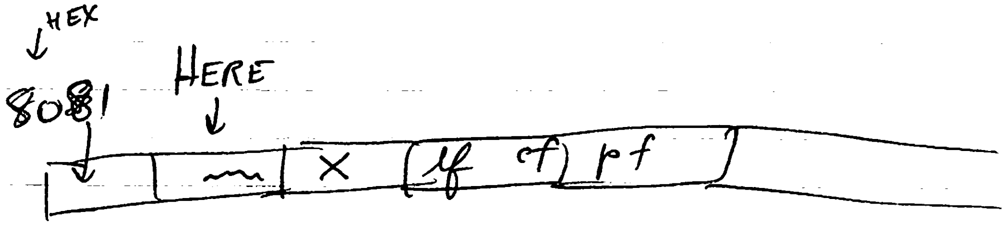
Top of Returnstack →



- uses word "null"
screen 45

When sreen 45 loaded:

8081 [illegible] is found by interpreter & put on data stack
HERE .. executed -- addr. it



! puts 8081 in addr of HERE

81
↕

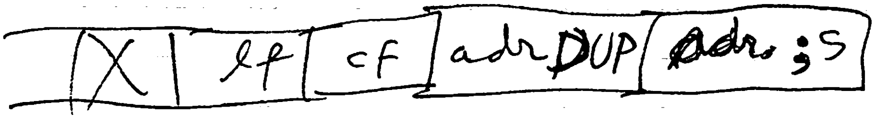
- this puts in a word with name 1 byte long
& name is ASCII null
↕
80

- purpose of null: to get us out of whatever we're doing

6/29/80 FORTH Class

Example for p. C9 r1

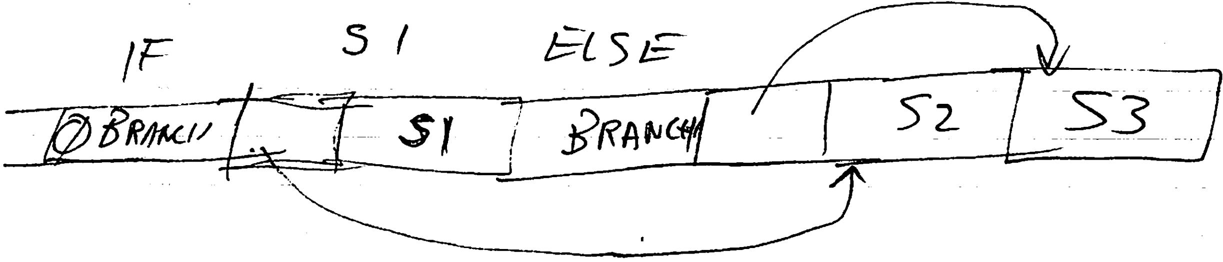
: X DUP ; ② ← time frame



P. C12 r1 NOTE: Data stack not used (by words) ~~by comp~~ when compiling so available for

P. C14
r1

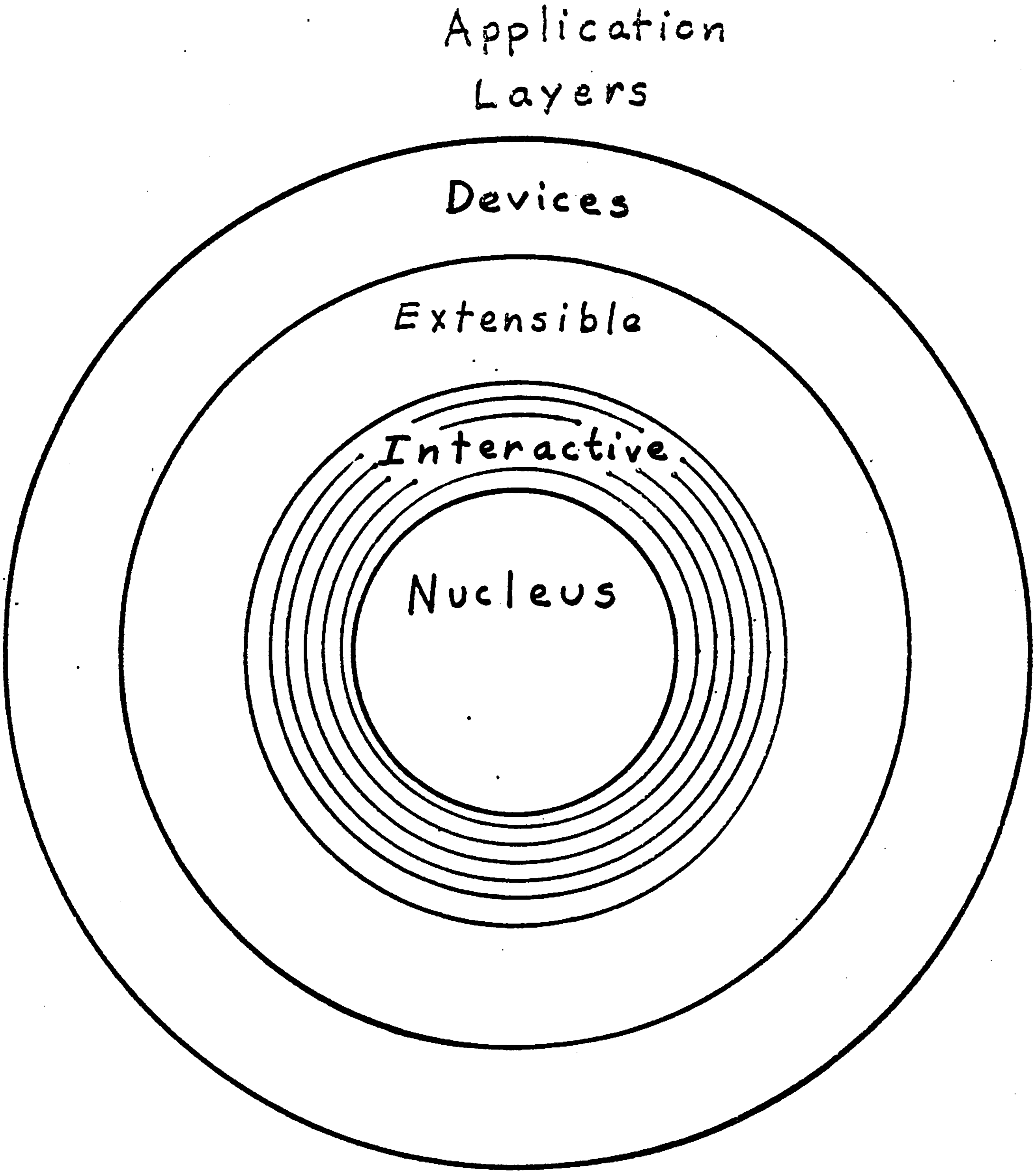
Desired dictionary result for "ELSE"



CORRECTIONS revised < & > that work!

```
< 2DUP XOR 0< IF DROP ELSE - THEN 0< ;

: U< 2DUP XOR 0< IF DROP 0< 0= ELSE - 0<
     THEN ;

SCR #56  : S→D DUP 0< MINUS ;

SCR #60  : FLUSH #BUF 1+ 0 DO 0
              BUFFER DROP LOOP ;
```

FORTH VOCABULARIES

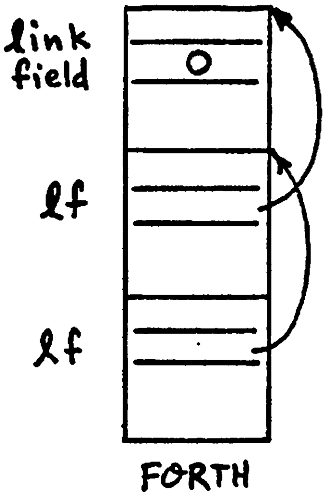


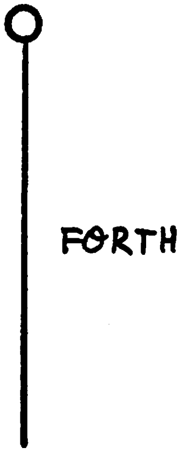
(about 5 yrs old)

V.2
a
fig

VOCABULARIES

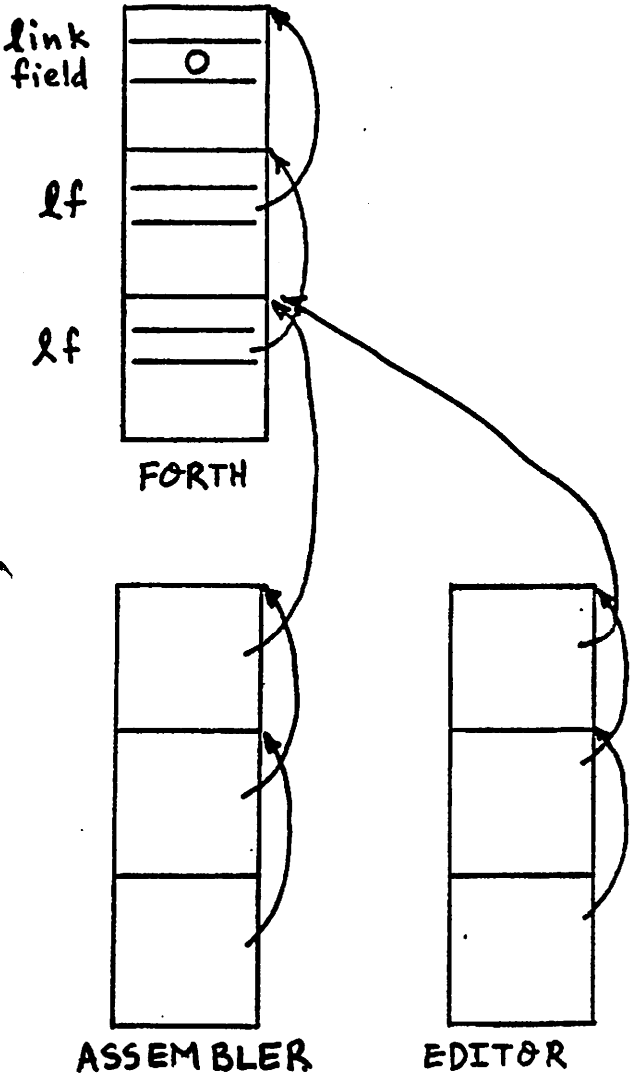
give definition names scope by restricting dictionary searches to a subset of the dictionary. This subset may be a general tree structure.

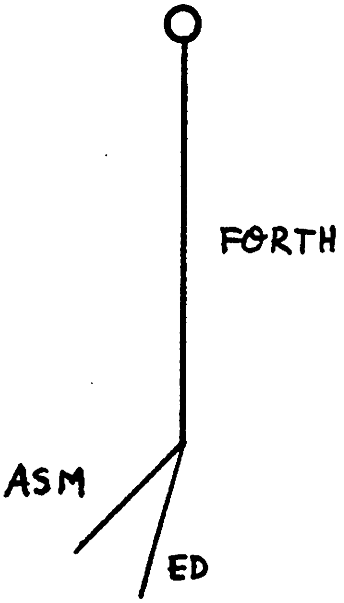




VOCABULARIES

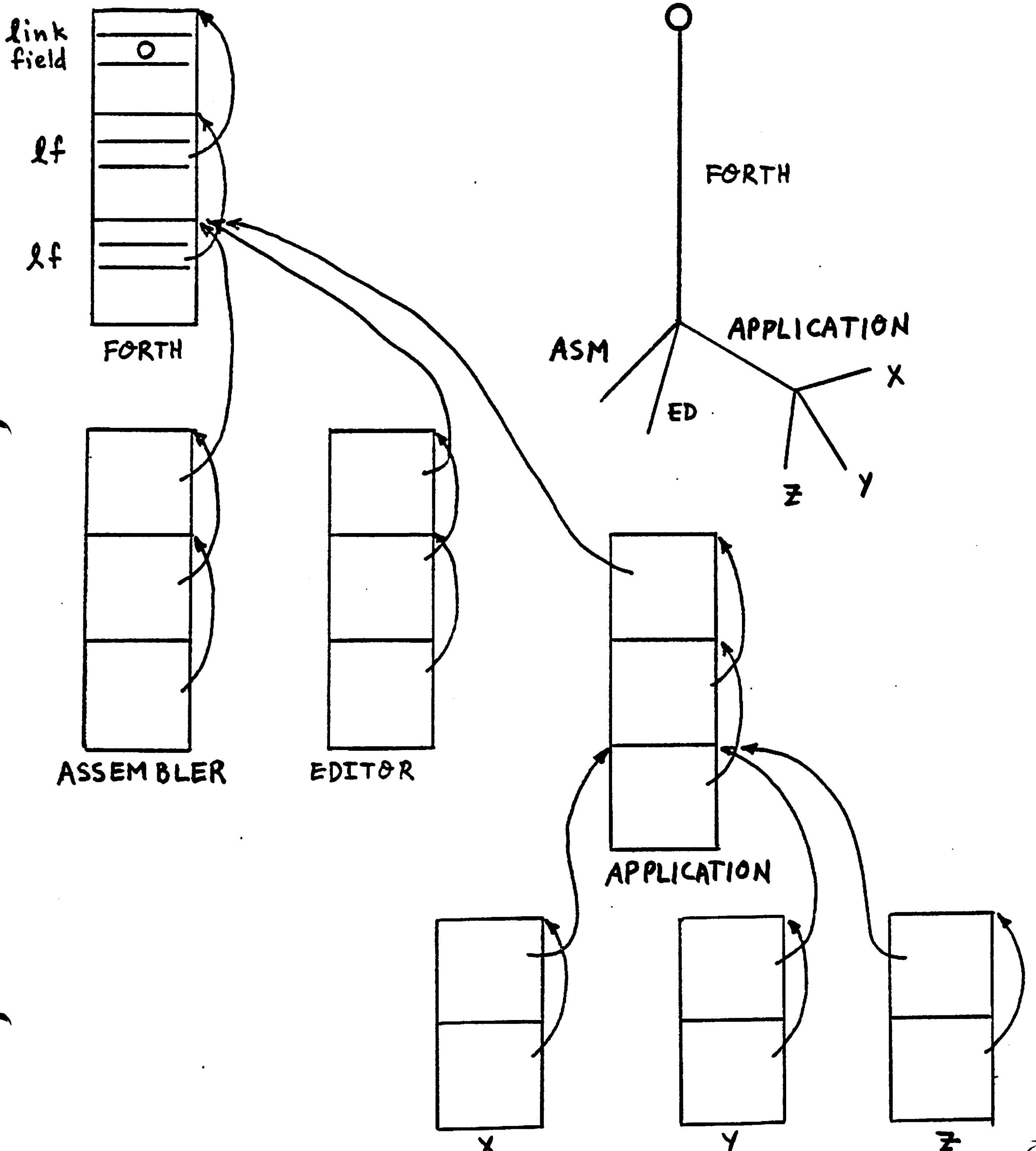
give definition names scope by restricting dictionary searches to a subset of the dictionary. This subset may be a general tree structure.





VOCABULARIES

give definition names scope by restricting dictionary searches to a subset of the dictionary. This subset may be a general tree structure.



Implementation requirements:

The dictionary is one memory area.
Space is allocated and deallocated like a stack.
Definitions may be added to any vocabulary at any time.

① Need a pointer to the last definition in each vocabulary. This pointer must be changed each time a definition is added to the vocabulary.

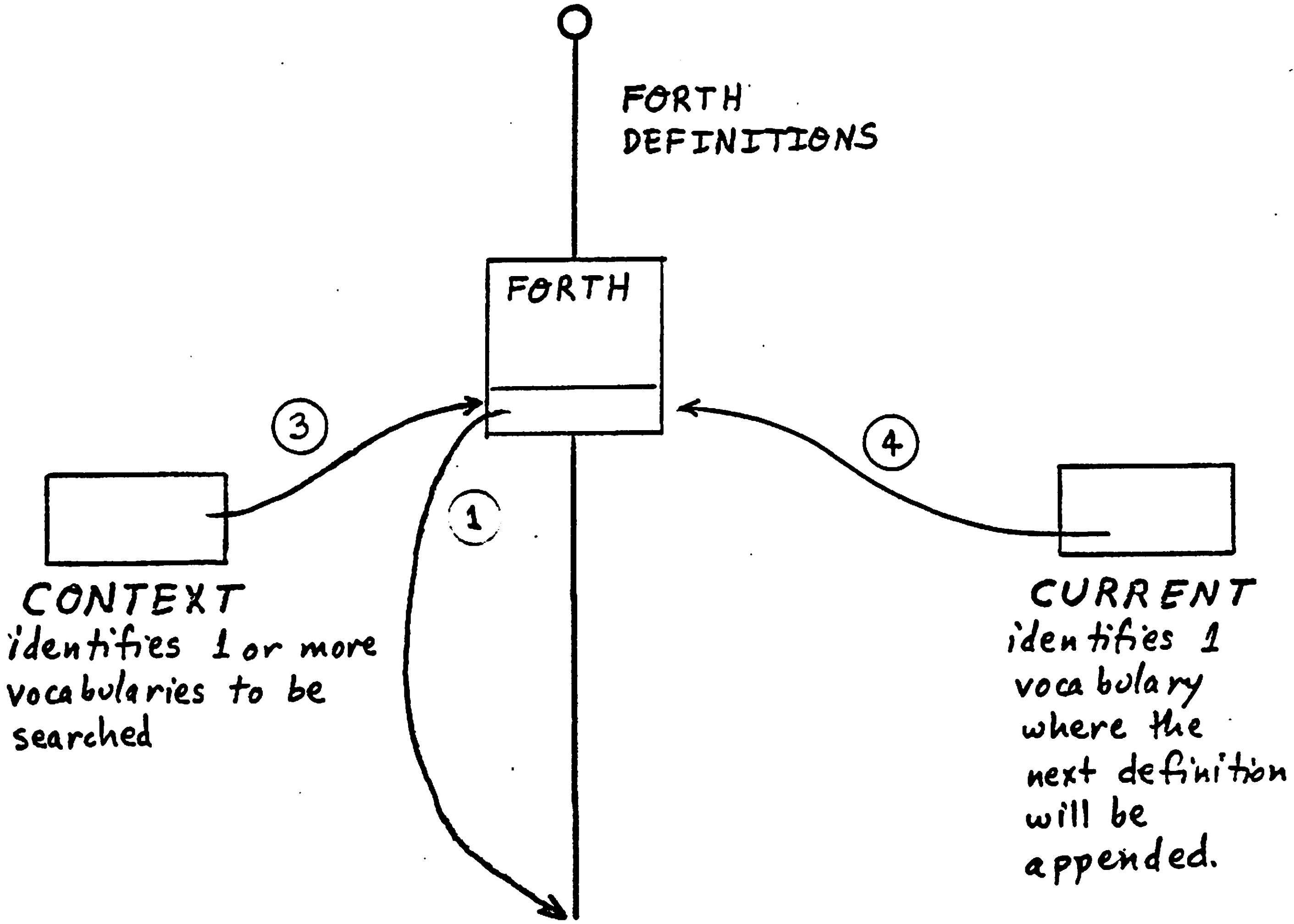
② Must be able to chain the first definition of a "leaf" vocabulary to the last definition of its parent vocabulary.

③ Must be able to identify the subtree which will be searched.

④ Must be able to identify the read and write vocabularies separately.

V.4
a
fig

fig-FORTH implementation technique:



V.4
b
fig

fig-FORTH implementation technique:

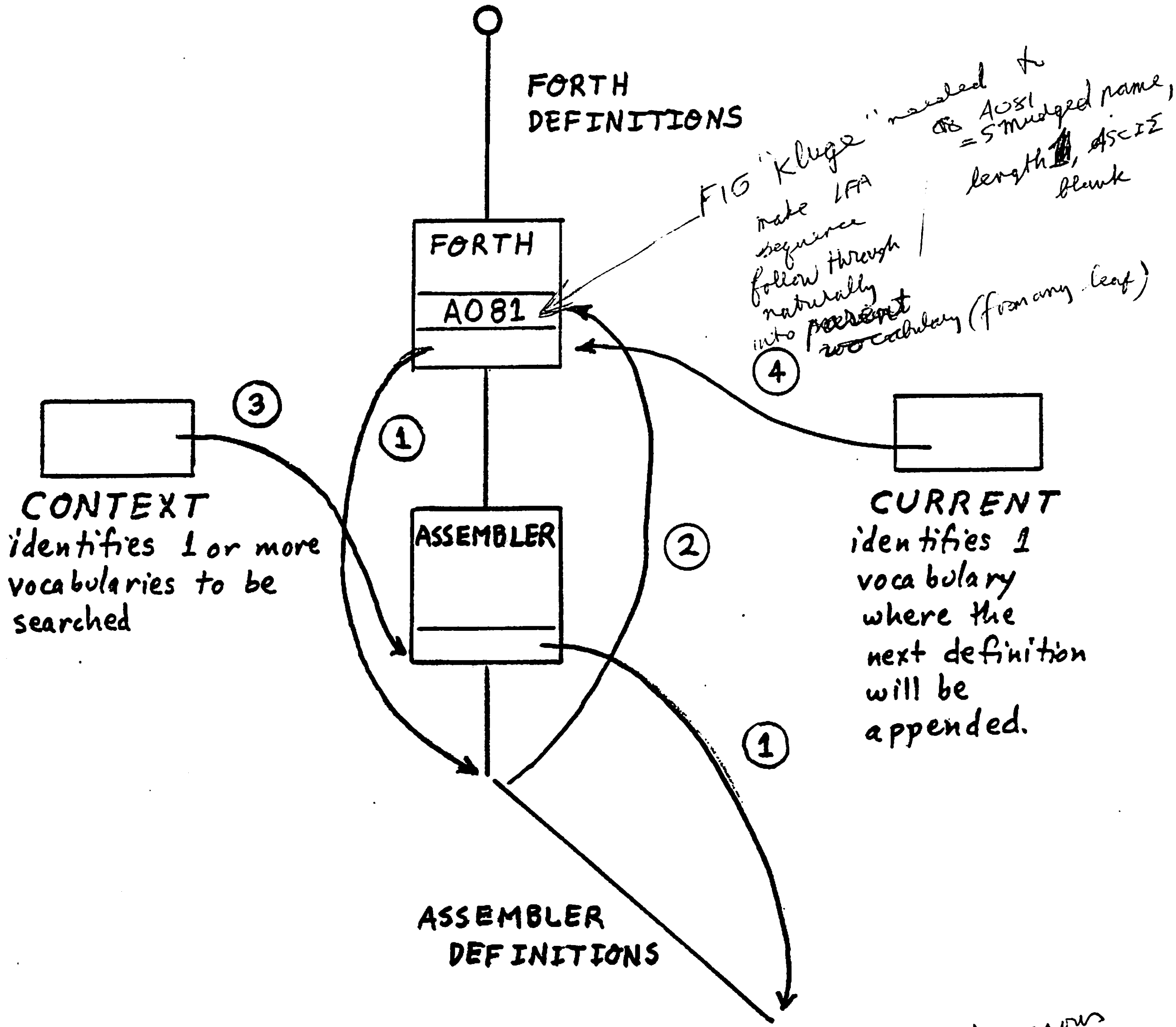


FIG "kluge" needed to make LFA sequence follow through naturally into ~~parent~~ vocabulary (from any leaf)

A081 = smudged name, length 1, ASCII blank

NOTE: FORGET is very dangerous when you have interwoven vocabularies

Defining a vocabulary

VOCABULARY name

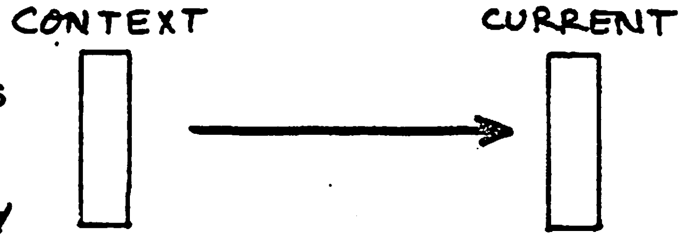
VOCABULARY FORTH
VOCABULARY ASSEMBLER
VOCABULARY EDITOR

Using vocabularies

vocabulary_name — *stores pointer to named vocabulary in CONTEXT.*
identifies vocabulary subtree which may be subsequently searched.

DEFINITIONS sets CURRENT from CONTEXT

- put new definitions in first CONTEXT vocabulary



: (colon) sets CONTEXT from CURRENT

- search CURRENT vocabulary first

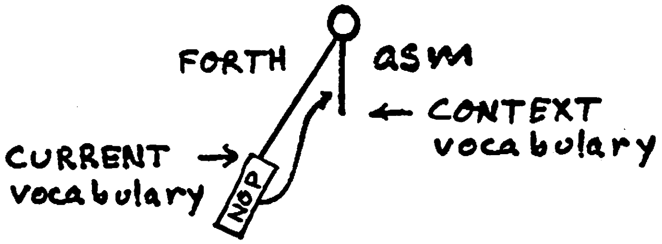
Reference definitions in the CONTEXT vocabularies.

Append new definitions to the CURRENT vocabulary.

Example:

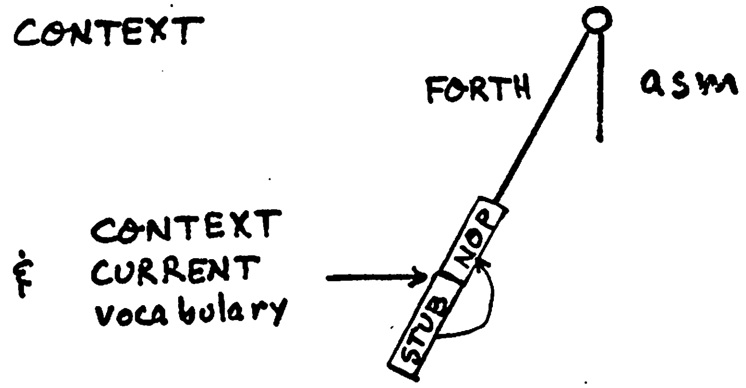
Assume FORTH is both the CONTEXT and CURRENT vocabulary.

ASSEMBLER
changes the
CONTEXT vocabulary



CODE NOP NEXT

: restores the CONTEXT vocabulary

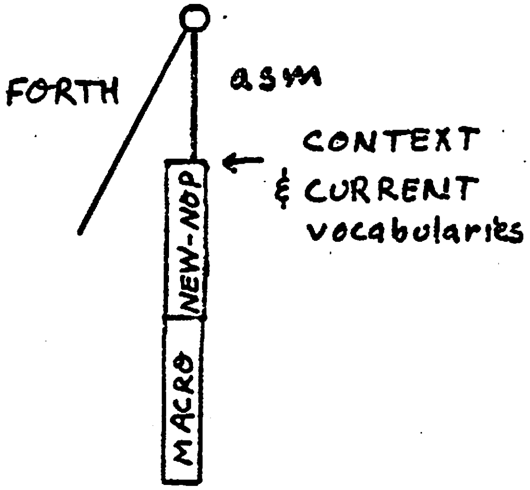


```
: STUB    NOP ;
```

V5
b
r2

ASSEMBLER DEFINITIONS

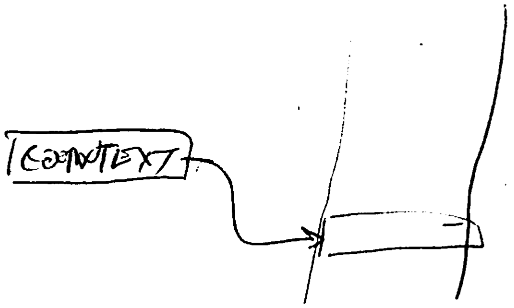
changes both CONTEXT
& CURRENT vocabularies
to assembler



```
CODE  NEW-NOP  NEXT
```

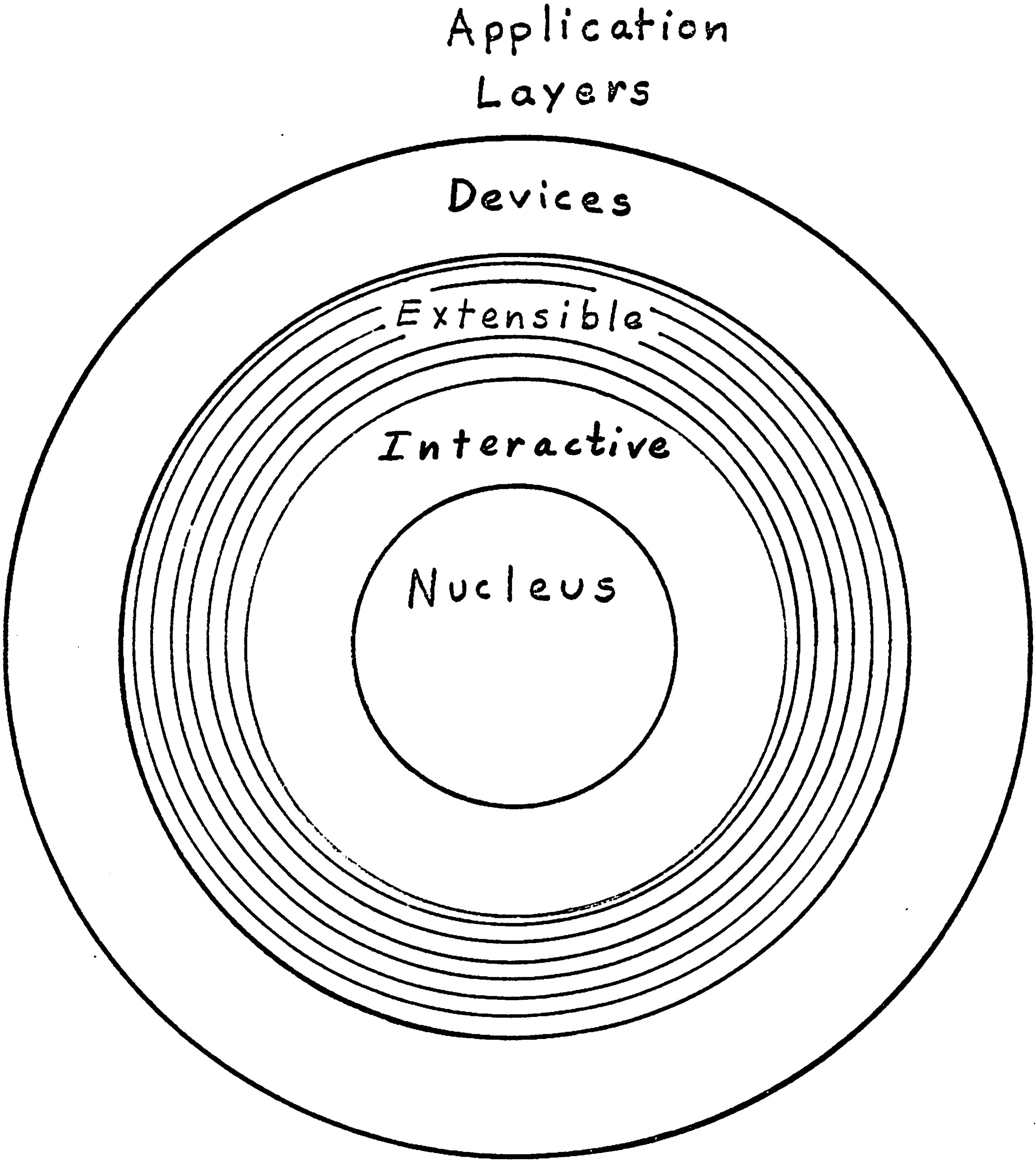
```
:  MACRO  NEW-NOP  NEW-NOP  ;
```

```
: ?VOC  CONTEXT @  4 -
        NFA ID. ;
```

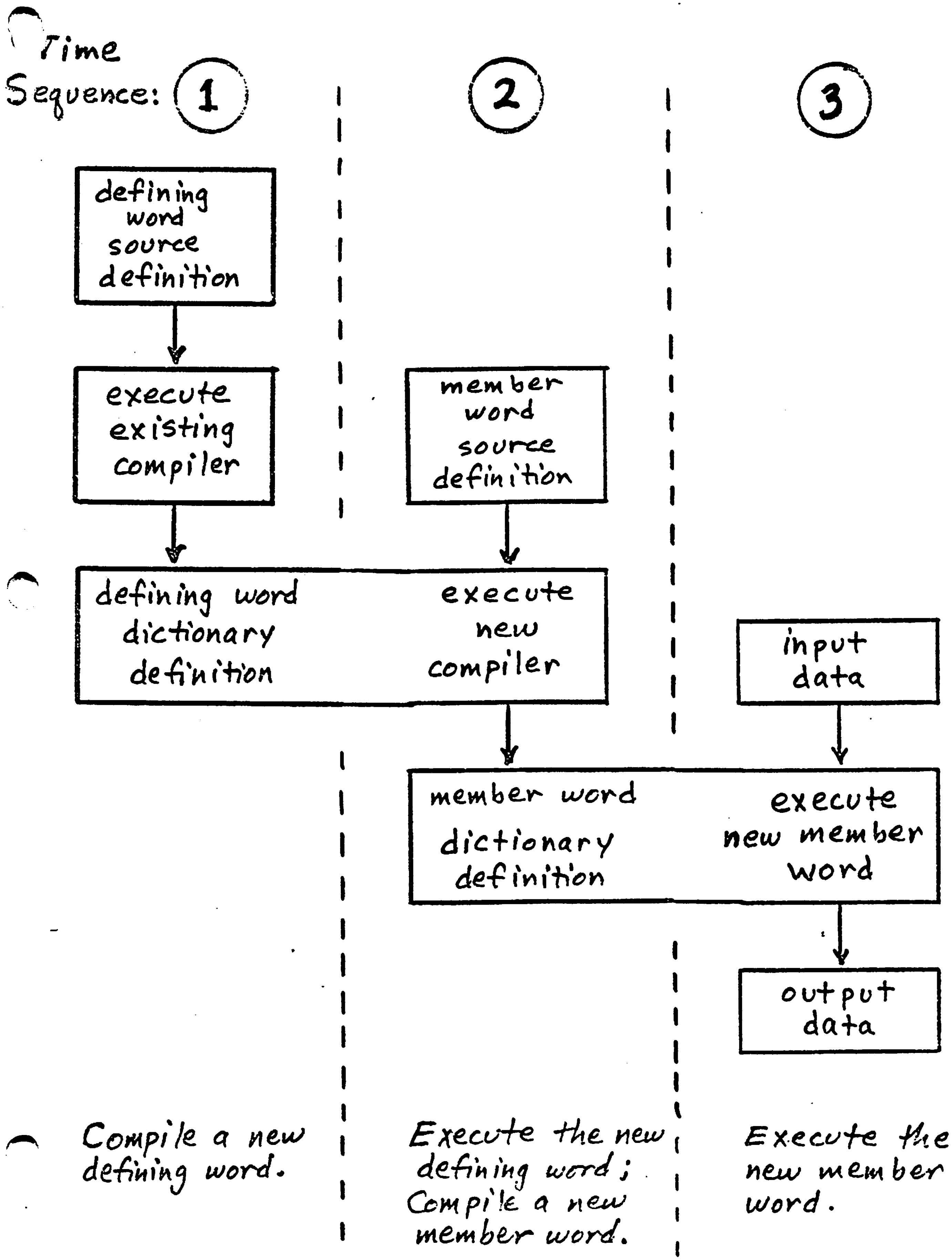


DEFINING WORDS

(or how to write a compiler in 25 words or less)



USING DEFINING WORDS



Compile a new defining word.

Execute the new defining word; Compile a new member word.

Execute the new member word.

DEFINING WORDS

are FORTH definitions which, when executed, create entire new definitions in the dictionary.

Predefined defining words:

:
CONSTANT
VARIABLE
USER
VOCABULARY
CODE

It is possible to create new defining words (ie, specialized compilers) which can subsequently be used to create a new family of member words.

Defining words are useful for creating data structures and procedures which share a common execution-time behavoir.

Proper use can substantially

reduce software development time,

reduce program size,

and improve readability

with no execution-time penalty.

D2
r2
fig

To define a new defining word,
an existing defining word (eg, :) is used.
This occurs at sequence ①.

The definition specifies the
compile-time activity ②
and the
execution-time activity ③
of each member of the family.

The form of a new defining word's definition is

: new-defining-word

② <BUILDS compile-time words

③ DOES> execution-time words
or
;CODE assembly language

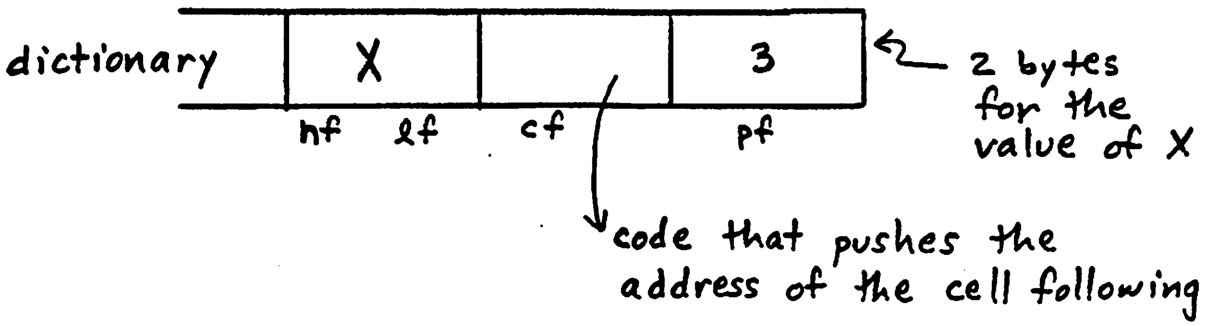
;

D3
r1
fig

Example of a high-level definition of a defining word: VARIABLE

② member creation-time

3 VARIABLE X



③ member execution-time

3 X !

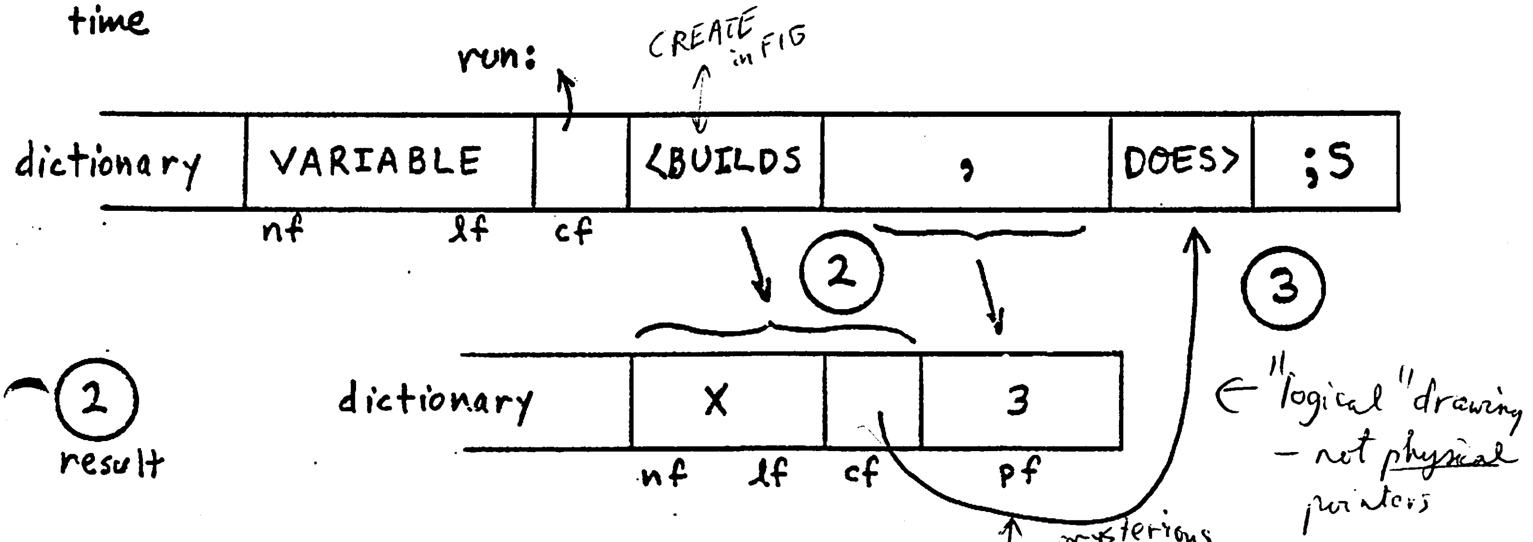
X ? Ⓒℝ 3 ok

① defining word creation-time

: VARIABLE

<BUILDS , DOES> ;

actually, FIG uses ;CODE for speed



mysterious "GREEN ARROW"! see yellow notes (142)

: CONSTANT <BUILDS , DOES> @ ;

Create a 1 Dimensional byte array

Determine the compile-time action of member words:

② use defining word

```
               ← new defining word
 5    CVECTOR    LINE
 ↑                ↑
 # bytes          array name
```

Determine the execution-time action of member words:

③ use member word

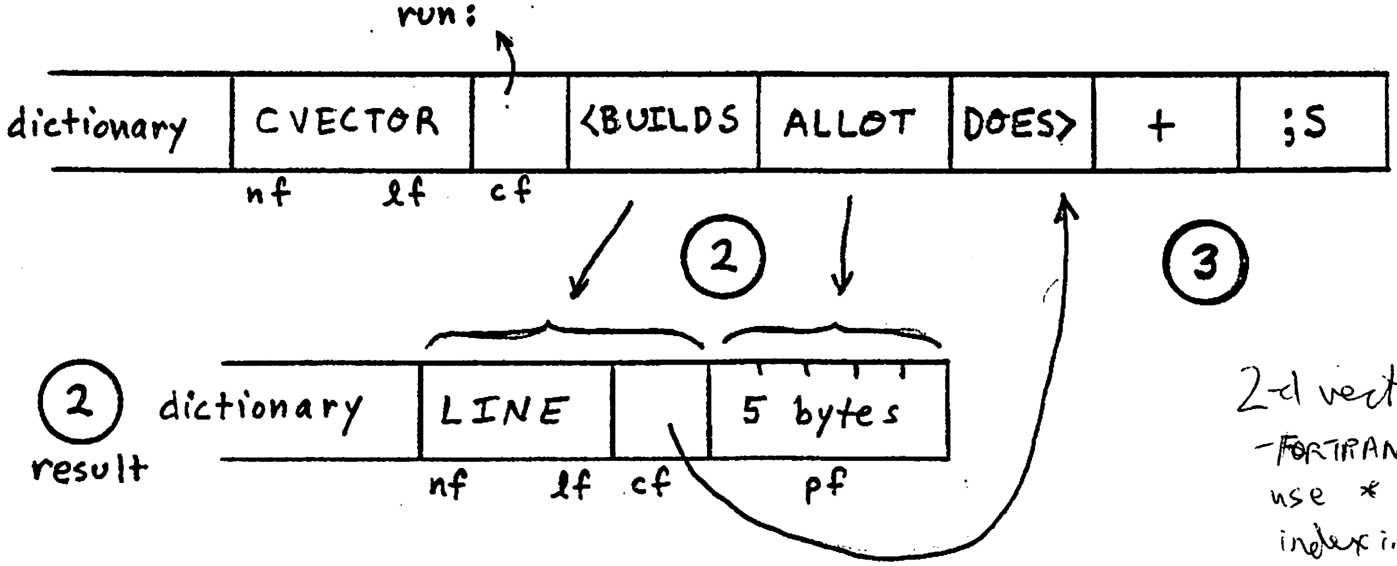
```
 3   LINE  C@
 ↑       ↖——— returns address of
 subscript      selected byte
```

Define the new defining word:

no. elts ---

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

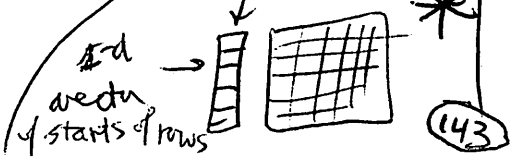
① create defining word



2-d vector array
-FORTRAN etc
use * & + to
index into 1-d
array
-better way

2 byte array:

```
: VECTOR <BUILDS 2* ALLOT DOES> SWAP 2* + ;
```



DS
r1
fig

Examples of using LINE:

```
: FILL-LINE   5 0 DO  65 I +  I LINE C!
                    LOOP ;

: PRINT-LINE   5 0 DO  I LINE C@  EMIT
              SPACE  LOOP ;

FILL-LINE (CR)  ok
PRINT-LINE (CR)  A B C D E ok
```

Variations on CVECTOR:

Subscripts starting from 1 (instead of 0)

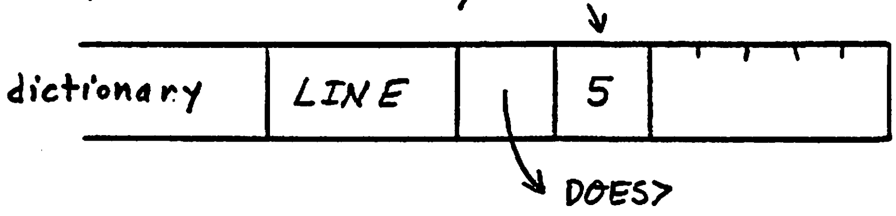
```
: CVECTOR    <BUILDS    ALLOT
             DOES>  +    1-     ;
```

Initialize member arrays to blanks when they are created

```
: BLANK&ALLOT    ( #bytes --- )
     HERE  OVER  BLANKS
     ALLOT   ;

: CVECTOR    <BUILDS    BLANK&ALLOT
             DOES>  +   ;
```

Check subscript range on each reference to all member words

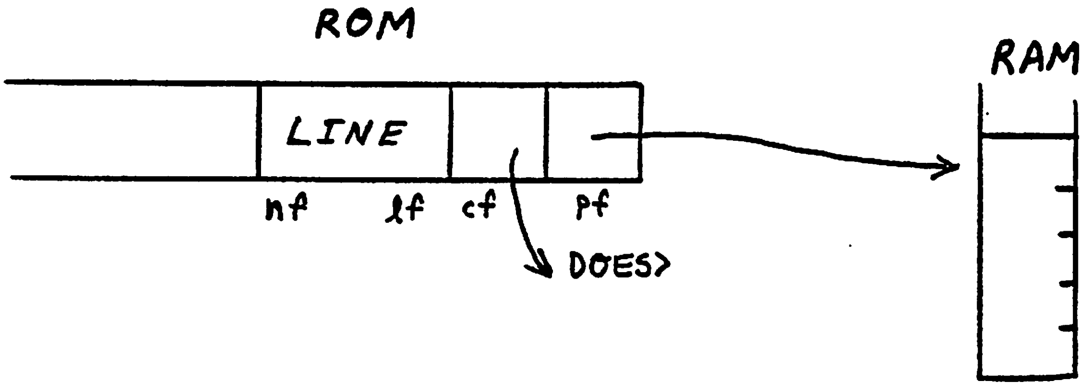
Must store array size in member's definition



```
                     OVER OVER
: CVECTOR   <BUILDS   DUP  ,  ALLOT
      DOES>  2DUP  @  U<
         IF   +  2+
         ELSE  @  .  .
                 ." Range error" ABORT
         THEN  ;
```

checks for negative subscripts ← checks signed values

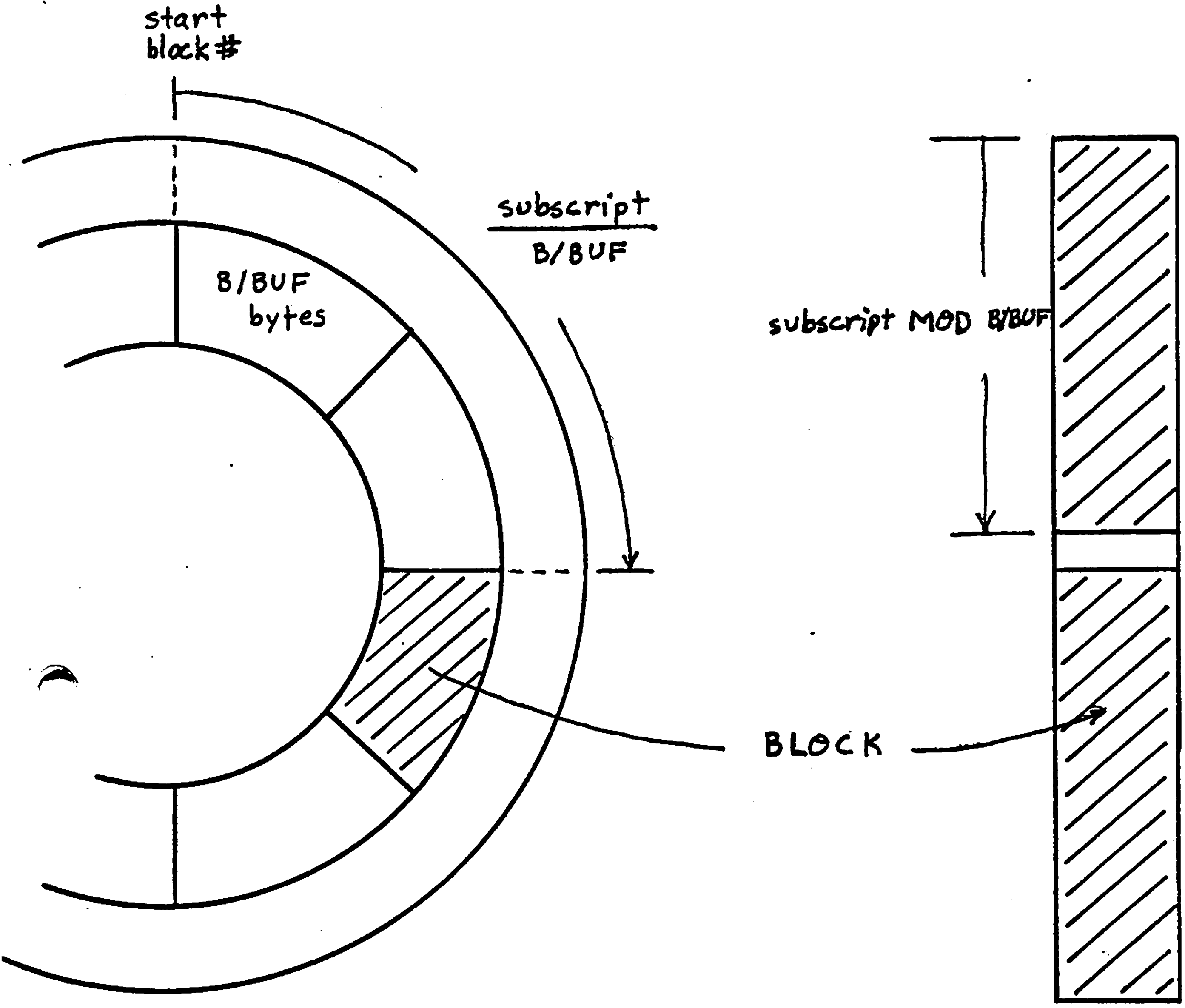
Definition in ROM, data in RAM (writable memory)



```
                                   in ROM      in RAM
: CVECTOR   <BUILDS   THERE   ,   ALLOT
      DOES>  @  +  ;
```

D7
r1
fig

Virtual array: definition in the dictionary, data on mass storage



no. elts ---

```
: CVECTOR   <BUILDS      start-block# ,    DROP
            DOES>   @   SWAP
                 B/BUF  /MOD   ROT   +
                    BLOCK   +      UPDATE   ;
```

D8
r1
fig

Defining word example: CASE: execution vector

Define some cases

```
:  0PET  ." DOG " ;        :  1PET  ." CAT " ;

:  2PET  ." RAT " ;        :  3PET  ." SNAKE " ;
```

Using defining word

② source definition

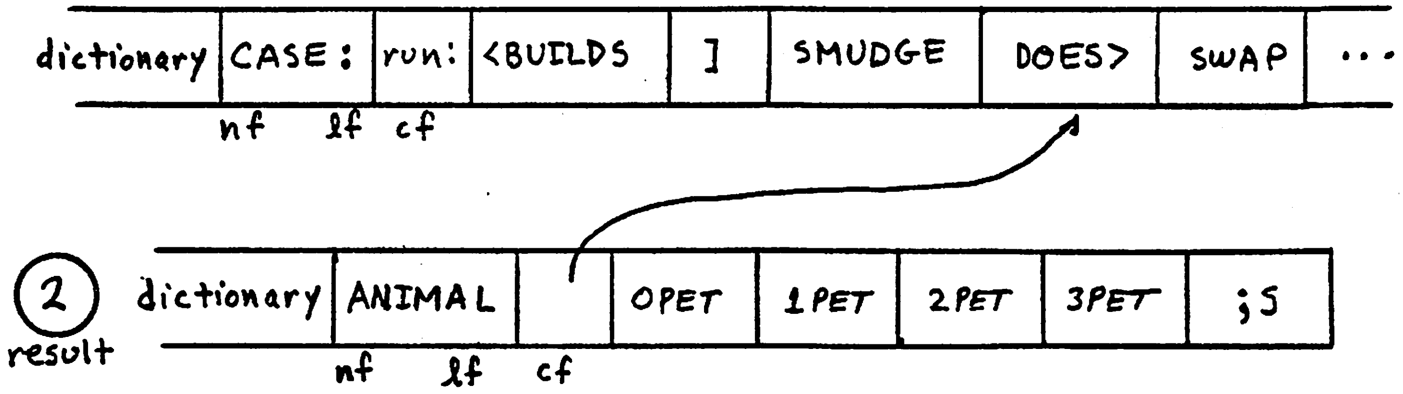
```
CASE:  ANIMAL  0PET  1PET  2PET  3PET  ;
```

Creating defining word

①

```
:  CASE:  <BUILDS  ]  SMUDGE

             DOES>  SWAP  2*  +  @

                  EXECUTE  ;
```

needed because ~~<BUILDS~~ ;S smudges so we need to unsmudge



Using member word

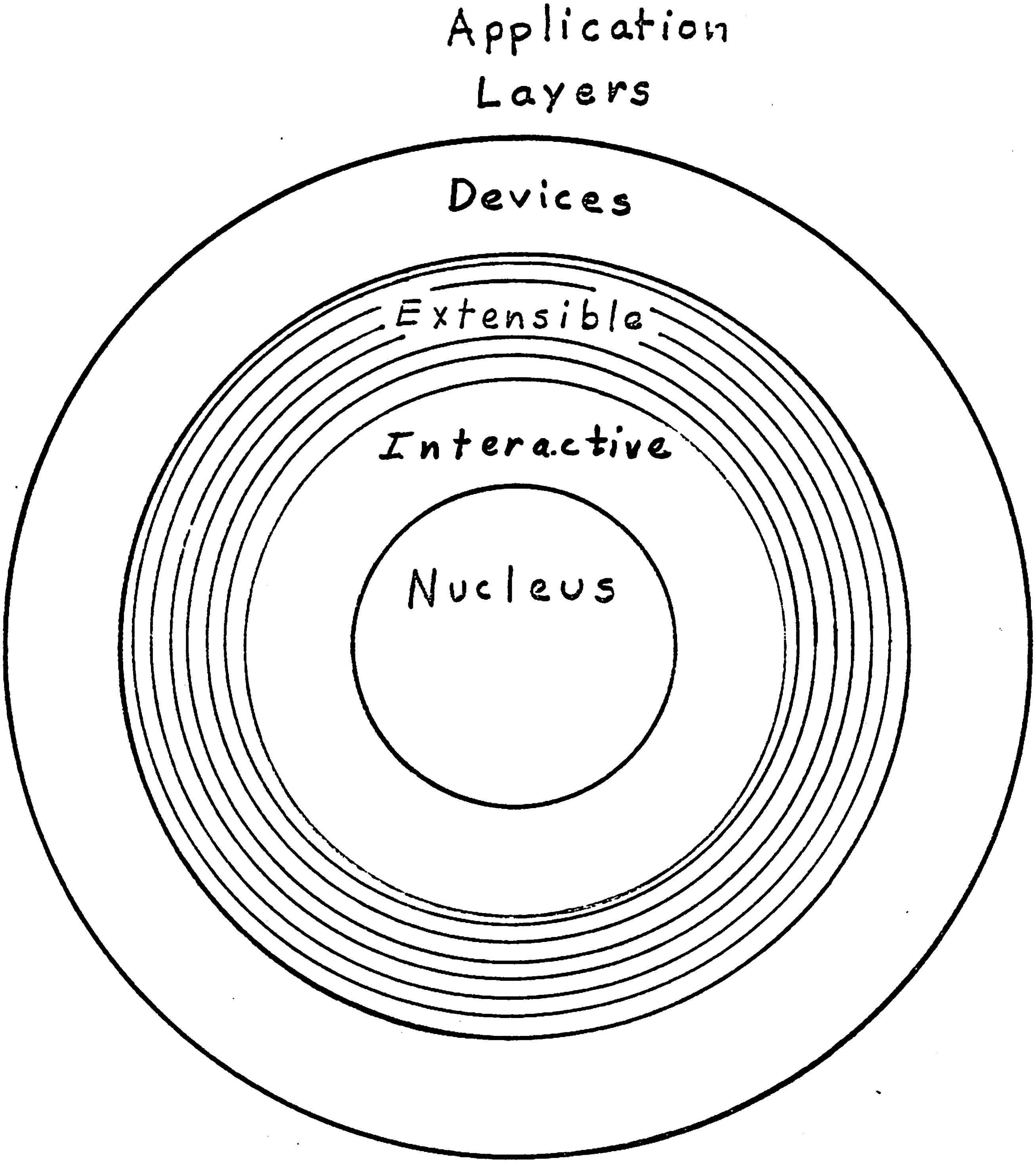
③

```
0  ANIMAL  (CR)  DOG  ok
1  ANIMAL  (CR)  CAT  ok
3  ANIMAL  (CR)  SNAKE  ok
```

← e.g. Kim used ~~this~~ for a random phrase generator

SOFTAPE :
;: is <BUILDS , DOES>

FORTH ASSEMBLER



FORTH ASSEMBLER

ATTRIBUTES:

"CODE" words interface exactly like ":" words
    universal reference
    stack arguments

Allows full machine speed
and full access to hardware details
    carry, overflow flags
    interrupts

Resident vocabulary
Source from keyboard or disk,
Object code to normal memory (normal mode)
    or to disk and alternate memory space
        (target compiling mode)
Macro capability
Structured programming control structures
"Meta assembler" (table driven) allows
    full control over assembly process
All capabilities of FORTH system available during
    assembly, eg. assembly-time calculations,
        dictionary search,
        editing.

USEAGE:

CODE name	body	ending
CREATEs dictionary head for "name" and invokes ASSEMBLER vocabulary (as the CONTEXT vocabulary)	assembler words literals FORTH words	jump to address interpreter or multitasker

Examples are for LSI-11 poly FORTH

Endings:

NEXT	(macro)	jump to address interpreter
POP JMP		discard top of stack, jump to NEXT
PUSH JMP		push register 0 onto stack, jump to NEXT
PUT JMP		store register 0 into top of stack, jump to NEXT
WAIT	(macro)	jump to multitasker (like PAUSE)

Assembler words:

1 operand instructions:

operand mnemonic

eg. CLR, NEG, ASL

2 operand instructions:

destination-operand source-operand mnemonic

eg. ADD MOV XOR

(Note: operand order is opposite on 8080's.)

Operands may be registers, numeric values (eg, immediate data, addresses), and addressing mode modifiers.

Registers :

number	name	(assignment)
0		scratch
1		scratch
2	W	
3	U	User variables base
4	I	Interpreter
5	S	Stack pointer
6	R	Return stack pointer
7	PC	processor's Program Counter

Immediate data, addresses:

value #

eg. CODE ONE 0 1 # MOV PUSH JMP

(*Traditional* assembler syntax: MOV #1,0)

Addressing mode modifiers:

Relative addressing reg)

eg. CODE MINUS S) NEG NEXT

(*Traditional* syntax: NEG (S))

Relative, post increment reg)+

eg. CODE DROP S)+ TST NEXT

(*Traditional* syntax: TST (S)+)

Relative, pre decrement reg -)

eg. LABEL PUSH S -) 0 MOV NEXT

(*Traditional* syntax: MOV 0,(S-))

```
Indexed                        displacement   reg)

     eg.   CODE SWAP          0       2 S)   MOV
                          2  S)       S  )   MOV
                             PUT    JMP
```

Conditional control structures:

result-flag IF true-phrase $\underbrace{\text{ELSE false-phrase}}_{\text{optional}}$ THEN

BEGIN loop-body result-flag END

result flags:

0<	Negative flag set
0>	Negative flag clear
0=	zero flag set
CS	Carry flag Set
VS	oVerflow flag Set

Macros: while in ~~the~~ ASSEBLER DEFINITIONS

```
: macro-name      assembler words     ;
```

eg.

```
:  NEXT       W    I  )+  MOV
                W  )+  )  JMP         ;
```

Interrupts:

address-interrupt-code   address-interrupt-vector   INTERRUPT

installs interrupt

interrupt-code must be CPU code (not code field addr)

Interrupt routine form:

```
LABEL   A/D      ...        RTI
```

To install this code at address 177777_8 :

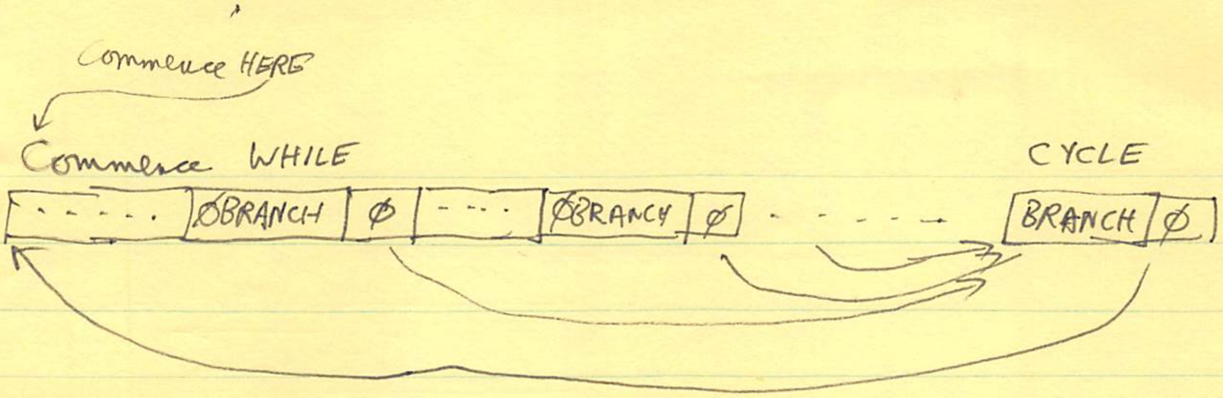
```
A/D   177777   INTERUPT
```

PS-E6
fig

```
 0 ( Solution:  Multiexit loop structure )
 1
 2 : <-BRANCH   HERE - ,    ;
 3 : ->BRANCH   HERE OVER -   SWAP !    ;
 4
 5 : COMMENCE   HERE   0    ;    IMMEDIATE
 6 : &WHILE   COMPILE 0BRANCH   HERE 0 ,    ;    IMMEDIATE
 7 : CYCLE   COMPILE BRANCH    0 ,
 8     BEGIN    -DUP WHILE    ->BRANCH    ?STACK    REPEAT
 9     -2 ALLOT    <-BRANCH    ;    IMMEDIATE
10
11 : MULTI-TEST   COMMENCE   DUP .    1 - DUP &WHILE
12     DUP .    1 - DUP &WHILE   DUP .    1 - DUP &WHILE
13     CR    CYCLE    DROP    ;
14

10 MULTI-TEST 10 9 8
7 6 5
4 3 2
1 OK
9 MULTI-TEST 9 8 7
6 5 4
3 2 1 OK
6 MULTI-TEST 6 5 4
3 2 1 OK
2 MULTI-TEST 2 1 OK
. 46 .?  Empty Stack
```

```
 0 ( Multi-exit sequence structure )
 1
 2 : &IF   [COMPILE] &WHILE    ;    IMMEDIATE
 3 : FIN   BEGIN   -DUP WHILE    ->BRANCH   ?STACK    REPEAT
 4     DROP    ;    IMMEDIATE
 5
 6
 7 : SEQ-TEST   COMMENCE   DUP .    1 -    DUP &IF
 8     DUP .    1 -    DUP &IF   DUP .    1 -    DUP &IF
 9     DUP .    FIN    DROP    ;
10
11
12
13
14
15
OK
10 SEQ-TEST 10 9 8 7 OK
4 SEQ-TEST 4 3 2 1 OK
3 SEQ-TEST 3 2 1 OK
2 SEQ-TEST 2 1 OK
1 SEQ-TEST 1 OK
0 SEQ-TEST 0 -1 -2 -3 OK
. 46 .?  Empty Stack
```

7/5/80


Commence HERE
Commence
WHILE
CYCLE
. . . . .
0BRANCH
0
. . . .
0BRANCH
0
. . . . . .
BRANCH
0

CVECTOR for virtual array

start block# ← decided initially

or alternatively
could keep a
variable
DISK-HERE

& write a word
DISK-ALLOT

Write BASED.

② 16 BASED. H.

(2) in H.

exec. time
1) save BASE
2) set BASE to value
3) restore

: BASED. <BUILDS , DOES> ~~BASE DUP~~ desire base base addr base addr
BASE ~~DUP~~ @ base addr cur. base desired base base addr cur. base
OVER ~~BASE~~ DUP base addr cur. base desired base base addr base addr
~~SWAP~~ ROT base addr base addr base addr desir base
SWAP ! base addr

~~BASE DUP @~~ desired base base addr cur. base
~~ROT~~ base addr

BASE @ SWAP @ BASE ! SWAP . BASE ! ~~SWAP~~ ;

note: this is same as CONSTANT

class ---

: LOADED-BY $\boxed{\text{<BUILDS ,}}$ DOES>

@ LOAD ;

The FIG FORTH "kludge" for <BUILDS DOES>