11 EXTENDING THE COMPILER: DEFINING WORDS AND COMPILING WORDS

In comparison with traditional languages, FORTH's compiler is completely backwards. Traditional compilers are huge programs designed to translate any foreseeable, legal combination of available operators into machine language. In FORTH, however, most of the work of compilation is done by a single definition, only a few lines long. Special structures like conditionals and loops are not compiled by the compiler but by the words being compiled (IF, DO, etc.).

Lest you scoff at FORTH's simple ways, notice that FORTH is unique among languages in the ease with which you can extend the compiler. Defining new, specialized compilers is as easy as defining any other word, as you will soon see.

When you've got an extensible compiler, you've got a very powerful language!

Just a Question of Time

Before we get fully into this chapter, let's review one particular concept that can be a problem to beginning FORTH programmers. It's a question of time.

We have used the term "run time" when referring to things that occur when a word is executed and "compile time" when referring to things that happen when a word is compiled. So far so good. But things get a little confusing when a single word has both a run-time behavior and a compile-time behavior.

In general there are two classes of words which behave in both ways. For purposes of this discussion, we'll call these two classes "defining words" and "compiling words."

A defining word is a word which, when executed, compiles a new definition. A defining word specifies the compile-time-and-run-time behavior of each member of the "family" of words that it defines. Using the defining word CONSTANT as an example, when we say

```
80 CONSTANT MARGIN
```

we are executing the compile-time behavior of CONSTANT; that is, CONSTANT is compiling a new constant-type dictionary entry called MARGIN and storing the value 80 into its parameter field. But when we say

```
MARGIN
```

we are executing the run-time behavior of CONSTANT; that is, CONSTANT is pushing the value 80 onto the stack. We'll pursue defining words further in the next few sections.

The other type of word which possesses dual behavior is the "compiling word." A compiling word is a word that we use inside a colon definition and that actually does something during compilation of that definition.

One example is the word ." , which at compile time compiles a text string into the dictionary entry with the count in the first byte, and at run time types it. Other examples are control-structure words like IF and LOOP, which also have compile-time behaviors distinct from their run-time behaviors. We'll explore compiling words after we've discussed defining words.

How to Define a Defining Word

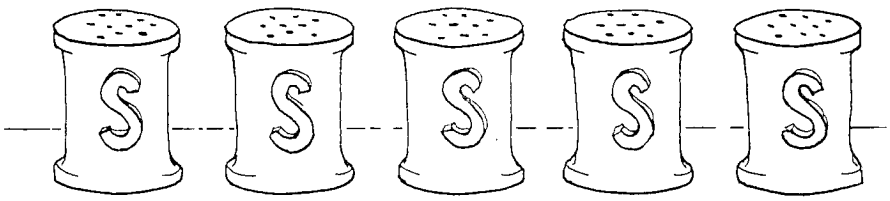
Here are the standard FORTH defining words we've covered so far:

```
:
VARIABLE
2VARIABLE
CONSTANT
2CONSTANT
CREATE
USER
```

What do they all have in common? Each of them is used to define a set of words with similar compile-time and run-time characteristics.

And how are all these defining words defined? First we'll answer this question metaphorically.

Let's say you're in the ceramic salt-shaker business. If you plan to make enough salt shakers, you'll find it's easiest to make a mold first. A mold will guarantee that all your shakers will be of the same design, while allowing you to make each shaker a different color.



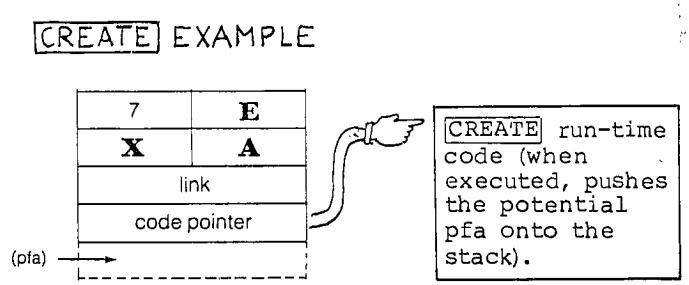
In making the mold, you must consider two things:

1. How the mold will work. (E.g., how will you get the clay into and out of the mold without breaking the mold or letting the seams show?)

2. How the shaker will work. (E.g., how many holes should there be? How much salt should it hold? Etc.)

To bring this analogy back to FORTH, the definition of a defining word must specify two things: the compile-time behavior and the run-time behavior for that type of word.

Hold that thought a moment while we look at the most basic of the defining words in the above list: CREATE. At compile time, CREATE takes a name from the input stream and creates a dictionary heading for it.



At run time, CREATE pushes the pfa of EXAMPLE onto the stack.

What happens if we use CREATE inside a definition? Consider this example, which is the definition for VARIABLE:

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

When we execute VARIABLE as in

```
VARIABLE ORANGES
```

we are indirectly using CREATE to create a dictionary head with the name ORANGES and a code pointer that points to CREATE's run-time code. Then we are allotting two bytes for the variable itself.

Since the run-time behavior of a variable is identical to that of a word defined by CREATE, VARIABLE does not need to have run-time code of its own; it can use CREATE's run-time code.

How do we specify a different run-time behavior in a defining word? By using the word DOES>, as shown here:

```
: DEFINING-WORD   CREATE (compile-time operations)
                  DOES> (run-time operations) ;
```

To illustrate, the following could be a valid definition for CONSTANT (although in fact CONSTANT is usually defined in machine code):

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

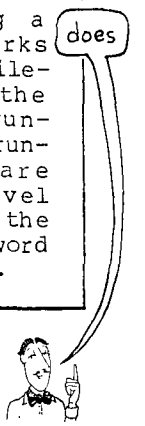
To see how this definition works, imagine we're using it to define a constant named TROMBONES, like this:

```
76 CONSTANT TROMBONES
```

compile-time portion	CREATE	Creates a new dictionary entry (e.g., TROMBONES).
	,	Compiles the value (e.g., 76) for the constant from the stack into the constant's parameter field.
run-time portion	DOES>	Marks the end of the compile-time behavior and the beginning of the run-time behavior. At run time, DOES> will leave the pfa of the word being defined on the stack.
	@	Fetches the contents of the constant, using the pfa that will be on the stack at run time.

The words that precede DOES> specify what the mold will do; the words that follow DOES> specify what the product of the mold will do.

DOES>	run time: (-- adr)	Used in creating a defining word; marks the end of its compile-time portion and the beginning of its run-time portion. The run-time operations are stated in higher-level FORTH. At run time, the pfa of the defined word will be on the stack.



Defining Words You Can Define Yourself

Here are some examples of defining words that you can create yourself.

Recall that in our discussion of "String Input Commands" in Chap. 10, we gave an example that employed character-string arrays called NAME, EYES, and ME. Every time we used one of these names, we followed it with a character count. In the input definition, we wrote

```
... PAD NAME 14 MOVE ...
```

and in the output definition we wrote

```
... NAME 14 -TRAILING TYPE ...
```

and so on.

Let's eliminate the count by creating a defining word called CHARACTERS, whose product definitions will leave the address and count on the stack when executed.

We'll use it like this: if we say

```
20 CHARACTERS ME
```

we will create an array called ME, with twenty bytes available for the character string.

When we execute ME, we'll get the address of the array and the

count on the stack. Now we can write:

```
PAD ME MOVE
```

instead of

```
PAD ME 20 MOVE
```

or

```
ME -TRAILING TYPE
```

instead of

```
ME 20 -TRAILING TYPE
```

Here's how we might define CHARACTERS:

```
: CHARACTERS
```

compile-time portion	CREATE	Creates a new dictionary entry (e.g., ME).
	DUP , ALLOT	Compiles the count (e.g., twenty) into the first cell of the array for future reference. Then allots an additional twenty bytes beyond the count for the string.
run-time portion	DOES>	Marks the beginning of run-time code, leaving the pfa of the product-word on the stack at run time.
	DUP	Copies the pfa.
	2+	Advances the address to point past the count, to the start of the character string.
	SWAP @	Swaps the string address with the count address and fetches the count. The stack now holds (adr count --).

We've just extended our compiler! Our new word CHARACTERS is a defining word that creates a data structure and procedure that we find useful. CHARACTERS not only simplifies our input and output definitions, it also allows us to change the length of any string, should the need arise, in one place only (i.e., where we define it).

Our next example could be useful in an application where a large number of byte arrays are needed. Let's create a defining word called STRING as follows:

```
: STRING   CREATE ALLOT DOES> + ;
```

to be used in the form

```
30 STRING VALVE
```

to create an array thirty bytes in length. To access any byte in this array, we merely say:

```
6 VALVE C@
```

which would give us the current setting of hydraulic valve 6 at an oil-pumping station. At run time, VALVE will add the argument 6 to the pfa left by [illegible], producing the correct byte address.

If our application requires a large number of arrays to be initialized to zero, we might include the initialization in an alternate defining word called 0STRING:

```
: ERASED   HERE OVER  ERASE ALLOT ;
: 0STRING   CREATE ERASED DOES> + ;
```

First we define ERASED to [illegible] the given number of bytes, starting at [illegible], before [illegible]ing the given number of bytes.

Then we simply substitute ERASED for ALLOT in our new version.

By changing the definition of a defining word, you can change the characteristics of all the member words of that family. This ability makes program development much easier. For instance, you can incorporate certain kinds of error checking while you are developing the program, then eliminate them after you are sure that the program runs correctly.

Here is a version of STRING which, at run time, guarantees that the index into the array is valid:

```
: STRING   CREATE DUP , ALLOT
   DOES>  2DUP @ U< NOT  ABORT" RANGE ERROR " + 2+ ;
```

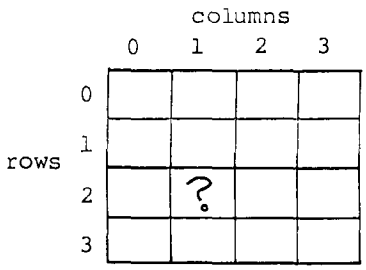
which breaks down as follows:

DUP , ALLOT	Compiles the count and allots the given number of bytes.
DOES> 2DUP @	At run time, given the argument on the stack, produces: (arg pfa arg count --).
U< NOT	Tests that the argument is not less than the maximum, i.e., the stored count. Since U< is an unsigned compare, negative arguments will appear as very high numbers and thus will also fail the test.
ABORT" RANGE ERROR"	Aborts if the comparison check fails.
+ 2+	Otherwise adds the argument to the pfa, plus an additional two to skip over the cell that contains the count.

Here's another way that the use of defining words can help during development. Let's say you suddenly decide that all of the arrays you've defined with STRING are too large to be kept in computer memory and should be kept on disk instead. All you have to do is redefine the run-time portion of STRING. This new STRING will compute which block on the disk a given byte would be contained in, read the block into a buffer using BLOCK, and return the address of the desired byte within the buffer. A string defined in this way could span many consecutive blocks (using the same technique as in Prob. 5, Chap. 10).

You can use defining words to create all kinds of data structures. Sometimes, for instance, it's useful to create multi-dimensional arrays. Here's an example of a defining word which creates two-dimensional byte arrays of given size:

```
: ARRAY  ( #rows #cols -- )
    CREATE  OVER , * ALLOT
    DOES> ( member: row col -- )
    DUP @  ROT *  + +  2+ ;†
```



To create an array four bytes by four bytes, we would say

```
4 4 ARRAY BOARD
```

To access, say, the byte in row 2, column 1, we could say

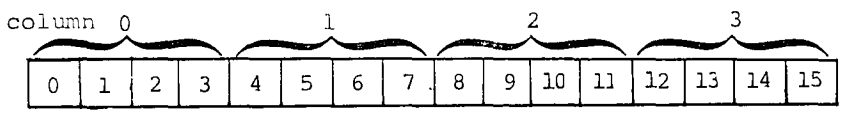
```
2 1 BOARD C@
```

Here's how our ARRAY works in general terms. Since the computer only allows us to have one-dimensional arrays, we must simulate the second dimension. While our imaginary array looks like this: ⟶

column: 0 1 2 3

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

our real array looks like this:



If you want the address of the byte in row 2, column 1, it can be computed by multiplying your column number (1) by the number of rows in each column (4) and then adding your row number (2), which indicates that you want the sixth byte in the real array.

---

†For Optimizers

This version will run even faster:

```
: ARRAY   OVER  CONSTANT  HERE 2+ ,  * ALLOT
    DOES>  2@  ROT *  + + ;
```

This calculation is what members of ARRAY must do at run time. You'll notice that, to perform this calculation, each member word needs to know how many rows are in each column of its particular array. For this reason, ARRAY must store this value into the beginning of the array at compile time.

For the curious, here are the stack effects of the run-time portion of ARRAY:

Operation	Contents of Stack
	row col pfa
DUP @	row col pfa #rows
ROT	row pfa #rows col
*	row pfa col-index
+ +	address
2+	corrected-address

It is necessary to add two to the computed address because the first cell of the array contains the number of columns.

Our final example is the most visually exciting, if not the most useful.

```
 0 ( SHAPES, USING A DEFINING WORD)     EMPTY
 1
 2 : STAR    42 EMIT ;
 3 : .ROW    CR 8 0 DO DUP 128 AND
 4             IF STAR ELSE SPACE THEN
 5                          2* LOOP DROP ;
 6
 7 : SHAPE    CREATE  8 0 DO  C,  LOOP
 8            DOES>  DUP 7 + DO  I C@ .ROW  -1 +LOOP  CR ;
 9
10 HEX    18 18 3C 5A 99 24 24 24 SHAPE MAN
11        81 42 24 18 18 24 42 81 SHAPE EQUIS
12        AA AA FE FE 38 38 38 FE SHAPE CASTLE
13                                                DECIMAL
```

.ROW prints a pattern of stars and spaces that correspond to the 8-bit number on the stack. For instance:

```
2 BASE ! ok
00111001 .ROW
  ***  * ok
DECIMAL ok
```

Our defining word SHAPE takes eight arguments from the stack and defines a shape which, when executed, prints an 8-by-8 grid that corresponds to the eight arguments. For example:

```
MAN
   **
   **
  ****
 * ** *
*  **  *
   *  *
   *  *
   *  *
ok
```

In summary, defining words can be extremely powerful tools. When you create a new defining word, you extend your compiler. Traditional languages do not provide this flexibility because traditional compilers are inflexible packages that say, "Use my instruction set or forget it!"

The real power of defining words is that they can simplify your problem. Using them well, you can shorten your programming time, reduce the size of your program, and improve readability. FORTH's flexibility in this regard is so radical in comparison to traditional languages that many people don't even believe it. Well, now you've seen it.

The next section introduces still another way to extend the ability of FORTH's compiler.

How to Control the Colon Compiler

Compiling words are words used inside colon definitions to do something at compile time. The most obvious examples of compiling words are control-structure words such as IF, THEN, DO, LOOP, etc. Because FORTH programmers don't often change the way these particular words work, we're not going to study them any further. Instead we'll examine the group of words that control the colon compiler and thus can be used to create any type of compiling word.

Recall that the colon compiler ordinarily looks up each word of a source definition and compiles each word's address into the dictionary entry--that's all. But the colon compiler does not

compile the address of a compiling word--it executes it.

How does the colon compiler know the difference? By checking the definition's "precedence bit." If the bit is "off," the address of the word is compiled. If the bit is "on," the word is executed immediately; such words are called "immediate" words.

The word IMMEDIATE makes a word "immediate." It is used in the form

```
: name   definition ;   IMMEDIATE
```

that is, it is executed right after the compilation of the definition.

To give an immediate example, let's define

```
: SAY-HELLO   ." HELLO " ;   IMMEDIATE
```

We can execute SAY-HELLO interactively, just as we could if it were not immediate.

```
SAY-HELLO HELLO ok
```

But if we put SAY-HELLO inside another definition, it will execute at compile time:

```
: GREET   SAY-HELLO   ." I SPEAK FORTH " ; HELLO ok
```

rather than at execution time:

```
GREET I SPEAK FORTH ok
```

Before we go on, let's clarify our terminology. FORTH folks adhere to a convention regarding the terms "run time" and "compile time." In this example, the terms are defined relative to GREET. Thus we would say that SAY-HELLO has a "compile-time behavior" but no "run-time behavior." Clearly, SAY-HELLO does have run-time behavior of its own, but relative to GREET it does not.

To keep our levels straight, let's call GREET in this example the "compilee"; that is, the definition whose compilation we're referring to. SAY-HELLO has no run-time behavior in relation to its compilee.

Here's an example of an immediate word you're familiar with: the definition of the compiling word BEGIN. It's simpler than you might have thought:

```
: BEGIN   HERE ;   IMMEDIATE
```

BEGIN simply saves the address of HERE at compile time on the

stack. Why? Because sooner or later an UNTIL or REPEAT is going to come along, and either has to know what address in the dictionary to return to in the event that it must repeat. This is the address that BEGIN left on the stack.

BEGIN's compile-time behavior is leaving [illegible]RE on the stack. But BEGIN compiles nothing into the compilee; there is no run-time behavior for BEGIN.

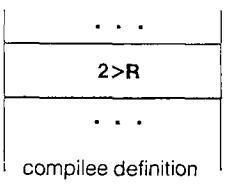
Unlike BEGIN, most compiling words do have a run-time behavior. To have a run-time behavior, a word has to compile into the compilee the address of the run-time behavior, which must already have been defined as a word.

A good example is DO. Like [illegible]N, DO must provide, at compile time, a HERE for LOOP or +L[illegible] to return to. But unlike E[illegible]N, DO also has a run-time behavior: it must push the limit and the index onto the return stack.

The run-time behavior of DO is defined by a lower-level word, sometimes called (DO) or 2>R. The definition of DO is this:

```
: DO   COMPILE 2>R  HERE ;   IMMEDIATE
```

The word COMPILE finds the address of the next word in the definition (in this case 2>R) and compiles its address into the compilee definition, so that at run time 2>R will be executed.†



---

†For the Very Curious

Another example is the definition of ;. At compile time, semicolon must do two things:

1. compile the address of EXIT into the dictionary entry being compiled, and
2. leave compilation mode.

Here's the definition of semicolon:

```
: ;   COMPILE EXIT  R> DROP ;   IMMEDIATE
```

The first phrase compiles [illegible]T, providing the run-time behavior. The second phrase, which is the compile-time behavior, gets us out of the compiler. The top return address at this [illegible]t is pointing inside the colon compiler, which is simply a [illegible]L loop. When semicolon has finished being [illegible]ed, execution will return not to the colon compiler, but to [illegible].

Don't worry about how we can use a semicolon to end the very definition that defines it. The explanation requires an understanding of polyFORTH's Target Compiler, which is beyond the scope of this book (see Appendix 2).

Another compiler-controlling word is [COMPILE].† This word can be used to compile an immediate word as though it were not immediate. Given our previous example, in which SAY-HELLO is an immediate definition, we might define

```
: GREET  [COMPILE] SAY-HELLO ." I SPEAK FORTH " ; ok
```

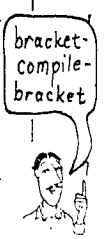
to force SAY-HELLO to be compiled rather than executed at compile time. Thus:

```
GREET HELLO I SPEAK FORTH ok
```

Be sure you understand the difference between COMPILE and [COMPILE]. COMPILE compiles the address of any (non-immediate) word into a compilee definition; think of it as deferred compilation. [COMPILE] compiles the address of any immediate word into the definition currently being defined; this is ordinary compilation, but of an immediate word which otherwise would have been executed.

To review, here are three words which are useful in creating new compiling words:

IMMEDIATE	(--)	Marks the most recently defined word as one which, when encountered during compilation, will be executed rather than be compiled.
COMPILE xxx	(--)	Used in the definition of a compiling word. When the compiling word, in turn, is used in a source definition, the code field address of xxx will be compiled into the dictionary entry so that when the new definition is executed, xxx will be executed.
[COMPILE] xxx	(--)	Used in a colon definition, causes the immediate word xxx to be compiled as though it were not immediate; xxx will be executed when the definition is executed.



†For Some Small-system, Non-polyFORTH, Users

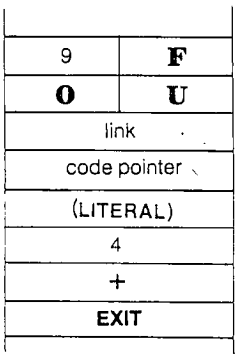
See footnote, page 218.

More Compiler-controlling Words

As you may recall, a number that appears in a colon definition is called a "literal." An example is the "4" in the definition

```
: FOUR-MORE   4 + ;
```

The use of a literal in a colon definition requires two cells. The first contains the address of a routine which, when executed, will push the contents of the second cell (the number itself) onto the stack.†



The name of this routine may vary; let's call it the "run-time code for a literal," or simply (LITERAL). When the colon compiler encounters a number, it first compiles the run-time code for a literal, then compiles the number itself.

The word you will use most often to compile a literal is LITERAL (no parentheses). LITERAL compiles both the run-time code and the value itself. To illustrate:

```
4  : FOUR-MORE   LITERAL + ;
```

Here the word LITERAL will compile as a literal the "4" that we put on the stack before beginning compilation. We get a dictionary entry that is identical to the one shown above.

For a more useful application of LITERAL, recall that in Chap. 8 we created an array called LIMITS that consisted of five cells, each of which contained the temperature limit for a different burner. To simplify access to this array, we created a word called LIMIT. The two definitions looked like this:

---

† For Memory Conservationists

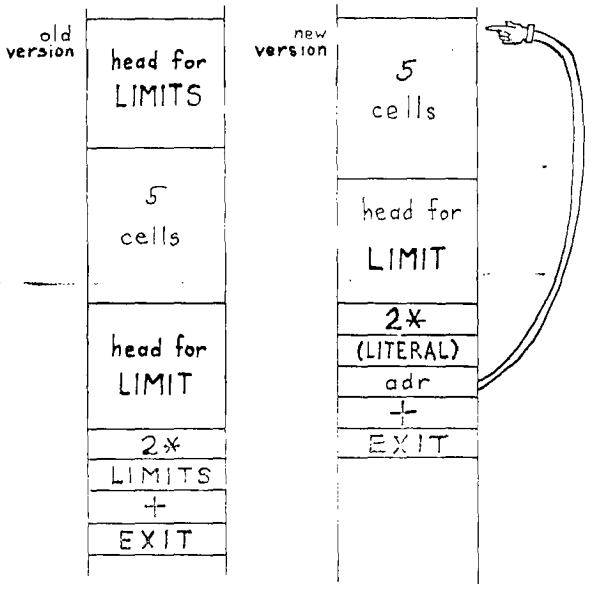
While a literal requires two cells, a reference to a constant requires only one cell. Since a constant takes only five cells to define, you can see that if you're going to use the same value six times or more, you will save memory by defining the value as a constant. There is hardly any difference between the time required to execute a constant and a literal.

```
VARIABLE LIMITS  8 ALLOT
: LIMIT   2* LIMITS + ;
```

Now let's assume that we will only access the array through the word LIMIT. We can eliminate the head of the array (eight bytes) by using this construction instead:

```
HERE  10 ALLOT
: LIMIT   2* LITERAL + ;
```

In the first line we put the address of the beginning of the array ([E .]) on the stack. In the second line, we compile this address as a literal into the definition of LIMIT.



Because we had to add an extra cell for the literal to the definition of LIMITS, our net saving is three cells.

There are two other compiler control words you should know. The words [and] can be used inside a colon definition to stop compilation and start it again, respectively. Whatever words appear between them will be executed "immediately," i.e., at compile time.

Consider this example:

```
: SAY-HELLO   ." HELLO " ;
: GREET   [ SAY-HELLO ] ." I SPEAK FORTH " ; HELLO ok
GREET I SPEAK FORTH ok
```

In this example, SAY-HELLO is not an immediate word, yet when we compile GREET, SAY-HELLO executes "immediately."

For a better example, imagine a colon definition in which we need to type line 3 of block 180. To get the address of line 3, we could use the phrase

```
180 BLOCK 3 64 * +
```

but it's time-consuming to execute

```
3 64 *
```

every time we use this definition. Alternatively, we could write

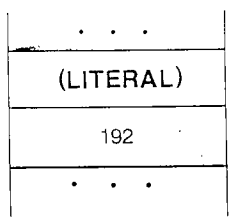
```
180 BLOCK  192 +
```

but it's unclear to human readers exactly what the 192 means.

The best solution is to write

```
180 BLOCK [ 3 64 * ] LITERAL +
```

Here the arithmetic is performed only once, at compile time, and the result is compiled as a literal.



Here's a silly example which may give you some ideas for more practical applications. This definition must be loaded from a disk block:

```
: LIST-THIS   [ BLK @ ]  LITERAL  LIST ;
```

When you execute LIST-THIS, you will list whichever block LIST-THIS is defined in. (At compile time, BLK contains the number of the block being loaded. LITERAL compiles this number into the definition as a literal, so that it will serve as the argument for LIST at run time.)

By the way, here's the definition of LITERAL:

```
: LITERAL   COMPILE (LITERAL)  , ;  IMMEDIATE
```

First it compiles the address of the run-time code, then it compiles the value itself (using comma).

To summarize, here are the additional compiler control words we introduced in this section:

LITERAL	compile time: (n --) run time: (-- n)	Used only inside a colon definition. At compile time, compiles a value from the stack into the definition as a literal. At run time, the value will be pushed onto the stack.
[	(--)	Leaves compile mode.
]	(--)	Enters compile mode.

A Handy Hint

Entering Long Definitions from Your Terminal

Let's say you want to enter a definition from your terminal, but the definition won't all fit on one line. The problem is, if you hit "return" in the middle of a colon definition, you will leave compilation mode. (Even if you don't hit "return," EXPECT only accepts eighty characters.)

How can you get FORTH to resume compilation as you enter subsequent lines? By starting them with]. For example:

```
: BOXTEST  6 > ROT 22 > ROT 19 > AND AND RETURN ok
] IF ." BIG ENOUGH " THEN ; RETURN ok
```

(Some FORTH systems stay in compilation mode until a ; is encountered; on such systems the right bracket is unnecessary.)

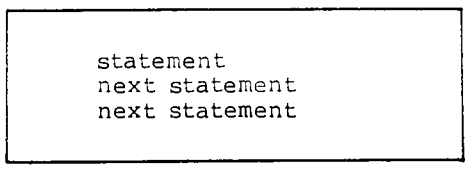
An Introduction to FOR...: Flowcharts

Flowcharts provide a way to visualize the logical structure of a definition, to see where the branches branch and where the loops loop. Old-fashioned flowcharting techniques haven't been adequate for describing FORTH's structured organization. Instead, various FORTH programmers have devised alternate schemes.

The question of which diagramming approach works best for FORTH remains open; programmers use whatever methods work best for them. The subject of flowcharting could occupy a chapter of its own, but we're running out of chapters.

The diagrams that we will use are loosely based on a type of flowchart called the "D-chart," invented by Prof. Edsger W. Dijkstra. Here's how our diagrams work:

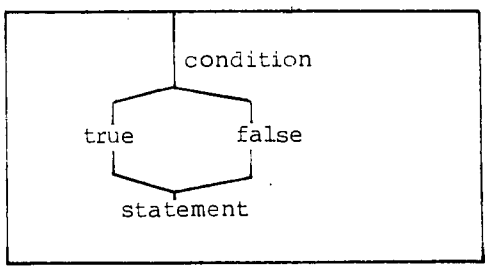
Sequential statements are written one below the other, without lines or boxes:



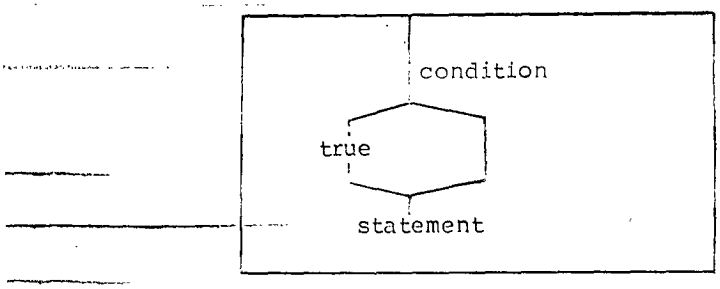
Lines are used to show non-sequential control paths (conditional branches and loops). The FORTH statement

```
condition IF true ELSE false THEN statement
```

would be diagrammed

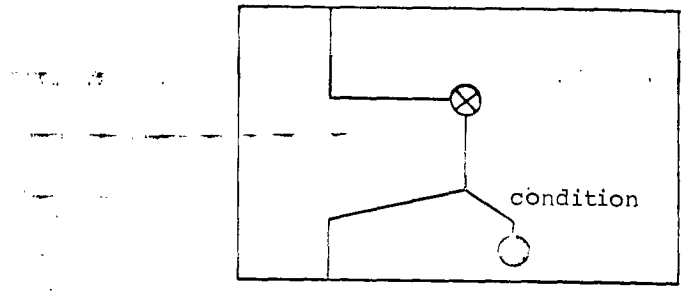


If either phrase is omitted, a vertical line is drawn in its place:



It is immaterial whether "true" is left or right.

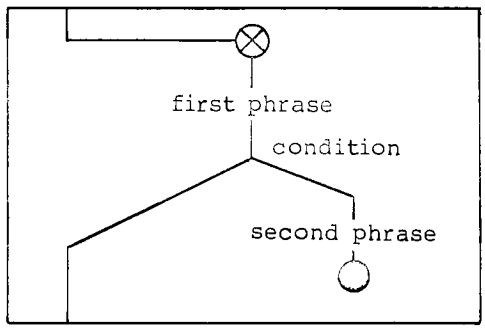
A BEGIN...UNTIL structure is diagrammed like this:



The entire loop structure is shifted to the right from the "normal" flow of execution, connected by a horizontal line at the top. If additional levels of nested loops were to be shown, they would be shifted still further to the right.

The black dot is the symbol for the end of the loop. It indicates that control is returned to the return point, symbolized by the circled X. The condition will cause the loop either to be repeated or to be exited. The diagonal line sloping down to the left indicates the return to the outer level of execution.

A BEGIN...WHILE...REPEAT loop is similar:

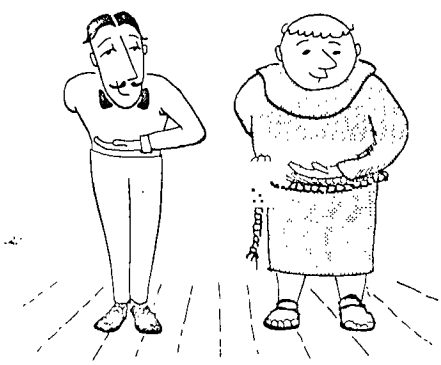


We've given this brief introduction to FORTH flowcharts so that we can visualize the structure of two very important words.

Curtain Calls

This section gives us a chance to say "Goodbye" to the text interpreter and the colon compiler and perhaps to see them in a new light.

Here is the definition of INTERPRET as it is found in many FORTH systems (see page 216 for a discussion of possible variations):

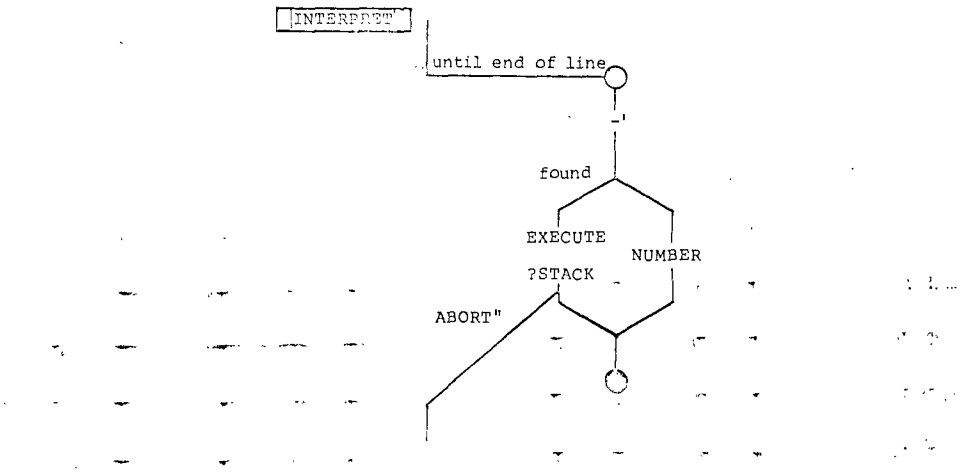


```
: INTERPRET   BEGIN -' IF  NUMBER ELSE  EXECUTE
     ?STACK  ABORT" STACK EMPTY" THEN  0 UNTIL ;
```

We've already covered each of the words contained in this definition; we can describe INTERPRET in English by simply "translating" its definition, like this:

Begin a loop. Within the loop, try to look up the next word from the input stream. If it's not defined, try to convert it as a number. If it is defined, execute it, then check to see whether the stack is empty. (If it is, exit the loop and print "STACK EMPTY.") Then repeat the infinite loop.

Now let's apply our flowcharting techniques to this definition.



As you can see, the FORTH text interpreter is a simple yet powerful structure. Now let's compare its structure with that of the colon compiler:

---

† For the Very Curious

You may have wondered, if INT ... RET is an infinite loop, how do we exit it and get back to QUIT? The answer varies for different implementations of FORTH, but the most common answer is this:

When you enter a line of text from the terminal and press "return," the word EXPECT places a "null" (zero) at the end of the input stream. This null is actually a defined FORTH word; its code field points directly to EXIT. The result: when INT ... gets to the end of the line, it finds null in the dictionary and executes it. EXIT immediately transports us up to QUIT. Simple and fast.

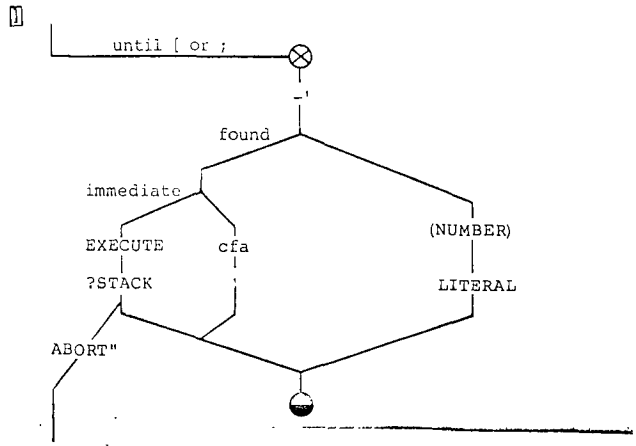
```
: ]   BEGIN -'  IF (NUMBER)  LITERAL
        ELSE ( check precedence bit)  IF  EXECUTE  ?STACK
           ABORT" STACK EMPTY"
        ELSE 2- ,  THEN THEN  0 UNTIL ;
```

The first thing you probably noticed is that the name of the colon compiler is not : but]. The definition of : invokes] after creating the dictionary head and performing a few other odd jobs.

The next thing you may have noticed is that the compiler is somewhat similar to the interpreter. Let's translate the above definition into English:

> Begin a loop. Within the loop, try to look up the next word from the input stream. If it's not defined, try to convert it as a number † and, if it is a number, compile it as a literal.
>
> If it is defined, then treat it as a word. If the word is immediate, then execute it and check to see if the stack is empty. If it is not immediate, change the pfa to a cfa (code-field address) and compile this address. Then repeat the infinite loop.

Picture it this way:



† For the Curious

The version of NUMBER that the colon compiler uses is the 16-bit version. That's why you can't have a double-length literal in a colon definition (except by making it two single-length literals).

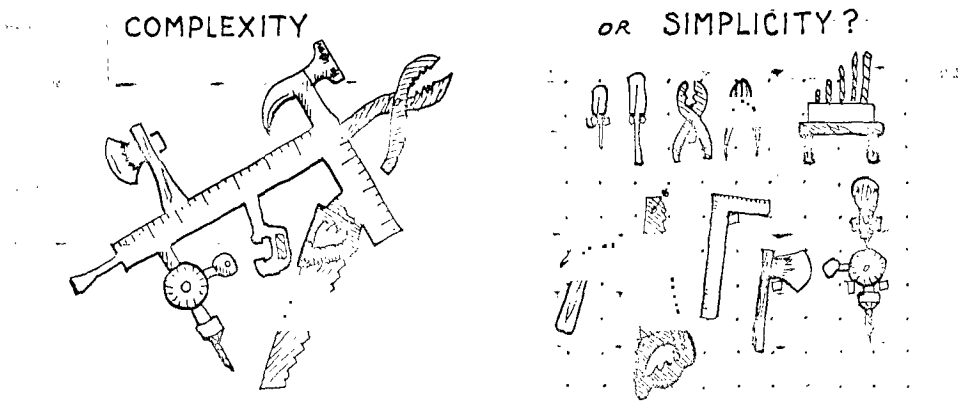
Compare this to the diagram of INTERPRET and you'll see that [could be called an interpreter with the ability to decide whether to execute or to compile any given word. It is the simplicity of this design that lets you add new compiling words so easily.

In summary, we've shown two ways to extend the FORTH compiler:

1. Add new, specialized compilers, by creating new defining words.

2. Extend the existing colon compiler by creating new compiling words.

While traditional compilers try to be universal tools, the FORTH compiler is a collection of separate, simple tools ... with room for more. Which approach seems more useful:



Here's a summary of the words we've covered in this chapter:

DOES>	run time: (-- adr)	Used in creating a defining word; marks the end of its compile-time portion and the beginning of its run-time portion. The run-time operations are stated in higher-level FORTH. At run time, the pfa of the defined word will be on the stack.
IMMEDIATE	(--)	Marks the most recently defined word as one which, when encountered during compilation, will be executed rather than be compiled.
COMPILE xxx	(--)	Used in the definition of a compiling word. When the compiling word, in turn, is used in a source definition, the code field address of xxx will be compiled into the dictionary entry so that when the new definition is executed, xxx will be executed.
[COMPILE] xxx	(--)	Used in a colon definition, causes the immediate word xxx to be compiled as though it were not immediate; xxx will be executed when the definition is executed.
LITERAL	compile time: (n --) run time: (--n)	Used only inside a colon definition. At compile time, compiles a value from the stack into the definition as a literal. At run time, the value will be pushed onto the stack.
[	(--)	Leaves compile mode.
]	(--)	Enters compile mode.

Review of Terms

Compile-time behavior
1. when referring to defining words: the sequence of instructions which will be carried out when the defining word is executed--these instructions perform the compilation of the member words;
2. when referring to compiling words: the behavior of a compiling word, contained within a colon definition, during compilation of the definition.

Compilee
a definition being compiled. In relation to a compiling word, the compilee is the definition whose compilation the compiling word affects.

Compiling word
a word used inside a colon definition to take some action during the compilation process.

Defining word
a word which, when executed, compiles a new dictionary entry. A defining word specifies the compile-time and run-time behavior of each member of the "family" of words that it defines.

Flowcharts
a graphic representation of the logical structure of a program or, in FORTH, of a definition.

Precedence bit
in FORTH dictionary entries, a bit which indicates whether a word should be executed rather than be compiled when it is encountered during compilation.

Run-time behavior
1. when referring to defining words: the sequence of instructions which will be carried out when any member word is executed;
2. when referring to compiling words: a routine which will be executed when the compilee is executed. Not all compiling words have run-time behavior.

Problems -- Chapter 11

1. Define a defining word named LOADED-BY that will define words which load a block when they are executed. Example:

```
6000 LOADED-BY CORRESPONDENCE
```

would define the word CORRESPONDENCE. When CORRESPONDENCE is executed, block 6000 would get loaded.

2. Define a defining word BASED. which will create number output words for specific bases. For example,

```
16 BASED. H.
```

would define H. to be a word which prints the top of the stack in hex but does not permanently change BASE.

```
DECIMAL
17 DUP H. .RETURN 11 17 ok
```

3. Define a defining word called PLURAL which will take the address of a word such as CR or STAR and create its plural form, such as CRS or STARS. You'll provide PLURAL with the address of the singular word by using tick. For instance, the phrase

```
' CR  PLURAL  CRS
```

will define CRS in the same way as though you had defined it

```
: CRS   ?DUP IF  0 DO CR  LOOP THEN ;
```

4. The French words for DO and LOOP are TOURNE and RETOURNE. Using the words DO and LOOP, define TOURNE and RETOURNE as French "aliases." Now test them by writing yourself a French loop.

5. The FORTH-79 Standard Reference Word Set contains a word called ASCII that can be used to make certain definitions more readable. Instead of using a numeric ASCII code within a definition, such as

```
: STAR   42 EMIT ;
```

you can use

```
: STAR   ASCII *  EMIT ;
```

The word ASCII reads the next character in the input stream, then compiles its ASCII equivalent into the definition as a literal. When the definition STAR is executed, the ASCII

value is pushed onto the stack.

Define the word ASCII.

6. Write a word called LOOPS which will cause the remainder of the input stream, up to the carriage return, to be executed the number of times specified by the value on the stack. For example,

```
7 LOOPS  42 EMIT  SPACE RETURN * * * * * * * ok
```