This technique, a rudimentary example of **vectoring**, saves memory and saves programming time by letting us vary something in the middle of a definition *after the definition has been entered in the dictionary*. However, this technique must be used with caution as it is akin to **self-modifying** code[17].

A similar procedure lets a subroutine call itself recursively, an enormous help in coding certain algorithms.

§§2 Defining "defining" words

The title of this section is neither a typo nor a stutter: **CREATE** finds its most important use in extending the powerful class of FORTH words called "defining" words. The colon compiler " : " is such a word, as are **VARIABLE** and **CONSTANT**.
The definition of **VARIABLE** is simple

```
: VARIABLE      CREATE  2  ALLOT  ;
```

Here is how we use it:

```
VARIABLE X   <cr>  ok
```

The inner workings of **VARIABLE** are these:

- **CREATE** makes a dictionary entry with the next name in the input stream — in this case, **X**.
- Then the number 2 is placed on the stack, and the word **ALLOT** increments the pointer that represents the current location in the dictionary by 2 bytes.

17. Self-modifying machine code is considered a serious "no-no" by modern structured programming standards. Although it is sometimes valuable, few modern cpu's are capable of handling it safely. More often, because cpu's tend to use pipelining and parallelism to achieve speed, a piece of code might be modified in memory, but – having been pre-fetched before modification – actually execute in unmodified form.

- This leaves a 2-byte vacancy to store the value of the variable (that is, the next dictionary header begins 2 bytes above the end of the one just defined).

When the outer interpreter loop encounters a new **VARIABLE**'s name in the input stream, that name's address is placed on the stack. But this is also the location where the 2 bytes of storage begins. Hence when we type in **X**, the TOS will contain the storage address named **X**.

As noted in §2.4 above, the phrase **X @** (pronounced "X fetch") places the contents of address **X** on the stack, dropping the address in the process. Conversely, to store a value in the named location **X**, we use **!** ("store"): thus

```
4 X !    <cr> ok
X @ . <cr> 4 ok
```

Double-length variables are defined *via* **DVARIABLE**, whose definition is

```
: DVARIABLE CREATE 4 ALLOT ;
```

FORTH has a method for defining words initialized to contain specific values: for example, we might want to define the number 17 to be a word. **CREATE** and " , " ("comma") let us do this as follows:

```
17 CREATE SEVENTEEN , <cr> ok
```

Now test it *via*

```
SEVENTEEN @ . <cr> 17 ok
```

Note: The word " **,** "("comma") puts TOS into the next 2 bytes of the dictionary and increments the dictionary pointer by 2.

A word **C,** ("see-comma") puts a byte-value into the next byte of the dictionary and increments the pointer by 1 byte.

§§3 Run-time vs. compile-time actions

In the preceding example, we were able to initialize the variable **SEVENTEEN** to 17 when we **CREATE**d it, but we still have to fetch it to the stack *via* **SEVENTEEN @** whenever we want it. This is not quite what we had in mind: we would like to find 17 in TOS when we say **SEVENTEEN**. The word **DOES>** gives us precisely the tool to do this.

As noted above, the function of **DOES>** is to specify a run-time action for the "child" words of a defining word. Consider the defining word **CONSTANT**, defined in high-level[18] FORTH by

```
: CONSTANT CREATE , DOES> @ ;
```

and used as

```
53 CONSTANT PRIME ok
```

Now test it:

```
PRIME . <cr> 53 ok
```

What happened?

- **CREATE** (hidden in **CONSTANT**) made an entry (named **PRIME** , the first word in the input stream following **CONSTANT**). Then " **,** " placed the TOS (the number 53) in the next two bytes of the dictionary.

18. Of course **CONSTANT** is usually a machine-code primitive, for speed.

- **DOES>** (inside **CONSTANT**) then appended the actions of all words between it and " ; " (the end of the definition of **CONSTANT**) to the child word(s) defined by **CONSTANT**.
- In this case, the only word between **DOES>** and **;** was @ , so all FORTH constants defined by **CONSTANT** perform the action of placing their address on the stack (anything made by **CREATE** does this) and fetching the contents of this address.

§§3-1 Klingons

Let us make a more complex example. Suppose we had previously defined a word **BOX** (n x y - -) that draws a small square box of n pixels to a side centered at (x, y) on the graphics display. We could use this to indicate the instantaneous location of a moving object – say a Klingon space-ship in a space-war game.

So we define a defining word that creates (not very realistic looking) space ships as squares n pixels on a side:

```
: SPACE-SHIP   CREATE ,  DOES>
    @ -ROT  ( - - n x y )     BOX ;
: SIZE ;           \ do-nothing word
```

Now, the usage would be (**SIZE** is included merely as a reminder of what 5 means – it has no function other than to make the definition look like an English phrase)

```
SIZE 5 SPACE-SHIP KLINGON  <cr> ok
71 35 KLINGON  <cr> ok
```

Of course, **SPACE-SHIP** is a poorly constructed defining word because it does not do what it is intended to do. Its child-word **KLINGON** simply draws itself at (x, y).

What we really want is for **KLINGON** to *undraw* itself from its old location, compute its new position according to a set of rules, and then redraw itself at its new position. This sequence of operations would require a definition more like

```
: OLD.POS@  ( adr - - adr n x y )  DUP  @  OVER
    2+  D@  ;
```

```
: SPACE-SHIP CREATE , 4 ALLOT  DOES>
   OLD.POS@  UNBOX  NEW.POS!
   OLD.POS@  BOX  DROP ;
```

where the needed specialized operation **UNBOX** would be defined previously along with **BOX**.

§§3–2 Dimensioned data (with intrinsic units)

Here is another example of the power of defining words and of the distinction between compile-time and run-time behaviors.

Physical problems generally work with quantities that have dimensions, usually expressed as mass (M), length (L) and time (T) or products of powers of these. Sometimes there is more than one system of units in common use to describe the same phenomena.

For example, traffic police reporting accidents in the United States or the United Kingdom might use inches, feet and yards; whereas Continental police would use the metric system. Rather than write different versions of an accident analysis program it is simpler to write one program and make unit conversions part of the grammar. This is easy in FORTH; impossible in FORTRAN, BASIC, Pascal or C; and possible, but exceedingly cumbersome in Ada[19].

We simply keep all internal lengths in millimeters, say, and convert as follows[20]:

19. An example (and its justification) of dimensioned data types in Ada is given by Do-While Jones, *Dr. Dobb's Journal*, March 1987. The FORTH solution below is much simpler than the Ada version.

20. This example is based on 16-bit integer arithmetic. The word */ means "multiply the third number on the stack by NOS, keeping 32 bits of precision, and divide by TOS". That is, the stack comment for */ is (a b c - - a*b/c).

```
: INCHES  254 10 */ ;
: FEET    [ 254 12 * ] LITERAL 10 */ ;
: YARDS   [ 254 36 * ] LITERAL 10 */ ;
: CENTIMETERS  10 * ;
: METERS      1000 * ;
```

The usage would be

```
10 FEET . <cr> 3048 ok
```

These are more definitions than necessary, of course, and the technique generates unnecessary code. A more compact approach uses a *defining word*, **UNITS** :

```
: D,  SWAP , , ; \ ! double-length # in next cells
: UNITS  CREATE D, DOES> D@ */ ;
```

Then we could make the table

```
254 10         UNITS INCHES
254 12 * 10    UNITS FEET
254 36 * 10    UNITS YARDS
 10 1          UNITS CENTIMETERS
1000 1         UNITS METERS
\ Usage:
\  10 FEET . <cr> 3048 ok
\   3 METERS . <cr> 3000 ok
\ ......
\ etc.
```

This is an improvement, but FORTH lets us do even better: here is a simple extension that allows conversion back to the input units, for use in output:

```
VARIABLE <AS>                  \ new variable
0 <AS> !                       \ initialize to "F"
: AS -1 <AS> ! ;             \ set <AS> = "T"
: UNITS   CREATE D,  DOES>
   D@                              \ get 2 #s
   <AS> @                          \ get current val.
         IF SWAP  THEN             \ flip if "true"
   */    0 <AS> ! ;          \ convert, reset <AS>

BEHEAD' <AS>         \ make it local for security[21]

\ unit definitions remain the same
\ Usage:
\  10     FEET       .  <cr>  3048  ok
\  3048 AS FEET      .  <cr>   10  ok
```

§§4 Advanced methods of controlling the compiler

FORTH includes a technique for switching from compile mode to interpret mode while compiling or interpreting. This is done using the words] and [. (Contrary to intuition,] turns the compiler on, [turns it off.)

One use of] and [is to create an "action table" that allows us to choose which of several actions we would like to perform[22].

For example, suppose we have a series of push-buttons numbered 1-6, and a word **WHAT** to read them.

That is, **WHAT** waits for input from a keypad; when button #3 is pushed, *e.g.*, **WHAT** leaves 3 on the stack.

We would like to use the word **BUTTON** in the following way:

```
WHAT  BUTTON
```

21. Headerless words are described in **FTR**, p. 325ff. The word **BEHEAD'** is HS/FORTH's method for making a normal word into a headerless one. See Ch. 5§1§§3 for further details.
22. Better methods will be described in Chapter 5.

BUTTON can be defined to choose its action from a table of actions called **BUTTONS** . We define the words as follows:

```
CREATE BUTTONS ] RING-BELL OPEN-DOOR
   ENTER LAUGH CRY SELF-DESTRUCT [
: BUTTON 1- 2* BUTTONS + @ EXECUTE ;
```

If, as before, I push #3, then the action **ENTER** will be executed. Presumably button #7 is a good one to avoid[23].

How does this work?

- **CREATE BUTTONS** makes a dictionary entry **BUTTONS**.
-] turns on the compiler: the previously-defined word-names **RING-BELL**, *etc.* are looked up in the dictionary and compiled into the table (as though we had begun with :), rather than being executed.
- [returns to interactive mode (as if it were ;), so that the next colon definition (**BUTTON**) can be processed.
- The table **BUTTONS** now contains the code-field addresses (CFA's) of the desired actions of **BUTTON** .
- **BUTTON** first uses **1-** to subtract 1 from the button number left on the stack by **WHAT** (so we can use 0-based numbering into the table – if the first button were #0, this would be unneeded).
- **2*** then multiplies by 2 to get the offset (from the beginning of **BUTTONS**) of the CFA representing the desired action.
- **BUTTONS +** then adds the base address of **BUTTONS** to get the absolute address where the desired CFA is stored.
- @ fetches the CFA for **EXECUTE** to execute.
- **EXECUTE** executes the word corresponding to the button pushed. Simple!

23. The safety of an execution table can be increased by making the first (that is, the zero'th) action **WARNING**, and making the first step of **BUTTON** a word **CHECK-DATA** that maps any number not in the range 1-6 into 0. Then a wrong button number causes a **WARNING** to be issued and the system resets.

You may well ask "Why bother with all this indirection, pointers, pointers to pointers, tables of pointers to tables of pointers, and the like?" Why not just have nested **IF ... ELSE ... THEN** constructs, as in Pascal?

There are three excellent reasons for using pointers:

- Nested **IF. . .THEN**'s quickly become cumbersome and difficult to decipher (TF). They are also **slow** (see Ch. 11).
- Changing pointers is generally much faster than changing other kinds of data — for example reading in code overlays to accomplish a similar task.
- The unlimited depth of indirection possible in FORTH permits arbitrary levels of abstraction. This makes the computer behave more "intelligently" than might be possible with more restrictive languages.

A similar facility with pointers gives the C language its abstractive power, and is a major factor in its popularity.

§9 Strings

By now it should be apparent that FORTH can do anything any other language can do. One feature we need in any sort of programming —scientific or otherwise— is the ability to handle alphanumeric strings. We frequently want to print messages to the console, or to put captions on figures, even if we have no interest in major text processing.

While every FORTH system must include words to handle strings(see, e.g., **FTR**, Ch. 9) —the very functioning of the outer interpreter, compiler, *etc.*, demands this— there is little unanimity in defining extensions. BASIC has particularly good string-handling features, so HS/FORTH and others provide extensions designed to mimic BASIC's string functions.

Typical FORTH strings are limited to 255 characters because they contain a count in their first byte[24]. The word **COUNT**

```
: COUNT  DUP  1 +  SWAP  C@  ;  ( adr - - n adr + 1)
```

expects the address of a counted string, and places the count and the address of the first character of the string on the stack. **TYPE**, a required '79 or '83 word, prints the string to the console.

It is straightforward to employ words that are part of the system (such as **KEY** and **EXPECT**) to define a word like **$"** that takes all characters typed at the keyboard up to a final " (close-quote —not a word but a string-terminator), makes a counted string of them, and places the string in a buffer beginning at an address returned by **PAD**[25].

The word **$.** ("string-emit") could then be defined as

```
: $.  COUNT  TYPE  ;   ( adr - - )
```

and would be used with **$"** like this:

```
$" The quick brown fox"  <cr>   ok
$. The quick brown fox  ok
```

Since this book in not an attempt to paraphrase **FTR**, it is strongly recommended that the details of using the system words to devise a string lexicon be studied there.

One might contemplate modifying the **FTR** lexicon by using a full 16-bit cell for the count. This would permit strings of up to 64k bytes (using unsigned integers[26]), wasting 1 byte of memory per short (255 bytes) string. Although few scientific applications need to manipulate such long strings, the program that generated the index to this book needed to read a page at a time, and thus to handle strings about 3–5 kbytes long.

24. A single byte can represent positive numbers 0-255.
25. A system variable that returns the current starting address of the "scratchpad".
26. FTR, Ch. 3.

§10 FORTH programming style

A FORTH program typically looks like this

```
\ Example of FORTH program
: WORD1  . . .  ;
: WORD2  OTHER-WORDS  ;
: WORD3  YET-OTHER-WORDS  ;
  . . .
:  LAST-WORD   WORDn . . . WORD3
     WORD2  WORD1   ;
LAST-WORD  <cr>      \ run program
```

Note: The word \ means "disregard the rest of this line". It is a convenient method for commenting code.

In other words, a FORTH program consists of a series of word definitions, culminating in a definition that invokes the whole shebang. This aspect gives FORTH programming a somewhat different flavor from programming in more conventional languages.

Brodie notes in **TF** that high-level programming languages are considered *good* if they require structured, top-down programming, and *wonderful* if they impose *information hiding*. Languages such as FORTRAN, BASIC and assembler that permit direct jumps and do not impose structure, top-down design and data-hiding are considered *primitive* or *bad*. To what extent does FORTH follow the norms of *good* or *wonderful* programming practice?

§§1 Structure

The philosophy of "structured programming" entered the general consciousness in the early 1970's. The idea was to make the logic of program control flow immediately apparent, thereby aiding to produce correct and maintainable programs. The language Pascal was invented to impose by fiat the discipline of structure. To this end, direct jumps (GOTOs) were omitted from the language[27].

FORTH programs are *automatically* structured because word definitions are nothing but subroutine calls. The language contains no direct jump statements (that is, no GOTO's) so it is *impossible* to write "spaghetti" code.

A second aspect of structure that FORTH imposes (or at least encourages) is *short* definitions. There is little speed penalty incurred in breaking a long procedure into many small ones, unlike more conventional languages. Each of the short words has one entry and one exit point, and does one job. This is the beaux ideal of structured programming!

§§2 "Top-down" design

Most authors of "how to program" books recommend designing the entire program from the general to the particular. This is called "top-down" programming, and embodies these steps:

- Make an outline, flow chart, data-flow diagram or whatever, taking a broad overview of the whole problem.
- Break the problem into small pieces (decompose it).
- Then code the individual components.

The natural programming mode in FORTH is "bottom-up" rather than "top-down" – the most general word appears last, whereas the definitions necessarily progress from the primitive to the complex. It is possible – and sometimes vital – to invoke a word before it is defined ("forward referencing"[28]). The dictionary and threaded compiler mechanisms make this nontrivial.

27. Ironically, most programmers refuse to get along without spaghetti code, so commercial Pascal's now include GOTO. Only FORTH among major languages completely eschews both line labels and GOTOs, making it the most structured language available.

28. The FORmula TRANslator in Ch. 11§4 uses this method to implement its recursive structure.

The naturalness of bottom-up programming encourages a somewhat different approach from more familiar languages:

- In FORTH, components are specified roughly, and then as they are coded they are immediately tested, debugged, redesigned and improved.
- The evolution of the components guides the evolution of the outer levels of the program.

We will observe this evolutionary style in later chapters as we design actual programs.

§§3 Information hiding

Information (or data) "hiding" is another doctrine of structured programming. It holds that no subroutine should have access to, or be able to alter (corrupt!) data that it does not absolutely require for its own functioning[29].

Data hiding is used both to prevent unforeseen interactions between pieces of a large program; and to ease designing and debugging a large program. The program is broken into small, manageable chunks ("black boxes") called **modules** or **objects** that communicate by sending messages to each other, but are otherwise mutually impenetrable. Information hiding and modularization are now considered so important that special languages – Ada, MODULA-2, C + + and Object Pascal – have been devised with it in mind.

To illustrate the problem information hiding is intended to solve, consider a FORTRAN program that calls a subroutine

29. Rather like the "cell" system in revolutionary conspiracies, where members of a cell know only each other but not the members of other cells. Mechanisms for receiving and transmitting messages between cells are double-blind. Hence, if an individual is captured or is a spy, he can betray at most his own cell and the damage is limited.

```
PROGRAM MAIN
   some lines
CALL SUB1(arg1, arg2, ... , argn, answer)
   some lines
END

SUB1(X1, ... , Xn, Y)
   some lines
   Y = something
RETURN
END
```

There are two ways to pass the arguments from MAIN to SUB1, and FORTRAN can use both methods.

- Copy the arguments from where they are stored in MAIN into locations in the address space of SUB1 (set aside for them during compilation). If the STATEMENTS change the values X1,...,Xn during execution of SUB1, the original values in the calling program will not be affected (because they are stored elsewhere and were copied during the CALL).
- Let SUB1 have the addresses of the arguments where they are stored in MAIN. This method is dangerous because if the arguments are changed during execution of SUB1, they are changed in MAIN and are forever corrupted. If these changes were unintended, they can produce remarkable bugs.

Although copying arguments rather than addresses seems safer, sometimes this is impossible either because the increased memory overhead may be infeasible in problems with large amounts of data, or because the extra overhead of subroutine calls may unacceptably slow execution.

What has this to do with FORTH?

- FORTH uses linked lists of addresses, compiled into a dictionary to which all words have equal right of access.
- Since everything in FORTH is a word –constants, variables, numerical operations, I/O procedures– it might seem impossible to hide information in the sense described above.
- Fortunately, word-names can be erased from the dictionary after their CFAs have been compiled into words that call them. (This erasure is called "beheading".)

- Erasing the names of variables guarantees they can be neither accessed nor corrupted by unauthorized words (except through a calamity so dreadful the program crashes).

§§4 Documenting and commenting FORTH code

FORTH is sometimes accused of being a "write-only" language. In other words, some complain that FORTH is cryptic. I feel this is basically a complaint against poor documentation and unhelpful word names, as Brodie and others have noted.

Unreadability is equally a flaw of poorly written FORTRAN, Pascal, C, *et al.*

FORTH offers a programmer who takes the trouble a goodly array of tools for adequately documenting code.

§§4–1 Parenthesized remarks

The word (—a left parenthesis followed by a space— says "disregard all following text up until the next right parenthesis[30] in the input stream. Thus we can intersperse explanatory remarks within colon definitions. This method was used to comment the Legendre polynomial example program in Ch. 1.

§§4–2 Stack comments

A particular form of parenthesized remark describes the effect of a word on the stack (or on the floating point fstack in Ch. 3). For example, the stack-effect comment (stack comment, for short)

```
( adr - - n)
```

would be appropriate for the word @ ("fetch"): it says @ expects to find an address (adr) on the stack, and to leave its contents (n) upon completion.

The corresponding comment for ! would be

30. The right parenthesis,), is not a word but a delimiter.

```
( n adr - - ) .
```

An fstack comment is prefaced by a double colon :: as

```
( :: x - - f[x]) .
```

Note that to replace parentheses within the comment we use brackets [ ] , since parentheses would be misinterpreted. Since the brackets appear to the right of the word (, they cannot be (mis-) interpreted as the FORTH words] or [.

With some standard conventions for names[31], and standard abbreviations for different types of numbers, the stack comment may be all the documentation needed, especially for a short word.

§§4–3 Drop line (\ )

The word \ (back-slash followed by space) has gained favor as a method for including longer comments. It simply means "drop everything in the input stream until the next carriage return". Instructions to the user, clarifications or usage examples are most naturally expressed in an included block of text with each line set off[32] by \ .

§§4–4 Self-documenting code

By eliminating ungrammatical phrases like CALL or GOSUB, FORTH presents the opportunity —*via* telegraphic names[33] for words— to make code almost as self-documenting and transparent as a simple English or German sentence. Thus, for example, a robot control program could contain a phrase like

```
2 TIMES LEFT EYE  WINK
```

which is clear (although it sounds like a stage direction for Brunhilde to vamp Siegfried). It would even be possible without much

31. See, *e.g.*, L. Brodie, TF, Ch. 5, Appendix E.
32. For those familiar with assembly language, \ is exactly analogous to ; in assembler. But since ; is already used to close colon definitions in FORTH, the symbol \ has been used in its place.
33. The matter of naming brings to mind Mark Twain's remark that the difference between the *almost*-right word and the right one is the difference between the lightning-bug and the lightning.

difficulty to define the words in the program so that the sequence could be made English-like:

```
WINK LEFT EYE 2 TIMES .
```

§§5 Safety

Some high level languages perform automatic bounds checking on arrays, or automatic type checking, thereby lending them a spurious air of reliability. FORTH has almost no error checking of any sort, especially at run time. Nevertheless FORTH is a remarkably safe language since it fosters fine-grained decomposition into small, simple subroutines. And each subroutine can be checked as soon as it is defined. This combination of simplicity and immediacy can actually produce safer, more predictable code than languages like Ada, that are ostensibly *designed* for safety.

Nonetheless, error checking —**especially array bounds-checking**— can be a good idea during debugging. FORTH lets us include checks in an unobtrusive manner, by placing all the safety mechanisms in a word or words that can be "vectored" in or out as desired[34].

34. See **FTR** for a more thorough discussion of vectoring. Brodie, **TF**, suggests a nice construct called **DOER ... MAKE** that can be used for graceful vectoring.