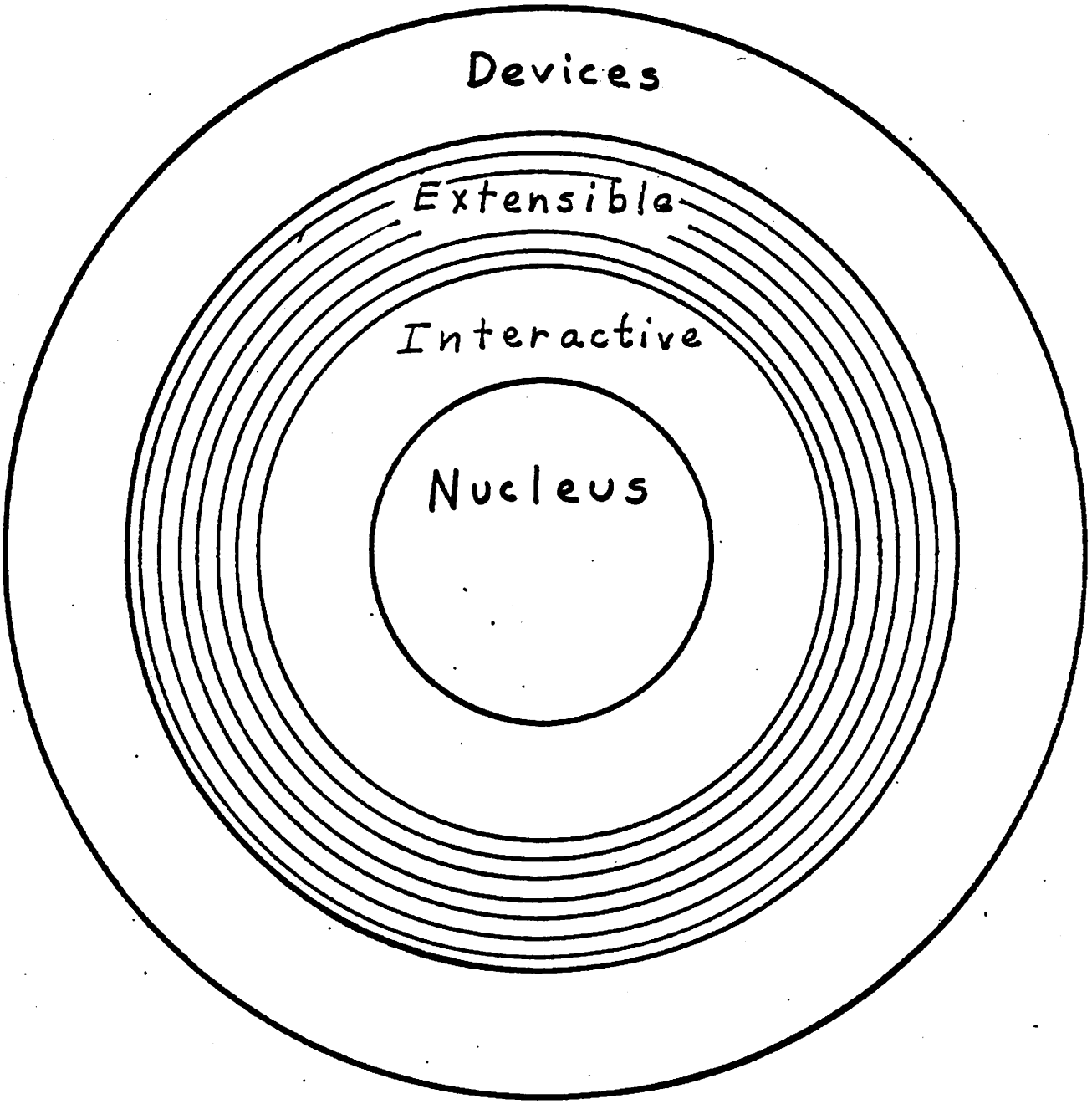
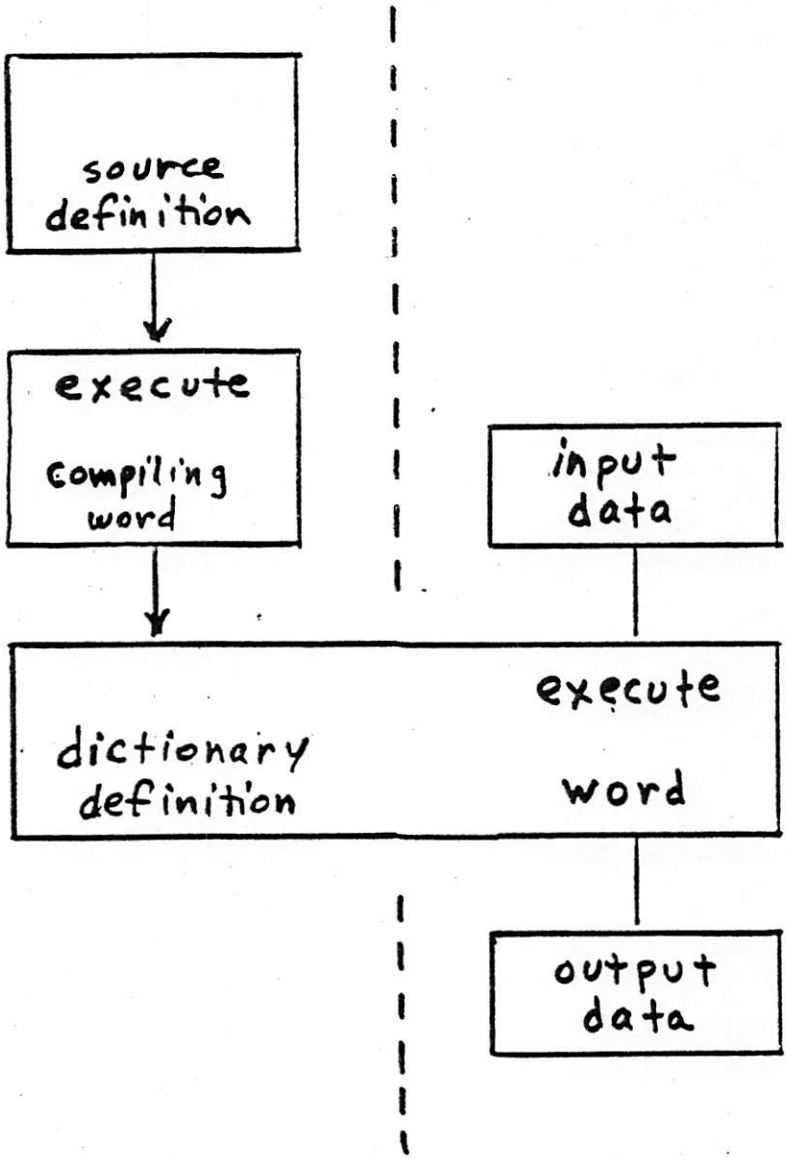
FORTH COMPILER

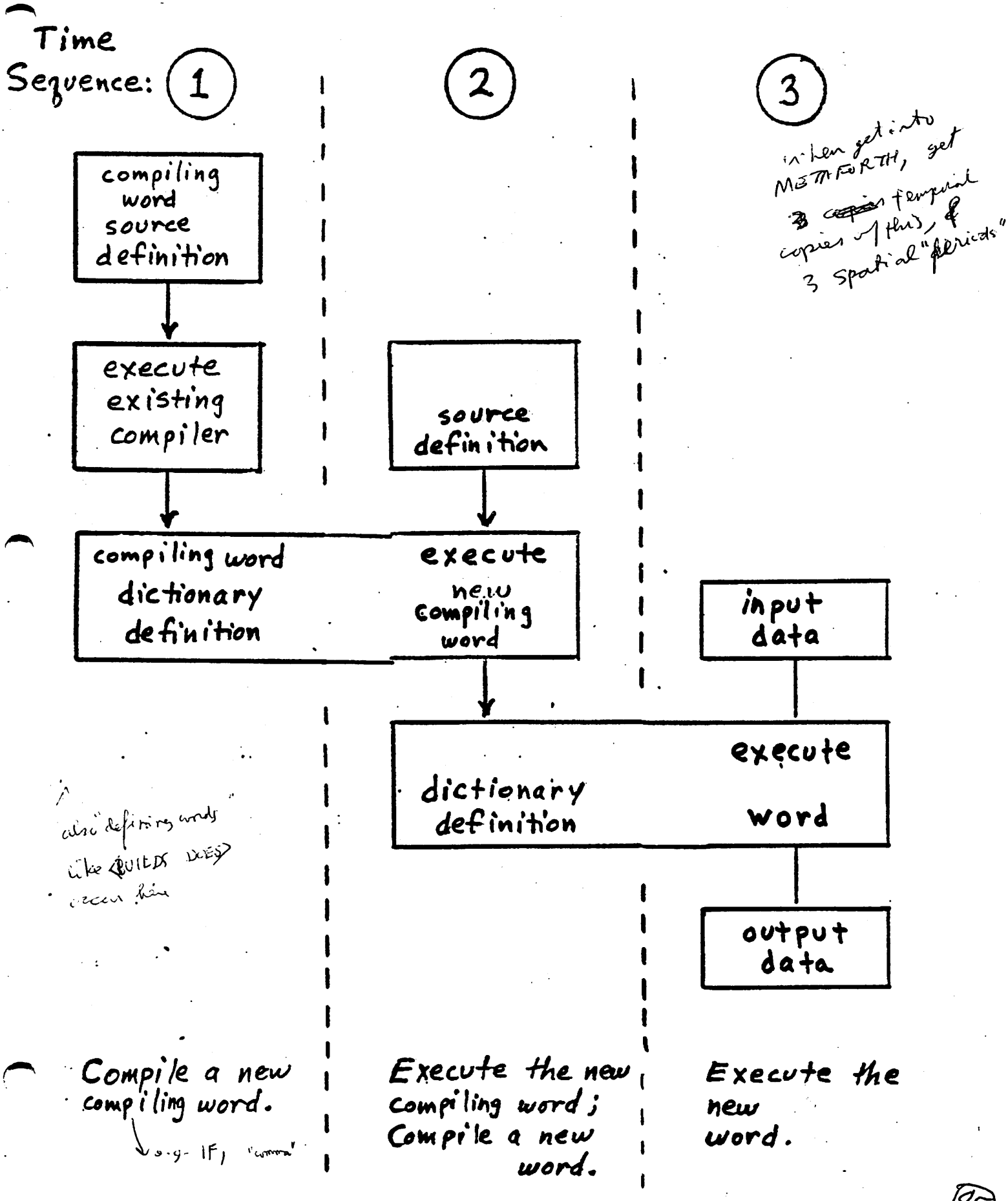
Application Layers



USING COMPILING WORDS



USING COMPILING WORDS



During compilation,

"normal words" are compiled by storing each code field address (CFA) 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.

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.

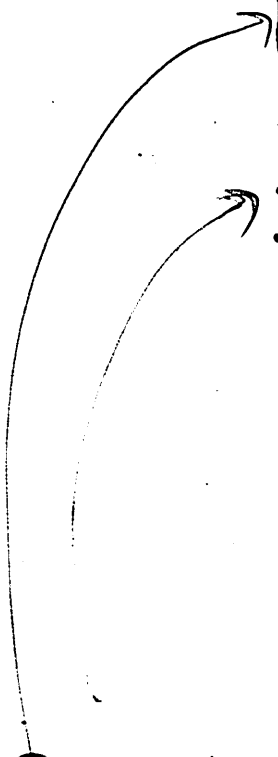
Using FILES outside a : definition
is the same as the non-immediate version.

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.

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 F83 word

The compilation of literal values:

a literal is a numeric character string

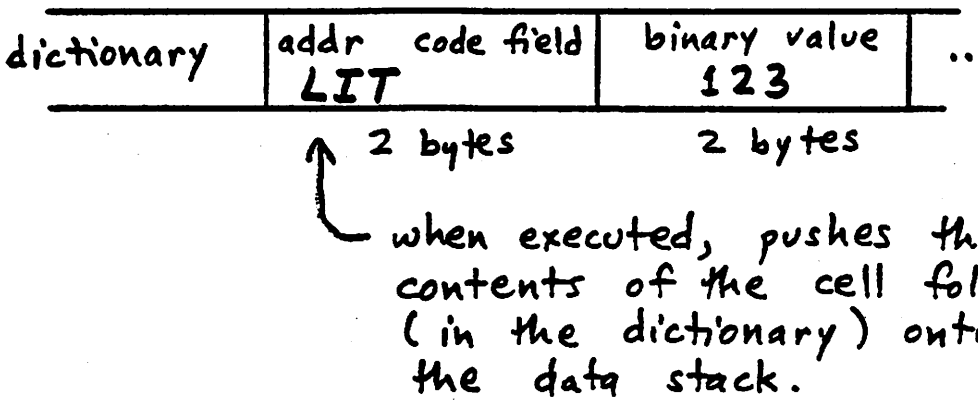
example: 123

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.

: def ~ 123 ~ ;

is compiled as



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	...

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

Deferred compilation:

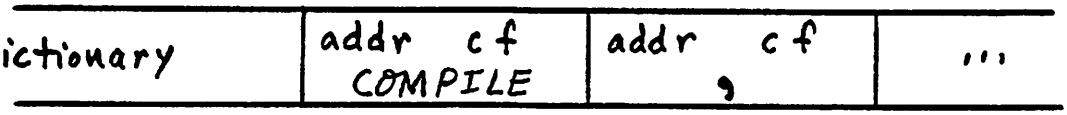
A non-immediate word in a : definition is compiled when it is encountered (ie, not deferred).

(2) : not-deferred ~ , ~ ;

(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 , ~ ;

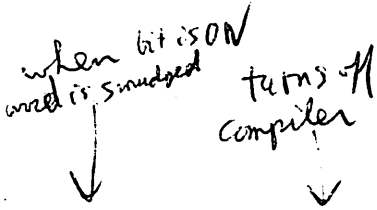


When this definition is executed, COMPILE is executed. This takes the 16 bit value which follows in the definition being executed and compiles that value into the dictionary.

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

because the 'immediate' word would execute anyway

Examples:




```
:  ;     ?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.

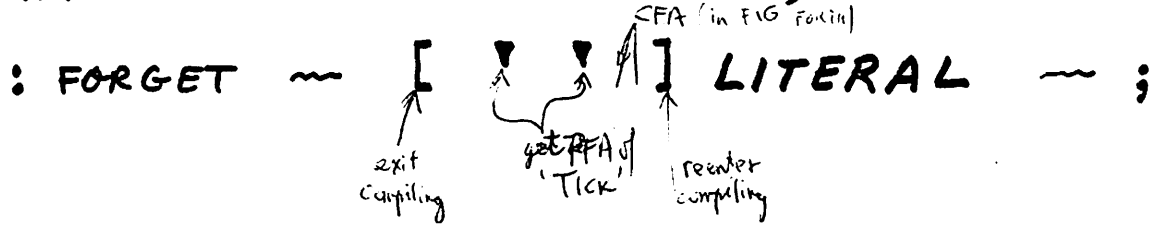
When LITERAL is executed outside of a : definition, it does nothing.

Compiling IMMEDIATE words:

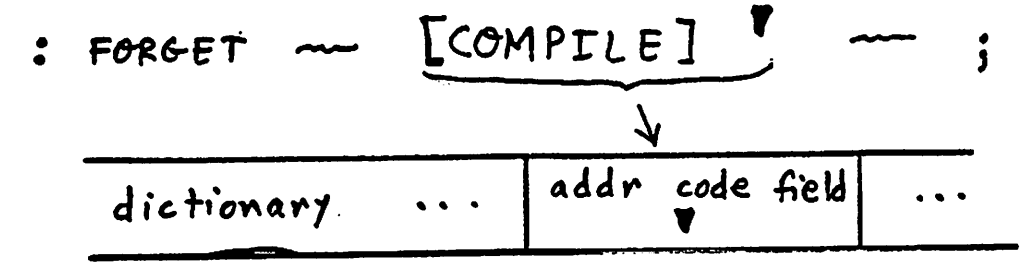
Compiling words sometimes need to force the compilation of IMMEDIATE words.

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



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



-forces immediate compilation

CONTROL STRUCTURES:

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

Unconditional branch:

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

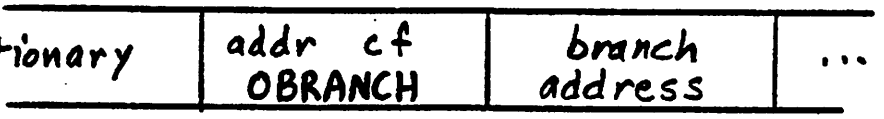
Depending on the implementation of the address interpreter, the branch may be

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.

Conditional branch:



When executed, OBRANCH pops the top of the data stack,

if it is ≠0 (false) then performs the branch (same as unconditional branch)

otherwise (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.

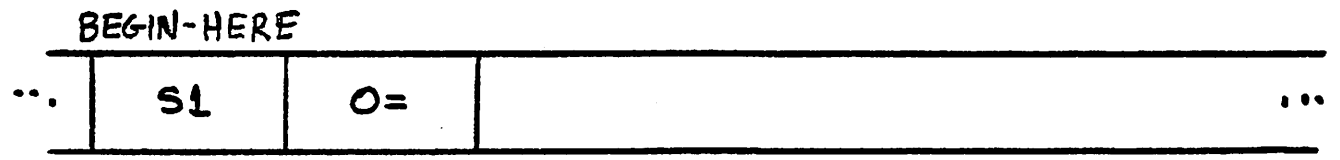
HERE returns the address of the next available location in the dictionary.

Example: 2 byte relative branch addresses

```
: BEGIN    HERE ;  IMMEDIATE

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

calculate backward branch

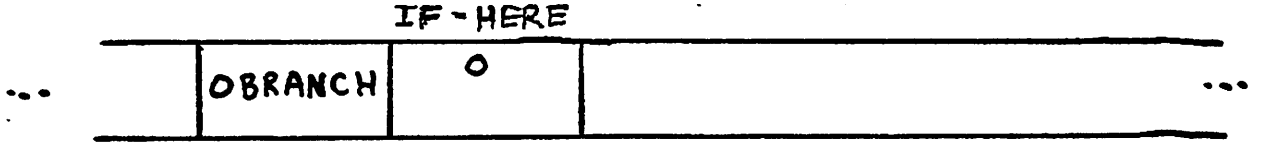
```
... BEGIN    S1    0=  UNTIL  S2 ...
```




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

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

calculate forward branch

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

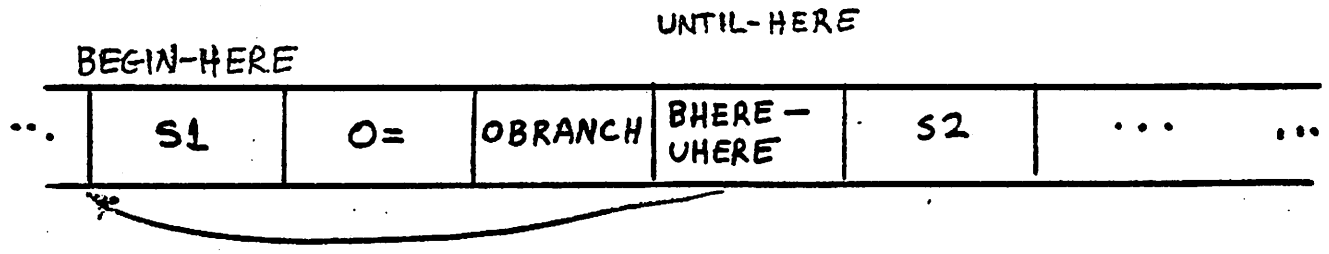


Example: 2 byte relative branch addresses

```
: BEGIN    HERE ;  IMMEDIATE

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

calculate backward branch

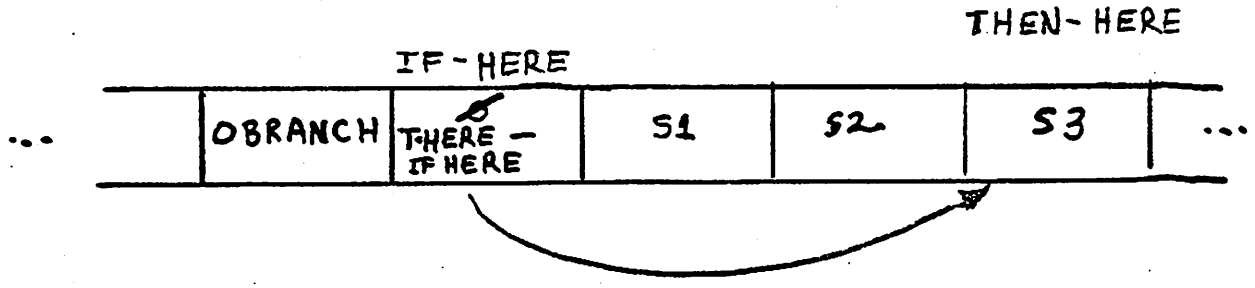
```
... BEGIN    S1    0=  UNTIL  S2  ...
```




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

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

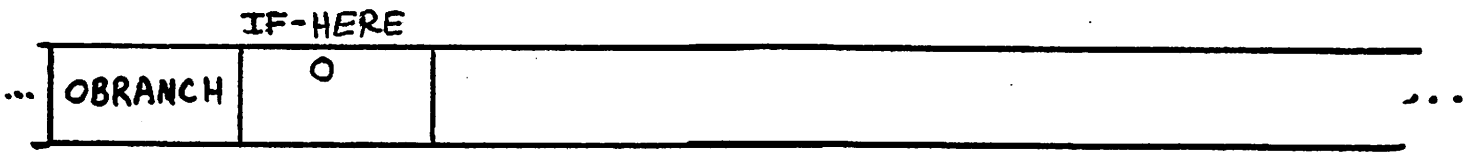
calculate forward branch

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



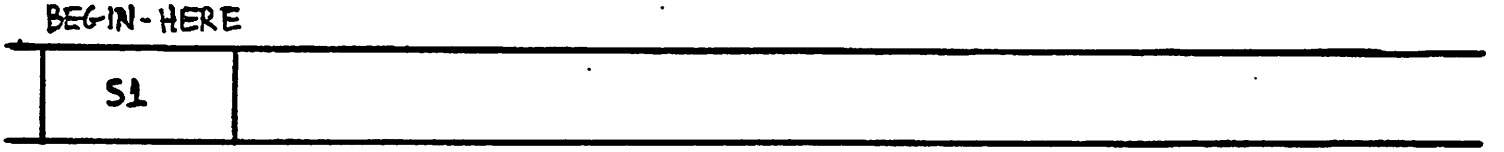
```
: ELSE   COMPILE BRANCH   HERE  0 ,
         SWAP  [COMPILE] THEN ;  IMMEDIATE
```

··· IF S1 ELSE S2 THEN S3 ···



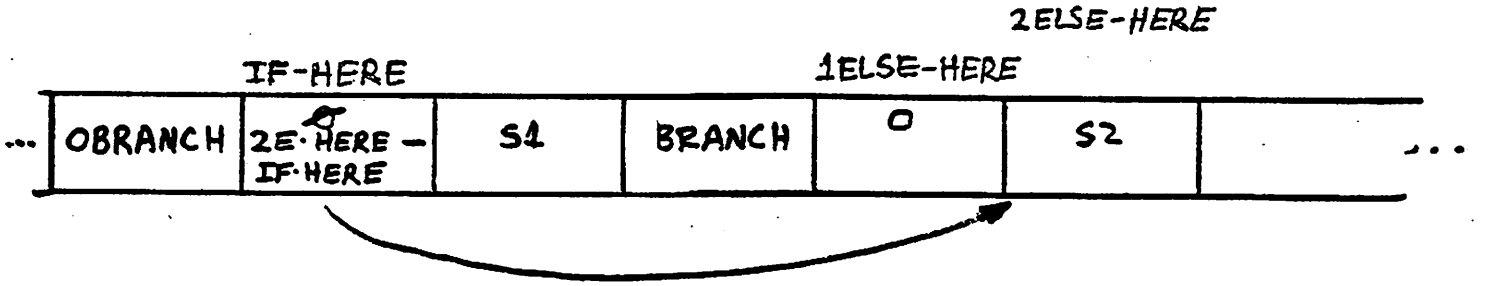

```
: WHILE  [COMPILE] IF  ;  IMMEDIATE

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

··· BEGIN S1 WHILE S2 REPEAT S3 ···



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

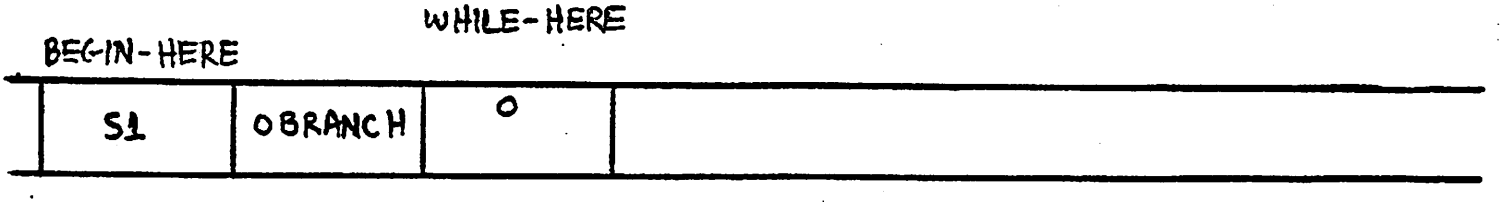
··· IF S1 ELSE S2 THEN S3 ···




```
: WHILE  [COMPILE] IF ; IMMEDIATE
```

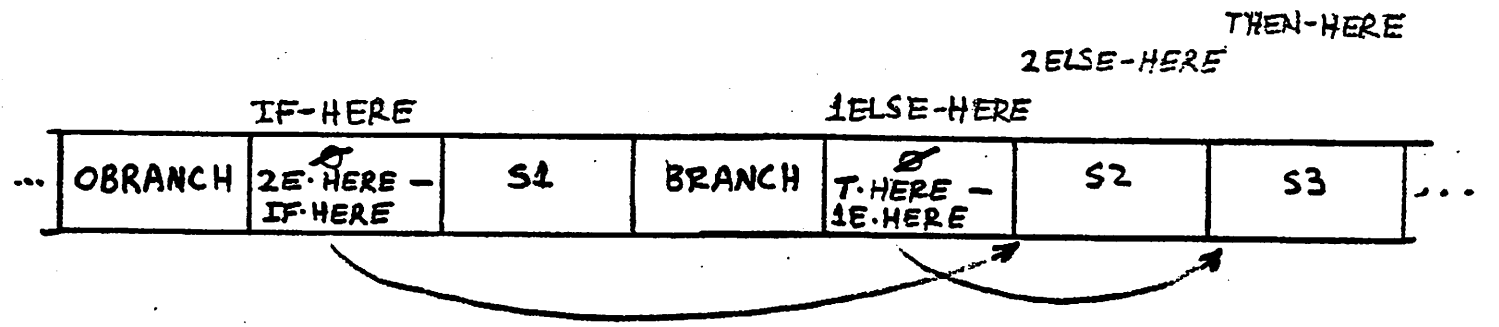
```
: REPEAT  >R  COMPILE BRANCH  HERE – ,
          R>  [COMPILE] THEN ; IMMEDIATE
```

··· BEGIN S1 WHILE S2 REPEAT S3 ···



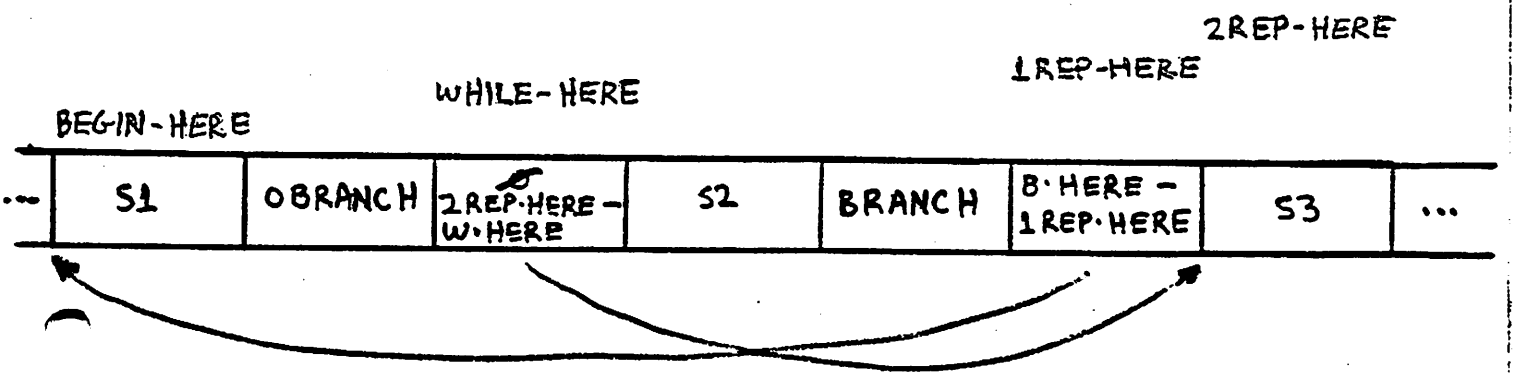
```
: ELSE   COMPILE BRANCH   HERE 0 ,
         SWAP  [COMPILE] THEN ;  IMMEDIATE
```

... 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 ...



```
 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