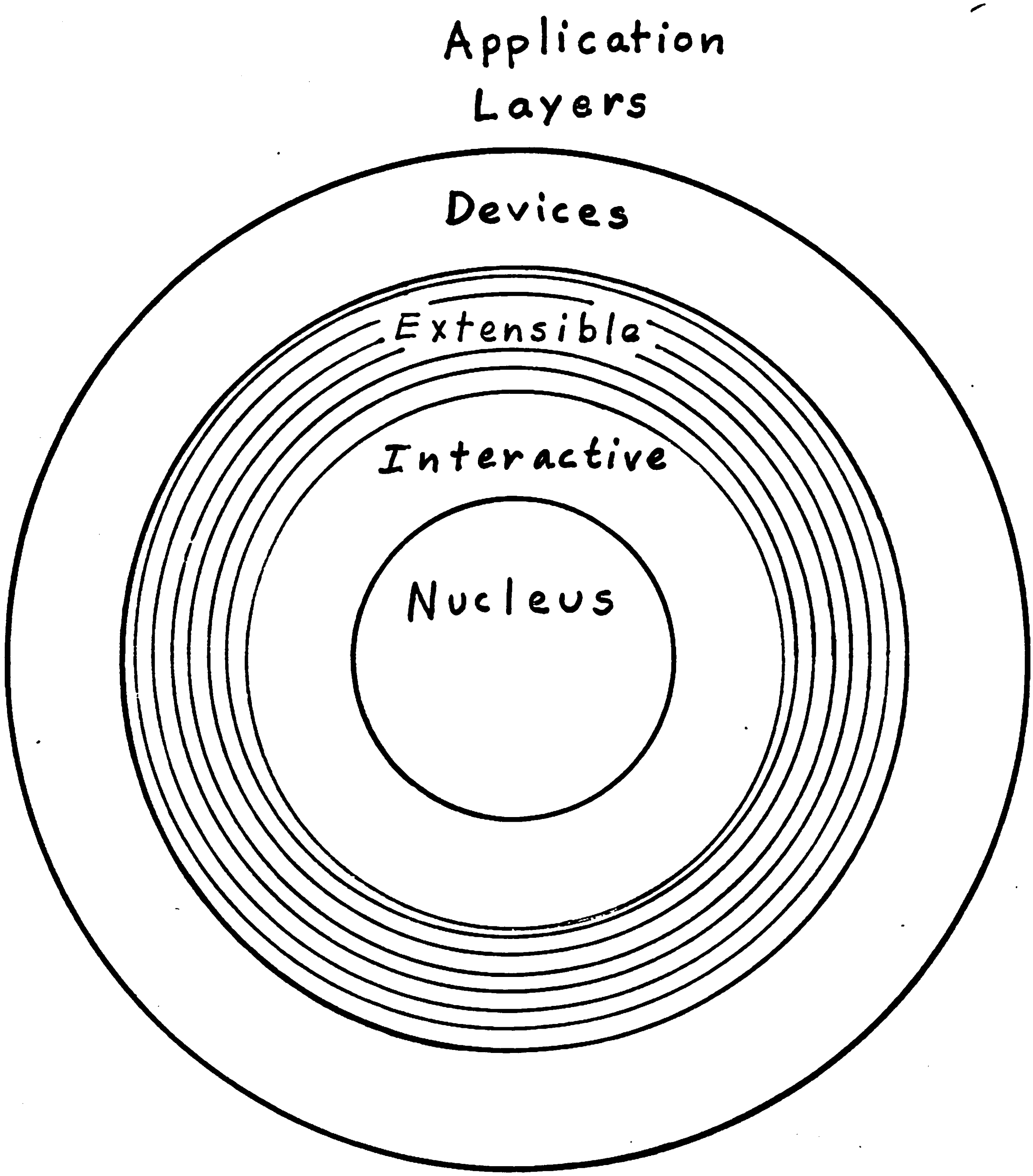
FORTH COMPILER



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

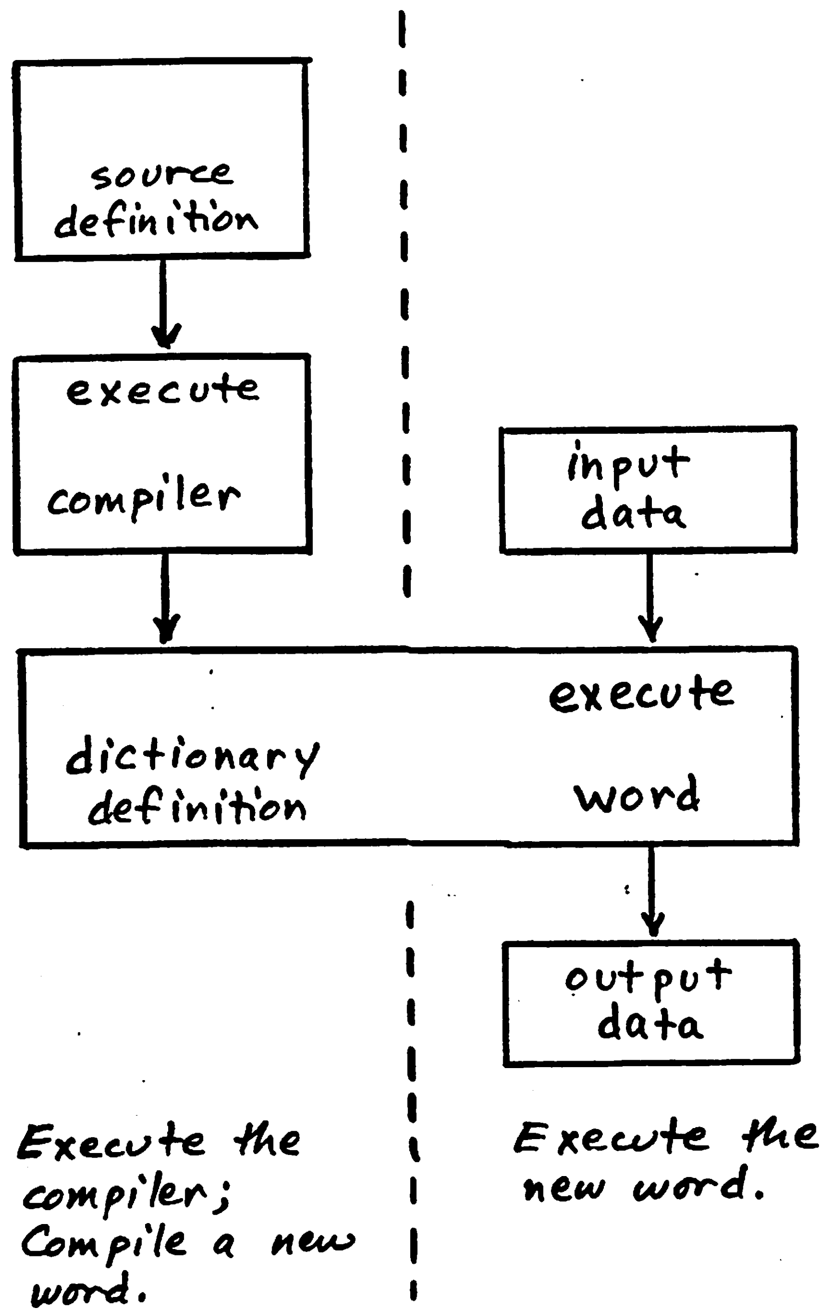
: new_word definition ;

(definition = previously defined words or numbers)

examples

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

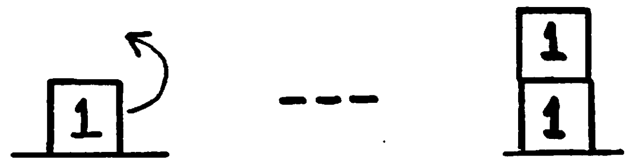
USING A FORTH COMPILER



16 bit STACK MANIPULATION OPERATORS

DROP

DUP

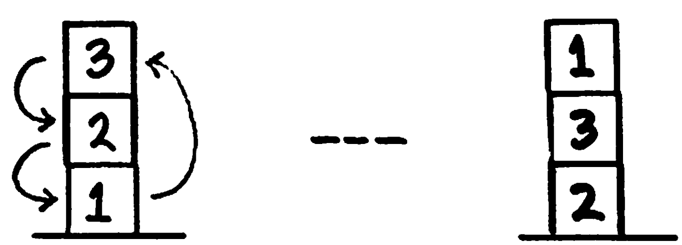


should never need more than 2 of these between other words

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

VARIABLES

— a symbol whose value can be changed.

defining a variable

value VARIABLE name

(value = initial value)

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

definition	useage
: 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 ! ---

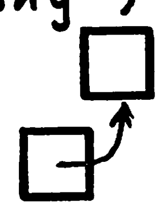
examples

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

This is a very simple and general capability.

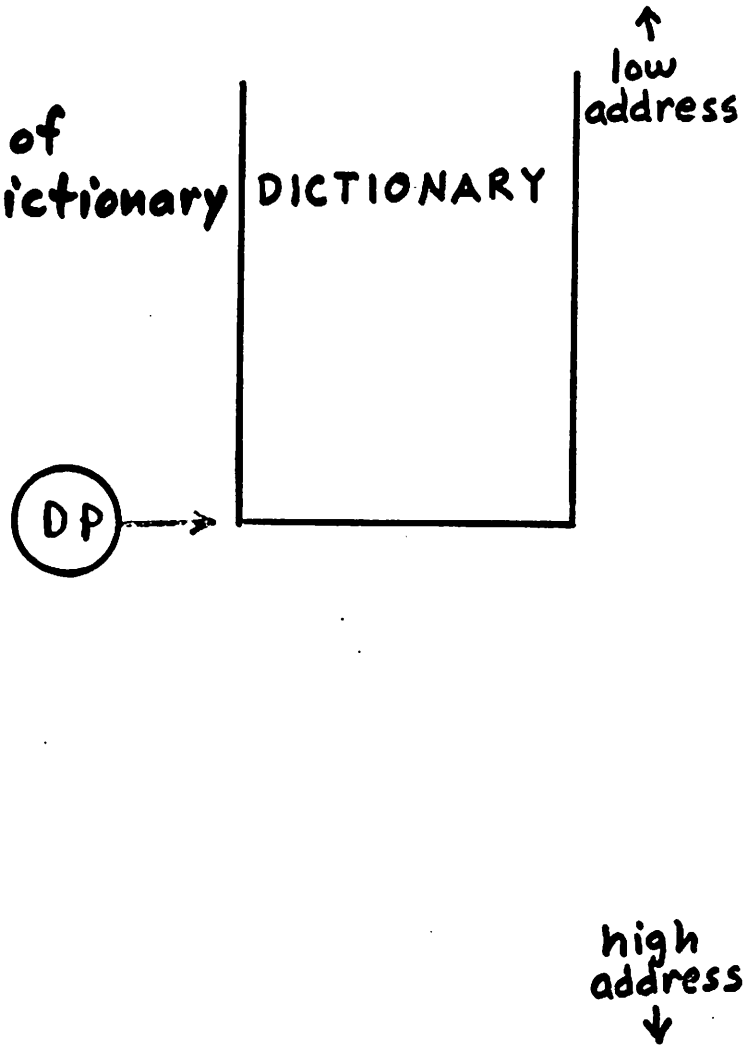
example (indirect addressing)

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



DICTIONARY ALLOCATION

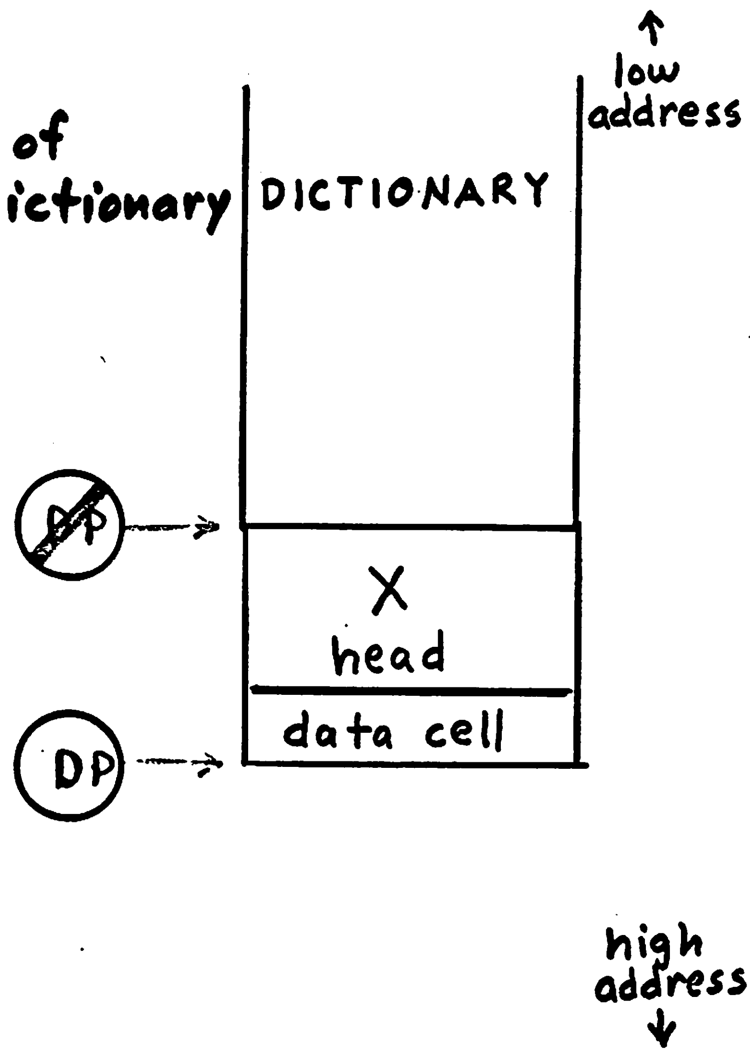
HERE --- address of top of dictionary stack



DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack

VARIABLE X

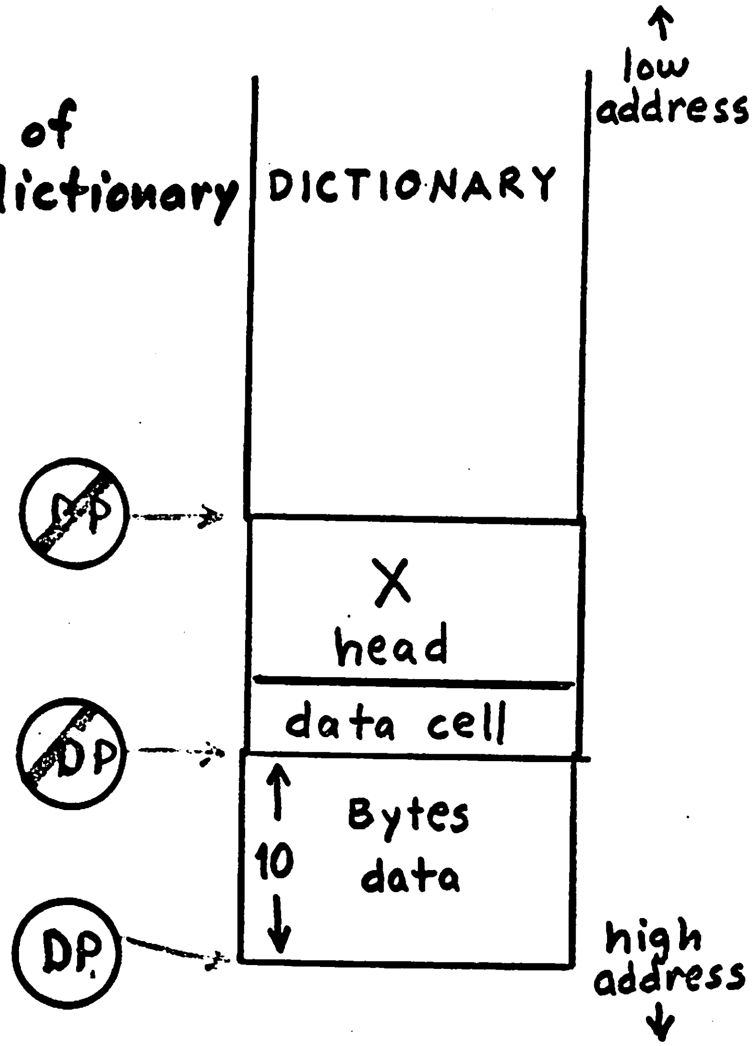


DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack

VARIABLE X

10 ALLOT



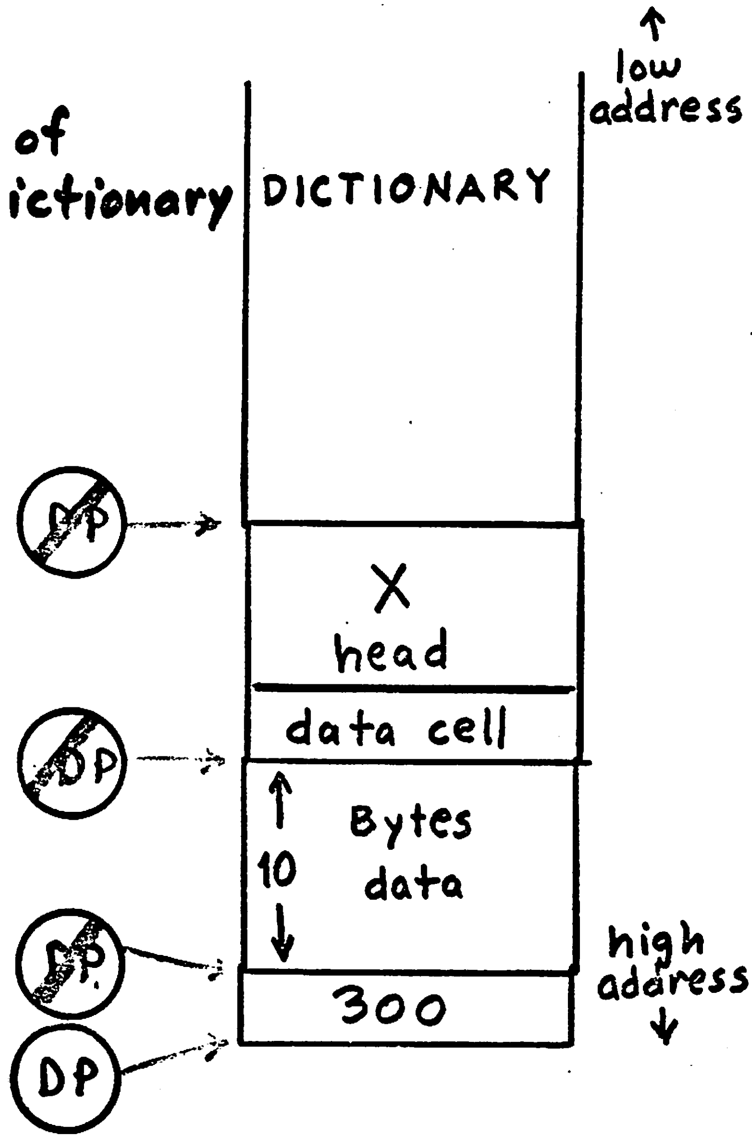
DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack

VARIABLE X

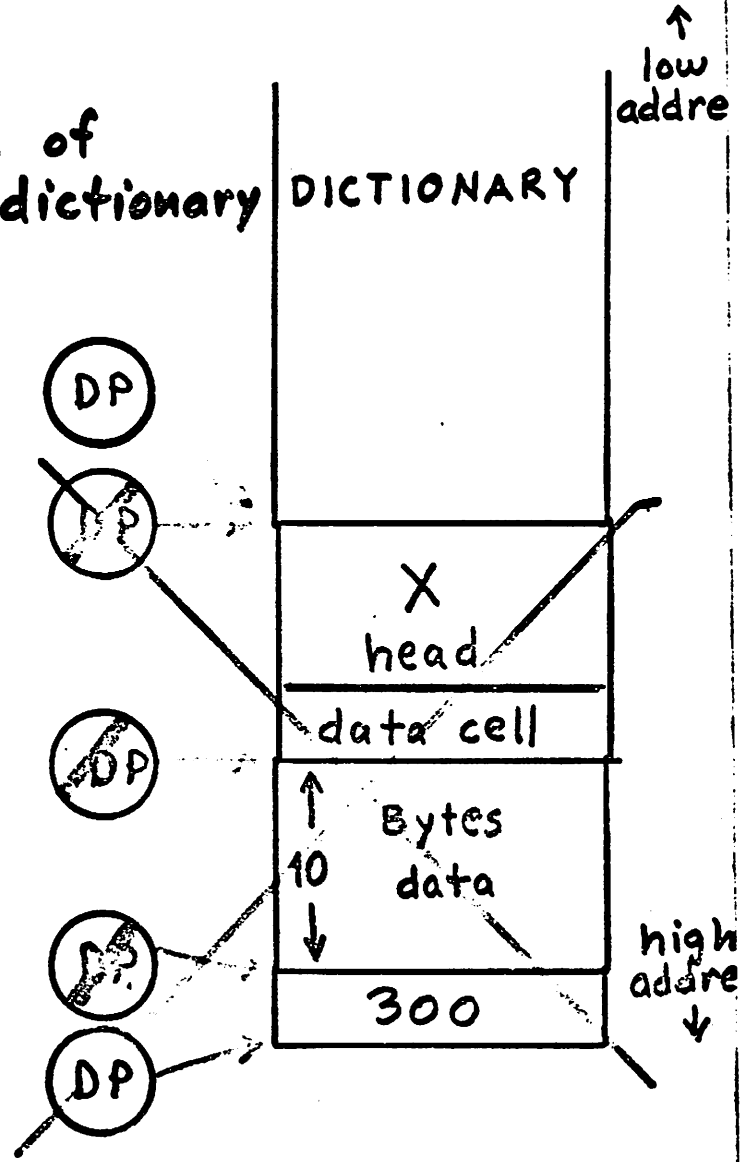
10 ALLOT

300 ,



DICTIONARY ALLOCATION

HERE --- address of top of dictionary stack



VARIABLE X

10 ALLOT

300 ,

FORGET X

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'

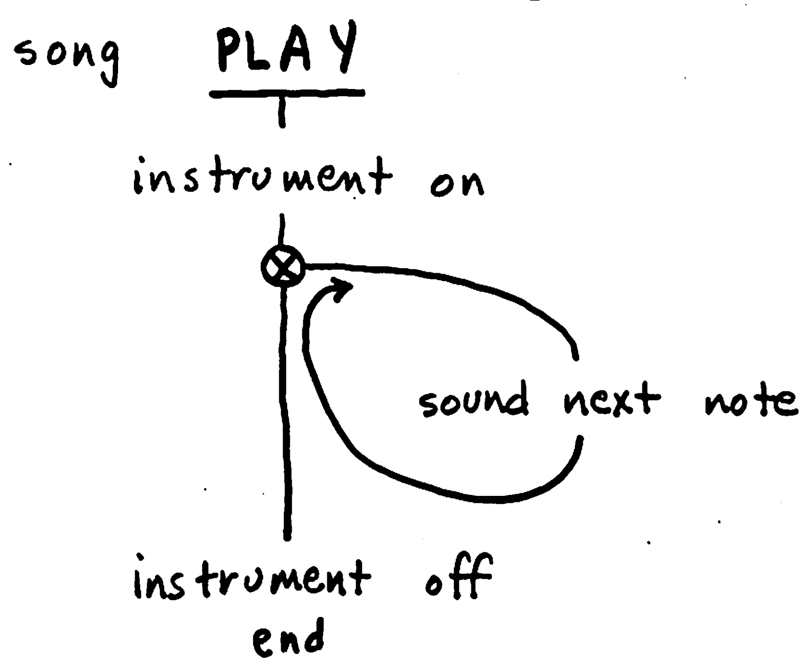
STRUCTURED PROGRAMMING

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

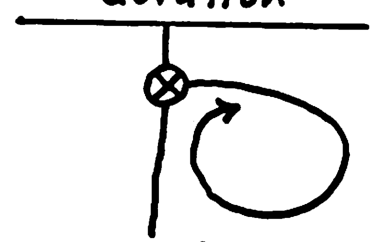
instrument off

quiet
end

sound next note

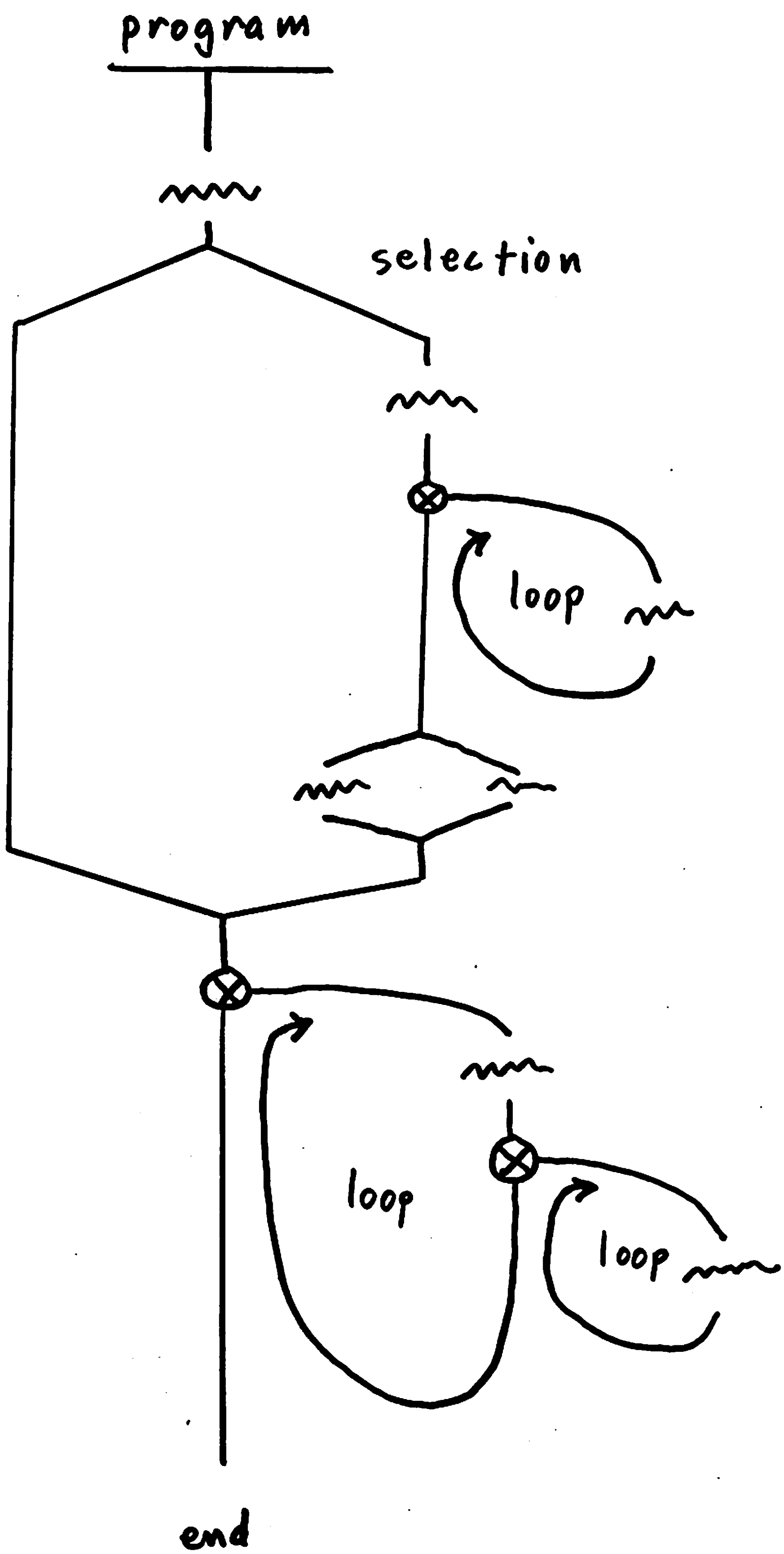
set frequency
start sound
wait for note's duration
stop sound
end

wait for note's duration



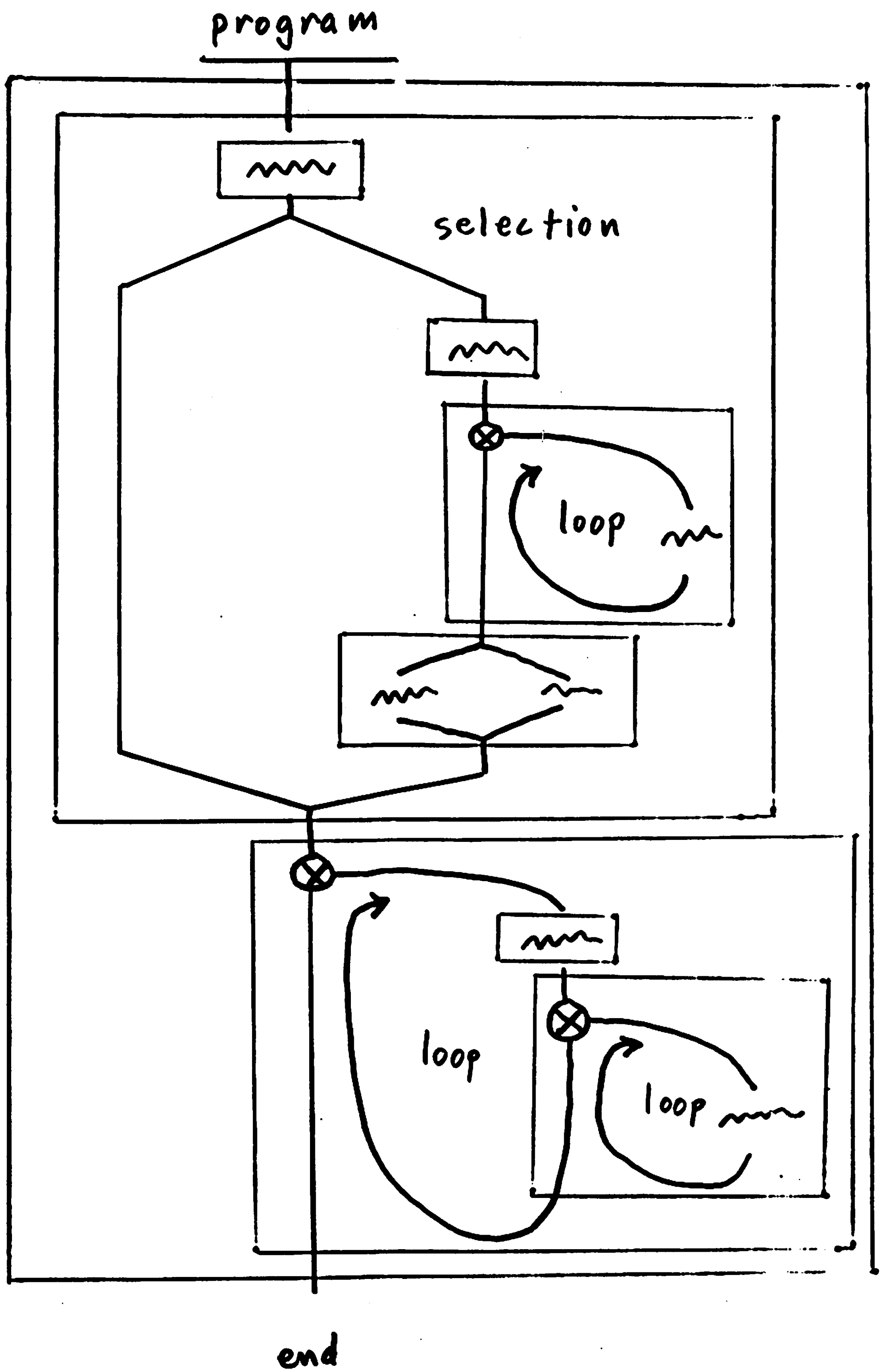
end

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.



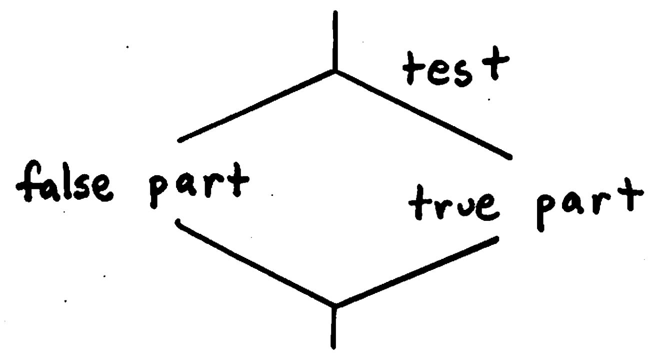
RULE: 1 control path in ; 1 control path out
data data

D-CHARTS

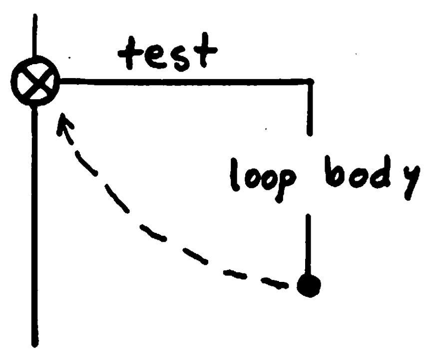
sequential operations:

step one
step two
step three
⋮

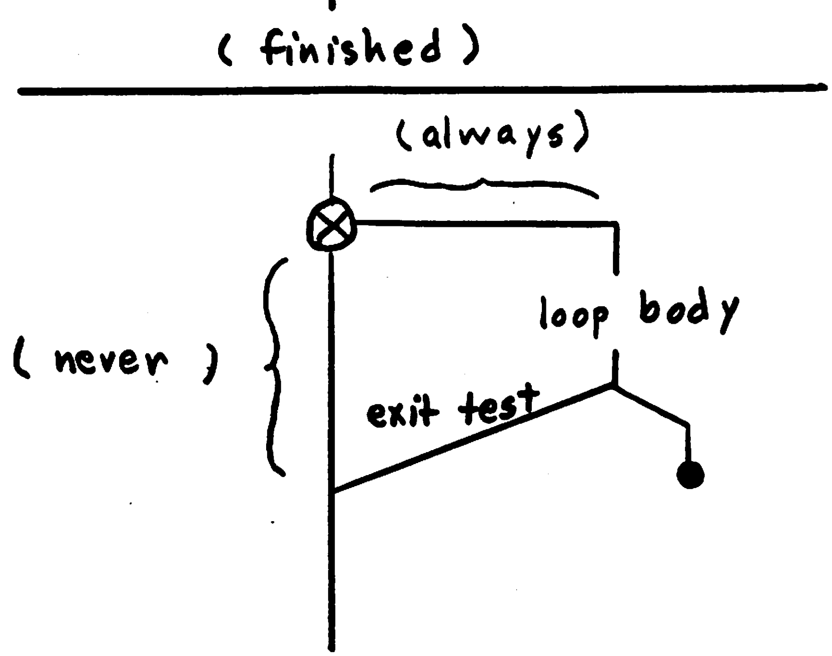
selection:



loop:

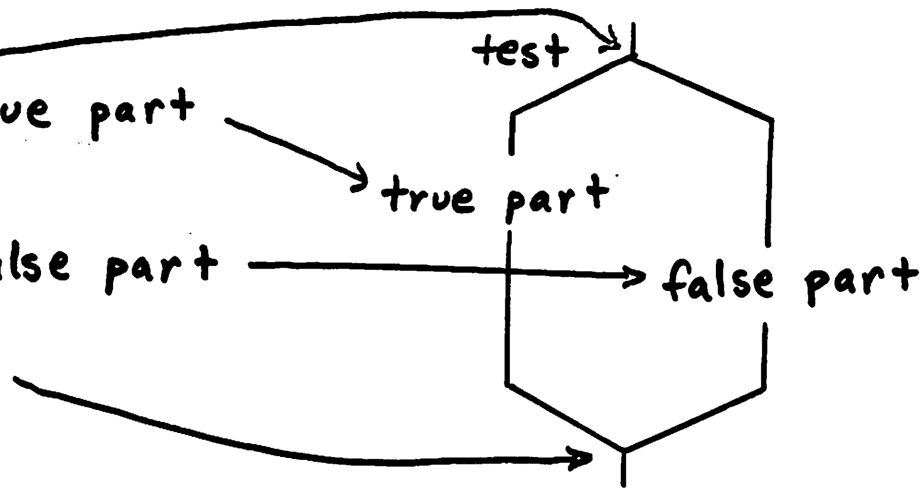


or



FORTH compiler control structure for SELECTION

```
value IF
    true part
  ELSE
    false part
  THEN
```



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 <u>only</u> within a : definition.

COMPARISON OPERATORS

16 bit signed integer:

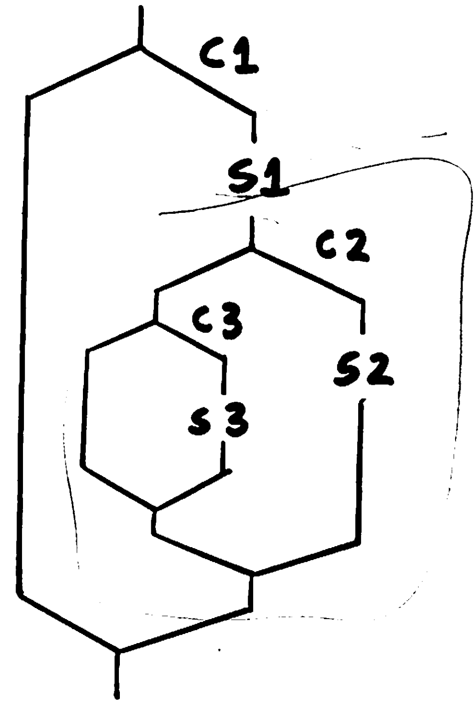
Operator	Stack		Result
0=	n	---	0 if $n \neq 0$ 1 if $n = 0$
0<	n	---	0 if $n \geq 0$ 1 if $n < 0$
=	n1 n2	---	0 if $n1 \neq n2$ 1 if $n1 = n2$
−	n1 n2	---	0 if $n1 = n2$ ≠0 if $n1 \neq n2$
<	n1 n2	---	0 if $n1 \geq n2$ 1 if $n1 < n2$
>	n1 n2	---	0 if $n1 \leq n2$ 1 if $n1 > n2$

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
```

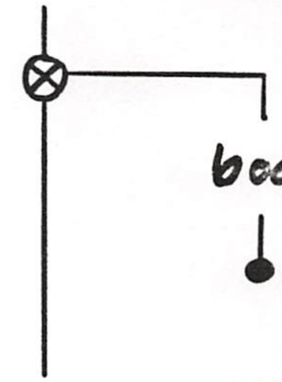


DO LOOPS

final initial DO

loop body

{ LOOP
or
inc +LOOP }



function:

DO removes 2 parameters, sets index ← initial *(stored on top of return stack & limit beneath that)*

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

LEAVE sets limit ← current loop index *(so I remains valid)*
so will exit next time at
LOOP or +LOOP

NOTE:

DO, LOOP, +LOOP, & LEAVE
can be used <u>only</u> within : definitions.

examples of DO loops:

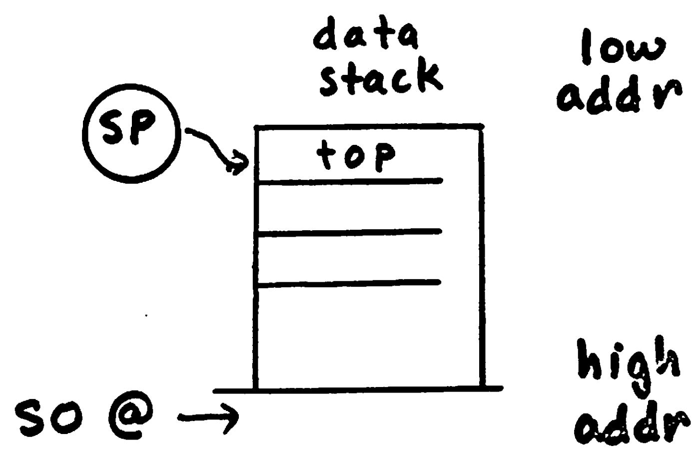
```
: COUNT  DO  I .  LOOP ;
4 0 COUNT (CR)  0 1 2 3  OK
0 4 COUNT (CR)  4  OK
-16 -20 COUNT   -20 -19 -18 -17  OK

: 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   S0 @   SP@  -   2 /  1 -  ;
```

stack dump :

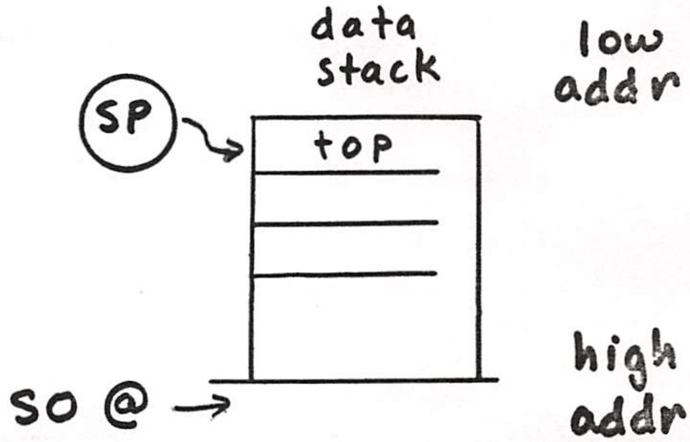
```
:  .S
         SP@  2 -    S0 @   2 -
         DO   I @ .     -2   +LOOP

;
```

<u>usage</u>

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>

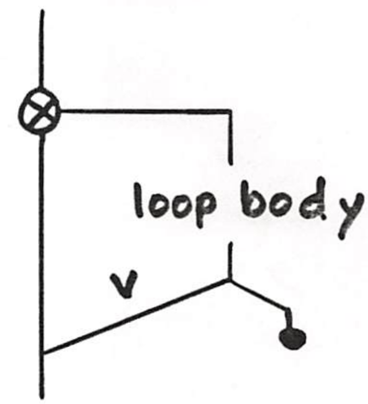
1 2 3 .S (CR) 1 2 3 OK

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

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 ≠ 0 (true)
- branch to BEGIN if v = 0 (false)

NOTE: BEGIN & UNTIL can be used **only** within : definitions

examples:

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

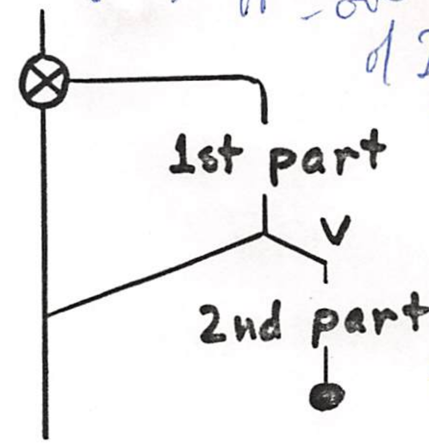
```
: HALVES  BEGIN
             DUP . 2/ DUP 0=
          UNTIL DROP ;
16 HALVES (CR) 16 8 4 2 1   OK
```

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

loop WHILE a condition remains true

Very efficient - overhead of DO LOOP is 20 times this

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



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.