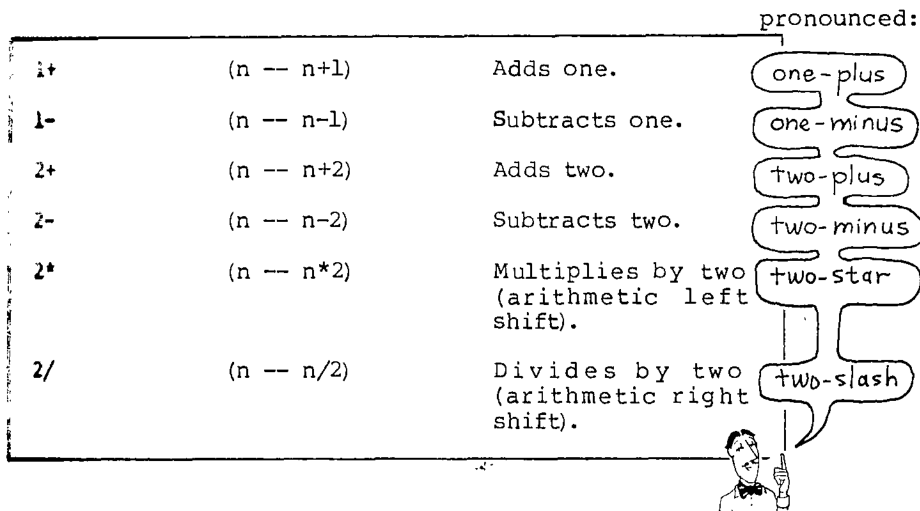
5 THE PHILOSOPHY OF FIXED POINT

In this chapter we'll introduce a new batch of arithmetic operators. Along the way we'll tackle the problem of handling decimal points using only whole-number arithmetic.

Quickie Operators

Let's start with the real easy stuff. You should have no trouble figuring out what the words in the following table do.†



			pronounced:
1+	(n -- n+1)	Adds one.	one-plus
1-	(n -- n-1)	Subtracts one.	one-minus
2+	(n -- n+2)	Adds two.	two-plus
2-	(n -- n-2)	Subtracts two.	two-minus
2*	(n -- n*2)	Multiplies by two (arithmetic left shift).	two-star
2/	(n -- n/2)	Divides by two (arithmetic right shift).	two-slash

The reason they have been defined as words in your FORTH system is that they are used very frequently in most applications and even in the FORTH system itself.

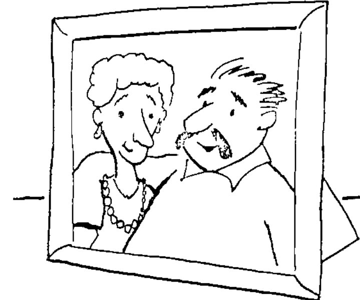
†For Beginners

We'll explain what "arithmetic left shift" is later on.

There are three reasons to use a word such as [1+], instead of one and [+], in your new definitions. First, you save a little dictionary space each time. Second, since such words have been specially defined in the "machine language" of each individual type of computer to take advantage of the computer's architecture, they execute faster than one and [+]. Finally, you save a little time during compilation.

Miscellaneous Math Operators

Here's a table of four miscellaneous math operators. Like the quickie operators, these functions should be obvious from their names.



Aunt Min and Uncle Max

ABS	(n -- |n|)	Returns the absolute value.
NEGATE	(n -- -n)	Changes the sign.
MIN	(n1 n2 -- n-min)	Returns the minimum.
MAX	(n1 n2 -- n-max)	Returns the maximum.

Here are two simple word problems, using [ABS] and [MIN]:

[ABS]

Write a definition which computes the difference between two numbers, regardless of the order in which the numbers are entered.

```
: DIFFERENCE   - ABS ;
```

This gives the same result whether we enter

```
52 37 DIFFERENCE . 15 ok    or
37 52 DIFFERENCE . 15 ok
```

Write a definition which computes the commission that furniture salespeople will receive if they've been promised $50 or 1/10 of the sale price, whichever is less, on each sale they make.

```
: COMMISSION   10 /  50 MIN ;
```

Three different values would produce these results:

```
600 COMMISSION . 50 ok
450 COMMISSION . 45 ok
50 COMMISSION . 5 ok
```

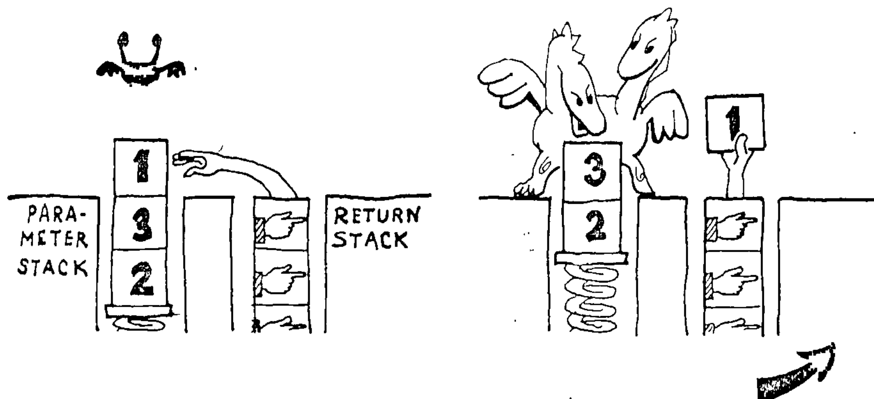
The Return Stack

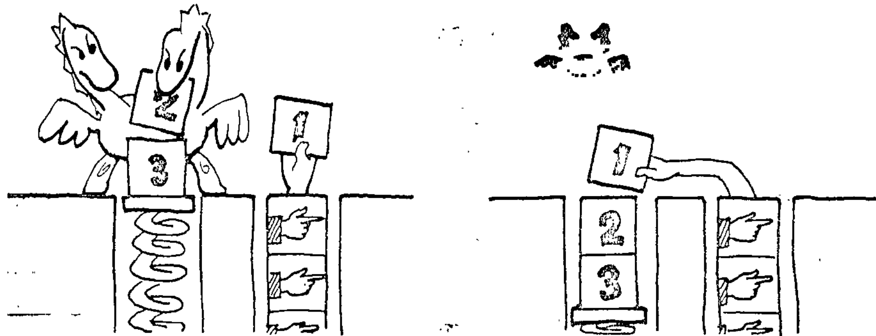
We mentioned before that there were still some stack manipulation operators we hadn't discussed yet. Now it's time.

Up till now we've been talking about "the stack" as if there were only one. But in fact there are two: the "parameter stack" and the "return stack." The parameter stack is used more often by FORTH programmers, so it's simply called "the stack" unless there is cause for doubt.

As you've seen, the parameter stack holds parameters (or "arguments") that are being passed from word to word. The return stack, however, holds any number of "pointers" which the FORTH system uses to make its merry way through the maze of words that are executing other words. We'll elaborate later on.

You the user can employ the return stack as a kind of "extra hand" to hold values temporarily while you perform operations on the parameter stack.





The return stack is a last-in first-out structure, just like the parameter stack, so it can hold many values. But here's the catch: whatever you put on the return stack you must remove again before you get to the end of the definition (the semicolon), because at that point the FORTH system will expect to find a pointer there. You cannot use the return stack to pass parameters from one word to another.

The following table lists the words associated with the return stack. Remember, the stack notation refers to the parameter stack.

Word	Stack	Description	Pronunciation
>R	(n --)	Takes a value off the parameter stack and pushes it onto the return stack.	to-R
R>	(-- n)	Takes a value off the return stack and pushes it onto the parameter stack.	R-from
I	(-- n)	Copies the top of the return stack without affecting it.	I
I'	(-- n)	Copies the second item of the return stack without affecting it.	I-prime
J	(-- n)	Copies the third item of the return stack without affecting it.	J

The words [>R] and [R>] transfer a value to and from the return stack, respectively. In the cartoon above, where the stack effect was:

```
(2 3 1 -- 3 2 1)
```

This is the phrase that did it:

```
>R SWAP R>
```

Each [>R] and its corresponding [R>] must be used together in the same definition or, if executed interactively, in the same line of input (before you hit the RETURN key).

The other three words--[I], [I'], and [J]--only copy values from the return stack without removing them. Thus the phrase:

```
>R SWAP I
```

would produce the same result as far as it goes, but unless you clean up your trash† before the next semicolon (or return key), you will crash the system.

To see how [>R], [R>], and [I] might be used, imagine you are so unlucky as to need to solve the equation:

$$ax^2 + bx + c$$

with all four values on the stack in the following order:

```
(a b c x -- )
```

(remember to factor out first).

†You might call such an error in your program a "litter bug."

Operator	Parameter Stack	Return Stack
	a b c x	
>R	a b c	x
SWAP ROT	c b a	x
I	c b a x	x
*	c b ax	x
+	c (ax + b)	x
R> *	c x(ax+b)	
+	x(ax+b)+c	

Go ahead and try it. Load the following definition:

```
: QUADRATIC   ( a b c x -- n)
     >R SWAP ROT I *  + R> *  + ;
```

Now test it:

```
2 7 9 3 QUADRATIC 48 ok
```

One more note (it's a little off the subject, but this is the first chance we've had to note it): you have now learned two different words with the name [I] (remember the EDITOR's "insert" word?). The reason the same name can refer to two separate definitions, depending on the context, is that the words are in different vocabularies.

We briefly mentioned earlier that the EDITOR is a vocabulary. You can get into the EDITOR vocabulary automatically by using certain EDITOR commands, such as [T]. Another vocabulary is called FORTH, which contains all the other predefined words we've covered so far. You can get back into the FORTH vocabulary by starting to compile a new definition (that is, when the interpreter sees the word [:]).

We mention all this now simply to amaze and impress you. The real discussion of vocabularies comes in a future chapter.

An Introduction to Floating-Point Arithmetic

There are many controversies surrounding FORTH. Certain principles which FORTH programmers adhere to religiously are considered foolhardy by the proponents of more traditional languages. One such controversy is the question of "fixed-point representation" versus "floating-point representation."

If you already understand these terms, skip ahead to the next section, where we'll express our views on the controversy. If you're a beginner, you may appreciate the following explanation.

First, what does floating point mean? Take a pocket calculator, for example. Here's what the display looks like after each entry:

You enter:	Display reads:
1 . 5 0 x	1.5
2 . 2 3	2.23
=	3.345

The decimal point "floats" across the display as necessary. This is called a "floating point display."

"Floating point representation" is a way to store numbers in computer memory using a form of scientific notation. In scientific notation, twelve million is written:

12×10^6

since ten to the sixth power equals one million. In many computers twelve million could be stored as two numbers: 12 and 6, where it is understood that 6 is the power of ten to be multiplied by 12, while 3.345 could be stored as 3345 and -3.

The idea of floating-point representation is that the computer can represent an enormous range of numbers, from atomic to astronomic, with two relatively small numbers.

What is fixed-point representation? It is simply the method of storing numbers in memory without storing the positions of each number's decimal point. For example, in working with dollars and cents, all values can be stored in cents. The _program_, rather than each individual _number_, can remember the location of the decimal point.

For example, let's compare fixed-point and floating-point representations of dollars-and-cents values.

Real-world Value	Fixed-point Representation	Floating-point Representation
1.23	123	123(-2)
10.98	1098	1098(-2)
100.00	10000	1(2)
58.60	5860	586(-1)

As you can see, with fixed-point all the values must conform to the same "scale." The decimal points must be properly "aligned" (in this case two places in from the right) even though they are not actually represented. With fixed-point, the computer treats all the numbers as through they were integers. If the program needs to print out an answer, however, it simply inserts the decimal point two places in from the right before it sends the number to the terminal or to the printer.

Why FORTH Programmers Advocate Fixed-Point

Many respectable languages and many distinguished programmers use floating-point arithmetic as a matter of course. Their opinion might be expressed like this: "Why should I have to worry about moving decimal points around? That's what computers are for."

That's a valid question--in fact it expresses the most significant advantage to floating-point implementation. For translating a mathematical equation into program code, having a floating-point language makes the programmer's life easier.

The typical FORTH programmer, however, perceives the role of a computer differently. A FORTH programmer is most interested in maximizing the efficiency of the machine. That means he or she wants to make the program run as fast as possible and require as little computer memory as possible.

To a FORTH programmer, if a problem is worth doing on a computer at all, it is worth doing on a computer well. The philosophy is, "If you just want a quick answer to a few calculations, you might as well use a hand-held calculator." You won't care if the calculator takes half a second to display the result. But if you have invested in a computer, you probably have to repeat the same set of calculations over and over and over again. Fixed-point arithmetic will give you the speed you need.

Is the extra speed that noticeable? Yes, it is. A floating-point multiplication or division can take three times as long as its equivalent fixed-point calculation. The difference is really noticeable in programs which have to do a lot of calculations

before sending results to a terminal or taking some action.† Most mini- and microcomputers don't "think" in floating-point; you pay a heavy penalty for making them act as though they do.

Here are some of the reasons you might prefer to have floating-point capability.

1. You want to use your computer like a calculator on floating-point data.

2. You value the initial programming time more highly than the execution time spent every time the calculation is performed.

3. You want a number to be able to describe a very large dynamic range (greater than -2 billion to +2 billion).

4. Your system includes a discrete hardware floating-point multiply (a separate "chip" whose only job is to perform floating-point multiplication at super high speeds).

---

†For Experts

Many professional FORTH programmers who have been writing complex applications for years have never had to use floating-point. And their applications often involve solutions of differential equations, Fast Fourier Transforms, non-linear least squares fitting, linear regression, etc. Problems that traditionally required a main-frame have been done on slower minicomputers and microprocessors, in some cases with an overall _increase_ in computation rate.

Most problems with physical inputs and outputs, including weather modeling, image reconstruction, automated electrical measurements, and the like all involve input and output variables that inherently have a dynamic range of no more than a few thousand to one, and thus fit comfortably into a 16-bit integer word. Intermediate calculation steps (such as summation) can be handled by the judicious use of scaling and double-length integers where required. For example, one common calculation step might involve multiplying each data point by a parameter (or by itself) and summing the result. In fixed point, this would be a 16 x 16-bit multiply and 32-bit summation. In floating-point, numbers are likely stored as 24-bit mantissa and 8-bit exponents. The 24-bit multiply will take about 1.5 times longer and the 32-bit addition 3-10 times longer than in fixed point. There is also the overhead of floating all the input data and fixing all the output data, approximately equal to one floating-point addition each. When these operations are performed thousands or millions of times, the overall saving by remaining in integer form is enormous.

All of these are valid reasons. Even Charles Moore, perhaps the staunchest advocate of simplicity in the programming community, has occasionally employed floating-point routines when the hardware supported it. Other FORTH programmers have written floating-point routines for their mini- and microcomputers. But the mainstream FORTH philosophy remains: "In most cases, you don't need to pay for floating-point."

FORTH backs its philosophy by supplying the programmer with a unique set of high-level commands called "scaling operators." We'll introduce the first of these commands in the next section. (The final example in Chap. 12 illustrates the use of scaling techniques.)

Star-slash the Scalar

Here's a math operator that is as useful as it is unusual: */.

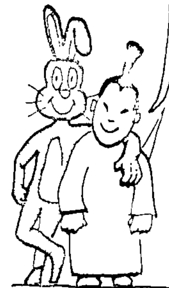
*/	(n1 n2 n3 -- n-result)	Multiplies, then divides (n1*n2/n3). Uses a 32-bit intermediate result.

As its name implies, */ performs multiplication, then division. For example, let's say that the stack contains these three numbers:

```
(225 32 100 -- )
```

*/ will first multiply 225 by 32, then divide the result by 100.

This operator is particularly useful as an integer-arithmetic solution to problems such as percentage calculations.



For example, you could define the word % like this:

```
: %   100 */ ;
```

so that by entering the number 225 and then the phrase:

```
32 %
```

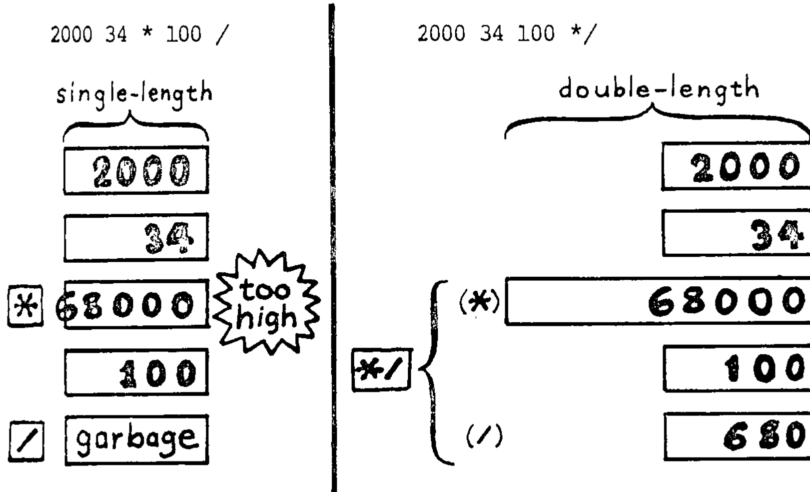
you'd end up with 32% of 225 (that is, 72) on the stack.†

*/ is not just a * and a / thrown together, though. It uses a "double-length intermediate result." What does that mean, you ask?

Say you want to compute 34% of 2000. Remember that single-precision operators, like * and /, only work with arguments and results within the range of -32768 to +32767. If you were to enter the phrase:

```
2000 34 *  100 /
```

you'd get an incorrect result, because the "intermediate result" (in this case, the result of multiplication) exceeds 32767, as shown in the left column in this pictorial simulation.



But */ uses a double-length intermediate result, so that its range will be large enough to hold the result of any two single-length numbers multiplied together. The phrase:

```
2000 34 100 */
```

returns the correct answer because the end result falls within the range of single-length numbers.

---

†For the curious

The method of first multiplying two integers, then dividing by 100 is identical to the approach most people take in solving such problems on paper.

225
.32
450
675
72.00

The previous example brings up another question: how to round off.

Let's assume that this is the problem:

> If 32% of the students eating at the school cafeteria usually buy bananas, how many bananas should be on hand for a crowd of 225? Naturally, we are only interested in whole bananas, so we'd like to round off any decimal remainder.

As our definition now stands, any value to the right of the decimal is simply dropped. In other words, the result is "truncated."

32% of:	Result:
225 = 72.00	72 -- exactly correct
226 = 72.32	72 -- correct, rounded down (truncated)
227 = 72.64	72 -- truncated, not rounded.

There is a way, however, with any decimal value of .5 or higher, to round upwards to the next whole banana. We could define the word R%, for "rounded percent," like this:

```
: R%   10 */  5 +  10 / ;
```

so that the phrase:

```
227 32 R% .
```

will give you 73, which is correctly rounded up.

Notice that we first divide by 10 rather than 100. This gives us an extra decimal place to work with, to which we can add five:

Operation	Stack Contents
	227 32 10
*/	726
5 +	731
10 /	73

The final division by ten sets the value to its rightful decimal position. Try it and see.†

A disadvantage to this method of rounding is that you lose one decimal place of range in the final result; that is, it can only go as high as 3,276 rather than 32,767. But if that's a problem, you can always use double-length numbers, which we'll introduce later, and still be able to round.

Some Perspective on Scaling

Let's back up for a minute. Take the simple problem of computing two-thirds of 171. Basically, there are two ways to go about it.

1. We could compute the value of the fraction 2/3 by dividing 2 by 3 to obtain the repeating decimal .666666, etc. Then we could multiply this value by 171. The result would be 113.9999999, etc., which is not quite right but which could be rounded up to 114.

2. We could multiply 171 by 2 to get 342. Then we could divide this by 3 to get 114.

Notice that the second way is simpler and more accurate.

Most computer languages support the first way. "You can't have a fraction like two-thirds hanging around inside a computer," it is believed, "you must express it as .666666, etc."

FORTH supports the second way. [*/] lets you have a fraction like two-thirds, as in:

```
171 2 3 */
```

Now that we have a little perspective, let's take a slightly more complicated example:

---

†For Experts

An even faster definition:

```
: R%   50 /  1+ 2/ ;
```

We want to distribute $150 in proportion to two values:†

```
 7,105      ?
 5,145      ?
12,       150
```

Again, we could solve the problem this way:

(7,105 / 12,250) x 150

and

(5,145 / 12,250) x 150

but for greater accuracy; we should say:

(7,105 x 150) / 12,250

and

(5,145 x 150) / 12,250

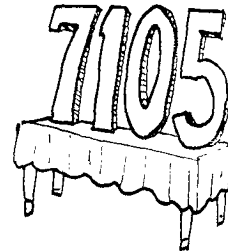
which in FORTH is written:

```
7105 150 12250 */ . 87 ok
```

then

```
5145 150 12250 */ . 63 ok
```

It can be said that the values 87 and 63 are "scaled" to 7105 and 5145. Calculating percentages, as we did earlier, is also a form of scaling. For this reason, `*/` is called a "scaling operator."



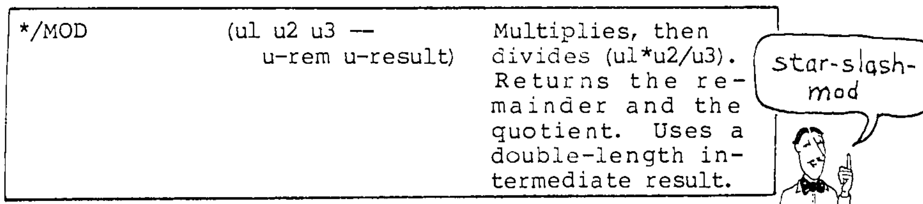
---

†For Beginners Who Like Word-problems

Here's a word-problem for the above example:

The boss says he'll divide a $150 bonus between the two top-selling marketing representatives according to their monthly commissions. When the receipts are counted, the top two commissions are $7,105 and $5,145. How much of the bonus does each marketing rep get?

Another scaling operator in FORTH is */MOD:



*/MOD	(u1 u2 u3 — u-rem u-result)	Multiplies, then divides (u1*u2/u3). Returns the remainder and the quotient. Uses a double-length intermediate result.

We'll let you dream up a good example for */MOD yourself.

Using Rational Approximations†

So far we've only used scaling operations to work on rational numbers. They can also be used on rational approximations of irrational constants, such as pi or the square root of two. For example, the real value of pi is

3.14159265358, etc.

but to stay within the bounds of single-length arithmetic, we could write the phrase:

```
31416 10000 */
```

and get a pretty good approximation.

Now we can write a definition to compute the area of a circle, given its radius. We'll translate the formula:

πr^2

into FORTH. The value of the radius will be on the stack, so we DUP it and multiply it by itself, then star-slash the result:

---

†For Math-block Victims:

You can skip this section if it starts making your brain itch. But if you're feeling particularly smart today, we want you to know that ...

A rational number is a whole number or a fraction in which the numerator and denominator are both whole numbers. Seventeen is a rational number, as is 2/3. Even 1.02 is rational, because it's the same as 102/100. $\sqrt{2}$, on the other hand, is irrational.

```
: PI   DUP *  31416 10000 */ ;
```

Try it with a circle whose radius is ten inches:

```
10  PI  . 314 ok
```

But for even more accuracy, we might wonder if there is a pair of integers besides 31416 and 10000 that is a closer approximation to pi. Surprisingly, there is. The fraction:

```
355 113 */
```

is accurate to more than six places beyond the decimal, as opposed to less than four places with 31416.

Our new and improved definition, then, is:

```
: PI   DUP *  355 113 */ ;
```

It turns out that you can approximate nearly any constant by many different pairs of integers, all numbers less than 32768, with an error of less than 10^{-8}.†

---

†For Really Dedicated Mathephiles

Here's a handy table of rational approximations to various constants:

Number	Approximation	Error
π = 3.141 ...	355/ 113	8.5×10^{-8}
$\sqrt{2}$ = 1.414 ...	19601/13860	1.5×10^{-9}
$\sqrt{3}$ = 1.732 ...	18817/10864	1.1×10^{-9}
e = 2.718 ...	28667/10546	5.5×10^{-9}
$\sqrt{10}$ = 3.162 ...	22936/ 7253	5.7×10^{-9}
$\sqrt[12]{2}$ = 1.059 ...	26797/25293	1.0×10^{-9}
$\log_{10} 2/1.6384$ = 0.183 ...	2040/11103	1.1×10^{-8}
ln2/16.384 = 0.042 ...	485/11464	1.0×10^{-7}
.001° /22-bit rev = 0.858 ...	18118/21109	1.4×10^{-9}
arc-sec/22-bit rev = 0.309 ...	9118/29509	1.0×10^{-9}
c = 2.9979248	24559/ 8192	1.6×10^{-9}

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

1+	(n -- n+1)	Adds one.
1-	(n -- n-1)	Subtracts one.
2+	(n -- n+2)	Adds two.
2-	(n -- n-2)	Subtracts two.
2*	(n -- n*2)	Multiplies by two (arithmetic left shift)
2/	(n -- n/2)	Divides by two (arithmetic right shift)
ABS	(n -- |n|)	Returns the absolute value.
NEGATE	(n -- -n)	Changes the sign.
MIN	(n1 n2 -- n-min)	Returns the minimum.
MAX	(n1 n2 -- n-max)	Returns the maximum.
>R	(n --)	Takes a value off the parameter stack and pushes it onto the return stack.
R>	(-- n)	Takes a value off the return stack and pushes it onto the parameter stack.
I	(-- n)	Copies the <u>top</u> of the return stack without affecting it.
I'	(-- n)	Copies the <u>second</u> item of the return stack without affecting it.
J	(-- n)	Copies the <u>third</u> item of the return stack without affecting it.
*/	(n1 n2 n3 -- n-result)	Multiplies, then divides (u1*n2/n3). Uses a 32-bit intermediate result.
*/MOD	(u1 u2 u3 -- u-rem u-result)	Multiplies, then divides (u1*u2/u3). Returns the remainder and the quotient. Uses a double-length intermediate result.

Review of Terms

Double-length intermediate result — a double-length value which is created temporarily by a two-part operator, such as [*/], so that the "intermediate result" (the result of the first operation) is allowed to exceed the range of a single-length number, even though the initial arguments and the final result are not.

Fixed-point arithmetic — arithmetic which deals with numbers which do not themselves indicate the location of their decimal points. Instead, for any group of numbers, the program assumes the location of the decimal point or keeps the decimal location for all such numbers as a separate number.

Floating-point arithmetic — arithmetic which deals with numbers which themselves indicate the location of their decimal points. The program must be able to interpret the true value of each individual number before any arithmetic can be performed.

Parameter Stack — in FORTH, the region of memory which serves as common ground between various operations to pass arguments (numbers, flags, or whatever) from one operation to another.

Return stack — in FORTH, a region of memory distinct from the parameter stack which the FORTH system uses to hold "return addresses" (to be discussed in Chap. 9), among other things. The user may keep values on the return stack temporarily, under certain conditions.

Scaling — the process of multiplying (or dividing) a number by a ratio. Also refers to the process of multiplying (or dividing) a number by a power of ten so that all values in a set of data may be represented as integers with the decimal point assumed to be in the same place for all values.

Problems — Chapter 5

1. Translate the following algebraic expression into a FORTH definition:

   $$-\frac{ab}{c}$$

   given (a b c --)

2. Given these four numbers on the stack:

   (6 70 123 45 --)

   write an expression that prints the largest value.

Practice in Scaling

3. In "calculator style," convert the following temperatures, using these formulas:

   $$^{\circ}C = \frac{^{\circ}F - 32}{1.8}$$

   $$^{\circ}F = (^{\circ}C \times 1.8) + 32$$

   $$^{\circ}K = {}^{\circ}C + 273$$

   (For now, express all arguments and results in whole degrees.)

   a) 0° F in Centigrade
   b) 212° F in Centigrade
   c) -32° F in Centigrade
   d) 16° C in Fahrenheit
   e) 233° K in Centigrade

4. Now define words to perform the conversions in Prob. 3. Use the following names:

   F>C F>K C>F C>K K>F K>C

   Test them with the above values.