1 FUNDAMENTAL FORTH

In this chapter we'll acquaint you with some of the unique properties of the FORTH language. After a few introductory pages we'll have you sitting at a FORTH terminal. If you don't have a FORTH terminal, don't worry. We'll show you the result of each step along the way.

A Living Language

Imagine that you're an office manager and you've just hired a new, eager assistant. On the first day, you teach the assistant the proper format for typing correspondence. (The assistant already knows how to type.) By the end of the day, all you have to say is "Please type this."

On the second day, you explain the filing system. It takes all morning to explain where everything goes, but by the afternoon all you have to say is "Please file this."

By the end of the week, you can communicate in a kind of shorthand, where "Please send this letter" means "Type it, get me to sign it, photocopy it, file the copy, and mail the original." Both you and your assistant are free to carry out your business more pleasantly and efficiently.

Good organization and effective communication require that you

1. define useful tasks and give each task a name, then

2. group related tasks together into larger tasks and give each of these a name, and so on.

FORTH lets you organize your own procedures and communicate them to a computer in just this way (except you don't have to say "Please").

As an example, imagine a microprocessor-controlled washing machine programmed in FORTH. The ultimate command in your example is named WASHER. Here is the definition of WASHER, as written in FORTH:

```
    : WASHER   WASH SPIN RINSE SPIN ;
```

In FORTH, the colon indicates the beginning of a new definition. The first word after the colon, WASHER, is the name of the new procedure. The remaining words, WASH, SPIN, RINSE, and SPIN, comprise the "definition" of the new procedure. Finally, the semicolon indicates the end of the definition.

Each of the words comprising the definition of WASHER has already been defined in our washing-machine application. For example, let's look at our definition of RINSE:

```
    : RINSE   FILL AGITATE DRAIN ;
```

As you can see, the definition of RINSE consists of a group of words: FILL, AGITATE, and DRAIN. Once again, each of these words has been already defined elsewhere in our washing-machine application. The definition of FILL might be

```
    : FILL   FAUCETS OPEN  TILL-FULL  FAUCETS CLOSE ;
```

In this definition we are referring to _things_ (faucets) as well as to _actions_ (open and close). The word TILL-FULL has been defined to create a "delay loop" which does nothing but mark time until the water-level switch has been activated, indicating that the tub is full.

If we were to trace these definitions back, we would eventually find that they are all defined in terms of a group of very useful commands that form the basis of all FORTH systems. For example, polyFORTH includes about 300 such commands. Many of these commands are themselves "colon definitions" just like our example words; others are defined directly in the machine language of the particular computer. In FORTH, a defined command is called a "word."†

†For Old Hands

This meaning of "word" is not to be associated with a 16-bit value, which in the FORTH community is referred to as a "cell."

The ability to define a word in terms of other words is called "extensibility." Extensibility leads to a style of programming that is extremely simple, naturally well-organized, and as powerful as you want it to be.

Whether your application runs an assembly line, acquires data for a scientific environment, maintains a business application, or plays a game, you can create your own "living language" of words that relate to your particular need.

In this book we'll cover the most useful of the standard FORTH commands.

All This and ... Interactive!

One of FORTH's many unique features is that it lets you "execute"† a word by simply naming the word. If you're working at a terminal keyboard, this can be as simple as typing in the word and pressing the RETURN key.

Of course, you can also use the same word in the definition of any other word, simply by putting its name in the definition.

FORTH is called an "interactive" language because it carries out your commands the instant you enter them.

We're going to give an example that you can try yourself, showing the process of combining simple commands into more powerful commands. We'll use some simple FORTH words that control your terminal screen or printer. But first, let's get acquainted with the mechanics of "talking" to FORTH through your terminal's keyboard.

Take a seat at your real or imaginary FORTH terminal. We'll assume that someone has been kind enough to set everything up for you, or that you have followed all the instructions given for loading your particular computer.

† For Beginners

To "execute" a word is to order the computer to carry out a command.

Now press the key labeled:

RETURN†

The computer will respond by saying

```
ok
```

The RETURN key is your way of telling FORTH to acknowledge your request. The ok is FORTH's way of saying that it's done everything you asked it to do without any hangups. In this case, you didn't ask it to do anything, so FORTH obediently did nothing and said ok. (The ok may be either in upper case or in lower case, depending on your terminal.)

Now enter this:

```
15 SPACES
```

If you make a typing mistake, you can correct it by hitting the "backspace" key. Back up to the mistake, enter the correct letter, then continue. When you have typed the line correctly, press the RETURN key. (Once you press RETURN, it's too late to correct the line.)

In this book, we use the symbol RETURN to mark the point where you must press the RETURN key. We also underline the computer's output (even though the computer does not) to indicate who is typing what.

Here's what has happened:

```
15 SPACESRETURN                ok
```

As soon as you pressed the return key, FORTH printed fifteen blank spaces and then, having processed your request, it responded ok (at the end of the fifteen spaces).

Now enter this:

```
42 EMITRETURN *ok
```

The phrase "42 EMIT" tells FORTH to print an asterisk (we'll

†For People at Terminals

RETURN may have a different name on your terminal. Other possible names are NEW LINE and ENTER.

Backspace may also have a different name on your terminal, such as DEL or RUBOUT.

discuss this command later on in the book.) Here FORTH printed the asterisk, then responded ok.

We can put more than one command on the same line. For example:

```
    15 SPACES  42 EMIT  42 EMIT RETURN                **ok
```

This time FORTH printed fifteen spaces and two asterisks. A note about entering words and/or numbers: we can separate them from one another by as many spaces as we want for clarity. But they must be separated by at least one space for FORTH to be able to recognize them as words and/or numbers.

Instead of entering the phrase

```
    42 EMIT
```

over and over, let's define it as a word called "STAR."

Enter this

```
    : STAR   42 EMIT ; RETURN ok
```

Here "STAR" is the name; "42 EMIT" is the definition. Notice that we set off the colon and semicolon from adjacent words with a space. Also, to make FORTH definitions easy for human beings to read, we conventionally separate the name of a definition from its contents with three spaces.

After you have entered the above definition and pressed RETURN, FORTH responds ok, signifying that it has recognized your definition and will remember it. Now enter

```
    STAR RETURN *ok
```

Voila! FORTH executes your definition of "STAR" and prints an asterisk.

There is no difference between a word such as STAR that you define yourself and a word such as $\boxed{\text{EMIT}}$ that is already defined. In this book, however, we will put boxes around those words that are already defined, so that you can more easily tell the difference.

Another system-defined word is $\boxed{\text{CR}}$, which performs a carriage return and line feed at your terminal.† For example, enter this:

† For Beginners

Be sure to distinguish between the key labeled RETURN and the FORTH word $\boxed{\text{CR}}$.

```
CR RETURN _
ok
```

As you can see, FORTH executed a carriage return, then printed an ok (on the next line).

Now try this:

```
CR STAR CR STAR CR STAR RETURN
*
*
*ok
```

Let's put a CR in a definition, like this:

```
: MARGIN   CR  30 SPACES ; RETURN ok
```

Now we can enter

```
 MARGIN STAR MARGIN STAR MARGIN STAR RETURN
```

and get three stars lined up vertically, thirty spaces in from the left.

Our MARGIN STAR combination will be useful for what we intend to do, so let's define

```
: BLIP   MARGIN STAR ; RETURN ok
```

We will also need to print a horizontal row of stars. So let's enter the following definition (we'll explain how it works in a later chapter):

```
: STARS   0 DO STAR LOOP ; RETURN ok
```

Now we can say

```
5 STARS RETURN *****ok
```

or

```
35 STARS RETURN ***********************************ok
```

or any number of stars imaginable!

We will need a word which performs MARGIN, then prints five stars. Let's define it like this:

```
: BAR   MARGIN  5 STARS ; RETURN ok
```

Now we can enter

```
BAR BLIP BAR BLIP BLIP  CR
```

and get a letter "F" (for FORTH) made up of stars. It should look like this:

```
*****
*
*****
*
*
```

The final step is to make this new procedure a word. Let's call the word "F":

```
    : F   BAR BLIP BAR BLIP BLIP  CR ; RETURN  ok
```

You've just seen an example of the way simple FORTH commands can become the foundation for more complex commands. A FORTH application, when listed,† consists of a series of increasingly powerful definitions rather than a sequence of instructions to be executed in order.

To give you a sample of what a FORTH application really looks like, here's a listing of our experimental application:

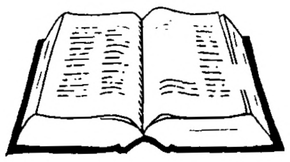
```
( LARGE LETTER-F)
: STAR   42 EMIT ;
: STARS   0 DO STAR LOOP ;
: MARGIN   CR 30 SPACES ;
: BLIP   MARGIN STAR ;
: BAR   MARGIN 5 STARS ;
: F   BAR BLIP BAR BLIP BLIP  CR ;
```

†For Beginners

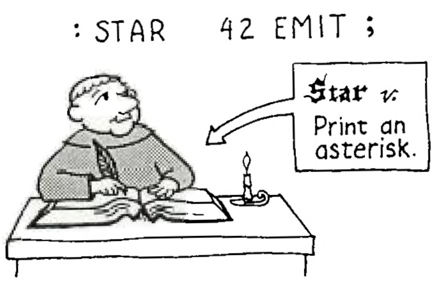
We'll explain more about listing, as it applies to FORTH, in Chapter 3.

The Dictionary

Each word and its definition are entered into FORTH's "dictionary." The dictionary already contained many words when you started, but your own words are now in the dictionary as well.



When you define a new word, FORTH translates your definition into dictionary form and writes the entry in the dictionary. This process is called "compiling."†



For example, when you enter the line

```
    : STAR   42 EMIT ;RETURN
```

the compiler compiles the new definition into the dictionary. The compiler does not print the asterisk.

Once a word is in the dictionary, how is it executed? Let's say you enter the following line directly at your terminal (not inside a definition):

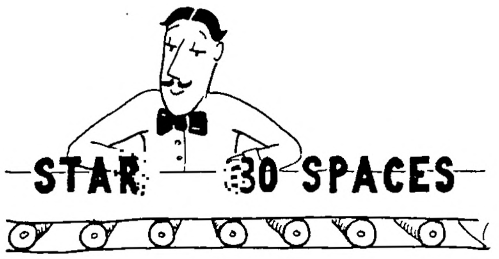
```
    STAR 30 SPACESRETURN
```

This will activate a word called $\boxed{\text{INTERPRET}}$, also known as the "text interpreter."

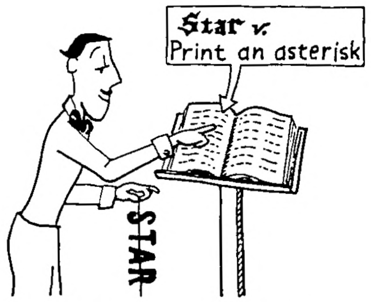
---

†For Beginners

Compilation is a general computer term which normally means the translation of a high-level program into machine code that the computer can understand. In FORTH it means the same thing, but specifically it means writing in the dictionary.



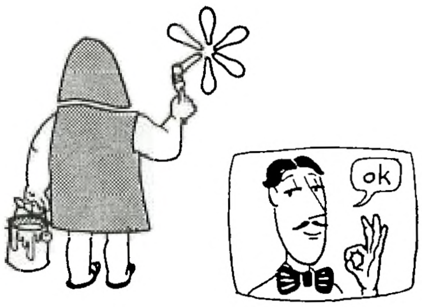
The text interpreter scans the input stream, looking for strings of characters separated by spaces.



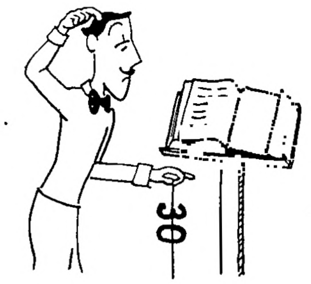
When he finds such a string, he looks it up in the dictionary.

Star v. Print an asterisk

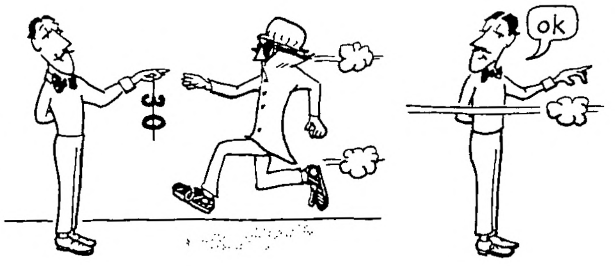
If he finds the word in the dictionary, he points out the definition to a word called EXECUTE--



--who then executes the definition (in this case, he prints an asterisk). The interpreter says everything's "ok."



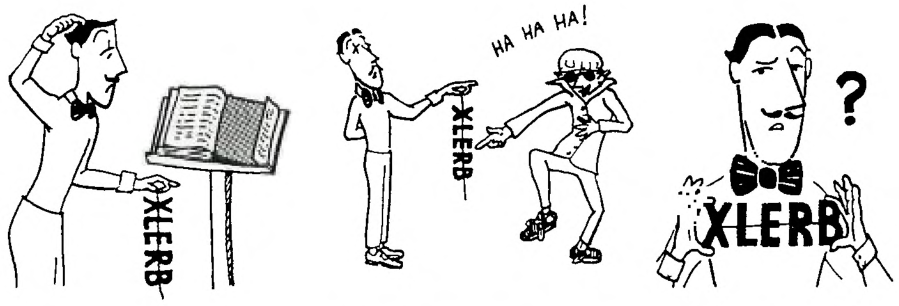
If the interpreter cannot find the string in the dictionary, he calls the number-runner (called NUMBER).



NUMBER knows a number when he sees one. If NUMBER finds a number, he runs it off to a temporary storage location for numbers.

What happens when you try to execute a word that is not in the dictionary? Enter this and see what happens:

```
    XLERB RETURN  XLERB ?
```



When the text interpreter can't find XLERB in the dictionary, it tries to pass it off on NUMBER. NUMBER shines it on. Then the interpreter returns the string to you with a question mark.

In some versions of FORTH, including polyFORTH, the compiler does not copy the entire name of the definition into the dictionary--only the first three characters and the number of characters. For example, in polyFORTH, the text interpreter cannot distinguish between STAR and STAG because both words are four characters in length and both begin S-T-A.†

While many professional programmers prefer the three-character rule because it saves memory, certain programmers and many hobbyists enjoy the freedom to choose any name. The FORTH-79 Standard allows up to thirty-one characters of a name to be stored in the dictionary.

To summarize: when you type a pre-defined word at the terminal, it gets interpreted and then executed.

Now remember we said that : is a word? When you type the word :, as in

```
    : STAR   42 EMIT ; RETURN
```

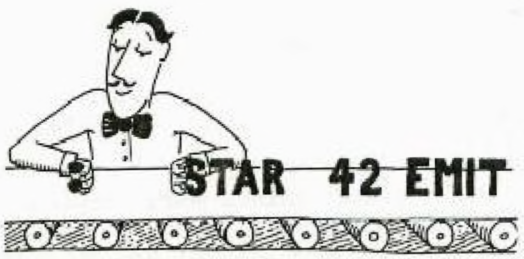
† For polyFORTH Users

The trick to avoiding conflicts is to

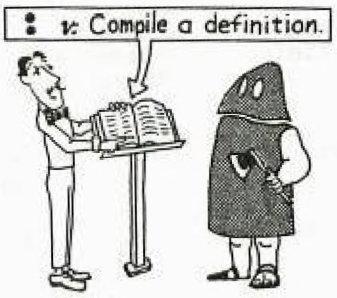
a) be conscious of your name choices, and
b) when naming a series of similar words, put the distinguishing character up front, like this:

```
        1LINE  2LINE  3LINE  etc.
```

the following occurs:

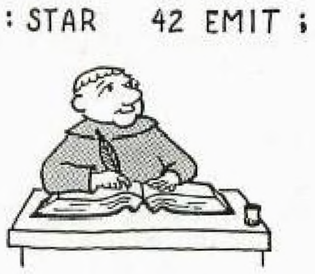


The text interpreter finds the colon in the input stream,



and points it out to EXECUTE.

EXECUTE says, "Please start compiling."



The compiler translates the definition into dictionary form and writes it in the dictionary.

When the compiler gets to the semicolon, he stops,

and execution returns to the text interpreter, who gives the message ok.

Say What?

In FORTH, a word is a character or group of characters that have a definition. Almost any characters can be used in naming a word. The only characters that cannot be used are:

return	because the computer thinks you've finished entering,†
backspace	because the computer thinks you're trying to correct a typing error,
space	because the computer thinks it's the end of the word, and
caret (↑ or ^)	because the editor (if you're using it) thinks you mean something else. We'll discuss the editor in Chap. 3.

Here is a FORTH word whose name consists of two punctuation marks. The word is $\boxed{."}$ and is pronounced dot-quote. You can use $\boxed{."}$ inside a definition‡ to type a "string" of text at your terminal. Here's an example:

```
    : GREET   ." HELLO, I SPEAK FORTH " ;RETURN ok
```

We've just defined a word called GREET. Its definition consists of just one FORTH word, $\boxed{."}$, followed by the text we want typed. The quotation mark at the end of the text will not be typed; it marks the end of the text. It's called a "delimiter."

---

†For Philosophers

No, the computer doesn't "think." Unfortunately, there's no better word for what it really does. We say "think" on the grounds that it's all right to say, "the lamp needs a new light bulb." Whether the lamp really needs a bulb depends on whether it needs to provide light (that is, incandescence is its karma). So let's just say the computer thinks.

‡FORTH-79 Standard

In systems that conform to the Standard, $\boxed{."}$ will execute outside of a colon definition as well.

When entering the definition of GREET, don't forget the closing ; to end the definition.

Let's execute GREET:

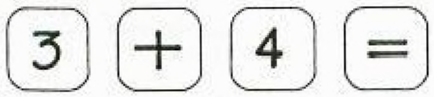
```
    GREET RETURN HELLO, I SPEAK FORTH ok
```

The Stack: FORTH's Worksite for Arithmetic

A computer would not be much good if it couldn't do arithmetic. If you've never studied computers before, it may seem pretty amazing that a computer (or even a pocket calculator) can do arithmetic at all. We can't cite all the mechanics in this book, but believe us, it's not a miracle.

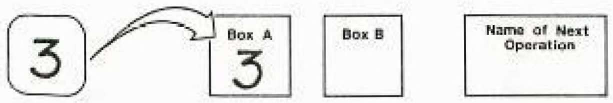
In general, computers perform their operations by breaking everything they do into ridiculously tiny pieces of information and ridiculously easy things to do. To you and me, "3 + 4" is just "7," without even thinking. To a computer, "3 + 4" is actually a very long list of things to do and remember.

Without getting too specific, let's say you have a pocket calculator which expects its buttons to be pushed in this order:

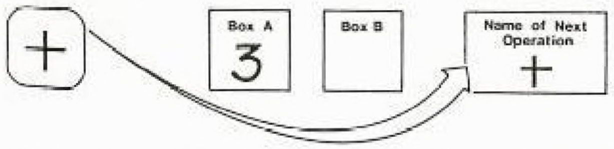


in order to perform the addition and display the result. Here's a generalized picture of what might occur:

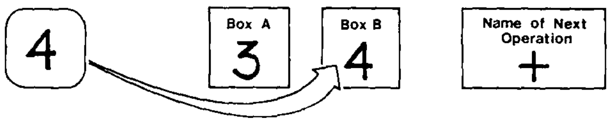
When you press



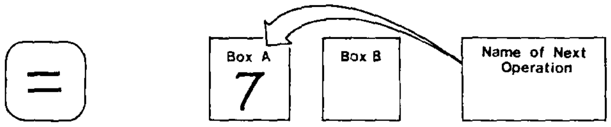
--the number 3 goes into one place (called Box A).



—the intended operation (addition) is remembered somehow.



—the number 4 is stored into a second place (called Box B).



—the calculator performs the operation that is stored in the "Next Operation" Box on the contents of the number boxes and leaves the result in Box A.

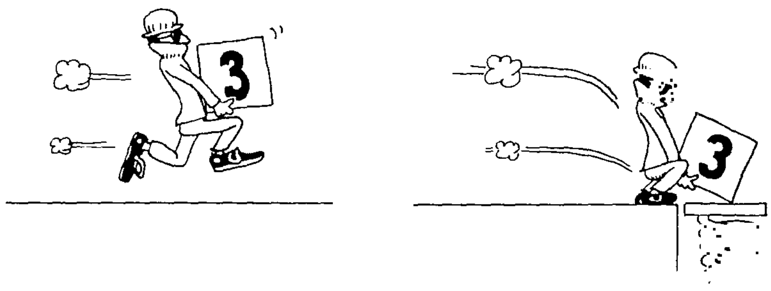
Many calculators and computers approach arithmetic problems in a way similar to what we've just described. You may not be aware of it, but these machines are actually storing numbers in various locations and then performing operations on them.

In FORTH, there is one central location where numbers are temporarily stored before being operated on. That location is called the "stack." Numbers are "pushed onto the stack," and then operations work on the numbers on the stack.

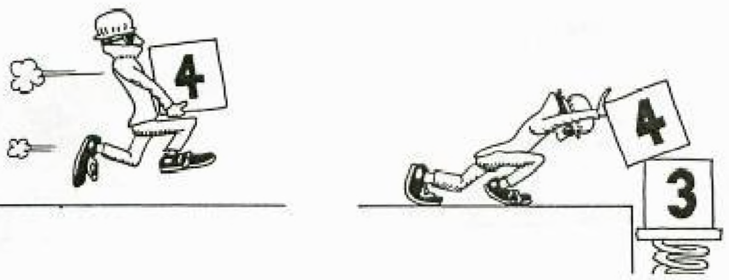
The best way to explain the stack is to illustrate it. If you enter the following line at your terminal:

```
3 4 + . RETURN 7 ok
```

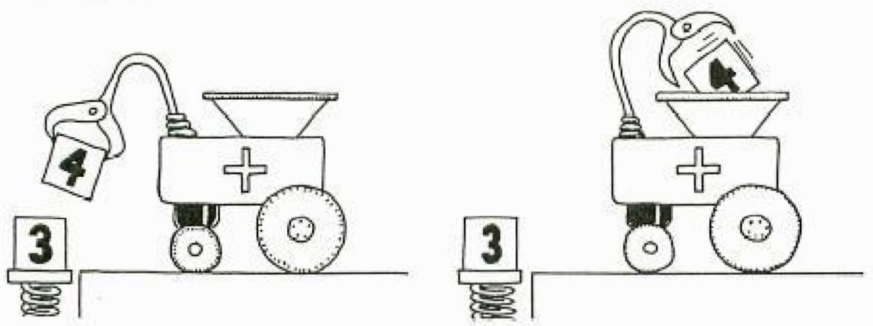
here's what happens, key by key.

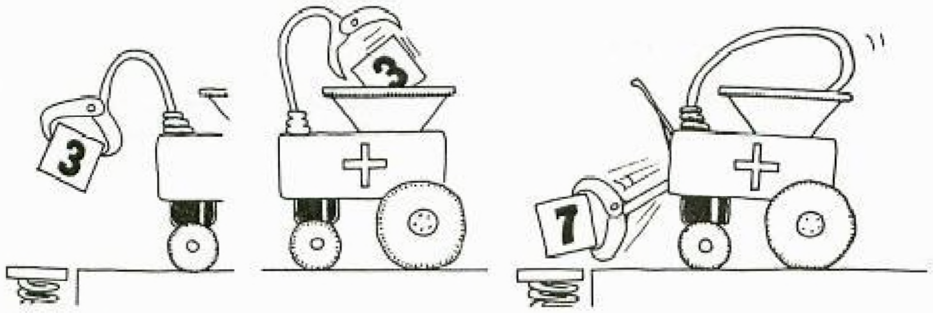


Recall that when you enter a number at your terminal, the text interpreter hands it over to NUMBER, who runs it to some location. That location, it can now be told, is the stack. In short, when you enter the number three from the terminal, you push it onto the stack.

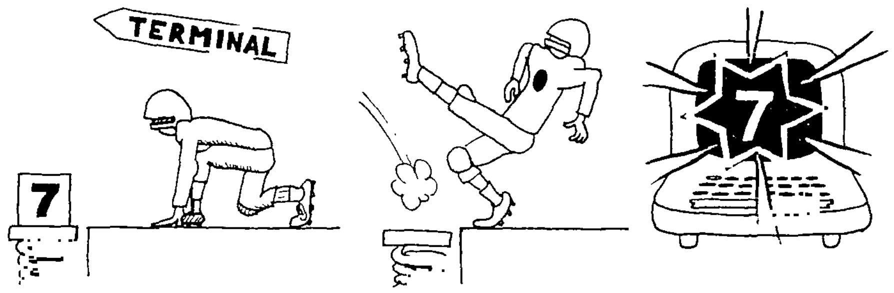


Now the four goes onto the "top" of the stack and pushes the three downward.





The next word in the input stream can be found in the dictionary. $\boxed{+}$ has been previously defined to "take the top two numbers off the stack, add them, and push the result back onto the stack."



The next word, $\boxed{.}$, is also found in the dictionary. It has been previously defined to take the number off the stack and print it at the terminal.

Postfix Power

Now wait, you say. Why does FORTH want you to type

```
3 4 +
```

instead of

```
3 + 4
```

which is more familiar to most people?

FORTH uses "postfix" notation (so called because the operator is affixed <u>after</u> the numbers) rather than "infix" notation (so called because the operator is affixed <u>in-between</u> the numbers) so that all words which "need" numbers can get them from the stack.†

†For Pocket-calculator Experts

Hewlett-Packard calculators feature a stack and postfix arithmetic.

For example:

> the word $\boxed{+}$ gets two numbers from the stack and adds them;
>
> the word $\boxed{.}$ gets one number from the stack and prints it;
>
> the word $\boxed{\text{SPACES}}$ gets one number from the stack and prints that many spaces;
>
> the word $\boxed{\text{EMIT}}$ gets a number that represents a character and prints that character;
>
> even the word STARS, which we defined ourselves, gets a number from the stack and prints that many stars.

When all operators are defined to work on the values that are already on the stack, interaction between many operations remains simple even when the program gets complex.

Earlier we pointed out that FORTH lets you execute a word in either of two ways: by simply naming it, or by putting it in the definition of another word and naming that word. Postfix is part of what makes this possible.

Just as an example, let's suppose we wanted a word that will always add the number 4 to whatever number is on the stack (for no other purpose than to illustrate our point). Let's call the word

```
    FOUR-MORE
```

We could define it this way:

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

and test it this way:

```
    3 FOUR-MORE .RETURN 7 ok
```

and again:

```
    -10 FOUR-MORE .RETURN -6 ok
```

The "4" inside the definition goes onto the stack, just as it would if it were outside a definition. Then the $\boxed{+}$ adds the two numbers on the stack. Since $\boxed{+}$ always works on the stack, it doesn't care that the "4" came from inside the definition and the three from outside.

As we begin to give some more complicated examples, the value of the stack and of postfix arithmetic will become increasingly apparent to you. The more operators that are involved, the more important it is that they all be able to "communicate" with each other.

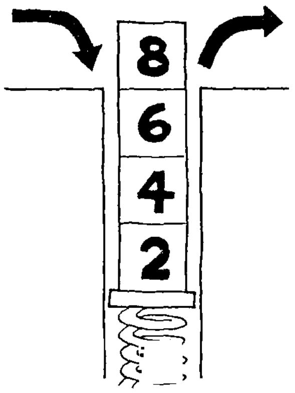
Keep Track of Your Stack

We've just begun to demonstrate the philosophy behind the stack and postfix notation. Before we continue, however, let's look more closely at the stack in action and get accustomed to its peculiarities.

FORTH's stack is described as "last-in, first-out" (LIFO). You can see from the earlier illustration why this is so. The three was pushed onto the stack first, then the four pushed on top of it. Later the adding machine took the four off first because it was on top. Hence "last-in, first-out."

In general, the only accessible value at any given time is the top value. Let's use another operation, the $\boxed{.}$ to further demonstrate. Remember that each $\boxed{.}$ removes one number from the stack and prints it. Four dots, therefore, remove four numbers and print them.

```
2 4 6 8 . . . . RETURN 8 6 4 2 ok
```



The system reads input from left to right and executes each word in turn.

For input, the rightmost value on the screen will end up on top of the stack.

For output, the rightmost value on the screen came from the bottom of the stack.

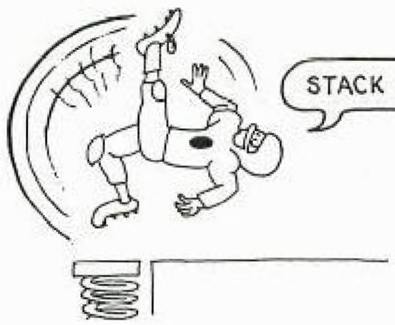
Let's see what kind of trouble we can get outselves into. Type:

```
10 20 30 . . . .
```

(that's four dots) then RETURN. What you get is:

```
10 20 30 . . . . RETURN 30 20 10 0 . STACK EMPTY†
```

Each dot removes one value. The fourth dot found that there was no value left on the stack to send to the terminal, and it told you so.



This error is called "stack underflow." (Notice that a stack underflow is not "ok.")

The opposite condition, when the stack completely fills up, is called "stack overflow." The stack is so deep, however, that this condition should never occur except when you've done something terribly wrong.

It's important to keep track of new words' "stack effects"; that is, the sort of numbers a word needs to have on the stack before you execute it, and the sort of numbers it will leave on the stack afterwards.

If you maintain a list of your newly created words with their meanings as you go, you or anyone else can easily understand the words' operations. In FORTH, such a list is called a "glossary."

To communicate stack effects in a visual way, FORTH programmers conventionally use a special stack notation in their glossaries or tables of words. We're introducing the stack notation now so that you'll have it under your belt when you begin the next chapter.

---

†For the Curious

Actually, dot always prints whatever is on the top, so if there is nothing on the stack, it prints whatever is just below the stack, which is usually zero. Only then is the error detected; the offending word (in this case dot) is returned to the screen, followed by the "error message."

Here's the basic form:

```
    (before -- after)
```

The dash separates the things that should be on the stack (before you execute the word) from the things that will be left there afterwards. For example, here's the stack notation for the word $\boxed{.}$:

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

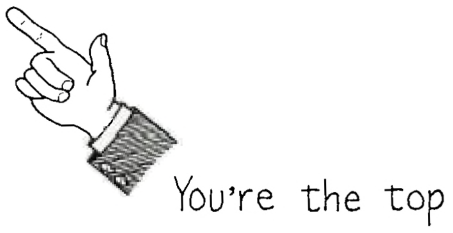
(The letter "n" stands for "number.") This shows that $\boxed{.}$ expects one number on the stack (before) and leaves <u>no</u> number on the stack (after).

Here's the stack notation for the word $\boxed{+}$.

```
    +       (n1 n2 -- sum)
```

When there is more than one n, we number them n1, n2, n3, etc., consecutively. The numbers 1 and 2 do <u>not</u> refer to position on the stack. Stack position is indicated by the order in which the items are written; the <u>rightmost</u> item on either side of the arrow is the <u>topmost</u> item on the stack. For example, in the stack notation of $\boxed{+}$, the n2 is on top:

```
    +       (n1 n2 -- sum)
```



Since you probably have the hang of it by now, we'll be leaving out the RETURN symbol except where we feel it's needed for clarity. You can usually tell where to press "return" because the computer's response is always underlined.

Here's a list of the FORTH words you've learned so far, including their stack notations ("n" stands for number; "c" stands for character):

: xxx yyy ;	(--)	Creates a new definition with the name xxx, consisting of word or words yyy.
CR	(--)	Performs a carriage return and line feed at your terminal.
SPACES	(n --)	Prints the given number of blank spaces at your terminal.
SPACE	(--)	Prints one blank space at your terminal.
EMIT	(c --)	Transmits a character to the output device.
." xxx"	(--)	Prints the character string xxx at your terminal. The " character terminates the string.
+	(n1 n2 -- sum)	Adds.
.	(n --)	Prints a number, followed by one space.

In the next chapter we'll talk about getting the computer to perform some fancier arithmetic.

Review of Terms

Compile	to generate a dictionary entry in computer memory from source text (the written-out form of a definition). Distinct from "execute."
Dictionary	in FORTH, a list of words and definitions including both "system" definitions (predefined) and "user" definitions (which you invent). A dictionary resides in computer memory in compiled form.

Execute	to perform. Specifically, to execute a word is to perform the operations specified in the compiled definition of the word.
Extensibility	a characteristic of a computer language which allows a programmer to add new features or modify existing ones.
Glossary	a list of words defined in FORTH, showing their stack effects and an explanation of what they do, which serves as a reference for programmers.
Infix notation	the method of writing operators between the operands they affect, as in "2 + 5."
Input stream	the text to be read by the text interpreter. This may be text that you have just typed in at your terminal, or it may be text that is stored on disk.
Interpret	(when referring to FORTH's text interpreter) to read the input stream, then to find each word in the dictionary or, failing that, to convert it to a number.
LIFO	(last-in, first-out) the type of stack which FORTH uses. A can of tennis balls is a LIFO structure; the last ball you drop in is the one you must remove first.
Postfix notation	the method of writing operators after the operands they affect, as in "2 5 +" for "2 + 5." Also known as Reverse Polish Notation.
Stack	in FORTH, a region of memory which is controlled in such a way that data can be stored or removed in a last-in, first-out (LIFO) fashion.
Stack overflow	the error condition that occurs when the entire area of memory allowed for the stack is completely filled with data.
Stack underflow	the error condition that occurs when an operation expects a value on the stack, but there is no valid data on the stack.
Word	in FORTH, the name of a definition.

Problems -- Chapter 1

Note: before you work these problems, remember these simple rules:

Every $\boxed{:}$ needs a $\boxed{;}$.

and

Every $\boxed{\texttt{."}}$ needs a $\boxed{\texttt{"}}$.

1. Define a word called GIFT which, when executed, will type out the name of some gift. For example, you might try:

```
: GIFT   ." BOOKENDS " ;
```

Now define a word called GIVER which will print out a person's first name. Finally, define a word called THANKS which includes the new FORTH words GIFT and GIVER, and prints out a message something like this:

```
DEAR STEPHANIE,
     THANKS FOR THE BOOKENDS. ok
```

2. Define a word called TEN.LESS which takes a number on the stack, subtracts ten, and returns the answer on the stack. (Hint: you can use $\boxed{+}$.)

3. After entering the words in Prob. 1, enter a new definition for GIVER to print someone else's name, then execute THANKS again. Can you explain why THANKS still prints out the first giver's name?