# 4 DECISIONS, DECISIONS, ...

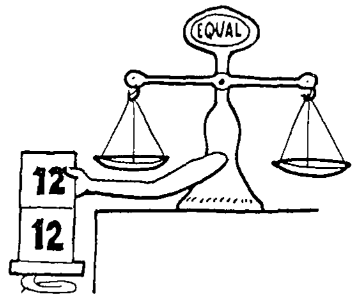
In this chapter we'll learn how to program the computer to make "decisions." This is the moment when you turn your computer into something more than an ordinary calculator.

## The Conditional Phrase

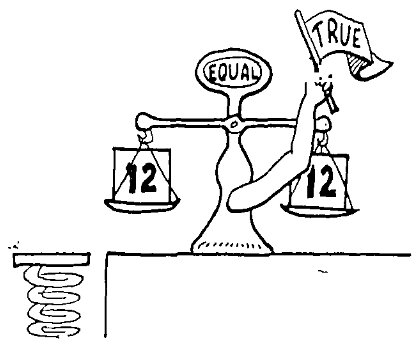
Let's see how to write a simple decision-making statement in FORTH. Imagine we are programming a mechanical egg-carton packer. Some sort of mechanical device has counted the eggs on the conveyor belt, and now we have the number of eggs on the stack. The FORTH phrase:

```
    12 =  IF  FILL-CARTON  THEN
```

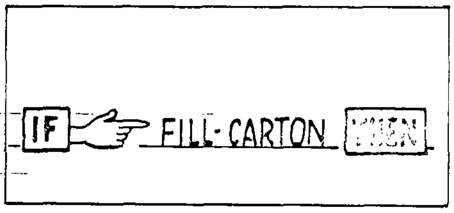
tests whether the number on the stack is equal to 12, and if it is, the word FILL-CARTON is executed. If it's not, execution moves right along to the words that follow $\boxed{\text{THEN}}$.



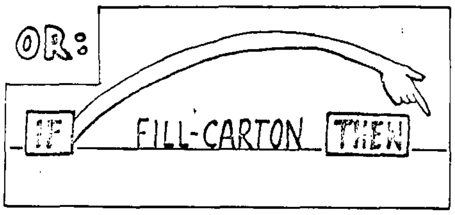
The word $\boxed{=}$ takes two values off the stack



and compares them to see whether they are equal.



If the condition is true, $\boxed{\text{IF}}$ allows the flow of execution to continue with the next word in the definition.



But if the condition is false, $\boxed{\text{IF}}$ causes the flow of execution to skip to $\boxed{\text{THEN}}$, from which point execution will proceed.

Let's try it. Define this example word:

```
: ?FULL   12 =  IF ." IT'S FULL " THEN ; ok
11 ?FULL ok
12 ?FULL IT'S FULL ok
```

Notice: an $\boxed{\text{IF}}$...$\boxed{\text{THEN}}$ statement must be contained within a colon definition. You can't just enter these words in "calculator style."

Don't be misled by the traditional English meanings of the FORTH words $\boxed{\text{IF}}$ and $\boxed{\text{THEN}}$. The words that follow $\boxed{\text{IF}}$ are executed if the condition is true. The words that follow $\boxed{\text{THEN}}$ are always executed, as though you were telling the computer, "After you make the choice, then continue with the rest of the definition." (In this example, the only word after $\boxed{\text{THEN}}$ is $\boxed{;}$, which ends the definition.)

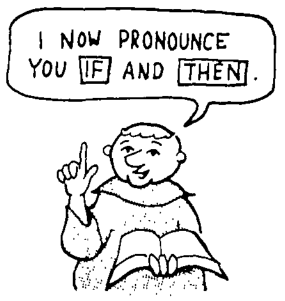
Let's look at another example. This definition checks whether the temperature of a laboratory boiler is too hot. It expects to find the temperature on the stack:

```
: ?TOO-HOT   220 > IF ." DANGER -- REDUCE HEAT " THEN ; ok
```

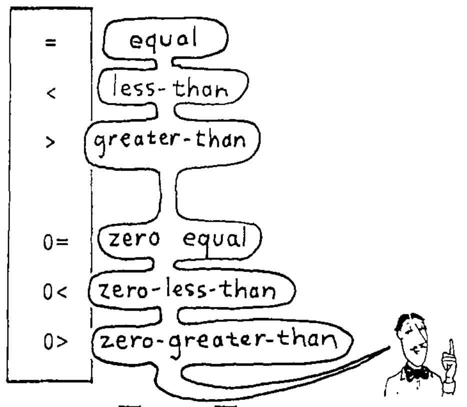
If the temperature on the stack is greater than 220, the danger message will be printed at the terminal. You can execute this one yourself, by entering the definition, then typing in a value just before the word.

```
290 ?TOO-HOT DANGER -- REDUCE HEAT ok
130 ?TOO-HOT ok
```

Remember that every IF needs a THEN to come home to. Both words must be in the same definition.



Here is a partial list of comparison operators that you can use before an IF...THEN statement:



The words < and > expect the same stack order as the arithmetic operators, that is:

| <u>Infix</u> | | <u>Postfix</u> |
|---|---|---|
| 2 < 10 | is equivalent to | 2 10 < |
| 17 > -39 | is equivalent to | 17 -39 > |

The words 0=, 0<, and 0> expect only one value on the stack. The value is compared with zero.

Another word, NOT, doesn't test any value at all; it simply reverses whatever condition has just been tested. For example, the phrase:

```
... = NOT IF ...
```

will execute the words after IF, if the two numbers on the stack are <u>not</u> equal.

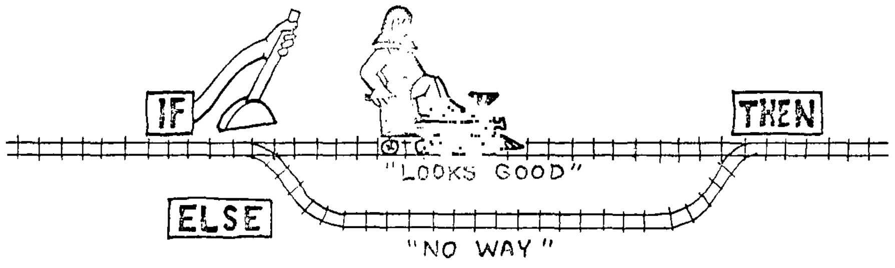
## The Alternative Phrase

FORTH allows you to provide an alternative phrase in an $\boxed{\text{IF}}$ statement, with the word $\boxed{\text{ELSE}}$.

The following example is a definition which tests whether a given number is a valid day of the month:

```
: ?DAY   32 < IF ." LOOKS GOOD "  ELSE ." NO WAY " THEN ;
```

If the number on the stack is less than thirty-two, the message "LOOKS GOOD" will be printed. Otherwise, "NO WAY" will be printed.



Imagine that $\boxed{\text{IF}}$ pulls a railroad-track switch, depending on the outcome of the test. Execution then takes one of two routes, but either way, the tracks rejoin at the word $\boxed{\text{TH}}$[illegible].

By the way, in computer terminology, this whole business of rerouting the path of execution is called "branching."†

Here's a more useful example. You know that dividing any number by zero is impossible, so if you try it on a computer, you'll get an incorrect answer. We might define a word which only performs division if the denominator is not zero. The following definition expects stack items in this order:

---

†For Old Hands

FORTH has no GOTO statement. If you think you can't live without GOTO, just wait. By the end of this book you'll be telling your GOTO where to GOTO.

```
    (numerator denominator -- )

    : /CHECK   DUP 0= IF ." INVALID "  DROP
                    ELSE / THEN ; †
```

Notice that we first have to DUP the denominator because the phrase

```
   0= IF
```

will destroy it in the process.

Also notice that the word DROP removes the denominator if division won't be performed, so that whether we divide or not, the stack effect will be the same.

## Nested IF...THEN Statements

It's possible to put an IF...THEN (or IF...ELSE...THEN) statement inside another IF...THEN statement. In fact, you can get as complicated as you like, so long as every IF has one THEN.

Consider the following definition, which determines the size of commercial eggs (extra large, large, etc.), given their weight in ounces per dozen:

```
    : EGGSIZE    DUP   18 <  IF ." REJECT "       ELSE
                 DUP   21 <  IF ." SMALL "        ELSE
                 DUP   24 <  IF ." MEDIUM "       ELSE
                 DUP   27 <  IF ." LARGE "        ELSE
                 DUP   30 <  IF ." EXTRA LARGE "  ELSE
                                ." ERROR "              ‡
                       THEN THEN THEN THEN THEN   DROP ;
```

†For Experts

There are better ways to do this, as we'll see.

‡For People at Terminals

Because this definition is fairly long, we suggest you load it from a disk block.

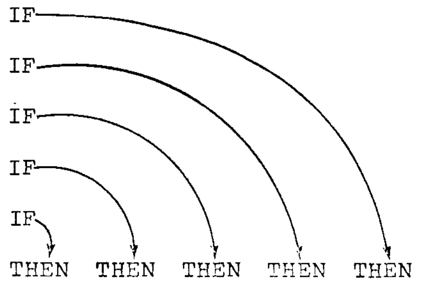
Once EGGSIZE has been loaded, here are some results you'd get:

```
    23 EGGSIZE MEDIUM ok
    29 EGGSIZE EXTRA LARGE ok
    40 EGGSIZE ERROR ok
```

We'd like to point out a few things about EGGSIZE:

The entire definition is a series of "nested" IF...THEN statements. The word "nested" does not refer to the fact that we're dealing with eggs, but to the fact that the statements nest inside one another, like a set of mixing bowls.

The five THENs at the bottom close off the five IFs in reverse order; that is:



Also notice that a DROP is necessary at the end of the definition to get rid of the original value.

Finally, notice that the definition is visually organized to be read easily by human beings. Most FORTH programmers would rather waste a little space in a block (there are plenty of blocks) than let things get any more confused than they have to be.

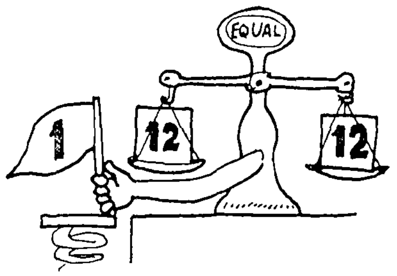
---

†For Trivia Buffs

Here is the official table on which this definition is based:

| | |
|---|---|
| Extra Large | 27-30 |
| Large | 24-27 |
| Medium | 21-24 |
| Small | 18-21 |

## A Closer Look at $\boxed{\text{IF}}$



How does the comparison operator ($\boxed{=}$, $\boxed{<}$, $\boxed{>}$, or whichever) let $\boxed{\text{IF}}$ know whether the condition is true or false? By simply leaving a one or a zero on the stack. A one means that the condition is true; a zero means that the condition is false.

In computer jargon, when one piece of program leaves a value as a signal for another piece of program, that value is called a "flag."

Try entering the following phrases at the terminal, letting $\boxed{.}$ show you what's on the stack as a flag.

```
    5 4 > . 1 ok
    5 4 < . 0 ok
```

(It's okay to use comparison operators directly at your terminal like this, but remember that an $\boxed{\text{IF}}$...$\boxed{\text{THEN}}$ statement must be wholly contained within a definition because it involves branching.)

$\boxed{\text{IF}}$ will take a one as a flag that means true and a zero as a flag that means false. Now let's take a closer look at $\boxed{\text{NOT}}$, which reverses the flag on the stack.

```
    0 NOT . 1 ok
    1 NOT . 0 ok
```

Now we'll let you in on a little secret: $\boxed{\text{IF}}$ will take <u>any non-zero value</u> to mean true.[†] So what, you ask? Well, the fact

---

[†]For the Doubting Few

Just to prove it, try entering this test:

```
    : TEST   IF ." NON-ZERO " ELSE ." ZERO " THEN ; ‡
```

Even though there is no comparison operator in the above definition, you'll still get

```
        0 TEST ZERO ok
        1 TEST [illegible] ok
     -400 TEST NON-ZERO ok
```

[‡]For Memory-Misers Who Read the above Footnote

```
    : TEST   IF ." NON-"  THEN ." ZERO " ;
```

that an arithmetic zero is identical to a flag that means "false" leads to some interesting results.

For one thing, if all you want to test is whether a number is zero, you don't need a comparison operator at all. For example, a slightly simpler version of /CHECK, which we saw earlier, could be

```
: /CHECK   DUP IF /  ELSE ." INVALID " DROP  THEN ;
```

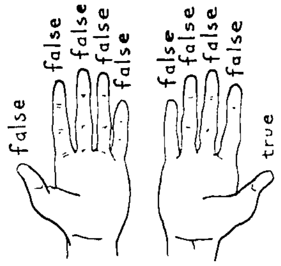
Here's another interesting result. Say you want to test whether a number is an even multiple of ten, such as 10, 20, 30, 40, etc. You know that the phrase

```
10 MOD
```

divides by ten and returns the remainder only. An even multiple of ten would produce a zero remainder, so the phrase

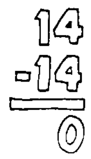
```
10 MOD 0=
```

gives the appropriate "true" or "false" flag.



If you think about it, both $\boxed{0=}$ and $\boxed{NOT}$ do exactly the same thing: they change zeros to ones and non-zeros to zeros. They have different names because one makes more sense dealing with numbers, the other with flags.

Still another interesting result is that you can use $\boxed{-}$ (minus) as a comparison operator which tests whether two values are "not equal." When you subtract two equal numbers, you get zero (false); when you subtract two unequal numbers, you get a non-zero value (true).

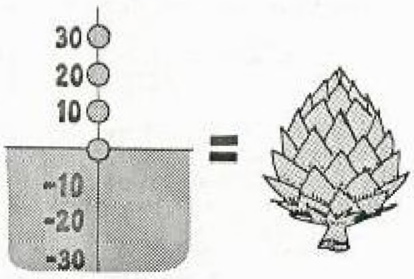


And a final result is described in the next section.

## A Little Logic

It's possible to take several flags from various tests and combine them into a single flag for one $\boxed{\text{IF}}$ statement. You might combine them as an "either/or" decision, in which you make two comparison tests. If either or both of the tests are true, then the computer will execute something. If neither is true, it won't.

Here's a rather simple-minded example, just to show you what we mean. Say you want to print the name "ARTICHOKE" if an input number is either negative or a multiple of ten.



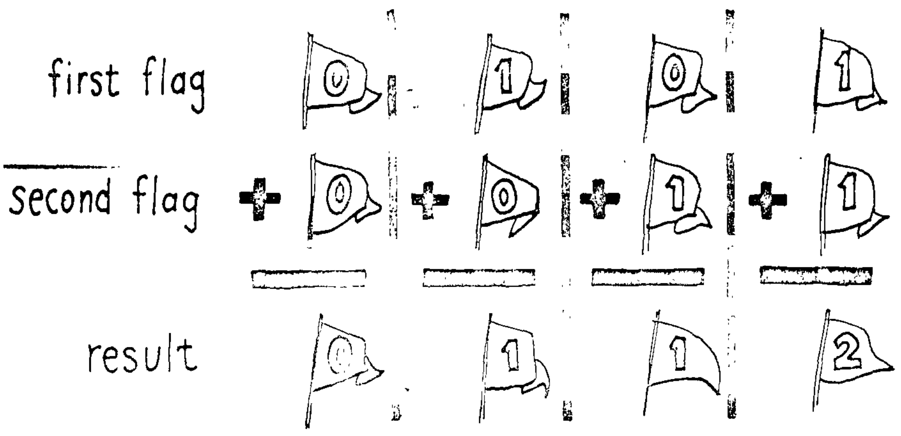
How do you do this in FORTH? Consider the phrase:

```
DUP 0< SWAP 10 MOD 0= +
```

Here's what happens when the input number is, say, 30:

| Operator | Contents of Stack | | Operation |
|---|---|---|---|
| | | 30 | |
| DUP | 30 | 30 | Duplicates it so we can test it twice. |
| 0< | 30 | 0 | Is it negative? No (zero). |
| SWAP | 0 | 30 | Swaps the flag with the number. |
| 10 MOD 0= | 0 | 1 | Is it evenly divisible by 10? Yes (one). |
| + | | 1 | Adds the flags. |

Adds the flags? What happens when you add flags? Here are four possibilities:



Lo and behold, the result flag is true if either or both conditions are true. In this example, the result is one, which means "true." If the input number had been -30, then both conditions would have been true and the sum would have been two. Two is, of course, non-zero. So as far as $\boxed{\text{IF}}$ is concerned, two is as true as one.

Our simple-minded definition, then, would be:

```
: VEGETABLE   DUP 0<  SWAP  10 MOD 0=  +
    IF ." ARTICHOKE " THEN ;
```

Here's an improved version of a previous example called ?DAY.

The old ?DAY only caught entries over thirty-one. But negative numbers shouldn't be allowed either. How about this:

```
: ?DAY   DUP 1 <  SWAP 31 >  +
    IF ." NO WAY " ELSE ." THANK YOU " THEN ;
```

The above two examples will always work because any "true" flags will always be exactly "1." In some cases, however, a flag may be any non-zero value, not just "1," in which case it's dangerous to add them with $\boxed{+}$. For example,

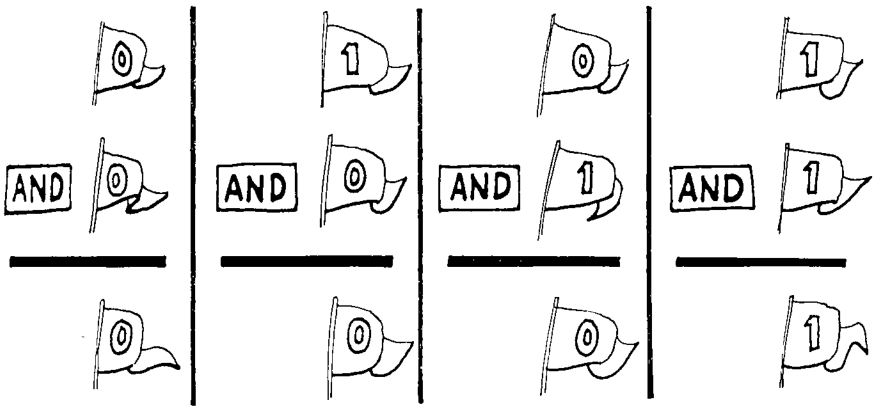
```
1 -1 + . 0 ok
```

gives us a mathematically correct answer but not the answer we want if 1 and -1 are flags.

For this reason, FORTH supplies a word called $\boxed{\text{OR}}$, which will return the correct flag even in the case of 1 and -1. An "or decision" is the computer term for the kind of flag combination we've been discussing. For example, if either the front door or the back door is open (or both), flies will come in.

Another kind of decision is called an "and" decision. In an "and" decision, both conditions must be true for the result to be true. For example, the front door and the back door must both be open for a breeze to come through. If there are three or more conditions, they must all be true.[†]

How can we do this in FORTH? By using the handy word $\boxed{\text{AND}}$. Here's what $\boxed{\text{AND}}$ would do with the four possible combinations of flags we saw earlier:



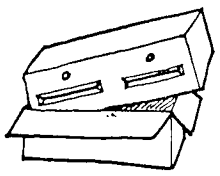
In other words, only the combination "1 1 AND" produces a result of one.

Let's say we're looking for a cardboard box that's big enough to fit a disk drive which measures:

height 6"

width 19"

length 22"



The height, width, and length requirements all must be satisfied for the box to be big enough. If we have the dimensions of a box on the stack, then we can define:

†For the Curious Newcomer

The use of words like "or" and "and" to structure part of an application is called "logic." A form of notation for logical statements was developed in the nineteenth century by George Boole; it is now called Boolean algebra. Thus the term "a Boolean flag" (or even just "a Boolean") simply refers to a flag that will be used in a logical statement.

```
: BOXTEST   ( length width height -- )
    6 >  ROT 22 >  ROT 19 >  AND AND
    IF ." BIG ENOUGH " THEN ;
```

Notice that we've put a comment inside the definition, to remind us of stack effects. This is particularly wise when the stack order is potentially confusing or hard to remember.

You can test BOXTEST with the phrase:

```
23 20 7 BOXTEST BIG ENOUGH ok
```

As your applications become more sophisticated, you will be able to write statements in FORTH that look like postfix English and are very easy to read. Just define the individual words within the definition to check some condition somewhere, then leave a flag on the stack.

An example is:

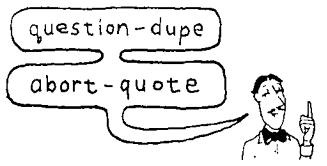
```
: SNAPSHOT   ?LIGHT ?FILM AND  IF PHOTOGRAPH THEN ;
```

which checks that there is available light and that there is film in the camera before taking the picture. Another example, which might be used in a computer-dating application, is:

```
: MATCH   HUMOROUS SENSITIVE AND
    ART.LOVING MUSIC.LOVING OR  AND  SMOKING NOT AND
    IF ." I HAVE SOMEONE YOU SHOULD MEET " THEN ;
```

where words like HUMOROUS and SENSITIVE have been defined to check a record in a disk file that contains information on other applicants of the appropriate sex.

## Two Words with Built-in IFs



?DUP

The word ?DUP duplicates the top stack value only if it is non-zero. This can eliminate a few surplus words. For example, the definition

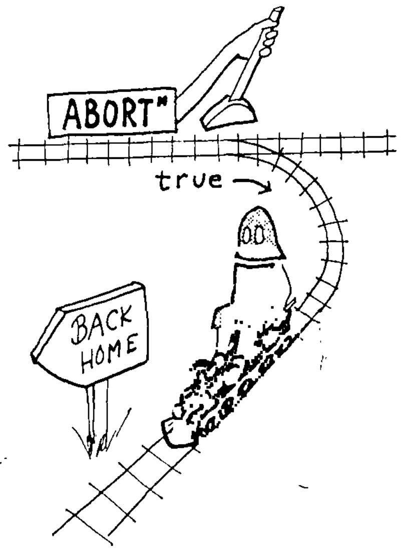
```
    : /CHECK   DUP IF / ELSE DROP THEN ;
```

can be shortened to:

```
    : /CHECK   ?DUP IF / THEN ;
```

ABORT"

It may happen that somewhere in a complex application an error might occur (such as division by zero) way down in one of the low-level words. When this happens you don't just want the computer to keep on going, and you also don't want it to leave anything on the stack.



If you think such an error might occur, you can use the word ABORT". ABORT" expects a flag on the stack: a "true" flag tells it to "abort," which in turn clears the stack and returns execution to the terminal, waiting for someone to type something. ABORT" also prints the name of the last interpreted word, as well as whatever message you want.†

Let's illustrate. We hope you're not sick of /CHECK by now, because here is yet another version:

```
    : /CHECK   DUP 0=  ABORT" ZERO DENOMINATOR " / ;
```

---

†FORTH-79 Standard

The Standard includes the word ABORT, which differs from ABORT" only in that it does not issue an error message.

In this version, if the denominator is zero, any numbers that happen to be on the stack will be dropped and the terminal will show:

```
    8 0 /CHECK /CHECK ZERO DENOMINATOR
```

Just as an experiment, try putting /CHECK inside another definition:

```
    : ENVELOPE   /CHECK  ." THE ANSWER IS " . ;
```

and try

```
    8 4 ENVELOPE THE ANSWER IS 2 ok
    8 0 ENVELOPE ENVELOPE ZERO DENOMINATOR
```

The point is that when /CHECK aborts, the rest of ENVELOPE is skipped. Also notice that the name ENVELOPE, not /CHECK, is printed.

A useful word to use in conjunction with ABORT" is ?STACK, which checks for stack underflow and returns a true flag if it finds it. Thus the phrase:

```
    ?STACK ABORT" STACK EMPTY "
```

aborts if the stack has underflowed.

FORTH uses the identical phrase, in fact. But it waits until all of your definitions have stopped executing before it performs the ?STACK test, because checking continuously throughout execution would needlessly slow down the computer.† You're free to insert a ?STACK ABORT" phrase at any critical or not-yet-tested portion of your application.

† For Computer Philosophers

FORTH provides certain error checking automatically. But because the FORTH operating system is so easy to modify, users can readily control the amount of error checking their system will do. This flexibility lets users make their own tradeoffs between convenience and execution speed.

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

| | | |
|---|---|---|
| IF xxx<br>ELSE yyy<br>THEN zzz | IF: (f -- ) | If f is true (non-zero) executes xxx; otherwise executes yyy; continues with zzz regardless. The phrase ELSE yyy is optional. |
| = | (nl n2 -- f) | Returns true if nl and n2 are equal. |
| - | (nl n2 -- n-diff) | Returns true (i.e., the non-zero difference) if nl and n2 are not equal. |
| < | (nl n2 -- f) | Returns true if nl is less than n2. |
| > | (nl n2 -- f) | Returns true if nl is greater than n2. |
| 0= | (n -- f) | Returns true if n is zero (i.e., reverses the truth value). |
| 0< | (n -- f) | Returns true if n is negative. |
| 0> | (n -- f) | Returns true if n is positive. |
| NOT | (f -- f) | Reverses the result of the previous test; equivalent to 0=. |
| AND | (nl n2 -- and) | Returns the logical AND. |
| OR | (nl n2 -- or) | Returns the logical OR. |
| ?DUP | (n -- n n) or<br>(0 -- 0) | Duplicates only if n is non-zero. |
| ABORT" xxx " | (f -- ) | If the flag is true, types out the last word interpreted, followed by the text. Also clears the user's stacks and returns control to the terminal. If false, takes no action. |
| ?STACK | ( -- f) | Returns true if a stack underflow condition has occurred. |

## Review of Terms

| | |
|---|---|
| Abort | as a general computer term, to abruptly cease execution if a condition occurs which the program is not designed to handle, in order to avoid producing nonsense or possibly doing damage. |
| "And" decision | two conditions that are combined such that if both of them are true, the result is true. |
| Branching | breaking the normally straightforward flow of execution, depending on conditions in effect at the time of exection. Branching allows the computer to respond differently to different conditions. |
| Comparison operator | in general, a command that compares one value with another (for example, determines whether one is greater than the other) and sets a flag accordingly, which normally will be checked by a conditional operator. In FORTH, a comparison operator leaves the flag on the stack. |
| Conditional operator | a word, such as IF, which routes the flow of execution differently depending on some condition (true or false). |
| Flag | as a general computer term, a value stored in memory which serves as a signal as to whether some known condition is true or false. Once the "flag is set," any number of routines in various parts of a program may check (or reset) the flag, as necessary. |
| Logic | in computer terminology, the system of representing conditions in the form of "logical variables," which can be either true or false, and combining these variables using such "logical operators" as "and," "or," and "not," to form statements which may be true or false. |
| Nesting | placing a branching structure within an outer branching structure. |
| "Or" decision | two conditions that are combined such that if either of them is true, the result is true. |

## Problems — Chapter 4

(answers in the back of the book)

1. What will the phrase

```
0= NOT
```

leave on the stack when the argument is

```
1?
0?
200?
```

2. Explain what an artichoke has to do with any of this.

3. Define a word called CARD which, given a person's age on the stack, prints out either of these two messages (depending on the relevant laws in your area):

```
ALCOHOLIC BEVERAGES PERMITTED     or
UNDER AGE
```

4. Define a word called SIGN.TEST that will test a number on the stack and print out one of three messages:

```
POSITIVE     or
ZERO     or
NEGATIVE
```

5. In Chap. 1, we defined a word called STARS in such a way that it always prints at least one star, even if you say

```
0 STARS * ok
```

Using the word STARS, define a new version of STARS that corrects this problem.

6. Write the definition for a word called WITHIN which expects three arguments:

```
(n low-limit hi-limit -- )
```

and leaves a "true" flag only if "n" is within the range

$$\text{low-limit} \le n < \text{hi-limit}$$

7. Here's a number-guessing game (which you may enjoy writing more than anyone will enjoy playing). First you secretly enter a number onto the stack (you can hide your number after entering it by executing the word PAGE, which clears the terminal screen). Then you ask another player to enter a guess followed by the word GUESS, as in

```
100 GUESS
```

The computer will either respond "TOO HIGH," "TOO LOW," or "CORRECT!" Write the definition of GUESS, making sure that the answer-number will stay on the stack through repeated guessing until the correct answer is guessed, after which the stack should be clear.

8. Using nested tests and $\boxed{\text{IF}}$...$\boxed{\text{ELSE}}$...$\boxed{\text{THEN}}$ statements, *write* a definition called SPELLER which will spell out a number that it finds on the stack, from -4 to 4. If the number is outside this range, it will print the message "OUT OF RANGE." For example:

```
2 SPELLER TWO ok
-4 SPELLER NEGATIVE FOUR ok
7 SPELLER OUT OF [illegible] ok
```

Make it as short as possible. (Hint: the FORTH word $\boxed{\text{ABS}}$ gives the absolute value of a number on the stack.)

9. Using your definition of WITHIN from Prob. 5, write another number-guessing game, called TRAP, in which you first enter a secret value, then a second player tries to home in on it by trapping it between two numbers, as in this dialogue:

```
0 1000 TRAP [illegible] ok
330 660 TRAP [illegible] ok
440 550 TRAP [illegible] ok
330 440 TRAP BETWE[illegible] ok
```

and so on, until the player guesses the answer:

```
391 391 TRAP YOU GOT IT! ok
```

Hint: you may have to modify the arguments to WITHIN so that TRAP does not say "BETWEEN" when only one argument is equal to the hidden value.