Starting

FORTH

With a foreword by Charles H. Moore
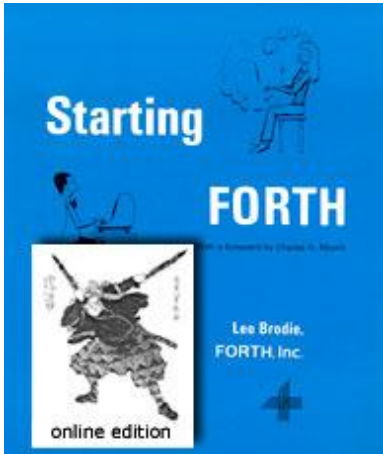
Leo Brodie,
FORTH, Inc.

TABLE OF CONTENTS

Code examples run on iForth and SwiftForth.*

* Download the
[free trial system](#).

Try the links to go to the original [FORTH INC](#) website

[About the online edition](#)

OTHER FORTH BOOKS -Textbook Teaches Forth Problem-Solving Methods

• [Forth Application Techniques](#) uses realistic examples and exercises to teach Forth approaches to solving typical tasks and programming challenges.

Today's Definitive Forth Reference Book

• [Forth Programmer's Handbook](#) (3rd ed.) is today's de facto, desktop reference book about Forth.

Learn Forth Faster

Our [Forth classes](#) will help you to accelerate through the learning curve.

Starting FORTH by Leo Brodie

Starting Forth, the classic Forth language tutorial, is available here in its official online edition.
Updates to the original Starting FORTH text include code examples that run on iForth and [SwiftForth](#)
systems. Aficionados (and detractors) of the original book's illustrations will find most of them restored in
this official edition by FORTH, Inc.

"…I hope this book is not so easy and enjoyable that it seems trivial. Be warned that there is heavy
content here and that you can learn much about computers and compilers as well as about programming."

—[Charles Moore](#), Forth's inventor

"A Forth to be reckoned with..."

—[Leo Brodie](#), Author

About the book

Starting FORTH has been the classic Forth tutorial and textbook since its first release. Many experienced
programmers have commented on its concise utility and completeness. Beginners will find a carefully
planned introduction to the Forth programming language that will prepare them for other books like [Forth
Application Techniques](#) and [Forth Programmer's Handbook](#).

FORTH, Inc. and the book's author, Leo Brodie, thank the global community for its continued enthusiasm
for this book. Many have sought a new copy, a used copy, or have asked for a reprinting. It is our pleasure
to now present this online edition, with great appreciation for work done by Marcel Hendrix to generate a
web version on which these pages were based.

Notes about this online edition

Changes made by FORTH, Inc. to Marcel's version were primarily related to internet coding standards
and visual presentation. We also restored the book's original illustrations here, in most cases. The
following notes are also relevant and important to those familiar with the original edition.

Notes by Marcel Hendrix

This transcript is not an exact reproduction. Forth code has been ANSified. All code should run on, at
least iForth. Where necessary, statements that were valid in 1981 have been exchanged with statements
more appropriate for 2003 (when this tribute was written)….

I have assumed a 32-bit, byte-addressing Forth with 8-bit characters. The address returned by WORD is
assumed to be HERE. This allows the common trick of ALLOTing length of *str* CHARS after using
WORD in order to compile string *str* to memory. Multitasking issues are ignored (e.g. no >TYPE, just
TYPE). Division is symmetric, not floored, and two's complement is assumed throughout. iForth works
splendidly with it, but other Forths will work too. Chapter 7 exploits extended uses of number conversion.
Most Forths are broken in this respect, but iForth and SwiftForth support these neat features.

Other Forth Books and Classes

As noted above, once you have progressed from the beginning stages and look for more-practical and
detailed information, several resources are available. The Forth books [Forth Application Techniques](#) and
[Forth Programmer's Handbook](#) are a good next step. In addition, [Forth classes and training](#) are available
at our offices or, by arrangement, at your facility.

Thank you again for your interest in this fascinating and powerful programming language and in Starting
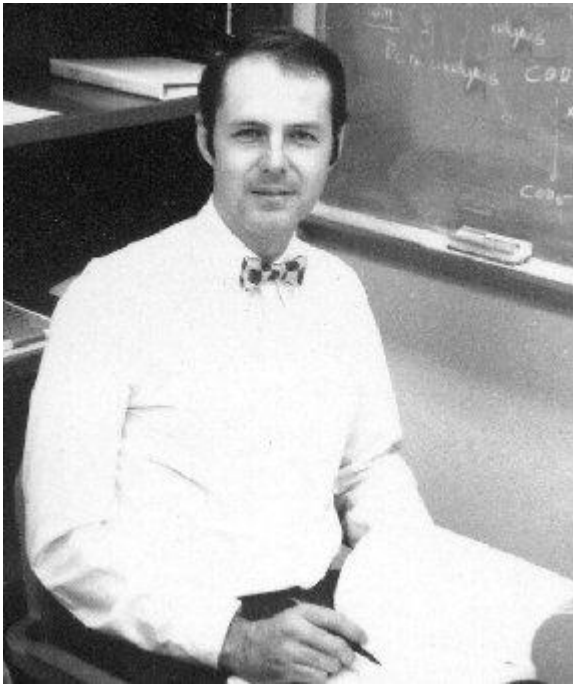Forth.

0. Introductions (1st edition)

About the Author

Leo Brodie's inability to express even the most complex technical concepts without adding a twist of humor comes from an early love of comedy. He specialized in playwriting at UCLA and has had several comedies produced there and in local theater. He has also written freelance magazine articles and has worked as a copywriter for an ad agency. When a company he was working for installed a computer, he became inspired to try designing a microprocessor-based toy. Although he never got the toy running, he learned a lot about computers and programming. He now works at FORTH, Inc. as a technical and marketing writer, where he can play on the computers as the muse determines without having to be a fanatical computer jockey, and is allowed to write books such as this.

Leo's other interests include singing, driving classic Volvos, and dancing to 50's music.

Foreword

The Forth community can celebrate a significant event with the publication of *Starting Forth*. A greater effort, talent, and commitment have gone into this book than into any previous introductory manual. I, particularly, am pleased at this evidence of the growing popularity of Forth, the language.

I developed Forth over the period of some years as an interface between me and the computers I programmed. The traditional languages were not providing the power, ease, or flexibility that I wanted. I disregarded much conventional wisdom in order to include exactly the capabilities needed by a productive programmer. The most important of these is the ability to add whatever capabilities later become necessary.

The first time I combined the ideas I had been developing into a single entity, I was working on an IBM 1130, a "third-generation" computer. The result seemed so powerful that I considered it a "fourth-generation computer language." I would have called it FOURTH, except that the 1130 permitted only five-character identifiers. So FOURTH became FORTH, a nicer play on words anyway.

One principle that guided the evolution of Forth, and continues to guide its application, is bluntly: Keep It Simple. A simple solution has elegance. It is the result of exacting effort to understand the *real* problem and is recognized by its compelling sense of rightness. I stress this point because it contradicts the conventional view that power increases with complexity. Simplicity provides confidence, reliability, compactness, and speed.

Starting Forth was written and illustrated by Leo Brodie, a remarkably capable person whose insight and imagination will become apparent. This book is an original and detailed prescription for learning. It deftly guides the novice over the thresholds of understanding that all Forth programmers must cross.

Although I am the only person who has never had to learn Forth, I do know that its study is a formidable one. As with a human language, the usage of many words must be memorized. For beginners, Leo's droll comments and superbly cast characters appear to make this study easy and enjoyable. For those like myself who already know Forth, a quick reading provides a delightful trip and fresh views of familiar terrain. But I hope this book is not so easy and enjoyable that it seems trivial. Be warned that there is heavy content here and that you can learn much about computers and compilers as well as about programming.

Forth provides a natural means of communication between man and the smart machines he is surrounding himself with. This requires that it share characteristics of human languages, including compactness, versatility, and extensibility. I cannot imagine a better language for writing programs, expressing algorithms, or understanding computers. As you read this book, I hope that you may come to agree.

Charles H. Moore
Inventor of Forth

About This Book

Welcome to *Starting Forth*, your introduction to an exciting and powerful computer language called Forth.

If you're a beginner who wants to learn more about computers, Forth is a great way to learn. Forth is more fun to write programs with than any language that I know of. (See the "Introduction for Beginners."[1])

If you are a seasoned professional who wants to learn Forth, this book is just what you need. Forth is a very different approach to computers, so different that everyone, from newcomers to old hands, learns Forth best from the ground up. If you're adept at other computer languages, put them out of your mind for now, and remember only what you know about *computers*. (See the "Introduction for Professionals.")

Since many people with different backgrounds are interested in Forth, I've arranged this book so that you'll only have to read what you need to know, with footnotes addressed to different kinds of readers. The first half of Chap. 7 provides a background to computer arithmetic for beginners only.

This book explains how to write simple applications in Forth. It includes all standard Forth words[2] that you need to write a high-level single-task application. This word set is an extremely powerful one, including everything from simple math operators to compiler-controlling words.

Excluded from this book are all commands that are related to the assembler, target compiler and other specialized utilities. These commands are available on some versions of Forth such as eForth and most commercial implementations.

I've chosen examples that will actually work on a Forth system with a terminal and a disk. Don't infer from this that Forth is limited to batch or string-handling tasks, since there is really no limit to Forth's usefulness.

Here are some features of this book that will make it easy to use:

- All commands are listed twice: first, in the section in which the word is introduced, and second, in the summary at the end of that chapter.
- Each chapter also has a review of terms and a set of exercise problems, with answers.
- Several "Handy Hints" have been included to reveal procedural tips or optional routines that are useful for learners but that don't merit an explanation as to how or why they work.

A personal note: Forth is a very unusual language. It violates many cardinal rules of programming. My first reaction to Forth was extremely skeptical, but as I tried to develop complicated applications I began to see its beauty and power. You owe it to yourself to keep an open mind while reading about some of its peculiarities. I'll warn you now: few programmers who learn Forth ever go back to other languages.

Good luck, and enjoy learning!

Leo Brodie
FORTH, Inc.

Acknowledgments

Introductions

Introduction for Beginners: What is a Computer Language?

At first when beginners hear the term "computer language," they wonder, "What kind of language could a computer possibly speak? It must be awfully hard for people to understand. It probably looks like:

```
976#!@NX714&+
```

if it looks like anything at all."

Actually a computer language should not be difficult to understand. Its purpose is simply to serve as a convenient compromise for communication between a person and a computer.

Consider the marionette. You can make a marionette "walk" simply by working the wooden control, without even touching the strings. You could say that rocking the control means "walking" in the language of the marionette. The puppeteer guides the marionette in a way that the marionette can understand and that the puppeteer can easily master.

Computers are machines just like the marionette. They must be told exactly what to do, in specific language. And so we need a language which possesses two seemingly opposite traits:

On the one hand, it must be precise in its meaning to the computer, conveying all the information that the computer needs to know to perform the operation. On the other hand, it must be simple and easy-to-use by the programmer.

Many languages have been developed since the birth of computers: Fortran is the elder statesman of the field; COBOL is still the standard language for data processing; BASIC was designed as a beginner's language along the road toward languages like Fortran and COBOL; C and Java are the general purpose application languages of the 90's. This book is about a very different kind of language: Forth. Forth's popularity has kept constant over the past several years, and its popularity is shared among programmers in all fields.

All the languages mentioned above, including Forth, are called "high-level" languages. It's important for beginners to recognize the difference between a high-level language and the computer it runs on. A high-level language looks the same to a programmer regardless of which make or model of computer it's running on. But each make or model has its own internal language, or "machine language." To explain what a machine language is, let's return to the marionette.

Imagine that there is no wooden control and that the puppeteer has to deal directly with the strings. Each string corresponds to exactly one part of the marionette's body. The harmonious combinations of movements of the individual strings could be called the marionette's "machine language."

Now tie the strings to a control. The control is like a high-level language. With a simple turn of the wrist, the puppeteer can move many strings simultaneously.
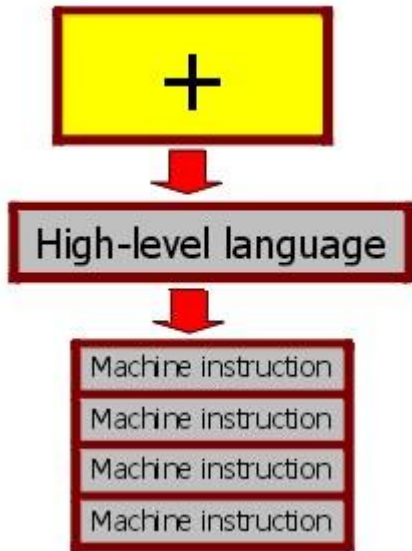
So it is with a high-level computer language, where the simple and familiar symbol "+" causes many internal functions to be performed in the process of addition.

Here's a clever thing about a computer: it can be programmed to translate high-level symbols (such as "+") into the computer's own machine language. Then it can proceed to carry out the machine instructions. A high-level language is a computer program that translates humanly understandable words and symbols into the machine language of the particular make and model of computer.

What's the difference between Forth and other high-level languages? To put it very briefly: it has to do with the compromise between man and computer. A language should be designed for the convenience of its human users, but at the same time for compatibility with the operation of the computer.

Forth is unique among languages because its solution to this problem is unique. This book will explain how.

Introduction for Professionals: Forth in the Real World

Forth enjoyed a rising tide of popularity up to around 1994,[3] perhaps most visibly among enthusiasts and hobbyists. After 1996 or so Forth's popularity has stayed relatively constant. But this development is only a new wrinkle in the history of Forth. Forth has been in use from 1972 on, in critical scientific and industrial applications. In fact, if you use a mini- or microcomputer professionally, chances are that Forth can run your application — more efficiently than the language you're presently using.

Now you'll probably ask rhetorically, "If Forth is so efficient, how come I'm not using it?" The answer is that you, like most people, don't know what Forth is.

To really get an understanding of Forth, you should read this book and, if possible, find a Forth system and try it for yourself. For those of you who are still at the bookstore browsing, however, this section will answer two questions: "What is Forth?" and "What is it *good* for?"

Forth is many things:

- a high-level language
- an assembly language
- an operating system
- a set of development tools
- a software design philosophy

As a language, Forth begins with a powerful set of standard commands, then provides the mechanics by which you can define your own commands. The structural process of building definitions upon previous definitions is Forth's equivalent of high-level coding. Alternatively, words may be defined directly in assembler mnemonics, using Forth's assembler. All commands are interpreted by the same interpreter and compiled by the same compiler, giving the language extreme flexibility.

The highest level of your code will resemble an English-language description of your application. Forth has been called a "meta-application language" — a language that you can use to create problem-oriented languages.

As an operating system, Forth does everything that traditional operating systems do, including interpretation, compilation, assembling, virtual memory handling, I/O, text editing, etc.

But because the Forth operating system is much simpler than its traditional counterparts due to Forth's design, it runs much more quickly, much more conveniently, and in much less memory.

What is Forth good for? Forth offers a simple means to maximize a processor's efficiency. For example:

Forth is fast. *High-level* Forth executes as fast as other high-level languages and between 20 to 75% slower than equivalent assembly-language programs, while time-critical code may be written in assembler to run at full processor speed. Without a traditional operating system, Forth eliminates redundancy and needless run-time error checking.

Forth compiled code is compact. Forth applications require less memory than their equivalent *assembly-language* programs and consume less power (important for hand-helds and portable gadgets!) Written in Forth, the entire operating system and its standard word set reside in less than 8K bytes. Support for a target application may require less than 1K bytes.

Forth is transportable. It has been implemented on just about every mini- and microcomputer known to the industry. Most microcontrollers and DSPs, even tiny ones, also have a Forth implementation.

Forth has been known to cut program development time by a factor of ten for equivalent assembly-language programming and by a factor of two for equivalent high-level programming in C or Java. Productivity increases because Forth epitomizes "structured programming" and because it is interactive and modular.

Here are a few samples of Forth in the real world:[4]
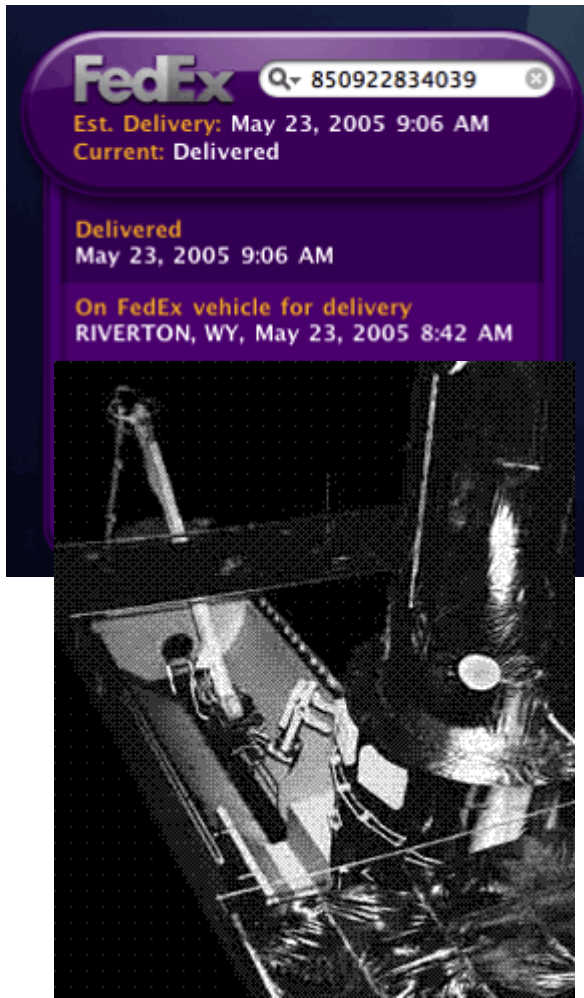
- AVCO/Textron Systems, building automation and auxiliary services for King Khaled International Airport (Saudi Arabia). System contains nine PDP 11/44s, 378 8086-based computers, and 320 8085-based security processors, collectively monitoring and controlling over 36,000 points.
- Eastman Kodak Company, quality control system monitoring photographic film density. Includes film motion control, automatic recognition of film density steps, and custom IEEE-488 bus interface.
- Federal Express, hand-held SuperTracker, carried by every FedEx delivery agent. Contains bar-code reader, keyboard, 2x20 line display. Performs extensive package entry and tracking functions, including cross index from airport code to all 10,000 US zip codes. Includes smart power-off sequencing to extend battery life.

- NASA Goddard Space Flight Center:

1. Control of 50-foot long, six-joint arm for Space Shuttle simulator. Extensive math routines convert two three-axis joystick commands into required joint velocities in six different co-ordinate systems.
2. Multitasking operating system, Forth language compiler, and libraries for UT69R000 radiation-hardened microprocessor used in Space Shuttle instrumentation.
3. Development of the Forth-based Small Payload Accommodations Interface Module (SPAIM), which interfaces the Shuttle Solar Backscatter Ultraviolet (SSBUV) instrument to the Space Shuttle's avionic systems. The SSBUV instrument is used to calibrate ozone-measuring instruments aboard NOAA satellites.

- Owens-Corning Fiberglas, Owens-Corning has used Forth for many years as the basic firmware in its distributed industrial controllers. These controllers perform a wide variety of functions, managing winders, weighing devices, etc., used in the manufacture of various fiberglass products. Plants in Korea and Mexico also use FORTH, Inc.'s EXPRESS to provide supervisory control and reporting functions.
- Saturn Corp, distributed HVAC system for entire Saturn automobile assembly plant, controlling over two hundred 40 hp. heating - cooling - humidifying units (with Z-80s) over a two-tier network using PCs as text and graphical system monitors. Outside air sensors provide inputs for intelligent set-point control and economic use of gas heating and chilled-water cooling systems.

- Sacramento Municipal Utilities District (California): photovoltaic arrays in the state capitol feature EXPRESS to provide user-configurable live trending, historical trending, alarm/exception reporting, rule-tracking, I/O system exerciser, class-based real-time database, graphical process displays, simultaneous multiple vendor I/O system scanning, I/O and process simulation for development, and multiple remote terminal access with full graphics. Custom drivers for the Digitronics Sixnet(TM) I/O system were provided in one week; EXPRESS already supports Modicon, Allen-Bradley, OPTO-22 OPTOMUX and PAMUX, plus others.

- University of Minnesota, PC-based system for telescope control and data taking (over IEEE-488 bus), data analysis and graphics display. Includes provision for remote observing, using a custom protocol to multiplex packets from three independent data streams over a single telephone line.
- VertexRSI (Div. of Tripoint Global), software for custom satellite tracking receivers. Includes frequency synthesizer control, remote RS-232 command port, vacuum fluorescent graphics display.
- A mobile phone manufacturer is introducing a new games engine derived from the SENDIT project. This uses a Forth-based virtual machine to reduce the size of games in the phone, and to permit more functionality to be provided in the phone without increasing memory size.
-

- A recent consultancy project based on MPE 8051 and ARM hardware, will introduce a new range of vending machines to the market.
- Construction Computer Software (CCS) in Cape Town produce the MARS and CANDY applications which are a standard all over the world. The CCS software is an example of a large-scale Windows application written in ProForth for Windows, and the VFX Forth version already consists of over 850,000 lines of code. CCS software was used to plan the new Chai Tak airport in Hong Kong. The CCS web site is here.

- Barefoot Auditor is used by Microsoft for collecting information about their own PCs, and was written using one of MPE's Forth systems. Barefoot Auditor has been available on several magazine cover disks recently, and more information is available from Pathfinder.

- Micross Electronics, use MPE's ProForth for Windows at the heart of their commercial laundry control systems, and MPE's Forth 6 cross compilers for the PLCs performing real time control. These systems are installed in many countries, and you may have slept in sheets washed by the Micross Tracknet control systems.

- Forth virtual machine runs payment terminals: Europay International's Open Terminal Architecture (OTA). OTA uses a virtual machine (VM) architecture to deliver payment terminal applications directly to payment terminals regardless of their hardware or CPU. The OTA VM has been installed on a range of CPUs and is now being deployed. The OTA project involved up to 30 programmers working in several locations on two continents. OTA is described here.

There's a catch we must admit. It is that Forth makes *you* responsible for your computer's efficiency. To draw an analogy: a manual transmission is tougher to master than an automatic, yet for many drivers it offers improved control over the vehicle.

Similarly, Forth is tougher to master than traditional high-level languages, which essentially resemble one another (i.e., after learning one, it is not difficult to learn another). Once mastered, however, Forth gives *you* the capability to minimize CPU time and memory space, as well as an organizing philosophy by which you can dramatically reduce project development time.

And remember, all of Forth's elements enjoy the same protocol, including operating system (sometimes), compiler, interpreters, text editor, virtual memory, assembler, and multiprogrammer. The learning curve for Forth is much shorter than that for all these separate elements added together.

If all of this sounds exciting to you, turn the page and start Forth.

1. 2007: also see the Wikipedia Forth entry

2. 2007: after the publication of Starting FORTH, the Forth language was formalized and widely implemented in conformance to the ANS Forth standard.

3. 2007: The ANS Forth Standard was finalized in 1994 and was adopted in near-identical form as the ISO Forth Standard.

4. 2007: Some applications of Forth cited in the original Starting FORTH are now outdated. More applications of Forth are described at FORTH, Inc., MPE and other web sites.

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.

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 do is say "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 in 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  FAUCETS OPEN  TILL-FULL  FAUCETS CLOSE ;
```

In this definition we are referring to *things* (faucets) as well as *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, a complete ANS Forth with all extensions includes 371 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."

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 that 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. 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 Forth on your particular computer.

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.

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, and 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 ↵ to mark the point where you must press the RETURN key. We also formatted the computer's output like this (even though the computer does not) to indicate who is typing what.

Here's what has happened:

```
   15 SPACES↵   ok
```

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

Now enter this:

```
   42 EMIT↵* ok
```

The phrase "42 EMIT" tells Forth to print an asterisk (we'll discuss this command later on in the book.) Here Forth printed an asterisk, then responded ok.

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

```
   15 SPACES  42 EMIT  42 EMIT↵     ** ok
```

This time Forth printed fifteen spaces and two asterisks. A note about entering words and/or numbers: we can separate them from 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 ;↵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 the definition from its contents with three spaces.

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

```
   STAR↵* 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 EMIT that is already defined. In this book, however, we will print those words that are already defined in this format, so that you can more easily tell the difference.

Another system-defined word is CR, which performs a carriage return and line feed at your terminal.⬚
For example, enter this:

```
CR↵ok
```

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

Now try this:

```
CR STAR CR STAR CR STAR↵*** ok
```

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

```
: MARGIN   CR 30 SPACES ;↵ok
```

Now we can enter

```
MARGIN STAR MARGIN STAR MARGIN STAR↵
```

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 ;↵ok
```

We will also need to print a 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 ;↵ok
```

Now we can say

```
5 STARS↵***** ok
```

or

```
35 STARS↵*********************************** 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 ;↵ok
```

Now we can enter

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

and get a letter "F" (for **F**orth) 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 ;↵ok
```

You've just seen an example of the way simple Forth commands can become a 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 ;
```
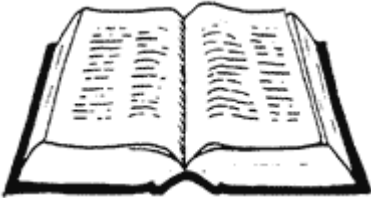

The Dictionary

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   [CHAR] * EMIT ;↵ok
```

the compiler compiles the new definition (it does the same as "42 EMIT" but doesn't use magic numbers) 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 SPACES↵
```
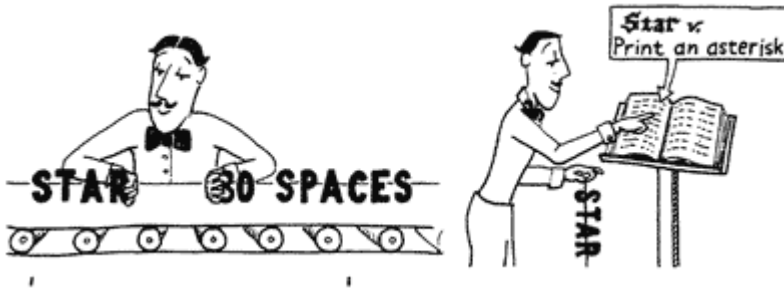
This will activate a word called INTERPRET, also known as the "text interpreter." The text interpreter scans the input stream, looking for strings of characters separated by spaces. When a string is found, it is looked up in the dictionary. If the word is in the dictionary, it is pointed out to a word called EXECUTE. EXECUTE executes the definition (in this case an asterisk is printed). Finally, 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↵XLERB ?

When the text interpreter cannot 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 (Some Forths print various error messages along with this.)

ANS Forth allows up to thirty-one characters of a name to be stored in the dictionary. A name should contain only graphic characters.

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    [CHAR] * EMIT ; ↵
```

the following occurs:
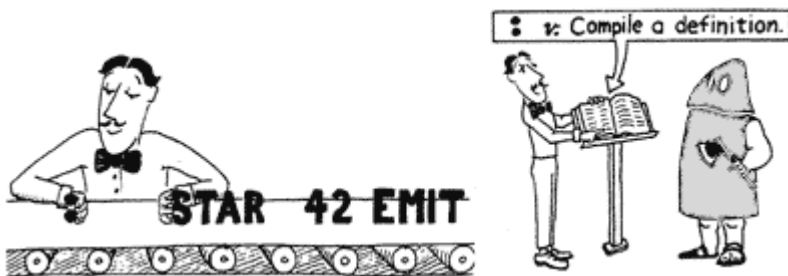
The text interpreter finds the colon in the input stream, and points it out to EXECUTE. The compiler translates the definition into dictionary form and writes it in the dictionary. When the compiler gets to the semicolon, he stops, and execution returns to the text interpreter, who gives the message ok.
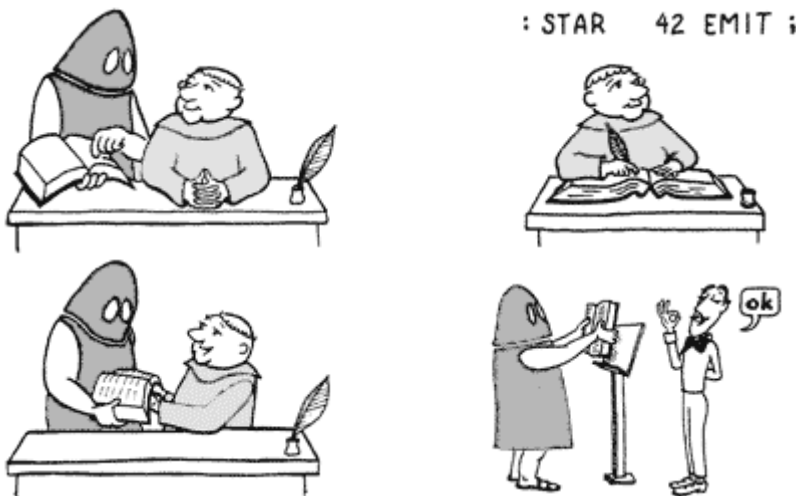
Say What?

In Forth, a word is a character or group of characters that have a definition. Almost any character can be used in naming a word. The reasons that some of the control characters cannot be used are:

return   because the computer thinks you've finished entering.
backspace because the computer thinks you are trying to correct a typing error.
space   because the computer thinks it's the end of the word.

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

```
: GREET   ." Hello, I speak Forth " ; ↵ok
```

We've just defined a word called GREET. Its definition consists of just one Forth word, .", followed by the text that 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."

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

Let's execute GREET:

```
GREET↵Hello, I speak Forth  ok
```

The Stack: Forth's Workspace for Arithmetic

A computer would not be much good if it couldn't do arithmetic. If you 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…
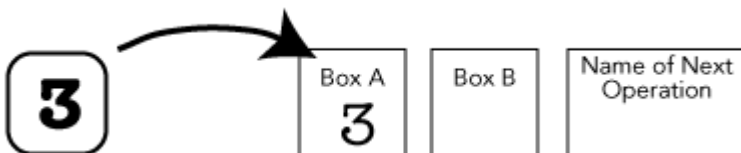
— the number 3 goes into one place (called Box A).
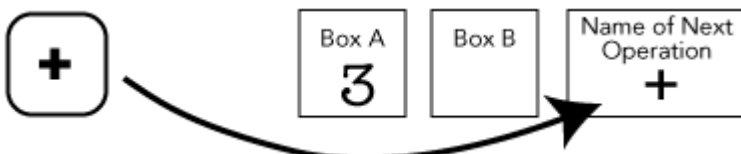
— the intended operation (addition) is remembered somehow.
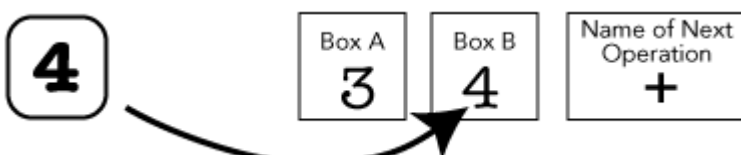
— the number 4 is stored into a second place (called Box B).
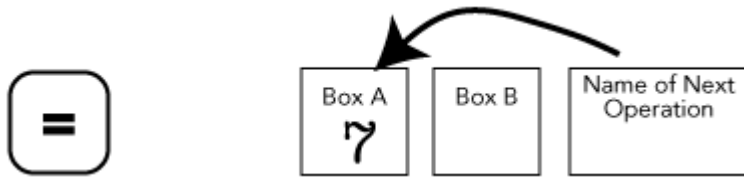
— the calculator performs the operation that is stored in the "Next Operation" Box on the contents of the number boxes and leaves the result in Box A.

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 + .↵7 ok
```

here is what happens, key by key.
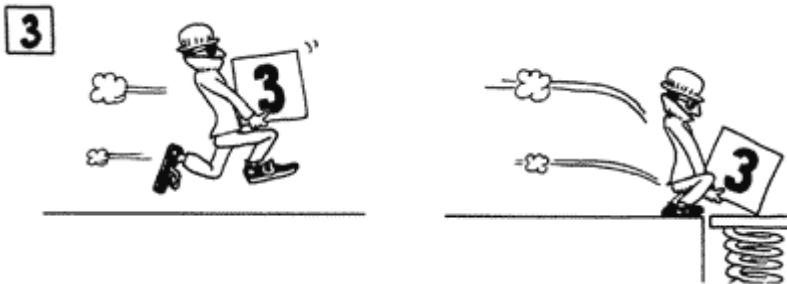
Recall that when you enter a number at your terminal, the text interpreter hands it over to NUMBER, who runs it to some location. That location, it can now be told, is the stack. In short, when you enter the number three from the terminal, you push it onto the stack.
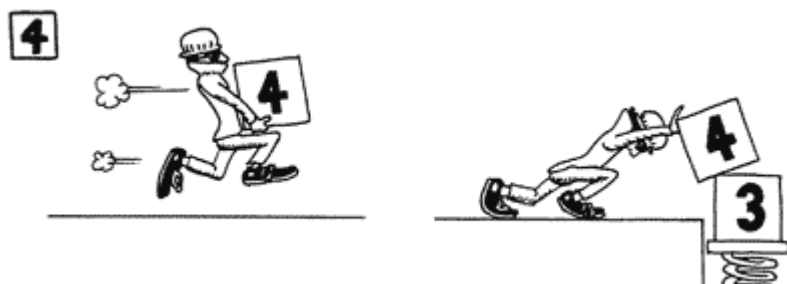
Now the four goes onto the "top" of the stack and pushes the three downward.
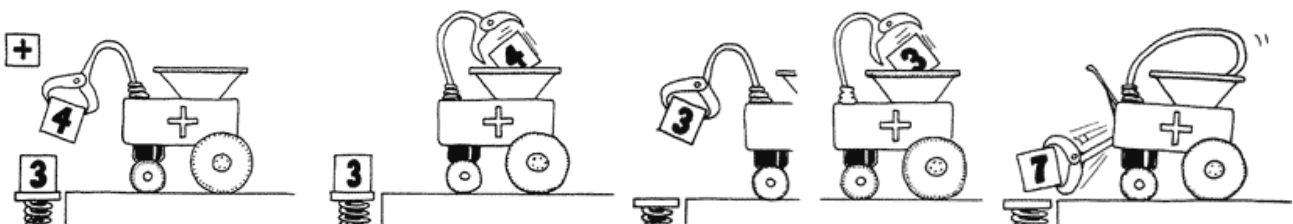
The next word in the input stream *can* be found in the dictionary. + has been previously defined to "take the top two numbers off the stack, add them, and push the result back onto the stack."
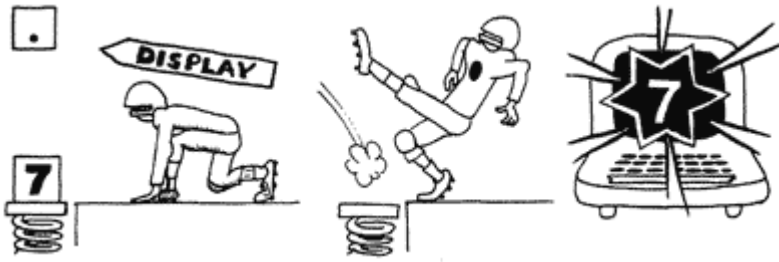
The next word, **.**, is also found in the dictionary. It has been previously defined to take the number off the stack and print it at the terminal.

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 *after* the numbers) rather than "infix" notation (so called because the operator is affixed *in-between* the numbers) so that all words which "need" numbers can get them from the stack.

For example:

- the word + gets two numbers from the stack and adds them;
- the word **.** gets one number from the stack and prints it;
- the word SPACES gets one number from the stack and prints that many spaces;
- the word 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  +  ; ↵
```

and test it this way:

```
3 FOUR-MORE .↵7 ok
```

and again:

```
-10 FOUR-MORE .↵-6 ok
```

The "4" *inside* the definition goes onto the stack, just as it would if it were outside a definition. Then the + adds the two numbers on the stack. Since + 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 a "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 **.** to further demonstrate. Remember that each **.** removes one number from the stack and prints it. Four dots, therefore, remove four numbers and print them.

```
2 4 6 8 . . . .↵8 6 4 2 ok
```
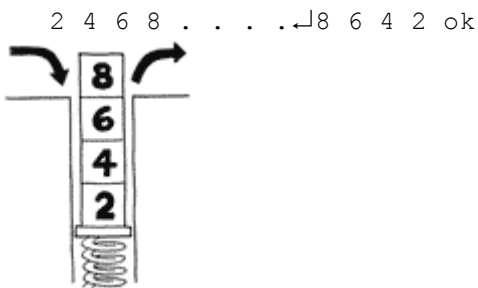

The system reads input from left to right and executes each word in turn.

- For input, the rightmost value on the screen will end up on the *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 ourselves into. Type:

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

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

```
10 20 30 . . . .↵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.
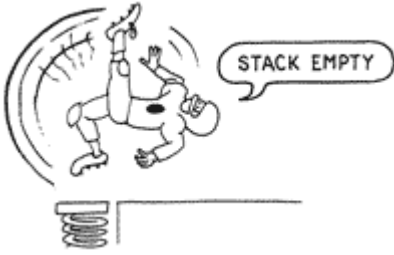
This error is called "stack underflow." (Notice that a stack underflow is *not* "ok.")

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 word's 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.

Here is 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 **.**:

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

(The letter "n" stands for "number.") This shows that **.** expects one number on the stack (before) and leaves *no* number on the stack (after).

Here's the stack notation for the word +.

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


You're the top

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

Since you probably have the hang of it, we may somtimes leave out the ↵ symbol. You can usually tell where to press "return" because the computer's response is always formatted like this.

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):

: xxxx 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 (pre-defined) 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 Revers 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 : needs a ;.
```

and

```
Every ." needs a ".
```

1. Define a word called GIFT which, when executed, will type out the name of some gift. For example, you might try:
2.   `: GIFT  ." Bookends " ;`
3. 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
```

[answer]

4. 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 +.) [answer]
5. 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? [answer]

2. How to Get Results

In this chapter, we'll dive right into some specifics that you need to know before we go on. Specifically, we'll introduce some of the arithmetic instructions besides + be able to write mathematical equations in Forth.

Forth Arithmetic — Calculator Style

Unlike calculators, computer terminals don't have special keys for multiplication or division. Instead we use * and /. ▫️

Here are the four simplest integer-
arithmetic operators in Forth:

+ (n1 n2 -- sum)   Adds.
- (n1 n2 -- diff)   Subtracts (n1-n2).
* (n1 n2 -- prod)   Multiplies.
/ (n1 n2 -- quot)   Divides (n1/n2).

In the first chapter, we learned that we can add two numbers by putting them both on the stack, then executing the word +, then finally executing the word **.** (dot) to get the result printed at our terminal.

```
17 5 + .↵22 ok
```

We can use this method with all of Forth's arithmetic operators. In other words, we can use Forth like a calculator to get answers, even without writing a "program." Try a multiplication problem:

```
7 8 * .↵56 ok
```

By now we've seen that the operator comes after the numbers. In the case of subtraction and division, though, we must also consider the *order of numbers* ("7 - 4" is not the same as "4 - 7").

Just remember this rule:

To convert to postfix, simply move the
operator to the end of the expression:

Infix	Postfix
3 + 4	3 4 +
500 - 300	500 300 -
6 x 5	6 5 *
20 / 4	20 4 /

So to do the subtraction problem:

```
7 - 4 =
```

simply type in

```
7 4 - .↵3 ok
```

For Adventuresome Newcomers Sitting at a Terminal

If you're one of those people who like to fool around and figure things out for themselves without reading this book, then you're bound to discover a couple of weird things. First off, as we told you, these

operators are *integer operators*. That not only means that you can't do calculations with decimal values, like

```
10.00 2.25 +
```

it also means that you can only get integer results, as in

```
21 4 / .↵5 ok   instead of 5.25 ok
```

Another thing is that if you try to multiply:

```
10000000 1000 * .
```

or some such large numbers, you'll get a crazy answer. So we're telling you up front that with the operators introduced so far and with **.** to print the results, you can't have any numbers that are higher than +2147483647 or lower than -2147483648. Numbers within this range are called "single-length signed numbers."

Notice, in the list of Forth words a few pages back, the letter "n," which stands for "number." Since Forth uses single-length numbers more often than other types of numbers, the "n" signifies that the number must be single-length. And yes, there are other operators that extend this range ("double-length" operators, which are indicated by "d").

All of these mysteries will be explained in time, so stay tuned.

The order of numbers stays the same. Let's try a division problem:

```
20 4 / .↵5 ok
```

The word / is defined to divide the second number on the stack by the top number.

SAMURAI DIVIDER

What do you do if you have more than one operator in an expression, like:

```
4 + (17 * 12)
```

you ask? Let's take it step-by-step: the parentheses tell you to first multiply seventeen by twelve, *then* add four. So in Forth you would write:

```
17 12 * 4 + .↵208 ok
```

and here's why:
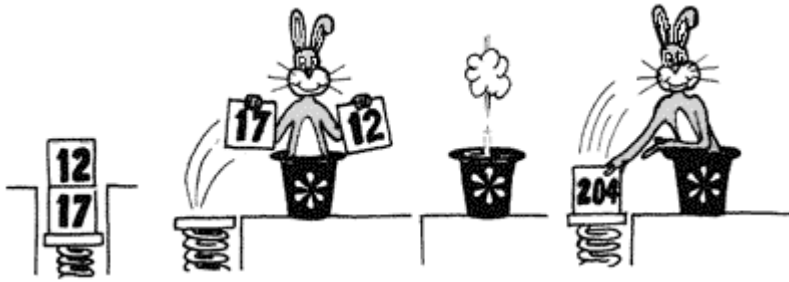
17 and 12 go onto the stack. * multiplies them and returns the result.

Then the four goes onto the stack, on top of 204. + rolls out the adding machine and adds them together, returning only the result.

Or suppose you want to add five numbers. You can do it in Forth like this:

```
17 20 + 132 + 3 + 9 + .↵181 ok
```
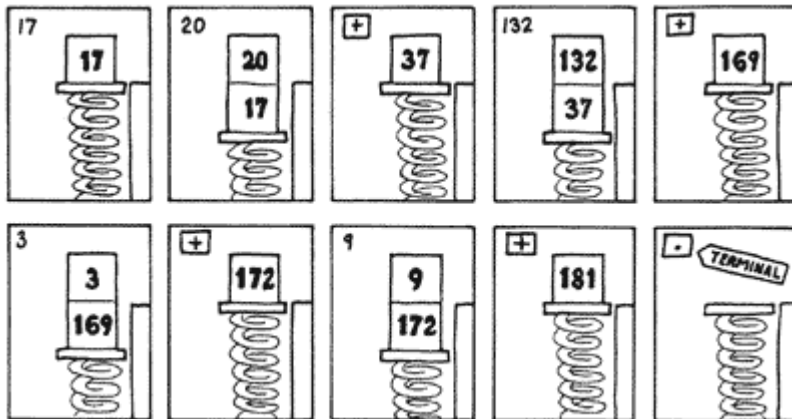

Now here's an interesting problem:

```
(3 + 9) * (4 + 6)
```

To solve it we have to add three to nine first, then add four to six, then finally multiply the two sums. In Forth, we can write

```
3 9 +  4 6 +  * .↵120 ok
```
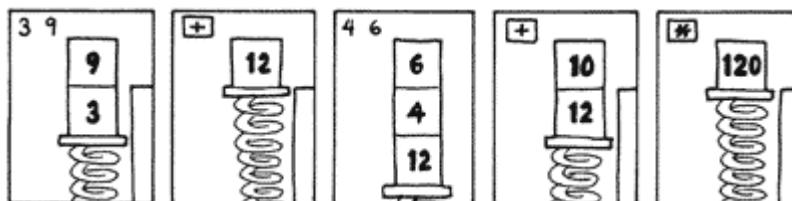
Notice that we very conveniently saved the sum twelve on the stack while we went on about the business of adding four to six.

Remember that we're not concerned yet with writing definitions. We are simply using Forth as a calculator.

If you're like most beginners, you probably would like to try your hand at a few practice problems until you feel more comfortable with postfix

.

Postfix Practice Problems (Quizzie 2-a)

Convert the following infix equations to postfix "calculator style." For example,

```
ab + c
```

would become

```
a b * c +
```

1. c(a+b)
2. (3a - b) / 4 + c
3. (0.5 ab) / 100
4. (n + 1) / n
5. x(7x + 5)

   Convert the following postfix expressions to infix:
6. a b - b a + /
7. a b 10 * /

[answer]

Forth Arithmetic — Definition Style
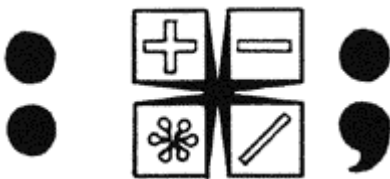
In Chap. 1 we saw that we could define new words in terms of numbers and other pre-defined words. Let's explore some further possibilities, using some of our newly-learned math operators.

Let's say we want to convert various measurements to inches. We know that

```
1 yard = 36 inches
```

and

```
1 foot = 12 inches
```

so we can define these two words:

```
: YARDS>IN  36 * ;↵ok
: FT>IN    12 * ;↵ok
```

where the names symbolize "yards-to-inches" and "feet-to-inches." Here's what they do:

```
10 YARDS>IN .↵360 ok
2 FT>IN .↵24 ok
```

If we *always* want our result to be in inches, we can define:

```
: YARDS  36 * ;↵ok
: FEET   12 * ;↵ok
: INCHES   ;↵ok
```

So that we can use the phrase

```
10 YARDS  2 FEET +  9 INCHES +  .↵393 ok
```

Notice that the word INCHES doesn't do anything except remind the human user what the nine is for. If we really want to get fancy, we can add these three definitions:

```
: YARD   YARDS ;↵ok
: FOOT   FEET ;↵ok
: INCH   ;↵ok
```

so that the user can enter the singular form of any of the nouns and still get the same result:

```
1 YARD  2 FEET +  1 INCH +  .↵61 ok
2 YARDS  1 FOOT + .↵84 ok
```

For Semantic Freaks

In mathematics, the word "argument" refers to an independent variable of a function. Computer linguists have borrowed this term to refer to numbers being operated on by operators. They have also borrowed the word "parameters" to describe pretty much the same thing.

So far we have only defined words whose definitions contain a single math operator. But it's perfectly possible to put many operators inside a definition, if that's what you need to do.

Let's say we want a word that computes the sum of five numbers on the stack. A few pages back we summed five numbers like this:

```
17 20 + 132 + 3 + 9 + .↵181 ok
```

But we can also enter

```
17 20 132 3 9 + + + + .↵181 ok
```
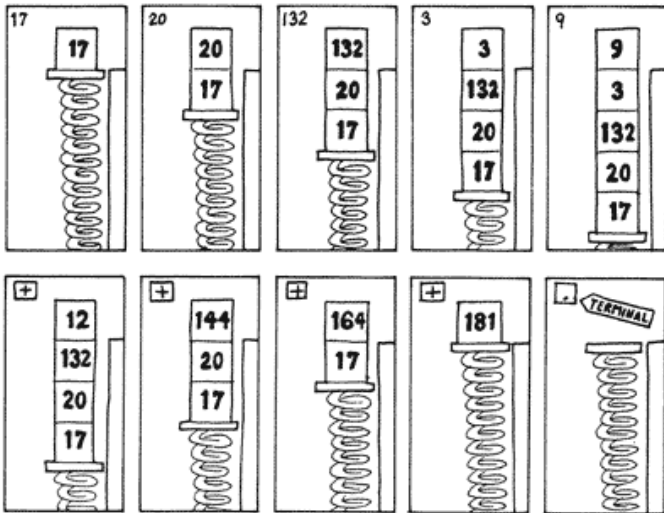
We get the same answer, even though we've clustered all the numbers into one group and all the operators into another group. We can write our definition like this:

```
: 5#SUM  + + + + ;↵ok
```

and execute it like this:

```
17 20 132 3 9 5#SUM .↵181 ok
```

If we were going to keep 5#SUM for future use, we could enter it into our ever-growing glossary, along with a note that it "expects five arguments" on the stack, which it will add together.

For Beginners Who Like Word Problems

If a jet plane flies at an average air speed of 600 mph and if it flies with a tail wind of 25 mph, how far will it travel in five hours?

If we define

```
: FLIGHT-DISTANCE  + * ;
```

we could enter

```
5 600 25 FLIGHT-DISTANCE .↵3125 ok
```

Try it with different values, including head winds (negative values).

Here is another equation to write a definition for:

```
(a + b) * c
```

As we saw in Quizzie 2-a, this expression can be written in postfix as

```
c a b + *
```

Thus we could write our definition

```
: SOLUTION   + * ;↵ok
```

as long as we make sure that we enter the arguments in the proper order;

```
c a b SOLUTION
```

Definition Style Practice Problems (Quizzie 2-b)

Convert the following infix expressions into Forth definitions and show the stack order required by your definitions. Since this is Quizzie 2-b, you can name your definitions 2B1, 2B2, etc.

ab + c would become : 2B1 * + ;
which expects this stack order: (c b a -- result)

1. (a - 4b) / 6 + c
2. a / (8b)
3. 0.5 ab / 100
4. a(2a + 3)
5. (a - b) / c

[answer]

The Division Operators

The word / is Forth's simplest division operator. *Slash* supplies only the quotient; any remainder is lost. If you type:

```
22 4 / .↵5 ok
```

You get only the quotient five, not the remainder two.

If you're thinking of a pocket calculator's per-cent operator, then five is not the full answer.

But / is only one of several division operators supplied by Forth to give you the flexibility to tell the computer exactly what you want it to do.

For example, let's say you want to solve this problem: "How many dollar bills can I get in exchange for 22 quarters?" The real answer, of course, is exactly 5, not 5.5. A computerized money changer, for example, would not know how to give you 5.5 dollar bills.

Here are two more Forth division operators:

/MOD	(n1 n2 -- rem quot)	Divides. Returns the remainder and quotient.
MOD	(n1 n2 -- rem)	Returns the remainder from division.

These operators are both signed, and "truncating." We'll see what this means in the chapter on computer numbers.

/MOD gives both the remainder and the quotient; MOD (from "modulo") gives the remainder only. (For /MOD, the stack notation in the table indicates that the quotient will be on top of the stack, and the remainder below. Remember, the *rightmost* represents the *topmost*.)

Let's try the first one:

```
22 4 /MOD . .↵5 2 ok
```


Here /MOD performs the division and puts both the quotient and the remainder on the stack. The first dot prints the quotient because the quotient was on top.

With what we've learned so far, we can easily define this word:

```
: QUARTERS  4 /MOD . ." ones and "
    . ." quarters " ;
```

So that you can type:

```
22 QUARTERS
```

with this result:

```
22 QUARTERS↵5 ones and 2 quarters ok
```

The second word in the table, MOD, leaves only the remainder. For example in:

```
22 4 MOD .↵2 ok
```

the two is the remainder.

Stack Maneuvers

If you worked Prob. 6 in the last set, you discovered that the infix equation

```
(a - b) / c
```

cannot be solved with a definition unless there is some way to rearrange values on the stack.

Well, there is a way: by using a "stack manipulation operator" called SWAP.

SWAP

The word SWAP is defined to switch the order of the top two stack items.
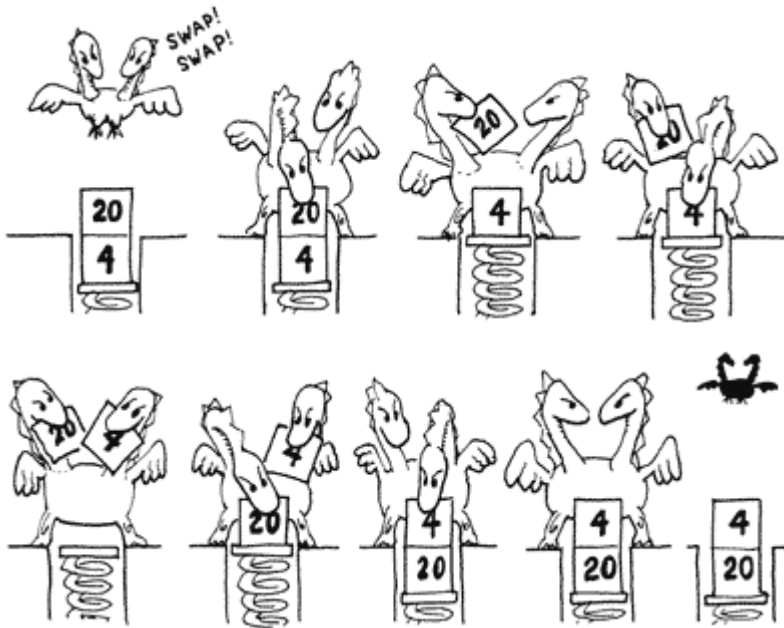
As with the other stack manipulation operators, you can test SWAP at your terminal in "calculator style"; that is, it doesn't have to be contained within a definition.

First enter

```
1 2 . .↵2 1 ok
```

then again, this time with SWAP:

```
1 2 SWAP . .↵1 2 ok
```

Thus Prob. 6 can be solved with this phrase:

```
- SWAP /
```

with (c a b --) on the stack.

Let's give a, b, and c these test values:

```
a = 10
```

```
b = 4
```

```
c = 2
```

then put them on the stack and execute the phrase, like so:

```
2 10 4 - SWAP / .↵3 ok
```

Here is a list of several stack manipulation operators, including SWAP

SWAP (n1 n2 -- n2 n1)   Reverses the top two stack items.
DUP   (n -- n n)   Duplicates the top stack item.
OVER   (n1 n2 -- n1 n2 n1)   Makes a copy of the second item and pushes it on top.
ROT   (n1 n2 n3 -- n2 n3 n1) Rotates the third item to the top.
DROP   (n --)   Discards the top stack item.
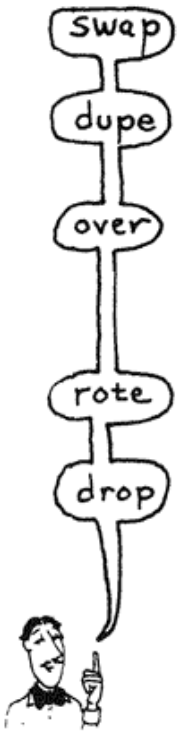
DUP

The next stack manipulation operator on the list, DUP, simply makes a second copy (duplicate) of the top stack item.

For example, if we have "a" on the stack, we can compute:

a^2

as follows:

```
DUP *
```

in which the following steps occur:

Operation	Contents of stack
	a
DUP	a a
*	a^2

OVER

Now somebody tells you to evaluate the expression:

```
a * (a + b)
```

given the following stack order:

```
( a b -- )
```


But, you say, I'm going to need a new manipulation operator: I want two copies of the "a," and the "a" is *under* the "b." Here's the word you need: OVER. OVER simply makes a copy of the "a" and leapfrogs it *over* the "b":

```
( a b -- a b a )
```

Now the expression

```
a * (a + b)
```

can easily be written

```
OVER + *
```

Here's what happens:

Operation	Contents of stack
	a b
OVER	a b a
+	a (b+a)
*	a*(b+a)

When writing equations in Forth, it's best to "factor them out" first. For example, if somebody asks you to evaluate:

$a^2 + ab$

in Forth, you'll find it quite complicated (and maybe even impossible) using the words we've introduced so far ... unless you factor out the expression to read:

```
a * (a + b)
```

which is the expression we just evaluated so easily.

ROT

The fourth stack manipulator on the list is ROT (pronounced *rote*), which is short for "rotate." ROT transforms the top three stack values from (a b c) to (b c a).
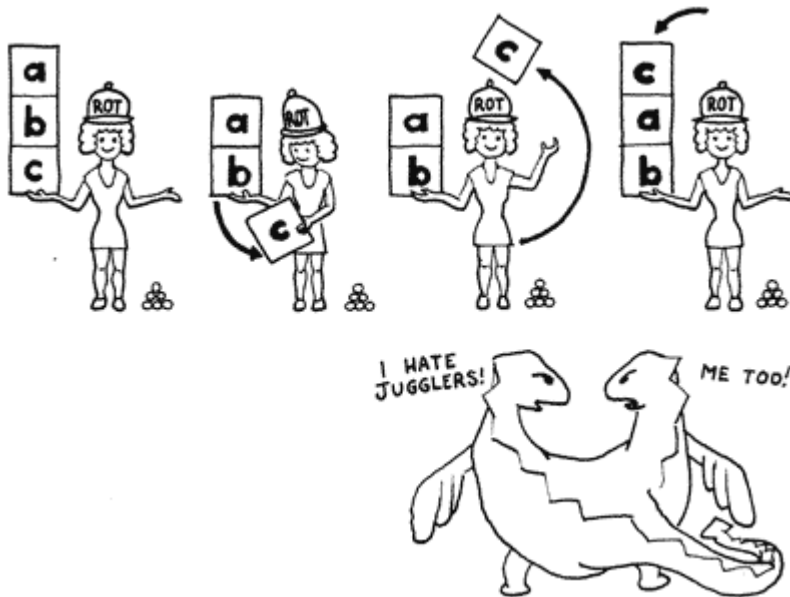
For example, if we need to evaluate the expression:

```
ab - bc
```

we should first factor out the "b"s:

```
b * (a - c)
```

Now if our starting-stack order is this:

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

we can use:

```
ROT - *
```

in which the following steps will occur:

Operation	Contents of stack
	c b a
ROT	b a c
-	b (a-c)
*	b*(a-c)

DROP

The final stack manipulation operator on the list is DROP. All it does is discard the top stack value.
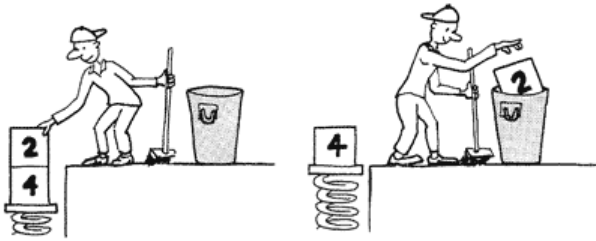
Pretty simple, huh? We'll see some good uses for DROP later on.

A Handy Hint
A Non-destructive Stack Print

Beginners who are just learning to manipulate numbers on the stack in useful ways very often find themselves typing a series of dots to see what's on the stack after their manipulations. The problem with dots, though, is that they don't leave the numbers on the stack for future manipulation.

The Forth word .S prints out all the values that happen to be on the stack "non-destructively"; that is, without removing them. Let's test it, first with nothing on the stack:

```
.S↵<0>  ok
```

As you can see, in this version of .S, we see at least one number. This is the number of items actually on the stack.

Now let's try with numbers on the stack:

```
1 2 3 .S↵<3> 1 2 3  ok

ROT .S <3>↵2 3 1  ok
```

Stack Manipulation and Math Definitions (Quizzie 2-c)

1. Write a phrase which flips three items on the stack, leaving the middle number in the middle; that is,

2. ```
   a b c
   ```

   becomes

   ```
   c b a
   ```

3. Write a phrase that does what OVER does, without using OVER.
4. Write a definition called -ROT, which rotates the top three stack items in the opposite direction from ROT; that is,

   ```
   a b c
   ```

   becomes

```
c a b
```

5. Write definitions for the following equations, given the stack effects shown:
    $(n+1) / n$ (n -- result)
    $x(7x + 5)$ (x -- result)
    $9a^2 - ba$ (a b -- result)

[answer]

Playing Doubles

The next four stack manipulation operators should look vaguely familiar:

2SWAP	(d1 d2 -- d2 d1)	Reverses the top two pairs of numbers.
2DUP	(d -- d d)	Duplicates the top pair of numbers.
2OVER	(d1 d2 -- d1 d2 d1)	Makes a copy of the second pair of numbers and pushes it on top.
2DROP	(d --)	Discards the top pair of numbers.

The prefix "2" indicates that these stack manipulation operators handle numbers in pairs. The letter "d" in the stack effects column stands for "double." "Double" has a special significance that we will discuss when we talk about "n" and "u."

The "2"-manipulators listed above are so straightforward, we won't even bore you with examples.

One more thing: there are still some stack manipulators we haven't talked about yet, so don't go crazy by trying too much fancy footwork on the stack.

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

+	(n1 n2 -- sum)	Adds.
-	(n1 n2 -- diff)	Subtracts (n1-n2).
*	(n1 n2 -- prod)	Multiplies.
/	(n1 n2 -- quot)	Divides (n1/n2).
/MOD	(n1 n2 -- rem quot)	Divides. Returns the remainder and quotient.
MOD	(n1 n2 -- rem)	Returns the remainder from division.
SWAP	(n1 n2 -- n2 n1)	Reverses the top two stack items.
DUP	(n -- n n)	Duplicates the top stack item.
OVER	(n1 n2 -- n1 n2 n1)	Makes a copy of the second item and pushes it on top.

ROT	(n1 n2 n3 -- n2 n3 n1)	Rotates the third item to the top.
DROP	(n --)	Discards the top stack item.
2SWAP	(d1 d2 -- d2 d1)	Reverses the top two pairs of numbers.
2DUP	(d -- d d)	Duplicates the top pair of numbers.
2OVER	(d1 d2 -- d1 d2 d1)	Makes a copy of the second pair of numbers and pushes it on top.
2DROP	(d --)	Discards the top pair of numbers.

Review of Terms

Double-length numbers
　　integers which encompass a range of over -18,446,744,073,709,551,615 to
　　+18,446,744,073,709,551,615 (and which we'll introduce officially in Chap. 7).
Single-length numbers
　　integers which fall within the range of -2 billion to +2 billion: the only numbers which are valid as
　　the arguments or results of any of the operators we've discussed so far.

Problems — Chapter 2

1. What's the difference between DUP DUP and 2DUP? [answer]
2. Write a phrase which will reverse the order of the top four items on the stack; that is,
   (1 2 3 4 -- 4 3 2 1) [answer]
3. Write a definition called

   3DUP

   which will duplicate the top three numbers on the stack; for example,
   (1 2 3 -- 1 2 3 1 2 3) [answer]

4. Write definitions for the following infix equations, given the stack effects shown:
   a^2 + ab + c (c a b -- result) [answer]

   (a-b) / (a+b) (a b -- result) [answer]
5. Write a set of words to compute prison sentences for hardened criminals such that the judge can
   enter:
6. 　 CONVICTED-OF ARSON HOMICIDE TAX-EVASION↵ok

7. 　 WILL-SERVE↵35 years ok

   or any series of crime beginning with the word CONVICTED-OF and ending with WILL-
   SERVE. Use these sentences

   ```
   HOMICIDE 20 years
   ARSON 10 years
   BOOKMAKING 2 years
   TAX-EVASION 5 years
   ```

   [answer]

8. You're the inventory programmer at Maria's Egg Ranch. Define a word called

   EGG.CARTONS

   which expects on the stack the total number of eggs laid by the chickens today and prints out the number of cartons that can be filled with a dozen each, as well as the number of left-over eggs.
   [answer]

3. The Editor (and Staff)

Another Look at the Dictionary

If you've been experimenting with a real computer, you may have discovered some things we haven't mentioned yet. In any case, it's time to mention them.

Discovery One:

You can define the same word more than once in different ways — only the most recent definition will be executed.

For example, if you have entered:

```
: GREET  ." Hello, I speak Forth. " ;↵ok
```

then you should get this result:

```
GREET↵Hello, I speak Forth. ok
```

And if you redefine:

```
: GREET  ." Hi there! " ;↵ok
```

you get the most recent definition:

```
GREET↵Hi there! ok
```

Has the first GREET been erased? No, it's still there, but the most recent GREET is executed because of the search order. The text interpreter always starts at the "back of the dictionary" where the most recent entry is. The definition he finds first is the one you defined last. This is the one he shows to EXECUTE.

We can prove that the old GREET is still there. Try this:

```
FORGET GREET↵ok
```

and

```
GREET↵Hello, I speak Forth. ok
```

(the old GREET again!).
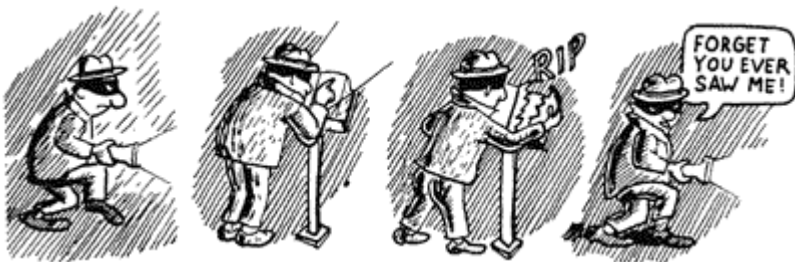
The word FORGET looks up a given word in the dictionary and, in effect, removes it from the dictionary along with anything you may have defined since that word. FORGET, like the interpreter, searches starting from the back; he only removes the most recently defined versions of the word (along with any words that follow). So now when you type GREET at the terminal, the interpreter finds the original GREET.

FORGET is a good word to know; he helps you to weed out your dictionary so it won't overflow. (The dictionary takes up memory space, so as with any other use of memory, you want to conserve it.)

Some Forths do not have FORGET. In that case you need to plan the forgetting in advance, e.g.:

```
MARKER -work
```

defines the null definition -work to mark the current system state for you. When you execute -work at some later time, the system state is restored to that in effect when -work was defined. In particular, all words defined after the marker word -work are completely removed from the dictionary.

Discovery Two:

When you enter definitions from the terminal (as you have been doing), your source text is not saved.

Only the compiled form of your definition is saved in the dictionary. So what if you want to make a minor change to a word already defined? This is where a "text editor" comes in. With this editor, you can save your source text and modify it if you want to. In this day and age we can assume that everyone has access to a text editor. The documentation of your Forth system should discuss the procedures to easily use your favorite text editor from within the Forth environment. (On a modern OS, double-click the file you want to edit. After finishing your editing business, type INCLUDE on the Forth command line. Add at least one trailing space, then drag your file in the Forth window and drop it on the command line. Type ↵.)

A text editor stores your source text on disk. So we'd better introduce the disk and the way the Forth system uses it.

How Forth Uses the Disk

All Forth systems use disk memory. Even though disk memory is not absolutely necessary for a Forth system, it's difficult to imagine Forth without it.

To understand what disk memory does, compare it with computer memory (RAM). The difference is analogous to the difference between a filing cabinet and a rolling card-index.

So far you've been using computer memory, which is like the card index. The computer can access this memory almost instantaneously, so programs that are stored in RAM can run very fast. Unfortunately, this kind of memory is sometimes very limited (e.g. in embedded controllers) and relatively expensive.

On the other hand, the disk is called a "bulk memory" device, because, like a filing cabinet, it can store a lot of information at a much cheaper price per unit of information than the memory inside the computer.

Both kinds of memory can be written to and read from.

The compiler compiles all dictionary entries into computer memory so that the definitions will be quickly accessible. The perfect place to store source text, however, is on the disk, which is what Forth does. You can either send source text directly from the keyboard to the interpreter (as you have been doing), or you can save your source text on the disk and then later read it off the disk and send it to the text interpreter.

Disk memory is divided into units called "blocks." Each block holds 1,024 characters of source text or binary data, traditionally organized as 16 lines of 64 characters. The ANS Forth standard does not specify how many blocks there are. The documentation of your Forth system should tell you this.

With current Forths, disk memory resides in OS files. There are ways to attach specific OS files to the "Forth disk." Due to the special 16 by 64 format of Forth blocks, OS utilities consider them as binary data and cannot generally print, list, filter or edit them. Forth systems have standardized facilities to handle some of these tasks by themselves.

Assuming you are using iForth, then the following should instruct disk memory to come from some file:

```
USE blocks.forth↵ok
```

To list a block, simply type the block-number and the word LIST, as in:

```
1 LIST
 0
 1 ( Large letter F                            MHX 21:29 07/01/89)
 2
 3 : STAR    [CHAR] * EMIT ;
 4 : STARS   0 DO  STAR  LOOP ;
 5 : MARGIN  CR 30 SPACES ;
 6 : BLIP    MARGIN STAR ;
 7 : BAR     MARGIN 5 STARS ;
 8 : F       BAR BLIP BAR BLIP BLIP CR ;
 9
10
11
12
13
14
15
ok
```

The above is what a block looks like when it's listed on your terminal.

To give you a better idea of how all of this could be used, we'll assume that block 1 contains the definitions shown above. Except for line 1, everything looks familiar: these are the definitions you used to print a large letter "F" at your terminal.

Now if you were to type:

```
1 LOAD
F
```

you would send block 1 to the input stream and then on to the text interpreter. The text interpreter does not care where his text comes from. Recognizing the colons, he will have all the definitions compiled, and then will execute the new word F.

Now for the unfinished business: line 1. The words inside the parentheses are for humans only; they are neither compiled nor executed. The word (("left parenthesis") tells the text interpreter to skip all the following text up to the terminating right parenthesis. Because (is a word, it must be set off with a space. The closing parenthesis is not a word, it is simply a character that is looked for by (, called a delimiter. (Recall that the delimiter for ." is the closing quote mark.)

To summarize, the three ANS Forth commands we've learned so far that concern disk blocks are:

LIST   (n --) Lists a disk block.

LOAD (n --) Loads a disk block (compiles or executes).

(xxx) (--)   Causes the string *xxx* to be ignored by the text interpreter. The character *)* is the delimiter.

Block-buffer Basics

We have discussed blocks mainly because of historical reasons. Blocks are hardly ever used for source text storage any more. The preferred way to handle source is in standard text files, using the word INCLUDE to load them:

```
INCLUDE blocks.forth↵ok
```

The main advantage is that `blocks.forth` can be edited and managed with standard text file utilities.

However, now we're at it, we'll mention a few other words to access and modify blocks on disk.

The basic word that brings a block in from the disk, after first finding an available buffer and storing its contents on disk if necessary, is BLOCK. For instance, if you say

```
1 BLOCK
```


the system will copy block 1 of the currently open file into one of the system buffers. BLOCK also leaves on the stack the address of the beginning of the buffer (1024 bytes, remember) that it used. The contents of this buffer are guaranteed to stay valid until you execute a word from the set of procedures with "multitasking impact," like EMIT or TYPE. If you at any time modify the buffer contents and then execute the word UPDATE, Forth will remember to first write the block back to disk when it needs to reuse the buffer. If, for some reason, you execute UPDATE and then decide that you don't want to have the blocks rewritten after all, use EMPTY-BUFFERS to invalidate them. This works because Forth does not immediately write the disk after you use UPDATE. To force writing out the buffers *right now*, use the word FLUSH.

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

USE xxx	(--)	Designate OS text file *xxx* as the "Forth disk."
LIST	(n --)	Lists a disk block.
LOAD	(n --)	Loads a disk block (compiles or executes).

(xxx)	(--)	Causes the string *xxx* to be ignored by the text interpreter. The character *)* is the delimiter.
UPDATE	(--)	Marks the most recently referenced block as modified. The block will later be automatically transferred to mass storage if its buffer is needed to store a different block or if FLUSH is executed.
EMPTY-BUFFERS	(--)	Marks all block buffers as empty without necessarily affecting their actual contents. Updated blocks are not written to mass storage.
BLOCK	(u -- addr)	Leaves the address of the first byte in block u. If the block is not already in memory, it is transferred from mass storage into whichever memory buffer has been least recently accessed. If the block occupying that buffer has been updated (i.e., modified), it is rewritten onto mass storage before block u is read into the buffer.
INCLUDE xxx	(--)	Load the text file *xxx* (compiles or executes).
FORGET xxx	(--)	Forgets all definitions back to and including *xxx*.
MARKER xxx	(--)	Creates a word *xxx* which, when executed, restores the dictionary to the state it had just prior to the definition of *xxx*. In particular, remove *xxx* and all subsequent word definitions.

Review of Terms

Block
    in Forth, a division of disk memory containing up to 1024 characters of source text.
Buffer
    a temporary storage area for data.
Null definition
    a definition that does nothing, written in the form:
    : NAME ;
    that is, a name only will be compiled into the dictionary. A null definition serves as a "bookmark" in the dictionary, for FORGET to find.
Pointer
    a location in memory where a number can be stored (or changed) as a reference to something else.
Source text
    in Forth, the written-out form of a definition or definitions in English-like words and punctuation, as opposed to the compiled form that is entered into the dictionary.

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 THEN.

The word = takes two values of the stack and compares them to see if they are equal.
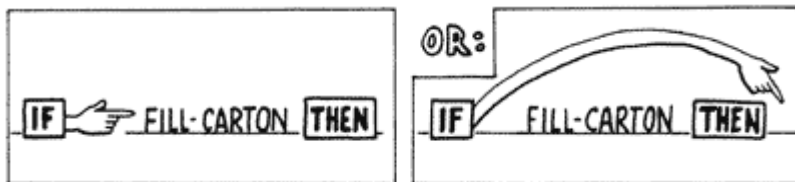
If the condition is true, IF allows the flow of execution to continue with the next word in the definition.

But if the condition is false, IF causes the flow of execution to skip to 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 IF...THEN statement must be contained within a definition. You can't just enter these words in "calculator style."

Don't be misled by the traditional English meanings of the Forth words IF and THEN. The words that follow IF are executed *if* the condition is true. The words that follow 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 THEN is ;, 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 ;
```

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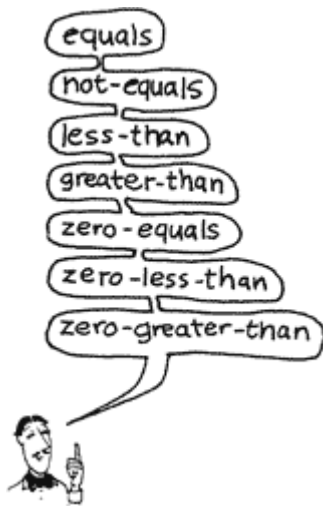
Here is a partial list of comparison operators that you can use before an IF...THEN statement:

=

<

>

0=

0<

0>

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

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

```
... = INVERT IF ...
```

will execute the words after IF, if the two numbers on the stack are *not* equal.

The Alternative Phrase

Forth allows you to provide an alternative phrase in an IF statement, with the word 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 IF pulls a railroad-track switch, depending on the outcome of the test. Execution then takes one of two possible routes, but either way, the tracks rejoin at the word THEN.

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:

```
( numerator denominator -- quotient )
: /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 ;
```

Once EGGSIZE has been entered, 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:
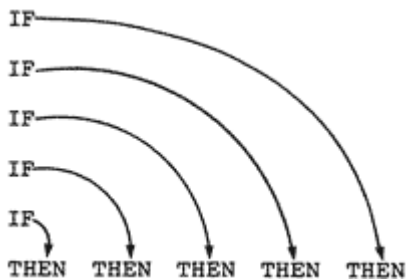
Also notice that a DROP is necessary at the end of the definition to get rid of the original value.

Finally, notice that the definition is visually organized to be read easily by human beings. Most Forth programmers would rather waste a little space than let things get any more confused than they have to be.

A Closer Look at IF

How does the comparison operator (=, <, >, or whichever) let IF know whether the condition is true or false? By simply leaving TRUE or FALSE on the stack. A TRUE (all bits high) means that the condition is true; a FALSE (all bits low) means that the condition is false.
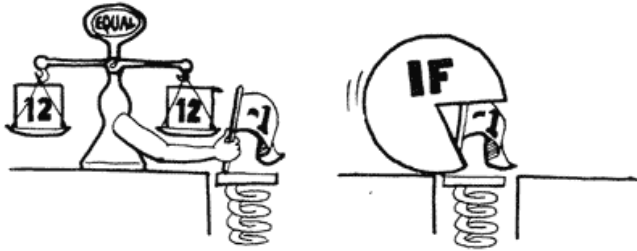
In computer jargon, when one piece of program leaves a value as a signal for another piece of program, that value is called a "flag."

Try entering the following phrases at the terminal, letting **.** show you what's on the stack as a flag.

```
5 4 > .↵-1 ok
5 4 < .↵0 ok
```

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

IF will take a TRUE as a flag that means true and a FALSE as a flag that means false. Now let's take a closer look at INVERT, which reverses the flag on the stack.

```
FALSE INVERT .↵-1 ok
TRUE INVERT .↵0 ok
```

Now we'll let you in on a little secret: IF will take *any non-zero value* to mean true.

To prove it, try entering this test:

```
: TEST  IF ." non-" THEN ." zero " ;
```

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

```
0 TEST↵zero ok
1 TEST↵non-zero ok
-400 TEST↵non-zero ok
```

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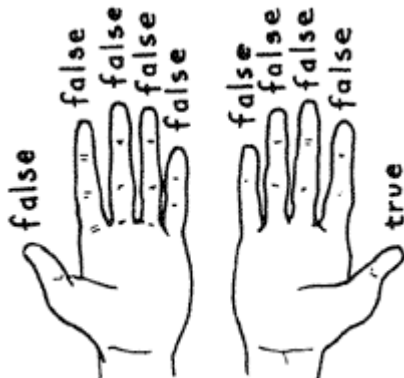
Here's another interesting result. Say you want to test whether a number is an even multiple of ten, such as 10, 20, 30, 40 etc. You know that the phrase

```
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.

Still another interesting result is that you can use - (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. However, now we must talk a bit about "well-formed flags."

If you think about it, both 0= and INVERT do almost the same thing. However, 0= changes the number 0 to the number -1 and any non-zero number to 0, while INVERT changes all zero bits in a number to one bits and the one bits in that number to zero bits. Only when the number is a "well-formed flag", i.e., either 0 or -1, the result of 0= and INVERT is the same. All comparison operators return well-formed flags, fit for either 0= or INVERT. However, when you use - to compare two numbers, as we did above, the flag will not be well-formed when the two numbers differ in value, and only 0= can be used to safely reverse the meaning of the comparison.

A final result is described in the next section.
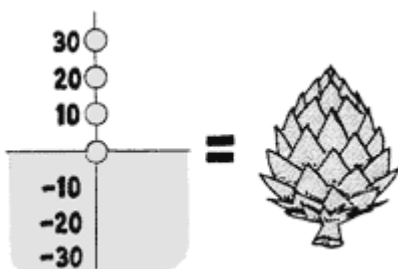
A Little Logic

It's possible to take several flags from various tests and combine them into a single flag for one 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 (true).
+                                      -1   Add 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 -1, which means "true." If the input number had been -30, then both condition would have been true and the sum would have been minus two. Minus two is, of course, non-zero. So as far as IF is concerned, -2 is as true as -1.

Our simple-minded definition, then would be:

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

Here is 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 ." Looks good " 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 +. 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 OR, which will return the correct flag even in case of 1 and -1. An "or decision" is the computer term for the kind of flag 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.

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.

How can we do this "and decision" in Forth? By using the handy word AND. Here's what AND would do with the four possible combinations of flags we saw earlier:
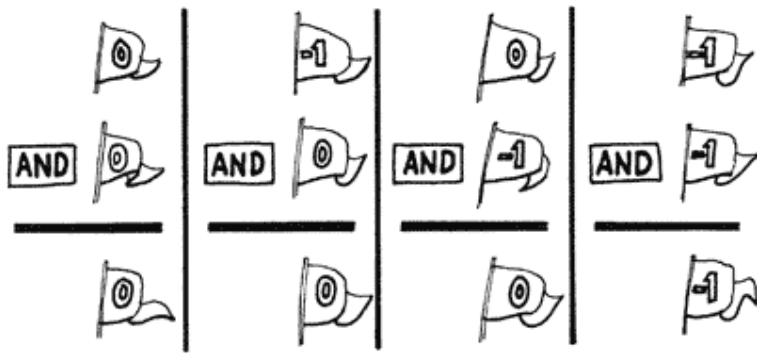
In other words, only the combination "-1 -1 AND" produces a result of "true." Let's say we're looking for a cardboard box that's big enough to fit a disk drive h measures:
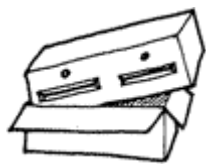

```
height  6"
width  19"
length 22"
```

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

```
: 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 following 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 0= 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.
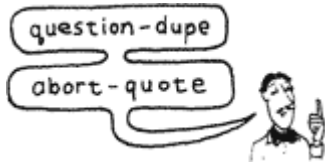
Two Words with Built-in IF

?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 ;
```
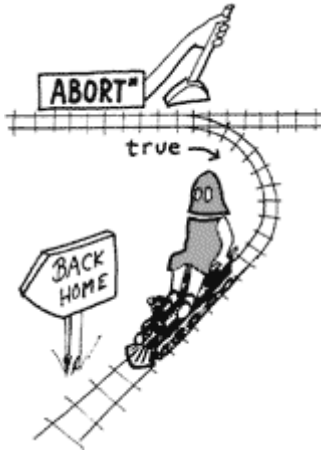

ABORT"

It may happen that somewhere in a complex application an error might occur (such as a 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 stacks 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 " / ;
```

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
Error -2
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
Error -2
zero denominator   ?
```

The point is that when /CHECK aborts, the rest of ENVELOPE is skipped.

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 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 ELSE yyy THEN zzz	IF: (f --)	If *f* is true (non-zero) executes *xxx*; otherwise executes *yyy*; continues execution with *zzz* regardless. The phrase ELSE *yyy* is optional.
=	(n1 n2 -- f)	Returns true if n1 and n2 are equal.
-	(n1 n2 -- n-diff)	Returns true (i.e., the non-zero difference) if n1 and n2 are not equal.
<	(n1 n2 -- f)	Returns true if n1 is less than n2.
>	(n1 n2 -- f)	Returns true if n1 is greater than n2.
0=	(n -- f)	Returns true if n is zero (i.e., reverse the truth value).
0<	(n -- f)	Returns true if n is negative.
0>	(n -- f)	Returns true if n is positive.
AND	(n1 n2 -- and)	Returns the logical AND.
OR	(n1 n2 -- or)	Returns the logical OR.
?DUP	(n -- n n) or (0 -- 0)	Duplicates only if n is non-zero.
ABORT" xx"	(f --)	If the flag is true, types out an error message, followed by the text. Also clears the 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 execution. 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.

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* one of them is true, the result is true.

Problems — Chapter 4

1. What will the phrase
2.   `0= 0=`

    leave on the stack when the argument is

        `-1?`

        `0?`

       `200?`

    [answer]

3. Explain what an artichoke has to do with any of this.
4. 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):
5.   `ALCOHOLIC BEVERAGES PERMITTED   or`
6.   `UNDER AGE`

    [answer]

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

8.   POSITIVE   or
9.   ZERO   or
10.   NEGATIVE

[answer]

11. 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

12.   0 STARS↵* ok

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

13. Write the definition for the word WITHIN which expects three arguments:

14.   (n lo-limit hi-limit --)

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

        low-limit <= n < hi-limit

[answer]

15. 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

16.   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. [answer]

17. Using nested tests and IF...ELSE...THEN statements, write a definition called SPELLER which will spell out a number on the stack, from -4 to 4. If the number is outside this range, it will print the message "OUT OF RANGE." For example:

18.   2 SPELLER↵two ok
19.   -4 SPELLER↵negative four ok
20.   7 SPELLER↵OUT OF RANGE ok

Make it as short as possible. (Hint: The Forth word ABS gives the absolute value of a number on the stack.) [answer]

21. Using your definition of WITHIN from Prob. 6, 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:

22.   0 1000 TRAP↵BETWEEN ok
23.   330 660 TRAP↵BETWEEN ok
24.   440 550 TRAP↵NOT BETWEEN ok
25.   330 440 TRAP↵BETWEEN 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 of the arguments is equal to the hidden value. [answer]

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.

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).

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.

The only reason to use a word such as 1+, instead of one and +, is tradition. In modern Forths 1+ saves neither space nor compile or execution time.

Miscellaneous Math Operators

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

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.

Aunt Min and Uncle Max

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
37 52 DIFFERENCE .↵15 ok
```

MIN
Write a definition which computes the commission that furniture salespeople will receive if they've been promised $50 or 1/10 of the sales 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 as kind of "extra hand" to hold values temporarily while you perform operations on the parameter stack.
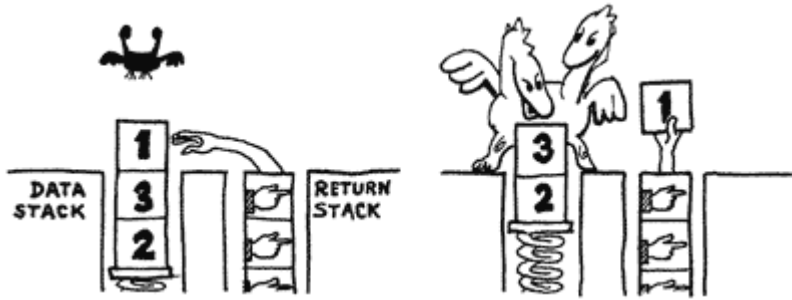
The return stack is a last-in first-out structure, just like the parameter stack, so it can hold many values. But here's the catch: whatever you put on the return stack you must remove again before you get to the end of the definition (the semicolon), because at that point the Forth system will expect 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.

>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 *top* of the return stack without affecting it.
R@ (-- n) Copies the *top* of the return stack without affecting it.
J (-- n) Copies the *third* item of the return stack without affecting it.
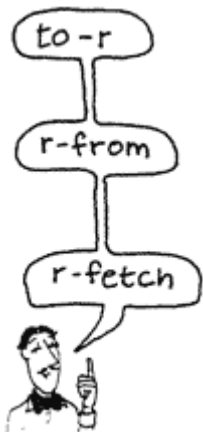
The words >R and R> transfer a value to and from the return stack, respectively. Say we want the following stack effect:

(2 3 1 -- 3 2 1)

this is the phrase that will do it:

>R SWAP R>

Each >R and its corresponding R> must be used together in the same definition.

The other three words — I or R@ and J — only *copy* values from the return stack without removing them. Thus the phrase:

```
>R SWAP R@
```

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

To see how >R, 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).

Operator	parameter stack	return stack
	a b c x	
>R	a b c	x
SWAP ROT	c b a	x
R@	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 R@ *  + R> *  + ;
```

Now test it:

```
2 7 9 3 QUADRATIC↵48 ok
```

An Introduction to Floating-Point Arithmetic

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

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:

$$12 \times 10^6$$

since ten to the sixth power equals one million. In a computer twelve million is 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 though 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 Fixed-Point is Useful

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. Unfortunately, not all processors or controllers offer hardware floating-point support. Therefore, in some environments, programs that use floating-point features are redirected through an emulation library. Emulation code can be up to three times slower than the equivalent fixed-point calculation. Of course, this difference is only really noticeable in programs which have to do a lot of calculations before sending results to a terminal or taking some action. The catch is that code from an emulation library is also many times larger than its fixed-point counterpart, which is quite uneconomical for small embedded controllers and such.

You should note carefully that *when* a processor supports hardware floating-point, it is almost always much faster and more compact than the fixed-point equivalent. The speed difference can be between 3 and 15 times.

Everything you can do with floating-point, you can do with fixed-point too, as we'll show in the following. But there is one thing you should minimize as much as possible, and that is switching back and forth between fixed and floating-point formats. Format conversion and additional scaling steps cost as much or even more time than doing the calculations themselves.

Forth helps programmers use fixed-point by supplying them 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 double-length intermediate result.
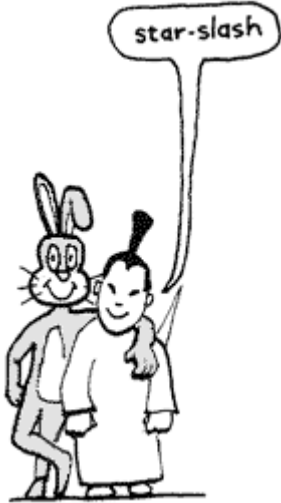
As its name implies, */ performs multiplication, then division. For example, let's say that the stack contains these three numbers:

```
( 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 %
```

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
 0.32 x
 4.50
67.5
72.00
```

*/ is not * and a / thrown together, though. It uses a "double-length intermediate result." What does that mean, you ask?
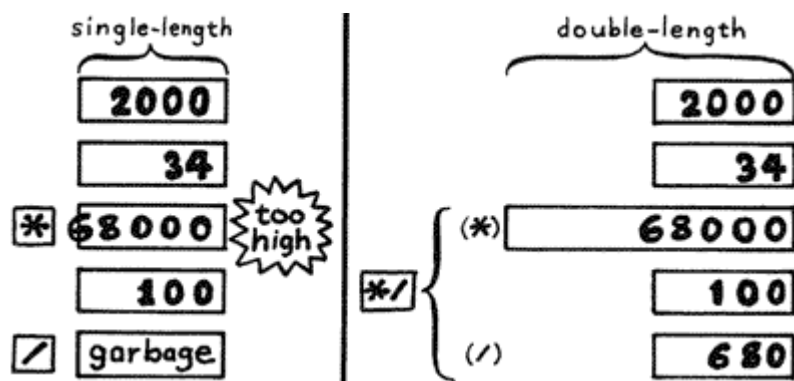
Say you want to compute 34% of 912,345,678. Remember that single-precision operators, like * and /, only work with arguments and results within the range of a single-length number. If you were to enter the phrase:

```
912345678 34 * 100 /
```

you'd get an incorrect result, because the "intermediate result" (in this case, the result of the multiplication), exceeds 2147483647, 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:

```
912345678 34 100 */
```

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

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 by 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 214,748,364 rather than 2,147,483,647. 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 .6666666, 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.
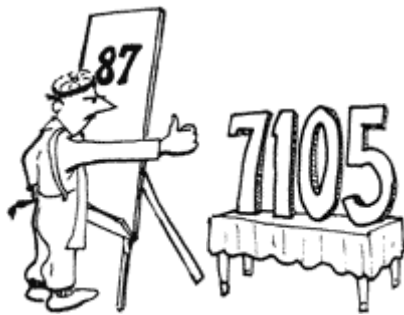
Most computer languages support the first way. "You can't have a fraction like two-thirds hanging around inside a computer," it is believed, "you must express it as .6666666, etc."

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:

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

```
 7,105        ?
 5,145        ?
12,250      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
```

and

```
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."

Another scaling operator in Forth is */MOD:

*/MOD	(n1 n2 n3 -- n-rem n-result)	Multiplies, then divides (n1*n2/n3). Returns the remainder and the quotient. Uses a double-length intermediate result.

star-slash-mod

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 p or the +2. For example, the real value of π is:

```
 3.14159265358979, 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:

pr^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:

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

Try it with a circle whose radius is 10 inches:

```
10 PI .↵314 ok
```

But for even more accuracy, we might wonder if there is a pair of integers beside 3146 and 10000 that is a closer approximation to p. 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 less than 10^{-8}.

Handy Table of Rational Approximations to Various Constants

Number	Approximation	Error
$\pi = 3.141$ ...	355 / 113	8.5×10^{-8}
$\pi = 3.141$ ...	1068966896 / 340262731	3.0×10^{-18}
$\sqrt{2} = 1.414$ ...	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}
$\log(2) / 1.6384 = 0.183$ ...	2040 / 11103	1.1×10^{-8}
$\ln(2) / 16.384 = 0.042$ ...	485 / 11464	1.0×10^{-7}

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 *top* of the return stack without affecting it.
R@	(-- n)	Copies the *top* of the return stack without affecting it.
J	(-- n)	Copies the *third* item of the return stack without affecting it.
*/	(n1 n2 n3 -- n-result)	Multiplies, then divides (n1*n2/n3). Uses a double-length intermediate result.
*/MOD	(n1 n2 n3 -- n-rem n-result)	Multiplies, then divides (n1*n2/n3). 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 when 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 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 points assumed to be in the same place for all values.

Problems — Chapter 5

1. Translate the following algebraic expression into a Forth definition:
2.   `-a b`
3.   `c`

   given (a b c --) [answer]

4. Given these four numbers on the stack:
5.   `( 6 70 123 45 -- )`

   write an expression that prints the largest value. [answer]

6. In "calculator style," convert the following temperatures, using these formulas:
   $^{o}C = (^{o}F - 32) / 1.8$
   $^{o}F = (^{o}C \times 1.8) + 32$
   $^{o}K = ^{o}C + 273$

   (For now, express all arguments and results in whole degrees.)
      0° F in Centigrade
     212° F in Centigrade
     -32° F in Centigrade
     16° C in Fahrenheit
     233° K in Centigrade [answer]

7. 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. [answer]

6. Throw It For a Loop

In Chap. 4 we learned to program the computer to make "decisions" by branching to different parts of a definition depending on the outcome of certain tests. Conditional branching is one of the things that make computers as useful as they are.

In this chapter, we'll see how to write definitions in which execution can conditionally branch back to an earlier part of the same definition, so that some segment will repeat again and again. This type of control construct is called a "loop." The ability to perform loops is probably the most significant thing that makes computers as powerful as they are. If we can program the computer to make out one payroll check, we can program it to make out a thousand of them.

For now we'll write loops that do simple things like printing numbers at your terminal. In later chapters, we'll learn to do much more with them.

Definite Loops — DO...LOOP

One type of loop structure is called a "definite loop." You, the programmer, specify the number of times the loop will loop. In Forth, you do this by specifying a beginning number and an ending number (in reverse order) before the word DO. Then you put the words which you want to have repeated between the words DO and LOOP. For example

```
: TEST   10 0 DO  CR ." Hello "  LOOP ;
```

will print a carriage return and "Hello" ten times, because zero from ten is ten.

```
TEST
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello ok
```

Like an IF...THEN statement, which also involves branching, a DO...LOOP statement must be contained within a (single) definition.

The ten is called the "limit" and the zero is called the "index."

```
FORMULA:
        limit index DO ... LOOP
```

Here is what happens inside a DO...LOOP:

First DO puts the index and the limit on the loop control stack.
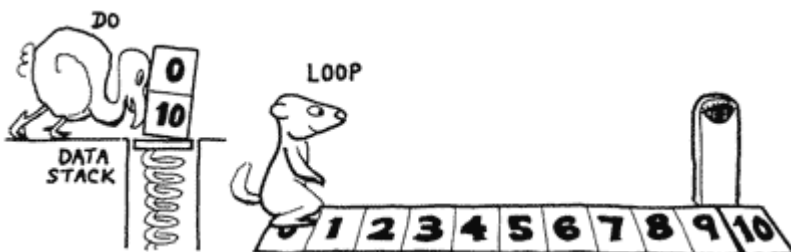
Then execution proceeds to the words inside the loop, up till the word LOOP.
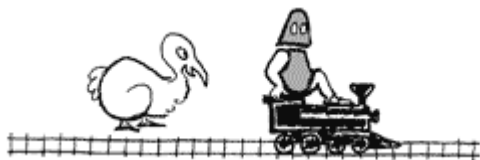
If the index is less than the limit, LOOP reroutes execution back to DO, and adds one to the index.
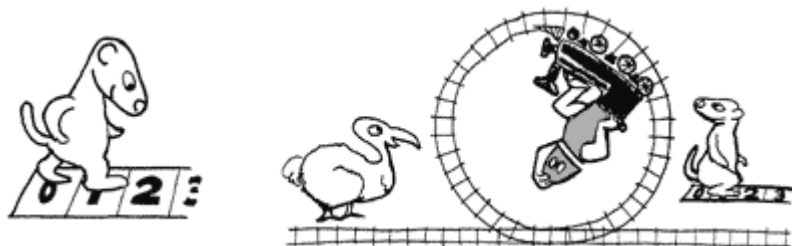
Eventually the index reaches ten, and LOOP lets execution move on to the next word in the definition.
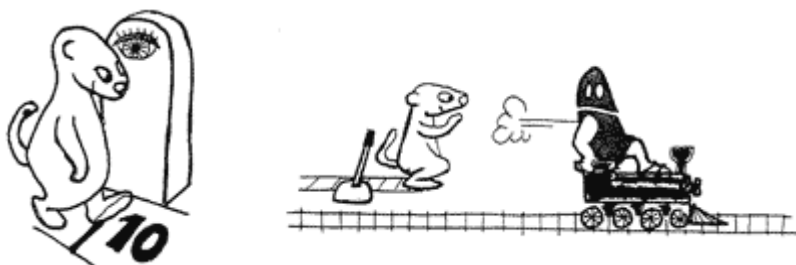
Remember that the Forth word I copies the top of the loop control stack onto the parameter stack. You can use I to get hold of the current value of the index each time around. Consider the definition

```
: DECADE   10 0 DO  I .  LOOP ;
```

which executes like this:

```
DECADE ↵0 1 2 3 4 5 6 7 8 9 ok
```

Of course, you could pick any range of numbers (within the range of -2147483648 to +2147483647):

```
: SAMPLE  -243 -250 DO  I .   LOOP ;
SAMPLE ↵-250 -249 -248 -247 -246 -245 -244 ok
```

Notice that even negative numbers increase by one each time. The limit is always higher than the index.

You can leave a number on the stack to serve as an argument to something inside a DO loop. For instance,

```
: MULTIPLICATIONS   CR 11 1 DO  DUP I * .  LOOP  DROP ;
```

will produce the following results:

```
7 MULTIPLICATIONS↵
7 14 21 28 35 42 49 56 63 70 ok
```

Here we're simply multiplying the current value of the index by seven each time around. Notice that we have to DUP the seven inside the loop so that a copy will be available each time and that we have to DROP it after we come out of the loop.

A compound interest problem gives us the opportunity to demonstrate some trickier stack manipulations inside a DO loop.

Given a starting balance, say $1000, and an interest rate, say 6%, let's write a definition to compute and print a table like this:

```
1000 6 COMPOUND
YEAR 1   BALANCE 1060
YEAR 2   BALANCE 1124
YEAR 3   BALANCE 1191
                            etc.
```

for twenty years.

First we'll load R%, our previously-defined word from , then we'll define

```
: COMPOUND  ( amount interest -- )
      CR SWAP 21 1 DO  ." YEAR " I . 3 SPACES
                       2DUP R% +  DUP ." BALANCE " . CR
              LOOP  2DROP ;
```
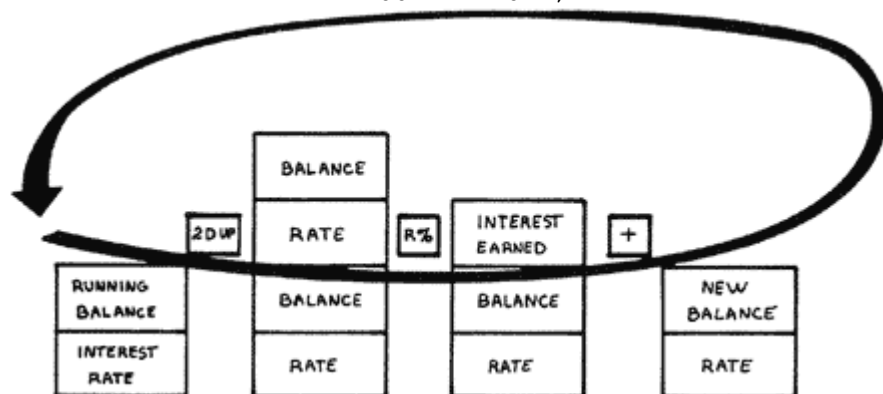

Each time through the loop, we do a 2DUP so that we always maintain a running balance and an unchanged interest rate for the next go-round. When we're finally done, we 2DROP them.

Getting IFfy

The index can also serve as a condition for an IF statement. In this way you can make something special happen on certain passes through the loop but not on others. Here's a simple example:

```
: RECTANGLE  256 0 DO   I 16 MOD 0= IF  CR  THEN
                           ." *"
               LOOP ;
```

RECTANGLE will print 256 stars, and at every sixteenth star it will also perform a carriage return at your terminal. The result should look like this:

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

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

And here's an example from the world of nursery rhymes. We'll let you figure this one out.

```
: POEM  CR 11 1 DO  I . ." Little "
                 I 3 MOD 0= IF ." indians " CR THEN
            LOOP
        ." indian boys. " ;
```

Nested Loops

In the last section we defined a word called MULTIPLICATIONS, which contains a DO...LOOP. If we wanted to, we could put MULTIPLICATIONS inside another DO...LOOP, like this:

```
: TABLE  CR 11 1 DO  I MULTIPLICATIONS  LOOP ;
```

Now we'll get a multiplication table that looks like this:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
                                etc.
10 20 30 40 50 60 70 80 90 100
```

because the I in the outer loop supplies the argument for MULTIPLICATIONS.

You can also nest DO loops inside one another all in the same definition:

```
: TABLE  CR 11 1 DO
             11 1 DO  I J *  5 U.R  LOOP
          CR LOOP ;
```

Notice this phrase in the inner loop:

```
I J *
```

In Chap. 5 we mentioned that the word J copies the third item on the loop control stack onto the parameter stack. It so happens that in this case the third item on the loop control stack is the index of the *outer* loop.

Thus the phrase "I J *" multiplies the two indexes to create the value in the table.

Now what about this phrase?

```
5 U.R
```

This is nothing more than a fancy . that is used to print numbers in table form so that they line up vertically. The five represents the number of spaces we've decided each column in the table should be. The output of the new table will look like this:

```
1    2    3    4    5    6    7    8    9   10
2    4    6    8   10   12   14   16   18   20
3    6    9   12   15   18   21   24   27   30   etc.
```

Each number takes five spaces, no matter how many digits it contains. (U.R stands for "*unsigned* number-print, *right* justified." The term "unsigned," you may recall, means you cannot use it for negative numbers.)

+LOOP

If you want the index to go up by some number other than one each time around, you can use the word +LOOP instead of LOOP. +LOOP expects on the stack the number by which you want the index to change. For example, in the definition

```
: PENTAJUMPS  50 0 DO  I .  5 +LOOP ;
```

the index will go up by five each time, with this result:

```
PENTAJUMPS↵0 5 10 15 20 25 30 35 40 45 ok
```

while in

```
: FALLING  -10 0 DO  I .  -1 +LOOP ;
```

the index will go down by one each time, with this result:

```
FALLING↵0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 ok
```
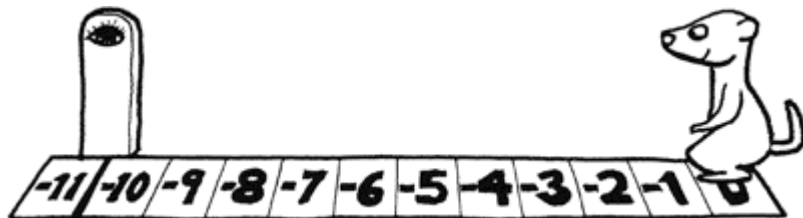

The argument for +LOOP, which is called the "increment," can come from anywhere, but it must be put on the stack each time around. Consider this experimental example:

```
: INC-COUNT  DO  I . DUP +LOOP  DROP ;
```

There is no increment inside the definition; instead, it will have to be on the stack when INC-COUNT is executed, along with the limit and index. Watch this:

Step up by one:

```
1 5 0 INC-COUNT↵0 1 2 3 4 ok
```

Step up by two:

```
2 5 0 INC-COUNT↵0 2 4 ok
```

Step down by three:

```
-3 -10 10 INC-COUNT↵10 7 4 1 -2 -5 -8 ok
```

Our next example demonstrates an increment that changes each time through the loop.

```
: DOUBLING  32767 1 DO  I . I +LOOP ;
```

Here the index itself is used as the increment (I +LOOP), so that starting with one, the index doubles each time, like this:

```
DOUBLING↵
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 ok
```
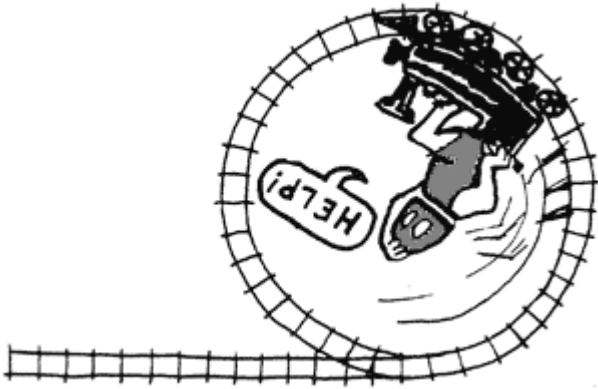
Notice that in this example we don't ever want the argument for +LOOP to be zero, because if it were we'd never come out of the loop. We would have created what is known as an "infinite loop."

DOing it — Forth Style

There are a few things to remember before you go off and write some DO loops of your own.

First, keep this simple guide in mind:

Reasons for termination

Execution makes its exit from a loop when, in going up, the index has *reached* or *passed* the limit.

Or, when in going down, the index has passed the limit — not when it has merely reached it.

But a DO loop always executes *at least once* (this example will loop millions of times on a true ANS Forth system, so be prepared):

```
: TEST  100 10 DO  I . -1 +LOOP ;
TEST↵10 9 8 7 ...
```

Second, remember that the words DO and LOOP are branching commands and that therefore they can only be executed inside a definition. This means that you cannot design/test your loop definitions in "calculator style" unless you simulate the loop yourself.

Let's see how a fledgling Forth programmer might go about design/testing the definition of COMPOUND (from the first section of this chapter). Before adding the ." messages, the programmer might begin by jotting down this version on a piece of paper:

```
: COMPOUND  ( amt int -- )
        SWAP 21 1 DO  I . 2DUP R% + DUP . CR LOOP  2DROP ;
```

The programmer might test this version at the terminal, using **.** or .S to check the result of each step. The "conversation" might look like this:

```
           1000 6 SWAP   .S
first      6 1000  ok
time
through    2DUP   .S
           6 1000 6 1000    ok
```

In simulation, the programmer omits the <limit> <index> DO phrase and any references to it.

```
           R%   .S
           6 1000 60  ok

           +  .S
           6 1060  ok
```

In simulation, the programmer can omit the DUP . phrase.

```
second     2DUP R% +  .S
time       6 1124  ok

           2DROP  .S
           Stack empty
```

Everything seems to be working, so the programmer pretends the last loop has finished and checks that the stack is clear.

A Handy Hint:
How to Clear the Stack

Sometimes a beginner will unwittingly write a loop which leaves a whole lot of numbers on the stack. For example

```
: FIVES  100 0 DO  I 5 . LOOP ;
```

instead of

```
: FIVES  100 0 DO  I 5 * . LOOP ;
```

If you see this happen to anyone (surely it will never happen to you!) and if you see the beginner typing in an endless succession of dots to clear the stack, recommend typing in

```
XX
```

XX is not a Forth word, so the text interpreter will execute the word ABORT", which among other things clears all stacks. The beginner will be endlessly grateful.

Indefinite Loops

While DO loops are called definite loops, Forth also supports "indefinite" loops. This type of loop will repeat indefinitely or until some event occurs. A standard form of indefinite loop is

```
BEGIN ... UNTIL
```

The BEGIN...UNTIL loop repeats until a condition is "true."

The usage is

```
BEGIN xxx f UNTIL
```
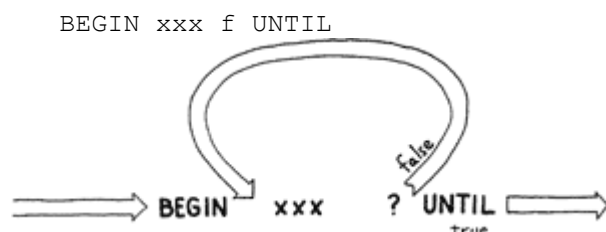

where "xxx" stands for the words that you want to be repeated, and "f" stands for a flag. As long as the flag is zero (false), the loop will continue to loop, but when the flag becomes non-zero (true), the loop will end.

An example of a definition that uses a BEGIN...UNTIL statement is one we mentioned earlier, in our washing machine example:

```
: TILL-FULL  BEGIN ?FULL UNTIL ;
```

which we used in the higher-level definition

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

?FULL will be defined to electronically check a switch in the washtub that indicates when the water reaches the correct level. It will return zero if the switch is not activated and a one if it is. TILL-FULL does nothing but repeatedly make this test over and over (millions of times per second) until the switch is finally activated, at which time execution will come out of the loop. Then the ; in TILL-FULL will return the flow of execution to the remaining words in FILL, and the water faucets will be turned off.

Sometimes a programmer will deliberately want to create an infinite loop. In Forth, the best way is with the form

```
: BEGIN xxx 0 UNTIL
```

The zero supplies a "false" flag to the word UNTIL, so the loop will repeat eternally.

Beginners usually want to avoid infinite loops, because executing one means that they lose control of the computer (in the sense that only the words inside the loop are being executed). But infinite loops do have their uses. For instance, the text interpreter is part of an infinite loop called QUIT, which waits for input, interprets it, executes it, prints "ok," then waits for input once again. In most microprocessor-controlled machines, the highest-level definition contains an infinite loop that defines the machine's behavior.

Another form of indefinite loop is used in this format:

```
BEGIN xx f WHILE yyy REPEAT
```

Here the test occurs halfway through the loop rather than at the end. As long as the test is true, the flow of execution continues with the rest of the loop, then returns to the beginning again. If the test is false, the loop ends.
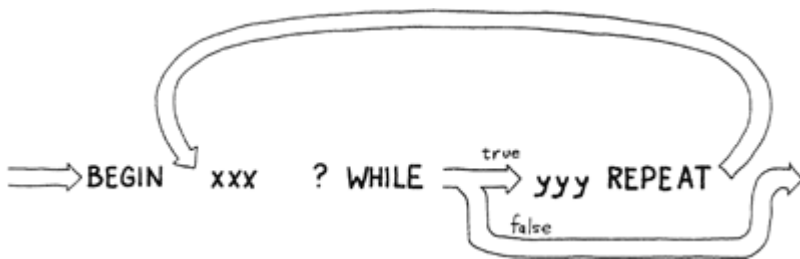
Notice that the effect of the test is opposite that in the BEGIN...UNTIL construction. Here the loop repeats *while* something is true (rather than *until* it's true).

The indefinite loop structures lend themselves best to cases in which you're waiting for some external event to happen, such as the closing of a switch or thermostat, or the setting of a flag by another part of an application that is running simultaneously. So for now, instead of giving examples, we just want you to remember that the indefinite loop structures exist.

The Indefinitely Definite Loop

There is a way to write a definite loop so that it stops short of the prescribed limit if a truth condition changes state, by using the word LEAVE. LEAVE causes the loop to end immediately.

Watch how we rewrite our earlier definition of COMPOUND. Instead of just letting the loop run twenty times, let's get it to quit after twenty times or as soon as our money has doubled, whichever occurs first.

We'll simply add this phrase:

```
2000 > IF  LEAVE  THEN
```

like this:

```
: DOUBLED
  6 1000 21 1 DO  CR ." YEAR " I 2 U.R
        2DUP R% +  DUP ."    BALANCE " .
        DUP 2000 > IF  CR CR ." more than doubled in "
                         I . ." years " LEAVE
                THEN
   LOOP 2DROP ;
```

The result will look like this:

```
DOUBLED
     YEAR  1   BALANCE 1060
     YEAR  2   BALANCE 1124
     YEAR  3   BALANCE 1191
     YEAR  4   BALANCE 1262
     YEAR  5   BALANCE 1338
     YEAR  6   BALANCE 1418
     YEAR  7   BALANCE 1503
     YEAR  8   BALANCE 1593
     YEAR  9   BALANCE 1609
     YEAR 10   BALANCE 1790
     YEAR 11   BALANCE 1897
     YEAR 12   BALANCE 2011

     more than doubled in 12 years ok
```

One of the problems at the end of this chapter asks you to rework DOUBLED so that it expects the parameters of interest and starting balance, and computes by itself the double balance that LEAVE will try to reach.

Two Handy Hints: PAGE and QUIT

To give a neater appearance to your loop outputs (such as tables and geometric shapes), you might want to clear the screen first by using the word PAGE. You can execute PAGE interactively like this:

```
PAGE RECTANGLE
```

which will clear the screen before printing the rectangle that we defined earlier in this chapter. Or you could put PAGE at the beginning of the definition. like this:

```
: RECTANGLE  PAGE 256 0 DO  I 16 MOD
     0= IF  CR  THEN  ." *"  LOOP ;
```

If you don't want the "ok" to appear upon completion of execution, use the word QUIT. Again, you can use QUIT interactively:

```
RECTANGLE QUIT
```

or you can make QUIT the last word in the definition (just before the semicolon).

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

DO ... LOOP	DO: (limit index --) LOOP: (--)	Sets up a finite loop, given the index range.
DO ... +LOOP	DO: (limit index --) +LOOP: (--)	Like DO ... LOOP except adds the value of n (instead of always one) to the index.
LEAVE	(--)	Terminate the loop immediately.
BEGIN ... UNTIL	UNTIL: (f --)	Sets up an indefinite loop which ends when f is true.
BEGIN xxx WHILE yyy REPEAT	WHILE: (f --)	Sets up an indefinite loop which always executes xxx and also yyy if f is true. Ends when f is false.
U.R	(u width --)	Prints the unsigned single-length number, right-justified within the field width.
PAGE	(--)	Clears the terminal screen and resets the terminal's cursor to the upper left-hand corner.
QUIT	(--)	Terminates execution for the current task and returns control to the terminal.

Review of Terms

definite loop
> a loop structure in which the words contained within the loop repeat a definite number of times. In Forth, this number depends on the starting and ending counts (index and limit) which are placed on the stack prior to the execution of the word DO.

infinite loop
> a loop structure in which the words contained within the loop continue to repeat without any chance of an external event stopping them, except for closing the Forth window or shutting down or resetting the computer.

indefinite loop
> a loop structure in which the words contained within the loop continue to repeat until some truth condition changes state (true-to-false or false-to-true). In Forth, the indefinite loops begin with the word BEGIN.

Problems — Chapter 6

In Problems 1 through 6, you will create several words which will print out patterns of stars (asterisks). These will involve the use of DO loops and BEGIN...UNTIL loops.

1. First create a word named STARS which will print out n stars on the same line, given n on the stack:

```
2.      10 STARS↵********** ok
```

[answer]

3. Next define BOX which prints out a rectangle of stars, given the width and height (number of lines), using the stack order (width height --).

```
4.      10 3 BOX
5.          **********
6.          **********
7.          ********** ok
```

[answer]

8. Now create a word named \STARS which will print a skewed array of stars (a rhomboid), given the height on the stack. Use a DO loop and, for simplicity, make the width a constant ten stars.

```
9.      3 \STARS
10.          **********
11.           **********
12.            ********** ok
```

[answer]

13. Now create a word which slants the stars the other direction: call it /STARS. It should take the height as a stack input and use a constant ten width. Use a DO loop. [answer]
14. Now redefine this last word, using a BEGIN...UNTIL loop. [answer]
15. Write a definition called DIAMONDS which will print out the given number of diamond shapes, as shown in this example.

```
16.      2 DIAMONDS
17.                        *
18.                       ***
19.                      *****
20.                     *******
21.                    *********
22.                   ***********
23.                  *************
24.                 ***************
25.                *****************
26.               *******************
27.              *********************
28.               *****************
29.                ***************
30.                 *************
31.                  ***********
32.                   *********
33.                    *******
34.                     *****
35.                      ***
36.                       *
37.                       *
38.                      ***
39.                     *****
40.                    *******
41.                   *********
42.                  ***********
43.                 *************
44.                ***************
45.               *****************
46.              *******************
47.              *********************
48.               *****************
49.                ***************
50.                 *************
51.                  ***********
52.                   *********
53.                    *******
54.                     *****
55.                      ***
```

```
56.                          *
```

[answer]

57. In our discussion of LEAVE we gave an example which computed 6% compound interest on a starting balance of $1000 for 20 years or until the balance had doubled, whichever came first. Rewrite this definition so that it will expect a starting balance and interest rate on the stack and will LEAVE when this starting balance has doubled. [answer]

58. Define a word called ** that will compute exponential values, like this:

```
59.      7 2 ** . 49 ok
```

(seven squared)

```
   2 4 ** . 16 ok
```

(two to the fourth power). For simplicity, assume positive exponents only (but make sure ** works correctly when the exponent is one — the result should be the number itself). [answer]

7. A Number of Kinds of Numbers

So far we've only talked about signed single-length numbers. In this chapter we'll introduce unsigned numbers and double-length numbers, as well as a whole passel of new operators to go along with them.
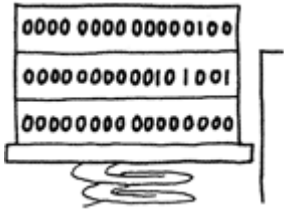
This chapter is divided in two sections:

For beginners — this section explains how a computer looks at numbers and exactly what is meant by the terms signed or unsigned and by single length or double length.

For everyone — this section continues our discussion of Forth for beginners and experts alike, and explains how Forth handles signed and unsigned, single- and double-length numbers.

Section 1 — For Beginners

Signed versus Unsigned Numbers

All digital computers store numbers in binary form. In Forth, the we speak of the stack in terms of the implementation's "cell size" (common sizes are 16, 32, and 64 bits, but other cell sizes are possible). Below is a view of the least significant sixteen bits of a cell, showing the value of each bit:
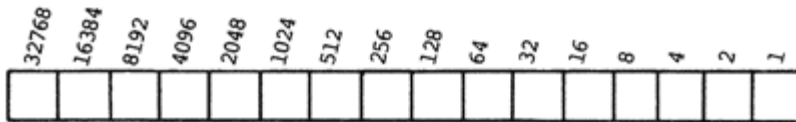
If every bit were to contain a 1, the total for just these sixteen bits would be 65,535. Thus in 32 bits we can express any value between 0 and 4,294,967,295. Because this kind of number does not let us express negative values, we call it an "unsigned number." We indicate unsigned numbers with the letter "u" in our tables and stack notations.

But what about negative numbers? In order to be able to express a positive or negative number, we need to sacrifice one bit that will essentially indicate sign. This bit is the one at the far left, the "high-order bit." In 31 bits we can express a number as high as 2,147,483,647. When the sign bit contains 1, then we can go an equal distance back into the negative numbers. Thus within 32 bits we can represent any number from -2,147,483,648 to +2,147,483,647. This should look familiar to you as the range of a single-length number, which we denote with the letter "n."
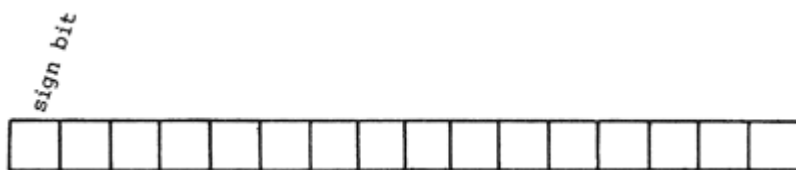
Before we leave you with any misconceptions, we'd better clarify the way negative numbers are represented. You might think that it's a simple matter of setting the sign bit to indicate whether a number is positive or negative, but it doesn't work that way.

To explain how negative numbers are represented, let's return to decimal notation and examine a counter such as that found on many web pages.

`20400`

Let's say the counter has three digits, not five. As more people visit the page, the counter wheels turn and the number increases. Starting once again with the counter at 0, now imagine you badly regret having visited the page and could "un-visit" it by rolling the counter wheels backward. The first number you see is 999, which is, in a sense, the same as -1. The next number will be 998, which is the same as -2, and so on.

The representation of signed numbers in a computer is similar.

Starting with the 32-bit number

`0000,0000,0000,0000,0000,0000,0000,0000`

and going backwards one number, we get

`1111,1111,1111,1111,1111,1111,1111,1111    (thirty-two ones)`

which stands for 4,294,967,295 in unsigned notation as well as for -1 in signed notation. The number

`1111,1111,1111,1111,1111,1111,1111,1110`

which stands for 4,294,967,294 in unsigned notation, represents -2 in signed notation.

Here's a chart that shows how a binary number on the stack can be used either as an unsigned number or as a signed number:

as an unsigned number		as a signed number
65535	1111 1111 1111 1111 1111 1111 1111 1111	
...	...	
32768	1000 0000 0000 0000 0000 0000 0000 0000	
32767	0111 1111 1111 1111 1111 1111 1111 1111	32767
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0
	1111 1111 1111 1111 1111 1111 1111 1111	-1
	...	...
	1000 0000 0000 0000 0000 0000 0000 0000	-32768

This bizarre-seeming method for representing negative values makes it possible for the computer to use the same procedures for subtraction as for addition.

To show how this works, let's take a very simple problem:

```
  2
 -1
```

Subtracting one from two is the same as adding two plus negative one. In single-length binary notation, the two looks like this:

`0000,0000,0000,0000,0000,0000,0000,0010`

while negative-one looks like this:

```
1111,1111,1111,1111,1111,1111,1111,1111
```

The computer adds them up the same way we would on paper; that is when the total of any column exceeds one, it carries a one into the next column. The result looks like this:

```
  0000,0000,0000,0000,0000,0000,0000,0010
+1111,1111,1111,1111,1111,1111,1111,1111
10000,0000,0000,0000,0000,0000,0000,0001
```

As you can see, the computer had to carry a one into every column all the way across, and ended up with a one in the thirty-third place. But since the stack is only thirty-two bits wide, the result is simply

```
0000,0000,0000,0000,0000,0000,0000,0001
```

which is the correct answer, one.

We needn't explain how the computer converts a positive number to negative, but we will tell you that the process is called "two's complementing."

Arithmetic Shift

While we're on the subject of how a computer performs certain mathematical operations, we'll explain what is meant by the mysterious phrases back in Chap. 5: "arithmetic left shift" and "arithmetic right shift."

	A Forth Instant Replay	
2*	(n -- n*2)	Multiplies by two (arithmetic left shift)
2/	(n -- n/2)	Divides by two (arithmetic right shift)
LSHIFT	(n u -- n*2^u)	Logical left shift over u positions
RSHIFT	(n -- n/2^-u)	Logical right shift over u positions

To illustrate, let's pick a number, say six, and write it in binary form:

```
0000,0000,0000,0000,0000,0000,0000,0110
```

(4+2). Now let's shift every digit one place to the left, and put a zero in the vacant place in the one's column.

```
0000,0000,0000,0000,0000,0000,0000,1100
```

This is the binary representation of twelve (8+4), which is exactly double the original number. This works in all cases, and it also works in reverse. If you shift every digit one place to the *right* and fill the vacant digit with a zero, the result will always be *half* of the original value.

In arithmetic shift, the sign bit does not get shifted. This means that a positive number will stay positive and a negative number will stay negative when you divide or multiply it by two.

When the high-order bit shifts with all the other bits, the term is "logical shift." In Forth you can do a logical shift of up to 32 places with the words LSHIFT and RSHIFT.

The important thing for you to know is that most computers can shift digits much more quickly than they can go through all the folderol of normal division or multiplication. When speed is critical, it's much better to say

```
2*
```

than

```
2  *
```

and it may even be better to say

```
2*  2*  2*
```

than

```
8  *
```

depending on your particular model of computer, but this topic is getting too technical for right now.

An Introduction to Double-length Numbers

A double-length number is just what you probably expected it would be: a number that is represented in two cells instead of one. In a 32-bit Forth implementation, signed double-length numbers have a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (signed) or 0 to 18,446,744,073,709,551,615 (unsigned).
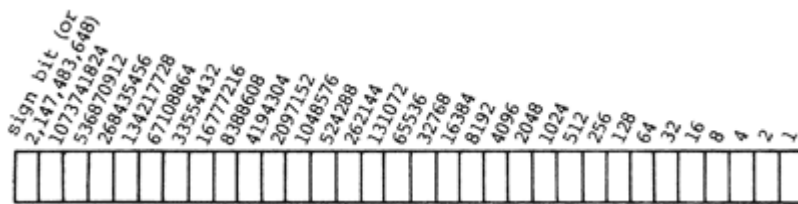
In Forth, a double-length number takes the place of two single-length numbers on the stack. Operators like 2DUP are useful either for double-length numbers or for pairs of single-length numbers.

One more thing we should explain: to the non-Forth-speaking computer world, the term "double word" could mean a 32-bit value, or four bytes. But in Forth, "word" means a defined command. So in order to avoid confusion, Forth programmers refer to a single number on the stack as a "cell." A double-length number requires two cells.

Other Number Bases

As you get more involved in programming, you'll need to employ other number bases besides decimal and binary, particularly hexadecimal (base 16) and possible octal (base 8). Since we'll be talking about these two number bases later on in this chapter, we think you might like an introduction now.

Computer people began using hexadecimal and octal numbers for one main reason: computers think in binary and human beings have a hard time reading long binary numbers. For people, it's much easier to convert binary to hexadecimal than binary to decimal, because sixteen is an even power of two, while ten is not. The same is true with octal. So programmers usually use hex or octal to express the binary numbers that the computer uses for things like addresses and machine codes. Hexadecimal (or simply "hex") looks strange at first since it uses the letters A through F.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1

2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Let's take a single-length binary number:

```
00000000000000000111101110100001
```

To convert this number to hexadecimal, we first subdivide it into eight units of four bits each:

```
| 0000 | 0000 | 0000 | 0000 | 0111 | 1011 | 1010 | 0001 |
```

then convert each 4-bit unit to its hex equivalent:

```
|0|0|0|0|7|B|A|1|
```

or simply 7BA1.

Octal numbers use only the numerals 0 through 7. Because nowadays most computers use hexadecimal representation, we'll skip an octal conversion example.

We'll have more on conversions in the section titled "Number Conversions" later in this chapter.

The ASCII Character Set

If the computer uses binary notation to store numbers, how does it store characters and other symbols? Binary, again, but in a special code that was adopted as an industry standard many years ago. The code is called the American Standard Code for Information Interchange, usually abbreviated ASCII.

Table 7-1 shows each ASCII character in the system, its ISO 646-1983, ISO 7-bit coded character set for information interchange, International Reference Version equivalent (IRV), and its hexadecimal form.

The characters in the first column (ASCII codes 0-1F hex) are called "control characters" because they indicate that the terminal or computer is supposed to do something like ring its bell, backspace, start a new line, etc. The remaining characters are called "printing characters" because they produce visible characters including letters, the numerals zero through nine, all available symbols and even the blank space (hex 20). The only exception is DEL (hex 7F), which is a signal to the computer to ignore the last character sent.

In Chap. 1 we introduced the word EMIT. EMIT takes an ASCII code on the stack and sends it to the terminal so that the terminal will print it as a character. For example,

```
65 EMIT↵>A ok
66 EMIT↵B ok
```

etc. (We're using the decimal, rather than the hex, equivalent because that's what your computer is most likely expecting right now.)

Why not test EMIT on every printing character, "automatically"?

```
: PRINTABLES  127 32 DO  I EMIT SPACE  LOOP ;
```

PRINTABLES will emit every printable character in the ASCII set; that is, the characters from decimal 32 to decimal 126. (We're using the ASCII codes as our DO loop index.)

```
PRINTABLES↵  ! " # $ % & ' ( ) * + , - . / ...ok
```

Table 7.1 — Standard graphic characters & equivalents							
Hex ASCII	**Hex ASCII**	**Hex IRV ASCII**	**Hex IRV ASCII**	**Hex IRV ASCII**	**Hex IRV ASCII**	**Hex IRV ASCII**	**Hex IRV ASCII**
00 NUL	10 DLE	20	30 0 0	40 @ @	50 P P	60 ' '	70 p p
01 SOH	11 DC1	21 ! !	31 1 1	41 A A	51 Q Q	61 a a	71 q q
02 STX	12 DC2	22 " "	32 2 2	42 B B	52 R R	62 b b	72 r r
03 ETX	13 DC3	23 # #	33 3 3	43 C C	53 S S	63 c c	73 s s
04 EOT	14 DC4	24 - $	34 4 4	44 D D	54 T T	64 d d	74 t t
05 ENQ	15 NAK	25 % %	35 5 5	45 E E	55 U U	65 e e	75 u u
06 ACK	16 SYN	26 & &	36 6 6	46 F F	56 V V	66 f f	76 v v
07 BEL	17 ETB	27 ' '	37 7 7	47 G G	57 W W	67 g g	77 w w
08 BS	18 CAN	28 ((	38 8 8	48 H H	58 X X	68 h h	78 x x
09 HT	19 EM	29))	39 9 9	49 I I	59 Y Y	69 i i	79 y y
0A LF	1A SUB	2A * *	3A : :	4A J J	5A Z Z	6A j j	7A z z
0B VT	1B ESC	2B + +	3B ; ;	4B K K	5B [[	6B k k	7B { {
0C FF	1C FS	2C , ,	3C < <	4C L L	5C \ \	6C l l	7C | |
0D CR	1D GS	2D - -	3D = =	4D M M	5D]]	6D m m	7D } }
0E SM	1E RS	2E . .	3E > >	4E N N	5E ^ ^	6E n n	7E ~ ~
0F SI	1F US	2F / /	3F ? ?	4F O O	5F	6F o o	

Some control (non-printing) characters that are good to know include the following:

name	operation	decimal equivalent
BS	backspace	8
LF	line feed	10
CR	carriage return	13

Experiment with these control characters, and see what they do.

ASCII is designed so that each character can be represented by one byte. The tables in this book use the letter "c" to indicate a byte value that is being used as a coded ASCII *c*haracter.

Bit Logic

The words AND and OR (which we introduced in) use "bit logic"; that is, each bit is treated independently, and there are no "carries"from one bit-place to the next. For example, let's see what happens when we AND these two binary numbers:

```
0000,0000,0000,0000,0000,0000,1111,1111
0000,0000,0000,0000,0110,0101,1010,0010  AND
```

```
0000,0000,0000,0000,0000,0000,1010,0010
```

For any result-bit to be "1," the respective bits in *both* arguments must be "1." Notice in this example that the argument on top contains all zeroes in the high-order bytes and all ones in the low-order byte. The effect on the second argument in this example is that the low-order eight bits are kept but the high-order twenty-four bits are all set to zero. Here the first argument is being used as a "mask," to mask out the high-order bytes of the second argument.

The word OR also uses bit logic. For example,

```
1000,0100,0010,0001,1000,1001,0000,1001
0110,0110,0110,0110,0000,0011,1100,1000   OR
1110,0110,0110,0111,1000,1011,1100,1001
```

A "1" in either argument produces a "1" in the result. Again, each column is treated separately,with no carries.

By clever use of masks, we could even use a 32-bit value to hold 32 separate flags. For example,we could find out whether this bit

```
1000,0100,0010,0001,1000,1001,0000,1001
                        ^
```

is "1" or "0" by masking out all other flags, like this:

```
1000,0100,0010,0001,1000,1001,0000,1001
0000,0000,0000,0000,1000,0000,0000,0000   AND
0000,0000,0000,0000,1000,0000,0000,0000
```

Since the bit was "1," the result is "true." Had it been "0," the result would have been "0" or"false."

We could set the flag to "0" without affecting the other flags by using this technique:

```
1000,0100,0010,0001,1000,1001,0000,1001
1111,1111,1111,1111,0111,1111,1111,1111   AND
1000,0100,0010,0001,0000,1001,0000,1001                ^
```

We used a mask that contains all "1"s except for the bit we wanted to set to "0." We can set the same flag back to "1" by using this technique:

```
1000,0100,0010,0001,0000,1001,0000,1001
0000,0000,0000,0000,1000,0000,0000,0000   OR
1000,0100,0010,0001,1000,1001,0000,1001                ^
```

Section 2 — For Everybody

Signed and Unsigned Numbers

Back in Chap. 1 we introduced the word NUMBER. If the word FIND can't find an incoming string in the dictionary, it hands it over to the word NUMBER. NUMBERthen attempts to convert the string into a number expressed in binary form. If NUMBERsucceeds, it pushes the binary equivalent onto the stack.
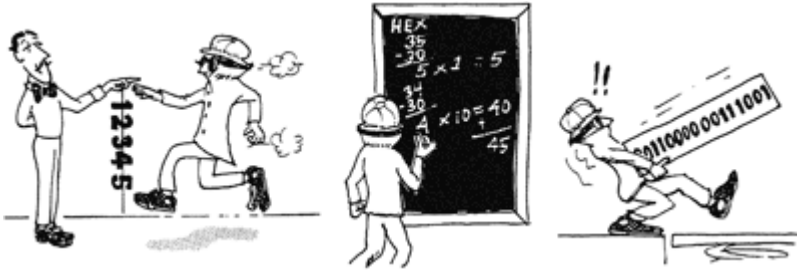
For Beginners

This means that NUMBER does not check whether the number you've entered as a single-length number exceeds the proper range. If you enter a giant number, NUMBER converts it but only saves the least significant thirty-two digits.

NUMBER does not do any range-checking. Because of this, NUMBER can convert either signed or unsigned numbers.

For instance, if you enter any number between 2147483648 and 4294967295, NUMBER will convert it as an unsigned number. Any value between -2147483648 and -1 will be stored as a two's-complement integer.

This is an important point: the stack can be used to hold either signed or unsigned numbers. Whether a binary value is interpreted as signed or unsigned depends on the operators that you apply to it. You decide which form is better for a given situation, then stick to your choice.

We've introduced the word ., which prints a value on the stack as a signed number:

```
4294967295 .↵-1 ok
```

The word U. prints the binary representation as an unsigned number:

```
4294967295 U.↵4294967295 ok
```
U. (u --)   Prints the unsigned single-length number, followed by a space.

In this book the letter "n" signifies *signed* single-length numbers, while the letter "u" signifies *unsigned* single-length numbers. (We've already introduced U.R, which prints an unsigned single-length number right-justified within a given column width.)

Here is a table of additional words that use unsigned numbers:

UM*	(u1 u2 -- ud)	Multiplies two single-length numbers. Returns a double-length result. All values are unsigned.
UM/MOD	(ud u1 -- u2 u3)	Divides a double-length by a single-length number. Returns a single-length quotient u2 and remainder u3. All values are unsigned.
U<	(u1 u2 -- f)	Leaves true if u1 < u2, where both are treated as single-length unsigned integers.

Number Bases

When you first start Forth, all number conversions use base ten (decimal), for both input and output.
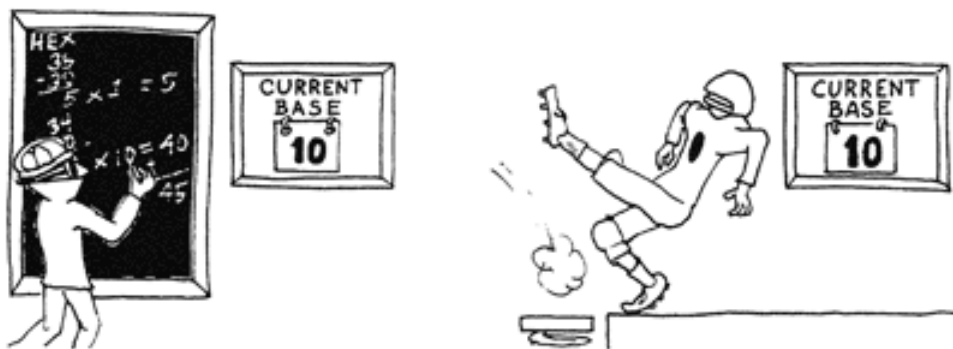
You can easily change the base by executing one of the following commands:

HEX	(--)	Sets the base to sixteen.
OCTAL	(--)	Sets the base to eight (available on some systems).
DECIMAL	(--)	Returns the base to ten.

When you change the number base, its stays changed until you change it again. So be sure to declare DECIMAL as soon as you're done with another number base.

These commands make it easy to do number conversions in "calculator style."

For example, to convert decimal 100 into hexadecimal, enter

```
DECIMAL 100 HEX .↵64 ok
```

To convert hex F into decimal (remember you are already in hex), enter

```
0F DECIMAL .↵15 ok
```

Make it a habit, starting right now, to precede each hexadecimal value with a zero, as in

```
0A 0B 0F
```

This practice avoids mix-ups with possibly predefined words as DEADBEEF, BAD, DEC etc.

Handy Hint

A definition of BINARY — or *Any*-ARY

Beginners who want to see what numbers look like in binary notation may enter this definition:

```
: BINARY  2 BASE ! ;
```

The new word BINARY will operate just like OCTAL or HEX but will change the number base to *two*. On systems which do not have the word OCTAL, experimenters may define

```
: OCTAL   8 BASE ! ;
```

Double-length Numbers

Most ANS Forth systems support double-length numbers to some degree. The standard way to enter a double-length number onto the stack (whether from the keyboard or from a file) is to punctuate it with a period at the end. When the text interpreter processes a number that is immediately followed by a decimal point and is not found as a definition name, it is converted to a double-cell number.

For example, when you type

```
200000.↵
```
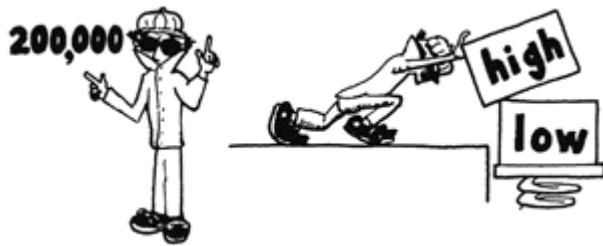

NUMBER recognizes the period at the as a signal that this value should be converted to double-length. NUMBER then pushes the value onto the stack as two consecutive "cells" (cell is the Forth term for a single-length item on the stack), the high order cell on top.

Some Forth implementations (including SwiftForth) will convert any number that contains the following characters as a double number:

```
 +  ,  -  .  /  :
```

The Forth word D. prints a double-length number without any punctuation.

D. (d --) Prints the signed double-length number, followed by one space.

In this book, the letter "d" stands for a double-length signed integer.

For example, having entered a double-length number, if you were now to execute D.,the computer would respond:

```
D.↵200000 ok
```

In the next section we'll show you how to define your own equivalents to D. which will print whatever punctuation you want along with the number.

Number Formatting — Double-length Unsigned

```
$200.00   12/31/80   999-6784   6:32:59   98.6
```

The above numbers represent the kinds of output you can create by defining your own "number-formatting words" in Forth. This section will show you how.

The simplest number-formatting definition we could write would be

```
: UD.   <# #S #> TYPE ;
```

UD. will print an unsigned double-length number. The words <# and #> (respectively pronounced *bracket-number* and *number-bracket*) signify the beginning and the end of the number-conversion process. In this definition, the entire conversion is being performed by the single word #S (pronounced *numbers*). #S converts the value on the stack into ASCII characters. It will only produce as many digits as are necessary to represent the number; it will not produce leading zeroes. But it always produces at least one digit, which will be zero if the value was zero. For example:

```
12,345 UD.↵12345ok
12. UD.↵12ok
0. UD.↵0ok
```

The word TYPE prints the characters that represent the number at your terminal. Notice that there is no space between the number and the "ok." To get a space,you would simply add the word SPACE, like this:

```
: UD.   <# #S #> TYPE SPACE ;
```

Now let's say we have a phone number on the stack, expressed as a double-length unsigned integer. For example, we may have typed in:

```
999-6784
```

(remember that the hyphen tells NUMBER to treat this as a double-length value). We want to define a word that will format this value back as a phone number. Let's call it .PH# (for "print the *ph*one number") and define it thus:

```
: .PH#   <# # # # # [CHAR] - HOLD #S #> TYPE SPACE ;
```


Our definition of .PH# has everything that UD. has, and more. The Forth word # (pronounced *number*) produces a single digit only. A number-formatting definition is reversed from the order in which the number will be printed, so the phrase

```
# # # #
```

produces the right-most four digits of the phone number.

Now it's time to insert the hyphen. Using [CHAR] we can get the code value of this ASCII character on the stack. The Forth word HOLD takes this ASCII code and inserts it into the formatted number character string.

We now have three digits left. We might use the phrase

```
# # #
```

but it is easier to simply use the word #S, which will automatically convert the rest of the number for us.

If you are more familiar with ASCII codes represented in hexadecimal form, you can use this definition instead:

```
HEX : .PH#  <# # # # # 02D HOLD #S #> TYPE SPACE ;
DECIMAL
```

Either way, the compiled definition will be exactly the same.

Now let's format an unsigned double-length number as a date, in the following form:

```
6/15/03
```

Here is the definition:

```
: .DATE  <# # # [CHAR] / HOLD   # # [CHAR] / HOLD  #S #>  TYPE SPACE ;
```


Let's follow the above definition, remembering that it is written in reverse order from the output.The phrase

```
# # [CHAR] / HOLD
```

produces the right-most two digits (representing the year) and the right-most slash. The next occurrence of the same phrase produces the middle two digits (representing the day) and the left-most slash. Finally #S produces the left-most two digits(representing the month).

We could have just as easily defined

```
# # [CHAR] / HOLD
```

as its own word and used this word twice in the definition of .DATE.

Since you have control over the conversion process, you can actually convert different digits in different number bases, a feature which is useful in formatting such numbers as hours and minutes. For example, let's say that you have the time in seconds on the stack, and you want a word which will print hh:mm:ss. You might define it this way:

```
: SEXTAL  6 BASE ! ;
: :00  # SEXTAL # DECIMAL [CHAR] : HOLD ;
: SEC  <# :00 :00 #S #>  TYPE SPACE ;
```

We will use the word :00 to format the seconds and minutes. Both seconds and minutes are modulo-60, so the right digit can go as high as nine, but the left digit can only go up to five. Thus in the definition of :00 we convert the first digit (the one on the right)as a decimal number, then go into "sextal" (base 6) and convert the left digit. Finally, we return to decimal and insert the colon character. After :00 converts the seconds and the minutes,#S converts the remaining hours.

For example, if we had 4500 seconds on the stack, we would get

```
4500. SEC↵1:15:00 ok
```

Table 7-2 summarizes the Forth words that are used in number formatting. (Note the "KEY" at the bottom, which serves as a reminder of the meanings of "n," "d," etc.)

Table 7-2 — Number Formatting

<#	Begins the number conversion process. Expects the *unsigned double-length* number on the stack.
#	Converts one digit and puts it into an output character string. # *always* produces a digit — if you're out of significant digits, you'll still get a zero for every #.
#S	Converts the number until the result is zero. Always produces *at least one digit* (0 if the value is zero).
c HOLD	Inserts, at the current position in the character string being formatted, a character *c* whose ASCII value is on the stack. HOLD (or a word which uses HOLD) must be used between <# and #>.
SIGN	Inserts a minus sign in the output string if the top of stack is negative. Usually used with ROT immediately before #> for a leading minus sign.
#>	Completes number conversion by leaving the character count and address on the stack (these are the appropriate arguments for TYPE).

Stack effects for number formatting

phrase	stack	type of arguments
<# ... #>	(ud -- addr u)	double-length unsigned
<# ... ROT SIGN #>	(n |d| -- addr u)	double-length signed (where n is the high-order cell of d and |d| is the absolute value of d).

KEY	
n, n1, ...	single-length signed
d, d1, ...	double-length signed
u, u1, ...	single-length unsigned
addr	address
c	ASCII character value

Number Formatting — Signed and Single-length

So far we have formatted only unsigned double-length numbers. The <#...#> form expects only unsigned double-length numbers, but we can use it for other types of numbers by making certain arrangements on the stack.
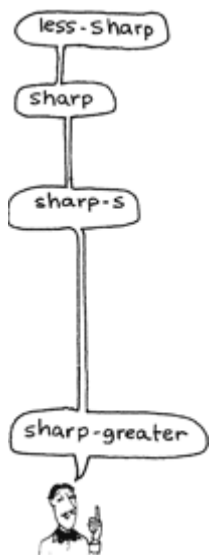
For instance, let's look at a simplified version of the system definition of D. (which prints a *signed* double-length number):

```
: D.  TUCK DABS <#  #S ROT SIGN  #>  TYPE SPACE ;
```

The phrase ROT SIGN inserts a minus string in the character string if the *third* number on the stack is negative. We have prepared for this test by putting a copy of the high-order cell (the one with the sign bit) at the bottom of the stack, by using the word TUCK.

Because <# expects only *unsigned* double-length numbers,we must take the absolute value of our double-length *signed* number, with the word DABS. We now have the proper arrangement of arguments on the stack for the <#...#> phrase. In some cases, such as accounting, we may want a negative number to be written

```
12345-
```

in which case we would place the phrase ROT SIGN at the *left* side of our <#...#>phrase, like this:

```
<#  ROT SIGN #S  #>
```

Let's define a word which will print a signed double-length number with a decimal point and two decimal places to the right of the decimal. Since this is the form most often used for writing dollars and cents, let's call it

.$

and define it like this:

```
: .$   TUCK DABS  <#  # # [CHAR] . HOLD  #S ROT SIGN  [CHAR] $ HOLD  #>  TYPE SPACE ;
```

Let's try it:

```
2000.00 .$↵$2000.00 ok
```

or even

```
2,000.00 .$↵$2000.00 ok
```

We recommend that you save .$, since we'll be using it in some future examples.

You can also write special formats for single-length numbers. For example, if you want to use an unsigned single-length number, simply put a zero on the stack before the word <#. This effectively changes the single-length number into a double-length number which is so small that it has nothing (zero) in the high-order cell.

To format a *signed* single-length number, again you must supply a zero as a high-order cell. But you must also leave a copy of the signed number in the third stack position for ROT SIGN, and you must leave the absolute value of the number in the second stack position. The phrase to do all this is

```
DUP ABS 0
```

Here are the "set-up" phrases that are needed to print various kinds of numbers:

Number to be printed	Precede <# by
double-length, unsigned	(nothing needed)
63-bit, plus sign	TUCK DABS (to save the sign in the third stack position for ROT SIGN)
single-length, unsigned	0 (to give a dummy high-order part)
31-bit, plus sign	DUP ABS 0 (to save the sign)

Double-length Operators

Here is a list of double-length math operators:

D.R	(d width --)	Prints the signed double-length number, right-justified within the field width.
D+	(d1 d2 -- d-sum)	Adds two double-length numbers.

D-	(d1 d2 -- d-diff)	Subtracts two double-length numbers (d1-d2).
DNEGATE	(d -- -d)	Changes the sign of a double-length number.
DMAX	(d1 d2 -- d-max)	Returns the maximum of two double-length numbers (d1-d2).
DMIN	(d1 d2 -- d-min)	Returns the minimum of two double-length numbers (d1-d2).
D=	(d1 d2 -- f)	Returns true if d1 and d2 are equal.
D0=	(d -- f)	Returns true if d is zero.
D<	(d1 d2 -- f)	Returns true if d1 is less than d2.
DU<	(ud1 ud2 -- f)	Returns true if ud1 is less than ud2. Both numbers are unsigned.

The initial "D" signifies that these operators may only be used for double-length operations, whereas the initial "2," as in 2SWAP and 2DUP, signifies that these operators may be used either for double-length numbers or for pairs of numbers.

Here's an example using D+:

```
200,000 300,000 D+ D.↵500000 ok
```

Mixed-length Operators

Here's a table of very useful Forth words which operate on a combination of single- and double-length numbers:

M+	(d n -- d-sum)	Adds a double-length number to a single-length number. Returns a double-length result.
SM/REM	(d n1 -- n2 n3)	Divide d1 by n1, giving the symmetric quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.
FM/MOD	(d n1 -- n2 n3)	Divide d1 by n1, giving the floored quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.
M*	(n1 n2 -- d-prod)	Multiplies two single-length numbers. Returns a double-length result. All values are signed.
M*/	(d +n1 n2 -- d-result)	Multiplies a double-length number by a single-length number and divides the triple-length result by a single-length number (d*n/n). Returns a double-length result. All values are signed.

Here's an example using M+:

```
200,000 7 M+ D.↵200007 ok
```

Or, using M*/, we can redefine our earlier version of % so that it will accept a double-length argument:

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

as in

```
200.50 15 % D.↵3007 ok
```

If you have loaded the definition of .$ we gave in the last Handy Hint, you can enter

```
200.50 15 % .$↵$30.07 ok
```

We can redefine our earlier definition of R% to get a rounded double-length result, like this:

```
: R%  10 M*/  5 M+  10 SM/REM NIP ;
```

then

```
987.65 15 R% .$↵$30.08 ok
```

Notice that M*/ is the only ready-made Forth word which performs multiplication on a double-length argument. To multiply 200,000 by 3, for instance, we must supply a "1" as a dummy denominator:

```
200,000 3 1 M*/ D.↵600000 ok
```

since

$$\frac{3}{1}$$

is the same as 3.

M*/ is also the only ready-made Forth word that performs division with a double-length result. So to divide 200,000 by 4, for instance, we must supply a "1" asa dummy numerator:

```
200,000 1 4 M*/ D.↵50000 ok
```

Numbers in Definitions

When a definition contains a number, such as

```
: SCORE-MORE  20 + ;
```

the number is compiled into the dictionary in binary form, just as it looks on the stack.

The number's binary value depends on the number base at the time you *compile* the definition. For example, if you were to enter

```
HEX  : SCORE-MORE  14 + ;  DECIMAL
```

the dictionary definition would contain the hex value 14, which is the same as the decimal value 20 (16+4). Henceforth, SCORE-MORE will always add the equivalent of the decimal 20 to the value on the stack, regardless of the current number base.

If, on the other hand, you were to put the word HEX *inside* the definition, then you would change the number base when you execute the definition.

For example, if you were to define:

```
DECIMAL
: EXAMPLE  HEX 20 . DECIMAL ;
```

the number would be compiled as the binary equivalent of decimal 20, since DECIMAL was current at compilation time.

At execution time, here's what happens:

```
EXAMPLE↵14 ok
```

The number is output in hexadecimal.

For the record, a number that appears inside a definition is called a "literal." (Unlike the words in the rest of the definition which allude to other definitions, a number must be taken literally.)

Here is a list of the Forth words we've covered in this chapter:

Unsigned operators

U.	(u --)	Prints the unsigned single-length number, followed by one space.
UM*	(u1 u2 -- ud)	Multiplies two single-length numbers. Returns a double-length result. All values are unsigned.
UM/MOD	(ud u1 -- u2 u3)	Divides a double-length by a single-length number. Returns a single-length quotient and remainder. All values are unsigned.
U<	(u1 u2 -- f)	Leaves true if u1 < u2, where both are treated as single-length unsigned integers.

Number bases

HEX	(--)	Sets the base to sixteen.
OCTAL	(--)	Sets the base to eight (available on some systems).
DECIMAL	(--)	Returns the base to ten.

Number formatting operators

<#	Begins the number conversion process. Expects the *unsigned double-length* number on the stack.
#	Converts one digit and puts it into an output character string. # *always* produces a digit — if you're out of significant digits, you'll still get a zero for every #.
#S	Converts the number until the result is zero. Always produces *at least one digit* (0 if the value is zero).
c HOLD	Inserts, at the current position in the character string being formatted, a character *c* whose ASCII value is on the stack. HOLD (or a word which uses HOLD) must be used between <# and #>.
SIGN	Inserts a minus sign in the output string if the top of stack is negative. Usually used with ROT immediately before #> for a leading minus sign.
#>	Completes number conversion by leaving the character count and address on the stack (these are the appropriate arguments for TYPE).

Stack effects for number formatting

phrase	stack	type of arguments
<# ... #>	(d -- addr u)	double-length unsigned
<# ... ROT SIGN #>	(n |d| -- addr u)	double-length signed (where n is the high-order cell of d and |d| is the absolute value of d).

Double-length operators

D+	(d1 d2 -- d-sum)	Adds two double-length numbers.
D-	(d1 d2 -- d-diff)	Subtracts two double-length numbers (d1-d2).
DNEGATE	(d -- -d)	Changes the sign of a double-length number.
DMAX	(d1 d2 -- d-max)	Returns the maximum of two double-length numbers (d1-d2).
DMIN	(d1 d2 -- d-min)	Returns the minimum of two double-length numbers (d1-d2).

D=	(d1 d2 -- f)	Returns true if d1 and d2 are equal.
D0=	(d -- f)	Returns true if d is zero.
D<	(d1 d2 -- f)	Returns true if d1 is less than d2.
DU<	(ud1 ud2 -- f)	Returns true if ud1 is less than ud2. Both numbers are unsigned.
D.R	(d width --)	Prints the signed double-length number, right-justified within the field width.

<div align="center">Mixed-length operators</div>

M+	(d n -- d-sum)	Adds a double-length number to a single-length number. Returns a double-length result.
SM/REM	(d n1 -- n2 n3)	Divide d1 by n1, giving the symmetric quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.
FM/MOD	(d n1 -- n2 n3)	Divide d1 by n1, giving the floored quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.
M*	(n1 n2 -- d-prod)	Multiplies two single-length numbers. Returns a double-length result. All values are signed.
M*/	(d +n1 n2 -- d-result)	Multiplies a double-length number by a single-length number and divides the triple-length result by a single-length number (d*n/n). Returns a double-length result. All values are signed.

<div align="center">KEY</div>

n, n1, ...	single-length signed
d, d1, ...	double-length signed
u, u1, ...	single-length unsigned
addr	address
c	ASCII character value

Review of Terms

Arithmetic left and right shift
    the process of shifting all bits in a number, except the sign bit, to the left or right, in effect doubling or halving the (assumed signed) number, respectively.

Logical left and right shift
    the process of shifting all bits in a number, including the sign bit, to the left or right, in effect doubling or halving the (assumed unsigned) number, respectively.

ASCII
    a standardized system of representing input/output characters as byte values. Acronym for American Standard Code for Information Interchange. (Pronounced *ask-key*)

Binary
    number base 2.

Byte
    the standard term for an 8-bit value.

Cell
    the Forth term for a single-cell value.

Decimal
    number base 10.

Hexadecimal
    number base 16.

Literal
    in general, a number of symbol which represents only itself; in Forth, a number that appears inside a definition.

Mask

a value which can be " superimposed" over another, hiding certain bits and revealing only those bits that we are interested in.

Number formatting
> the process of printing a number, usually in a special form such as 3/13/03 or $47.93.

Octal
> number base 8.

Sign bit high-order bit
> the bit which, for a signed number, indicates whether it is positive or negative and, for an unsigned number, represents the bit of the highest magnitude.

Two's complement
> for any number, the number of equal absolute value but opposite sign. To calculate 10 - 4, the computer first produces the two's complement of 4, (i.e., -4), then computes 10 + (-4).

Unsigned number
> a number which is assumed to be positive.

Unsigned single-length number
> an integer which falls within the range of 0 to 2147483647.

Word
> In Forth, a defined dictionary entry, elsewhere, a term for a 16-bit value.

Integer division
> produces a quotient q and a remainder r by dividing operand a by operand b. Division operations return q, r, or both. The identity b*q + r = a holds for all a and b.

Floored division
> is integer division in which the remainder carries the sign of the divisor or is zero, and the quotient is rounded to its arithmetic floor.

Symmetric division
> is integer division in which the remainder carries the sign of the dividend or is zero and the quotient is the mathematical quotient "rounded towards zero" or "truncated".

Problems — Chapter 7

1. Veronica Wainwright couldn't remember the upper limit for a signed single-length number, and she had no book to refer to, only a Forth terminal. So she wrote a definition called N-MAX, using a BEGIN... UNTIL loop. When she executed it, she got

2. ↵2147483647 ok

   What was her definition? [answer]

3. Since you now know that AND and OR employ bit logic, explain why the following example *must* use OR instead of +:

4.   : MATCH   humorous sensitive AND
5.   art-loving music-loving OR AND
6.   smoking 0= AND
7.   IF ." I have someone you should meet " THEN ;

8. Write a definition that "rings" your terminal's bell three times. Make sure that there is enough of a delay between the bells so that they are distinguishable. Each time the bell rings, the word "BEEP" should appear on the terminal screen. [answer]

9. Rewrite the temperature conversion definitions which you created for the problems in Chap. 5. This time assume that the input and resulting temperatures are to be double-length signed integers

which are scaled (i.e., multiplied) by ten. For example, if 10.5 degrees is entered, it is a 32-bit integer with a value of 105. [answer]

10. Write a formatted output word named .DEG which will display a double-length signed integer scaled by ten as a string of digits, a decimal point, and one fractional digit. For example:

```
11.  12.3 .DEG↵12.3 ok
```

[answer]

12. Solve the following conversions:
    0.0° F in Centigrade
    212.0° F in Centigrade
    20.0° F in Centigrade
    16.0° C in Fahrenheit
    -40.0° C in Fahrenheit
    100.0° K in Centigrade
    100.0° K in Fahrenheit
    233.0° K in Centigrade
    233.0° K in Fahrenheit

13. Write a routine which evaluates the quadratic equation $7x^2 + 20x + 5$ given x, and returns a double-length result.

14. How large an x will work without overflowing sixty-four bits as a signed number?

15. Write a word which prints the numbers 0 through 16 (decimal) in decimal, hexadecimal, and binary form in three columns. E.g.,

```
16.  DECIMAL  0  HEX  0  BINARY     0
17.  DECIMAL  1  HEX  1  BINARY     1
18.  DECIMAL  2  HEX  2  BINARY    10
19.  ...
20.  DECIMAL 16  HEX 10  BINARY 10000
```

[answer]

21. If you enter

```
22.  ..↵
```

(two periods *not* separated by a space) and the system responds "ok," what does this tell you?
[answer]

23. Write a definition for a phone-number formatting word that will also print the area code with a slash *if and only if* the number includes an area code. E.g.,

```
24.  555-1234 .PH#  555-1234↵ok
25.  310/999-6784 .PH#↵310/999-6784 ok
26.  [answer]
```

8. Variables, Constants, and Arrays

As we have seen throughout the previous seven chapters, Forth programmers use the stack to store numbers temporarily while they perform calculations or to pass arguments from one word to another. When programmers need to store numbers more permanently, they use variables and constants.

In this chapter, we'll learn how Forth treats variables and constants, and in the process we'll see how to directly access locations in memory.

Variables

Let's start with an example of a situation in which you'd want to use a variable — to store the day's date. First we'll create a variable called DATE. We do this by saying

```
VARIABLE DATE
```

If today is the twelfth, we now say

```
12 DATE !
```

that is, we put twelve on the stack, then give the name of the variable, then finally execute the word !, which is pronounced *store*. This phrase stores the number twelve into the variable DATE.

Conversely, we can say

```
DATE @
```

that is, we can name the variable, then execute the word @, which is pronounced *fetch*. This phrase fetches the twelve and puts it on the stack. Thus the phrase

```
DATE @ .↵12 ok
```

prints the date.

To make matters even easier, there is a Forth word whose definition is this:

```
: ?   @ . ;
```

So instead of "DATE-fetch-dot," we can simply type

```
DATE ?↵12 ok
```

The value of DATE will be twelve until we change it. To change it, we simply store a new number

```
13 DATE !↵ok
DATE ?↵13 ok
```

Conceivably we could define additional variables for the month and year:

```
VARIABLE DATE   VARIABLE MONTH   VARIABLE YEAR
```

then define a word called !DATE (for "store-the-date") like this:

```
: !DATE  YEAR !  DATE !  MONTH ! ;
```

to be used like this:

```
7 31 03 !DATE↵ok
```

then define a word called .DATE (for "print-the-date") like this:

```
: .DATE  MONTH ?  DATE ?  YEAR ? ;
```

Your Forth system already has a number of variables defined; one is called BASE. BASE contains the number base that you're currently working in. In fact, the definition of HEX and DECIMAL (and OCTAL, if your system has it) are simply

```
: DECIMAL  10 BASE ! ;
: HEX      16 BASE ! ;
: OCTAL     8 BASE ! ;
```

You can work in any number base by simply storing it into BASE.

For Experts

A three-letter code such as an airport terminal name, can be stored as a single-length unsigned number in base 36. For example:

```
: ALPHA  36 BASE ! ;↵ok
ALPHA↵ok
ZAP U.↵ZAP ok
```

Somewhere in the definitions of the system words which perform input and output number conversions, you will find the phrase

```
BASE @
```

because the current value of BASE is used in the conversion process. Thus a single routine can convert numbers in *any* base. This leads us to make a formal statement about the use of variables:

A Closer Look at Variables

In Forth, variables are appropriate for any value that is used inside a definition which may need to change at any time after the definition has already been compiled.

When you create a variable such as DATE by using the phrase

```
VARIABLE DATE
```

you are really compiling a new word, called DATE, into the dictionary. A simplified view would look like the view below.

```
┌─────────────────────────────────────────┐
│              4 D A T E                    │
├─────────────────────────────────────────┤
│ instruction code appropriate for variables│
├─────────────────────────────────────────┤
│   space for the actual value to be stored │
└─────────────────────────────────────────┘
```

DATE is like any other word in your dictionary except that you defined it with the word VARIABLE instead of the word **:**. As a result, you didn't have to define what your definition would *do*, the word VARIABLE itself spells out what is supposed to happen. And here *is* what happens:

When you say

```
12 DATE !
```

Twelve goes onto the stack, after which the text interpreter looks up DATE in the dictionary and, finding it, points it out to EXECUTE.
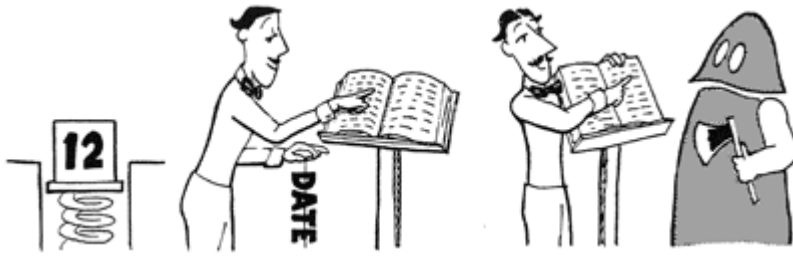
EXECUTE executes a variable by copying the address of the variable's "empty" cell (where the value will go) onto the stack.
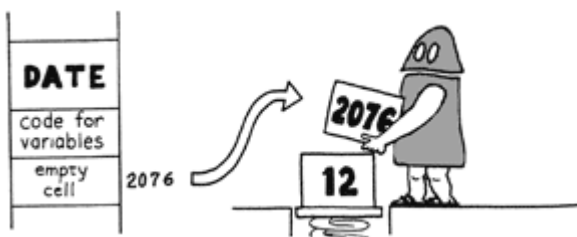
The word ! takes the address (on top) and the value (underneath), and stores the value into that location. Whatever number used to be at that address is replaced by the new number.

(To remember what order the arguments belong in, think of setting down your parcel, then sticking the address label on top.)
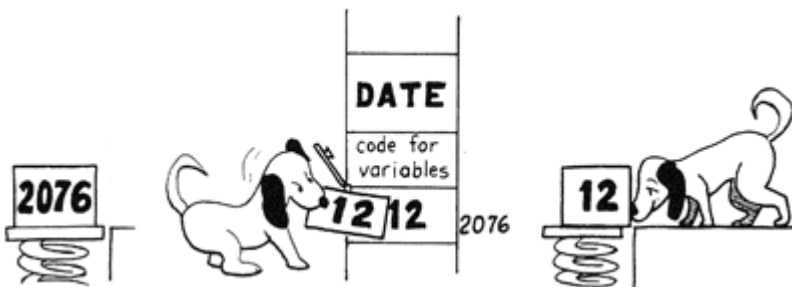
The word @ expects one argument only: an address, which in this case is supplied by the name of the variable, as in

```
DATE @
```

Using the value on the stack as an address, the word @ pushes the contents of that location onto the stack, "dropping" the address. (The contents of the location remain intact.)

Using a Variable as a Counter

In Forth, a variable is ideal for keeping a count of something. To reuse our egg-packer example, we might keep track of how many eggs go down the conveyor belt in a single day. (This example will work at your terminal, so enter it as we go.)
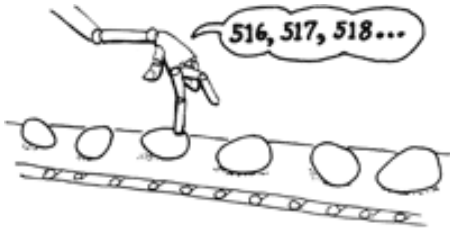
First we can define

```
VARIABLE EGGS
```

to keep the count in. To start with a clean slate every morning, we could store a zero into EGGS by executing a word whose definition looks like this:

```
: RESET  0 EGGS ! ;
```
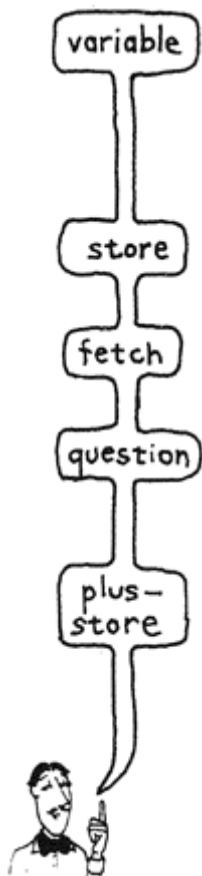

Then somewhere in our egg-packing application, we would define a word which executes the following phrase every time an egg passes an electric eye on the conveyor:

```
1 EGGS +!
```

The word +! Adds the given value to the contents of the given address. (It doesn't bother to tell you what the contents are.) Thus the phrase

```
1 EGGS +!
```

Increments the count of eggs by one. For purposes of illustration, let's put this phrase inside a definition like this:

```
: EGG  1 EGGS +! ;
```

At the end of the day, we would say

```
EGGS ?
```

to find out how many eggs went by since morning.

Let's try it:

```
RESET↵ok
EGG↵ok
EGG↵ok
EGG↵ok
EGGS ?↵3 ok
```

Here's a review of the words we've covered in the chapter so far:

VARIABLE xxx	(--) xxx: (-- addr)	Creates a variable named *xxx*; the word *xxx* returns its address when executed.
!	(n addr --)	Stores a single-length number into the address.
@	(addr – n)	Replaces the address with its contents.
?	(addr --)	Prints the contents of the address, followed by one space.
+!	(n addr --)	Adds a single-length number to the contents of the address.

Constants

While variables are normally used for values that may change, constants are used for values that *won't* change. In Forth, we create a constant and set its value at the same time, like this:


```
220 CONSTANT LIMIT
```

Here we have defined a constant named LIMIT, and given it the value 220. Now we can use the word LIMIT in place of the value, like this:

```
: ?TOO-HOT  LIMIT > IF  ." Danger -- reduce heat "  THEN ;
```

If the number on the stack is greater than 220, then the warning message will be printed.

Notice that when we say

```
LIMIT
```

we get the *value*, not the address. We don't need the "fetch."

This is an important difference between variables and constants. The reason for the difference is that with variables, we need the address to have the option of fetching or storing. With constants we always want

the value; we absolutely never store. (If you really need to store a new value into a "constant", you should use a VALUE.)

One use for constants is to name a hardware address. For example, a microprocessor-controlled portable camera application might contain this definition:

```
: PHOTOGRAPH   SHUTTER OPEN   TIME EXPOSE   SHUTTER CLOSE ;
```

Here the word SHUTTER has been defined as a constant so that execution of SHUTTER returns the hardware address of the camera's shutter. It might, for example, be defined:

```
HEX
FFFF3E27 CONSTANT SHUTTER
DECIMAL
```

The words OPEN and CLOSE might be defined simply as

```
: OPEN  1 SWAP ! ;
: CLOSE 0 SWAP ! ;
```

so that the phrase

```
SHUTTER OPEN
```

writes a "1" to the shutter address, causing the shutter to open.

Here are some situations when it's good to define numbers as constants:

1. When it's important that you make your application more readable. One of the elements of Forth style is that definitions should be self-documenting, as is the definition of PHOTOGRAPH above.
2. When it's more convenient to use a name instead of the number. For example, if you think you may have to change the value (because, for instance, the hardware might get changed) you will only have to change the value *once* — in the file where the constant is defined — then recompile your application.
3. (Only true for less sophisticated Forth compilers) When you are using the same value many times in your application. In the compiled form of a definition, reference to a constant requires less memory space.

CONSTANT xxx	(n --) xxx: (-- n)	Creates a constant named *xxx* with the value n; the word *xxx* returns n when executed.

Double-length Variables and Constants

You can define a double-length variable by using the word 2VARIABLE. For example,

```
2VARIABLE DATE
```

Now you can use the Forth words 2! (pronounced *two-store*) and 2@ (pronounced *two-fetch*) to access this double-length variable. You can store a double-length number into it by simply saying

```
800,000 DATE 2!
```

and fetch it back with

```
DATE 2@ D.↵800000 ok
```
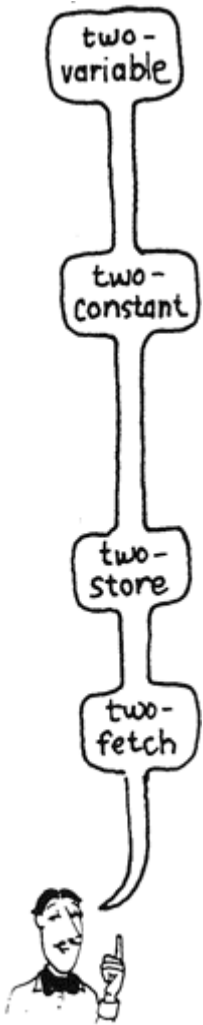
Or you can store the full month/date/year into it, like this:

```
7/17/03 DATE 2!
```

and fetch it back with

```
DATE 2@ .DATE↵7/17/03 ok
```

assuming that you've loaded the version of .DATE we gave in the last chapter.

You can define a double-length constant by using the Forth word 2CONSTANT, like this:

```
200,000 2CONSTANT APPLES
```

Now the word APPLES will place the double-length number on the stack.

```
APPLES D.↵200000 ok
```

Of course, we can do:

```
400,000 2CONSTANT MUCH
: MUCH-MORE  200,000 D+  MUCH D+ ;
```

in order to be able to say

```
APPLES MUCH-MORE D.↵800000 ok
```

As the prefix "2" reminds us, we can also use 2CONSTANT to define a *pair* of single-length numbers. The reason for putting two numbers under the same name is a matter of convenience and of saving space in the dictionary.

As an example, recall (from Chap. 5) that we can use the phrase

```
355 113 */
```

to multiply a number by a crude approximation of π. We could store these two integers as a 2CONSTANT as follows:

```
355 113 2CONSTANT PI
```

then simply use the phrase

```
PI */
```

as in

```
10000 PI */ .↵31415 ok
```

Here is a review of the double-length data-structure words:

2CONSTANT xxx	(d --) xxx: (-- d)	Creates a double-length constant named *xxx* with the value d; the word *xxx* returns d when executed.
2VARIABLE xxx	(--) xxx: (-- addr)	Creates a double-length variable named *xxx*; the word *xxx* returns its address when executed.
2!	(d addr --)	Stores a double-length number into the address.
2@	(addr -- d)	Returns the double-length contents of the address.

Arrays

```
┌─────────────────────────────────┐
│          4 D A T E              │
├─────────────────────────────────┤
│             code                │
├─────────────────────────────────┤
│  room for a single-length value │
└─────────────────────────────────┘
```

As you know, the phrase

```
VARIABLE DATE
```

creates a definition which conceptually looks like that at the right.

Now if you say

```
1 CELLS ALLOT
```

an additional cell is allotted in the definition, like this:

4 D A T E
code
room for a single-length value
room for a single-length value

The result is the same as if you had used 2VARIABLE. By changing the argument to ALLOT, however, you can define *any number* of variables under the same name. Such a group of variables is called an "array."

For example, let's say that in our laboratory, we have not just one, but *five* burners that heat various kinds of liquids.

We can make our word ?TOO-HOT check that all five burners have not exceeded their individual limit if we define LIMIT using an array rather than a constant.

Let's give the array the name LIMITS, like this:

```
VARIABLE LIMITS 4 CELLS ALLOT
```

The phrase "4 CELLS ALLOT" gives the array an extra four cells (five cells in all).

6 L I M I T S
code
room for burner 0's limit
room for burner 1's limit
room for burner 2's limit
room for burner 3's limit
room for burner 4's limit

address
3160
3164
3168
316C
3170

Suppose we want the limit for burner 0 to be 220. We can store this value by simply saying

```
220 LIMITS !
```

because LIMITS returns the address of the first cell in the array. Suppose we want the limit for burner 1 to be 340. We can store this value by adding 1 CELLS to the address of the original cell, like this:

```
340 LIMITS 1 CELLS + !
```

We can store limits for burners 2, 3, and 4 by adding the "offsets" 2 CELLS, 3 CELLS, and 4 CELLS, respectively, to the original address. We can define the convenient word

```
: LIMIT  ( burner# -- addr ) CELLS LIMITS + ;
```

to take a burner number on the stack and compute an address that reflects the appropriate offset.

Now if we want the value 170 to be the limit for burner 2, we simply say

```
170 2 LIMIT !
```

or similarly, we can fetch the limit for burner 2 with the phrase

```
2 LIMIT ?↵170 ok
```

This technique increases the usefulness of the word LIMIT, so that we can redefine ?TOO-HOT as follows:

```
: ?TOO-HOT ( temp burner# -- )
LIMIT @ > IF  ." Danger -- reduce heat "  THEN ;
```

which works like this:

```
210 0 ?TOO-HOT↵ok
230 0 ?TOO-HOT↵Danger -- reduce heat ok
300 1 ?TOO-HOT↵ok
350 1 ?TOO-HOT↵Danger -- reduce heat ok
```

etc.

Another Example — Using an Array for Counting

Meanwhile, back at the egg ranch:

Here's another example of an array. In this example, each element of the array is used as a separate counter. Thus we can keep track of how many cartons of "extra large" eggs the machine has packed, how many "large," and so forth.

Recall from our previous definition of EGGSIZE (in Chap. 4) that we used four categories of acceptable eggs, plus two categories of "bad eggs."

```
0 CONSTANT REJECT
1 CONSTANT SMALL
2 CONSTANT MEDIUM
3 CONSTANT LARGE
4 CONSTANT EXTRA-LARGE
5 CONSTANT ERROR
```

So let's create an array that is six cells long:

```
VARIABLE COUNTS 5 CELLS ALLOT
```

The counts will be incremented using the word +!, so we must be able to set all the elements of the array to zero before we begin counting. The phrase

```
COUNTS 6 CELLS 0 FILL
```

will fill 6 cells , starting at the address of COUNTS, with zeros. If your Forth system includes the word ERASE, it's better to use it in this situation. ERASE fills the given number of bytes with zeroes. Use it like this:

```
COUNTS 6 CELLS ERASE
```

FILL   (addr n b --) Fills n bytes of memory, beginning at the address, with value b.

ERASE (addr n --)   Stores zeroes into n bytes of memory, beginning at the address.

For convenience, we can put the phrase inside a definition, like this:

```
: RESET  COUNTS 6 CELLS ERASE ;
```

Now let's define a word which will give us the address of one of the counters, depending on the category number it is given (0 through 5), like this:

```
: COUNTER   CELLS COUNTS + ;
```

and another word which will add one to the counter whose number is given, like this:

```
: TALLY  COUNTER 1 SWAP +! ;
```

The "1" serves as the increment for +!, and SWAP puts the arguments for +! in the order they belong, i.e., (n addr --).

Now, for instance, the phrase

```
LARGE TALLY
```

will increment the counter that corresponds to large eggs.

Now let's define a word which converts the weight per dozen into a category number:

```
: CATEGORY ( weight -- category )
  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  NIP ;
```

(By the time we'll get to the NIP, we will have two values on the stack: the weight which we have been DUPping and the category number, which will be on top. We want only the category number; "NIP" eliminates the weight.)

For instance, the phrase

```
25 CATEGORY
```

will leave the number 3 (LARGE) on the stack. The above definition of CATEGORY resembles our old definition of EGGSIZE, but, in the true Forth style of keeping words as short as possible, we have removed the output messages from the definition. Instead, we'll define an additional word which expects a category number and prints an output message, like this:

```
: LABEL ( category -- )
  CASE
REJECT      OF ." reject "      ENDOF
SMALL       OF ." small "       ENDOF
MEDIUM      OF ." medium "      ENDOF
LARGE       OF ." large "       ENDOF
EXTRA-LARGE OF ." extra large " ENDOF
ERROR       OF ." error "       ENDOF
  ENDCASE ;
```

For example:

```
SMALL LABEL↵small ok
```

Now we can define EGGSIZE using three of our own words:

```
: EGGSIZE  CATEGORY  DUP LABEL  TALLY ;
```

Thus the phrase

```
23 EGGSIZE
```

will print

```
medium ok
```

at your terminal and update the counter for medium eggs.

How will we read the counters at the end of the day? We could check each cell in the array separately with a phrase such as

```
LARGE COUNTER ?
```

(which would tell us how many "large" cartons were packed). But let's get a little fancier and define our own word to print a table of the day's results in this format:

```
QUANTITY     SIZE
    1        reject
  112        small
  132        medium
  143        large
  159        extra large
    0        error
```

Since we have already devised category numbers, we can simply use a DO and index on the category number, like this:

```
: REPORT ( -- )
   PAGE ." QUANTITY       SIZE" CR CR
   6 0 DO     I COUNTER @ 5 U.R
             7 SPACES
             I LABEL CR
     LOOP ;
```

(The phrase "I COUNTER @ 5 U.R" takes the category number given by I, indexes into the array, and prints the contents of the proper element in a five-column field.)

Factoring Definitions

This is a good time to talk about factoring as it applies to Forth definitions. We've just seen an example in which factoring simplified our problem.

Our first definition of EGGSIZE from Chap. 4, categorized eggs by weight and printed the name of the categories at the terminal. In our present version we factored out the "categorizing" and the "printing" into two separate words. We can use the word CATEGORY to provide the argument either for the printing word or the counter-tallying word (or both). And we can use the printing word, LABEL, in both EGGSIZE and REPORT.

As Charles Moore, the inventor of Forth, has written:

A good Forth vocabulary contains a large number of small words. It is not enough to break a problem into small pieces. The object is to isolate words *that can be reused*.

For example, in the recipe:

Get a can of tomato sauce.
Open can of tomato sauce.
Pour tomato sauce into pan.
Get can of mushrooms.
Open can of mushrooms.
Pour mushrooms into pan.

you can "factor out" the getting, opening, and pouring, since they are common to both cans. Then you can give the factored-out process a name and simply write:

```
TOMATOES ADD
MUSHROOMS ADD
```

and any chef who's graduated from the Postfix School of Cookery will know exactly what you mean.

Not only does factoring make a program easier to write (and fix!), it saves memory space, too. A reusable word such as ADD gets defined only once. The more complicated the application, the greater the savings.

Here is another thought about Forth style before we leave the egg ranch. Recall our definition of EGGSIZE

```
: EGGSIZE  CATEGORY  DUP LABEL  TALLY ;
```

CATEGORY gave us a value which we wanted to pas on to both LABEL and TALLY, so we included the DUP. To make the definition "cleaner," we might have been tempted to take the DUP out and put it inside the definition of LABEL, at the beginning. Thus we might have written:

```
: EGGSIZE  CATEGORY  LABEL TALLY ;
```

where CATEGORY passes the value to LABEL, and LABEL passes it on to TALLY. Certainly this approach would have worked. But then, when we defined REPORT, we would have had to say

```
I LABEL DROP
```

instead of simply

```
I LABEL
```

Forth programmers tend to follow this convention: when possible, words should destroy their own parameters. In general, it's better to put the DUP inside the "calling definition" (EGGSIZE, here) than in the "called" definition (LABEL, here).

Another Example — "Looping" through an Array

We'd like to introduce a little technique that is relevant to arrays. We can best illustrate this technique by writing our own definition of a Forth word called DUMP. DUMP is used to print out the contents of a series of memory addresses. The usage is

```
addr count DUMP
```

For instance, we could enter

```
COUNTS 6 DUMP
```

to print the contents of our egg-counting array called COUNTS. Since DUMP is primarily designed as a programming tool to print out the contents of memory locations, it prints either byte-by-byte or cell-by-cell, depending on the type of addressing our computer uses. Our version of DUMP will print cell-by-cell.

Obviously DUMP will involve a DO loop. The question is: what should we use for an index? Although we might use the count itself (0 - 6) as the loop index, it's better to use the *address* as the index.

The address of COUNTS will be the starting index for the loop, while the address plus the count will serve as the limit, like this:

```
: DUMP ( addr cell-count -- )
   CELLS OVER + SWAP
         DO   CR I @ 5 U.R
   1 CELLS +LOOP ;
```

The key phrase here is

```
CELLS OVER + SWAP
```

which immediately precedes the DO.
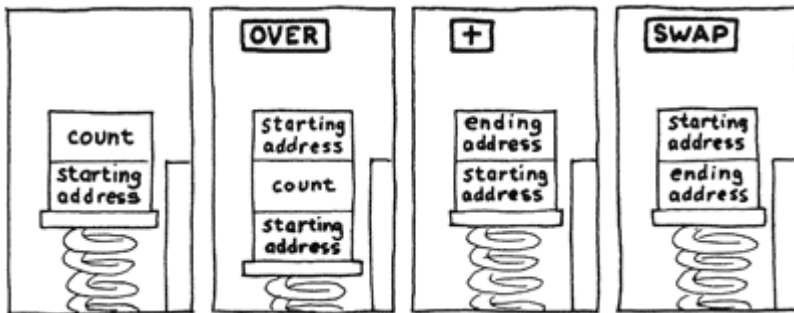
The ending and starting addresses are now on the stack, ready to serve as the limit and index for the DO loop. Since we are "indexing on the addresses," once we are inside the loop we merely have to say

```
I @  5 U.R
```

to print the contents of each element of the array. Since we are examining cells (@ fetches a single-length, single cell value), we increment the index by one cell each time, by using

```
1 CELLS +LOOP
```

Byte Arrays

Forth lets you create an array in which each element consists of a single byte rather than a full cell. This is useful any time you are storing a series of numbers whose range fits into that which can be expressed within eight bits.
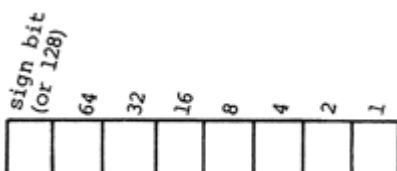
The range of an unsigned 8-bit number is 0 to 255. Byte arrays are also used to store ASCII character strings. The benefit of using a byte array instead of a cell array is that you can get the same amount of data in 25% (32-bit Forth) of the memory space.

The mechanics of using a byte array are the same as using a cell array except that

1. you don't have to use CELLS to manipulate the offset, since each element corresponds to one address unit, and
2. you must use the words C! and C@ instead of ! and @. These words, which operate on byte values only, have the prefix "C" because their typical use is accepting ASCII *c*haracters.

C! (b addr --) Stores an 8-bit value into the address.
C@ (addr -- b) Fetches an 8-bit value from the address.

Initializing an Array

Many situations call for an array whose values never change during the operation of the application and which may as well be stored into the array at the same time that the array is created, just as CONSTANTs are. Forth provides the means to accomplish this through the two words CREATE and **,** (pronounced *create* and *comma*).

Suppose we want permanent values in our LIMITS array. Instead of saying

```
VARIABLE LIMITS 4 CELLS ALLOT
```

we can say

```
CREATE LIMITS  220 , 340 , 170 , 100 , 190 ,
```

Usually the above line would be included from a disk file, but it also works interactively.

Like the word VARIABLE, CREATE puts a new name in the dictionary at compile time and returns the address of that definition when it is executed. But it does not "allot" any bytes for a value.

The word **,** takes a number off the stack and stores it into the array. So each time you express a number and follow it with **,** you add one cell to the array.
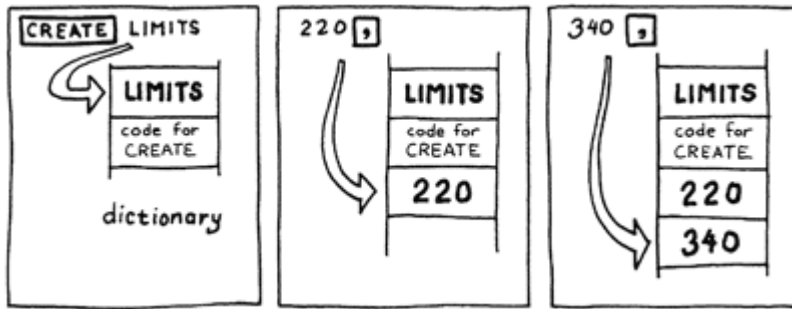
For Newcomers

Ingrained habits, learned from English writing, lead some newcomers to forget to type the final **,** in the line. Remember that **,** does not separate the numbers, it *compiles* them.

You can access the elements in a CREATE array just as you would the elements in a VARIABLE array. For example:

```
LIMITS CELL+ @ .↵340 ok
```

You can even store new values into the array, just as you would into a VARIABLE array.

To initialize a byte-array that has been defined with CREATE, you can use the word C, (c-comma). For instance, we could store each of the values used in our egg-sorting definition CATEGORY as follows:

```
CREATE SIZES 18 C, 21 C, 24 C, 27 C, 30 C, 255 C,
```

This would allow us to redefine CATEGORY using a DO loop rather than as a series of nested IF...THEN statements, as follows

```
: CATEGORY  6 0 DO  DUP SIZES I + C@  < IF  DROP I LEAVE THEN  LOOP ;
```

Note that we have added a maximum (255) to the array to simplify our definition regarding category 5.

Including the initialization of the SIZES array, this version takes only three lines of source text as opposed to six and takes less space in the dictionary, too.

For People Who Don't Like Guessing How It Works

The idea here is this: since there are five possible categories, we can use the category numbers as our loop index. Each time around, we compare the number on the stack against the element in SIZES, offset by the current loop index. As soon as the weight on the stack is greater than one of the elements in the array, we leave the loop and use I to tell us how many times we had looped before we "left." Since this number is our offset into the array, it will also be our category number.

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

CONSTANT xxx	(n --) xxx: (-- n)	Creates a constant named *xxx* with the value n; the word *xxx* returns n when executed.
VARIABLE xxx	(--) xxx: (-- addr)	Creates a variable named *xxx*; the word *xxx* returns its address when executed.
CREATE xxx	(--) xxx: (-- addr)	Creates a dictionary entry (head and code pointer only) named *xxx*; the word *xxx* returns its address when executed.
!	(n addr --)	Stores a single-length number into the address.

@	(addr -- n)	Replaces the address with its contents.
?	(addr --)	Prints the contents of the address, followed by one space.
+!	(n addr --)	Adds a single-length number to the contents of the address.
ALLOT	(n --)	Adds n bytes to the body of the most recently defined word.
,	(n --)	Compiles n into the next available cell in the dictionary.
C!	(b addr --)	Stores an 8-bit value into the address.
C@	(addr -- b)	Fetches an 8-bit value from the address.
FILL	(addr n b --)	Fills n bytes of memory, beginning at the address, with value b.
BASE	(n --)	A variable which contains the value of the number base being used by the system.
2CONSTANT xxx	(d --) xxx: (-- d)	Creates a double-length constant named *xxx* with the value d; the word *xxx* returns d when executed.
2VARIABLE xxx	(--) xxx: (-- addr)	Creates a double-length variable named *xxx*; the word *xxx* returns its address when executed.
2!	(d addr --)	Stores a double-length number into the address.
2@	(addr -- d)	Returns the double-length contents of the address.
C,	(b --)	Compiles b into the next available byte in the dictionary.
DUMP	(addr u --)	Displays u bytes of memory, starting at the address.
ERASE	(addr n --)	Stores zeroes into n bytes of memory, beginning at the address.

KEY

n, n1, ...	single-length signed
d, d1, ...	double-length signed
u, u1, ...	single-length unsigned
ud, ud1, ...	double-length unsigned
addr	address
c	ASCII character value
b	8-bit byte
f	Boolean flag

Review of Terms

Array
a series of memory locations with a single name. Values can be stored and fetched into the individual locations by giving the name of the array and adding an offset to the address.

Constant
a value which has a name. The value is stored in memory and usually never changes.

Factoring
as it applies to programming in Forth, simplifying a large job by extracting those elements which might be reused and defining those elements as operations.

Fetch

to retrieve a value from a given memory location.

Initialize

to give a variable (or array) its initial value(s) before the rest of the program begins.

Offset

a number which can be added to the address of the beginning of an array to produce the address of the desired location within the array.

Store

to place a value in a given memory location.

Variable

a location in memory which has a name and in which values are frequently stored and fetched.

Problems — Chapter 8

1. Write two words called BAKE-PIE and EAT-PIE. The first word increases the number of available PIES by one. The second decreases the number by one and thanks you for the pie. But if there are no pies, it types "What pie?" (make sure you start out with no pies.)

2. EAT-PIE↵What pie?

3. BAKE-PIE↵ok

4. EAT-PIE↵Thank you! ok

5. Write a word called FREEZE-PIES which takes all the available pies and adds them to the number of pies in the freezer. Remember that frozen pies cannot be eaten.

6. BAKE-PIE BAKE-PIE FREEZE-PIES↵ok

7. PIES ?↵0 ok

8. FROZEN-PIES ?↵2 ok

[answer]

9. Define a word called .BASE which prints the current value of the variable BASE in decimal. Test it by first changing BASE to some value other than ten. (This one is trickier than it may seem.)

10. DECIMAL .BASE 10↵ok

11. HEX .BASE 16↵ok

[answer]

12. Define a number-formatting word called M. which prints a double-length number with a decimal point. The position of the decimal point within the number is movable and depends on the value of a variable that you will define as PLACES. For example, if you store a "1" into PLACES, you will get

13. 200,000 M.↵20000.0 ok

that is, with the decimal point one place from the right. A zero in PLACES should produce no decimal point at all. [answer]

14. In order to keep track of the inventory of colored pencils in your office, create an array, each cell of which contains the count of a different colored pencil. Define a set of words so that, for example, the phrase

15. RED PENCILS

returns the address of the cell that contains the count of red pencils, etc. Then set these variables to indicate the following counts:

```
23 red pencils
15 blue pencils
12 green pencils
0 orange pencils
```

[answer]

16. A histogram is a graphic representation of a series of values. Each value is shown by the height or length of a bar. In this exercise you will create an array of values and print a histogram which displays a line of "*"s for each value. First create an array with about ten cells. Initialize each element of the array with a value in the range of zero to seventy. Then define a word PLOT which will print a line for each value. On each line print the number of the cell followed by a number of "*"s equal to the contents of that cell.

    For example, if the array has four cells and contains the values 1, 2, 3 and 4, then PLOT would produce:
17. 1 *
18. 2 **
19. 3 ***
20. 4 ****

[answer]

21. Create an application that displays a tic-tac-toe board, so that two human players can make their moves by entering them from the keyboard. For example, the phrase
22. 4 X!

    puts an "X" in box 4 (counting starts with 1) and produces this display:

```
   |    |
---------
X  |    |
---------
   |    |
```

    Then the phrase

```
3 O!
```

    puts an "O" in box 3 and prints the display:

```
   |    | O
---------
X  |    |
---------
   |    |
```

    Use a byte array to remember the contents of the board, with the value 1 to signify "X," a -1 to signify a "O," and a 0 to signify an empty box. [answer]

9. Under the Hood

Let's stop for a chapter to lift Forth's hood and see what goes on inside.

Some of the information contained herein we've given earlier, but, at the risk of redundancy, we're now going to view the Forth "machine" as a whole, to see how it all fits together.

Inside INTERPRET

Back in the first chapter we learned that the text interpreter, whose name is INTERPRET, picks words out of the input stream and tries to find their definitions in the dictionary. If it finds a word, INTERPRET has it executed.
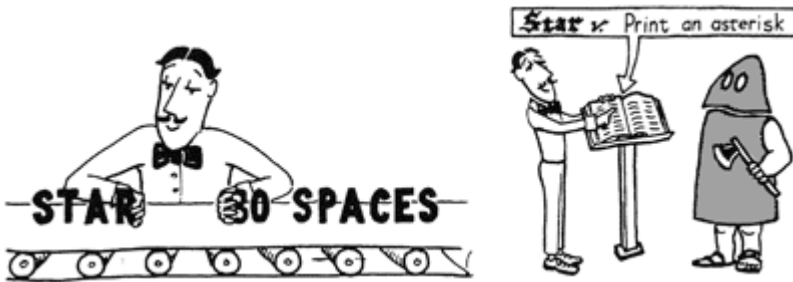
We can perform these separate operations ourselves by using words that perform the component functions of INTERPRET. For instance, the word **'** (pronounced *tick*) finds a definition in the dictionary and returns its *execution token*. If we have defined GREET as we did in Chap. 1, we can now say

```
' GREET U.↵ 4956608  ok
```

and discover the *execution token* of GREET (whatever it happens to be).

We may also directly use EXECUTE. INTERPRET will execute a definition, given its execution token ("xt") on the stack. Thus we can say

```
' GREET EXECUTE↵ Hello, I speak Forth ok
```

and accomplish the same thing as if we had merely said GREET, only in a more roundabout way.

If tick cannot find a word in the dictionary, it executes ABORT" and prints an error message.

Forth's text interpreter uses a word related to tick that returns a zero flag if the word is found. The name and usage of the word varies, but the conditional structure of the INTERPRET phrase always looks like this:

```
(find the word) IF     (convert to a number)
                ELSE   (execute the word)
                THEN
```

that is, if the string is *not* a defined word in the dictionary, INTERPRET tries to convert it as a number. If it is a defined word, INTERPRET executes it.

The word **'** has several uses. For instance, you can use the phrase

```
' GREET .
```

to find out whether GREET has been defined, without actually having to execute it (it will either print the xt or respond with an error).

You can also use the xt to DUMP the contents of the definition, like this:

```
' GREET 12 CELLS DUMP
 A054620: 68 13 40 00  00 00 00 00 - 60 3D 03 0A  15 48 65 6C  h.@.....'=...Hel
 A054630: 6C 6F 2C 20  49 20 73 70 - 65 61 6B 20  46 6F 72 74  lo, I speak Fort
 A054640: 68 20 20 20  38 02 41 00 - 00 00 00 00  00 00 00 00  h   8.A.........
 ok
```

Or you can use tick to implement something called "vectored execution." Which brings us to the next section…

Vectored Execution

While it *sounds* hairy, the idea of vectored execution is really quite simple. Instead of executing a definition *directly*, as we did with the phrase

```
' GREET EXECUTE
```

we can execute it *indirectly* by keeping its xt in a variable, then executing the contents of the variable, like this:

```
' GREET pointer !
pointer @ EXECUTE
```

The advantage is that we can change the pointer later, so that a single word can be made to perform different things at different times.

Here is an example that you can try yourself:

```
( 1 ) : HELLO    ." Hello " ;
( 2 ) : GOODBYE  ." Goodbye " ;
( 3 ) VARIABLE 'aloha  ' HELLO 'aloha !
( 4 ) : ALOHA    'aloha @ EXECUTE ;
```

In the first two lines, we've simply created words which print the strings "Hello" and "Goodbye." In line 3, we've defined a variable called 'aloha. This will be our pointer. We've initialized the pointer with the xt of HELLO. In line 4, we've defined the word ALOHA to execute the definition whose xt is in 'aloha.

Now if we execute ALOHA, we will get

```
ALOHA↵ Hello ok
```

Alternatively, if we execute the phrase

```
' GOODBYE 'aloha !
```

to store the xt of GOODBYE into 'aloha, we will get

```
ALOHA↵ Goodbye ok
```

Thus the same word, ALOHA, can do two different things.

Notice that we named our pointer 'aloha (which we would pronounce *tick-aloha*). Since *tick* provides an xt, we use it as a prefix to suggest "the xt of" ALOHA. It is a Forth convention to use this prefix for vectored execution pointers.

Tick always goes to the next word in the *input stream*. What if we put tick inside a definition? When we execute the definition, tick will find the next word in the input stream, not the next word in the definition. Thus we could define

```
: SAY  ' 'aloha ! ;
```

then enter

```
SAY HELLO↵ ok
ALOHA↵ Hello ok
```


or

```
SAY GOODBYE↵ ok
ALOHA↵ Goodbye ok
```

to store the xt of either HELLO or GOODBYE into 'aloha.

But what if we want tick to use the next word in the *definition*? We must use the word ['] (*bracket-tick-bracket*) instead of tick. For example:

```
: COMING   ['] HELLO   'aloha ! ;
: GOING    ['] GOODBYE 'aloha ! ;
```

Now we can say

```
COMING↵ ok
ALOHA↵ Hello ok
GOING↵ ok
ALOHA↵ Goodbye ok
```

Here are the commands we've covered so far:

' xxx (-- addr)	Attempts to find the execution token of *xxx* (the word that follows in the input stream) in the dictionary.
['] compile time (--) run time (-- addr)	Used only in a colon definition, compiles the execution token of the next word in the definition as a literal.

The Structure of a Dictionary Entry

All definitions, whether they have been defined by **:**, by VARIABLE, by VALUE, by CREATE, or by any other "defining word," share these basic parts:

```
name field
link field
code pointer field
data field
```

Using the variable DATE as an example, here's how these components are arranged within each dictionary entry. In this diagram, each horizontal line represents one cell in the dictionary:
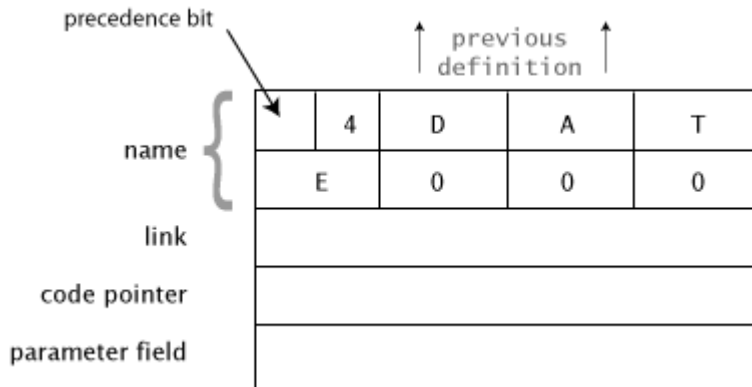
No two Forth systems are alike in this respect. There may be more basic parts, their size may differ, and the order of the components almost certainly differs.

In this book we're only concerned with the *functions* of the four components, not with their *order* inside a dictionary entry.

Name

In our example, the first byte contains the number of characters in the full name of the defined word (there are four letters in DATE). The next four bytes contain the ASCII representations of the four letters in the name of the defined word.

Notice the "precedence bit" in the diagram. This bit is used during compilation to indicate whether the word is supposed to be executed during compilation, or to simply be compiled into the new definition. More on this in Chap. 11.

Link
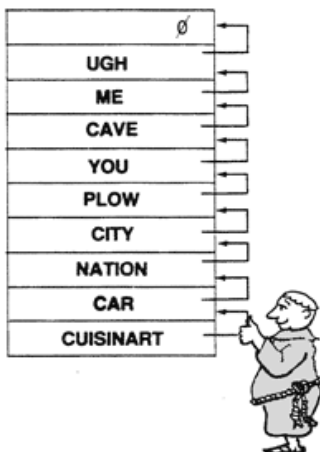
The "link" cell contains the address of the previous definition in the dictionary list. The link cell can be used in linearly searching the dictionary. To simplify things a bit, imagine that it works this way:

Each time the compiler adds a new word to the dictionary, he sets the link field to point to the address of the previous definition. Here he is setting the link field of CUISINART to point to the definition of CAR.
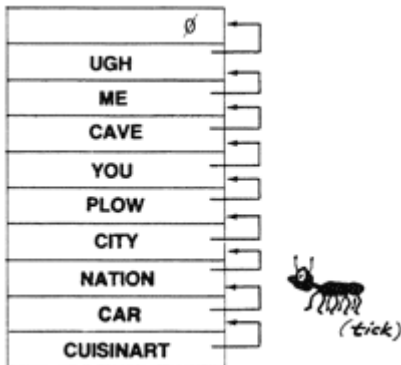
At search time, tick (or bracket-tick-bracket, etc.) starts with the most recent word and follows the "chain" backwards, using the address in each link cell to locate the next definition back.

The link field of the first definition in the dictionary contains a zero, which tells tick to give up; the word is not in the dictionary.

Code Pointer
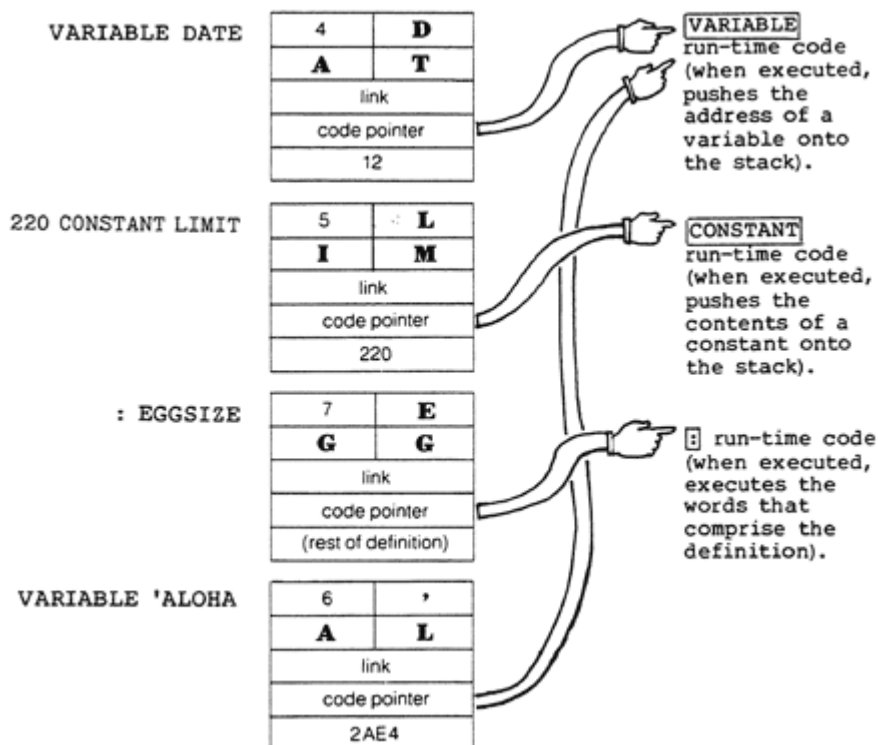
Next is the "code pointer." The xt contained in this pointer is what distinguishes a variable from a constant or a colon definition. It is the address of the instruction that is executed first when a particular type of word is executed. Conceptually, in the case of a variable, the pointer points to code that pushes the address of the variable on the data stack. In the case of a constant, the pointer points to code that pushes the contents of the constant on the data stack. In the case of a colon definition, the pointer points to code that executes the rest of the words in the colon definition. In practice there are many ways to implement this concept, including native code realizations.

The code that is pointed to is called the "run-time code" because it is used when a word of that type is executed (not when a word of that type is defined or compiled).

All variables (conceptually) have the same code pointer; all constants have the same code pointer of their own, and so on.

Data field

Following the code pointer is the data field. In variables and constants, the data field is only one cell. In a 2VARIABLE or a 2CONSTANT, the data field is two cells. In an array, the data field can be as long as you want it. In a colon definition, the length of the data field depends on the length of the definition, as we'll explain in the next section. Strictly speaking, the colon definition of a modern Forth does not have a data field.

The xt that is supplied by tick and expected by EXECUTE is the code pointer defined above. The beginning of the data field can be found with >BODY, a word that computes the data field given an xt. >BODY does not work for colon definitions. Some Forths may even forbid the use of >BODY on any system data structure (variables constants, user, etc.).

The Basic Structure of a Colon Definition

While the format of the head and code pointer is the same for all types of definitions, the format of the data field varies from type to type. Let's look at the data field of a colon definition.

The data field of a colon definition contains the *xts* of the previously defined words which comprise the definition. Here is the dictionary entry for the definition of PHOTOGRAPH, which is defined as

```
: PHOTOGRAPH   SHUTTER OPEN   TIME EXPOSE   SHUTTER CLOSE ;
```

10	**P**
H	**O**
link	
code pointer	
adr of **SHUTTER**	
adr of **OPEN**	
adr of **TIME**	
adr of **EXPOSE**	
adr of **SHUTTER**	
adr of **CLOSE**	
adr of **EXIT**	

(parameter field: brace spanning from adr of SHUTTER to adr of EXIT)
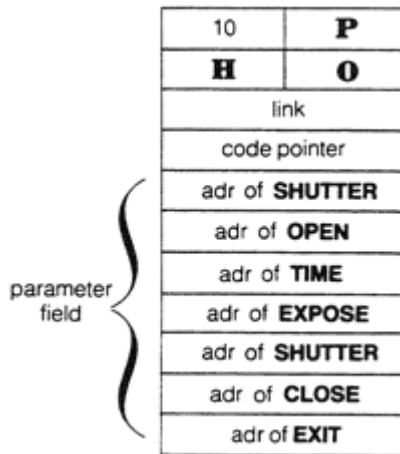
When PHOTOGRAPH is executed, the definitions that are pointed to by the successive xts are executed in turn. The mechanism which reads the list of xts and executes the definitions they point to is called the "address interpreter."

The word **;** at the end of the definition compiles the xt of a word called EXIT. As you can see in the figure, the xt of EXIT resides in the last cell of the dictionary entry. The address interpreter will execute EXIT when it gets to this address, just as it executes the other words in the definition. EXIT terminates the execution of the address interpreter, as we will see in the next section.

Nested Levels of Execution

The function of EXIT is to return the flow of execution to the next higher-level definition that refers to the current definition. Let's see how this works in simplified terms.

Suppose that DINNER consists of three courses:

```
: DINNER   SOUP ENTREE DESSERT ;
```

and that tonight's ENTREE consists simply of

```
: ENTREE   CHICKEN RICE ;
```
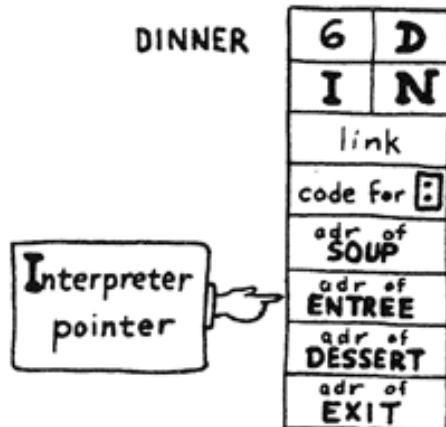

We are executing DINNER and we have just finished the SOUP. The pointer that is used by the address interpreter is called the "interpreter pointer". Since the next course after the SOUP is the ENTREE, our interpreter pointer is pointing to the cell that contains the xt of ENTREE.

The action the address interpreter performs can be seen as "subroutine calling" all the xts in the list, with the return stack used to keep return addresses, and the EXIT working as the machine's RTS (return from subroutine) instruction.
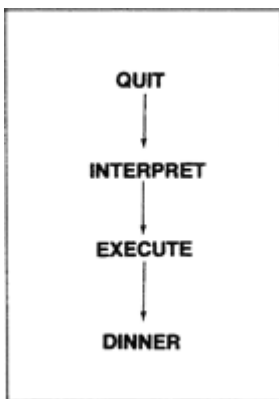
One Step Beyond

Now you're of course wondering: what happens when we finally execute the EXIT in DINNER. Whose return address is on the return stack? What do we return to?

Well, remember that DINNER has just been executed by EXECUTE, which is a component of
INTERPRET. INTERPRET is a loop which checks the entire input stream. Assuming that we entered ↵ after DINNER, then there is nothing more to interpret. So when we exit INTERPRET, where does that leave us? In the outermost definition of each terminal, called QUIT.

QUIT, in simplified form, looks like this:

```
: QUIT  BEGIN  (clear return stack)
               (accept input)
               INTERPRET
               ." ok " CR
               AGAIN ;
```

(The parenthetical comments represent words and phrases not yet covered.) We can see that after the word INTERPRET comes a dot-quote message, "ok," and a CR, which of course are what we see after interpretation has been completed.

Next is the phrase

```
AGAIN
```

which unconditionally returns us to the beginning of the loop, where we clear the return stack and once again wait for input.

If we execute QUIT at any level of execution, we will immediately cease execution of our application and re-enter QUIT's loop. The return stack will be cleared (regardless of how many levels of return addresses we had there, since we could never use any of them now) and the system will wait for input. You can see why QUIT can be used to keep the message "ok" from appearing at our terminal.

The definition of ABORT" uses QUIT.

Abandoning the Nest

It's possible to include EXIT in the middle of a definition. For example, if we were to redefine ENTREE as follows:

```
: ENTREE  CHICKEN EXIT RICE ;
```

then when we subsequently execute DINNER, we will exit right after CHICKEN and return to the next course after the ENTREE, i.e., DESSERT.
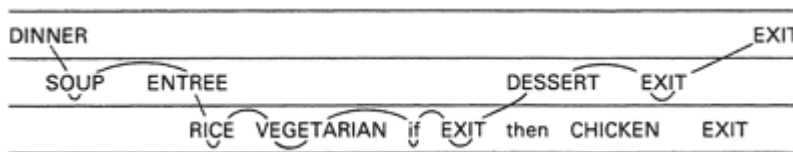
EXIT is commonly used to exit from deeply nested conditional structures.

EXIT (--) When compiled within a colon definition, terminates execution at that point.
QUIT (--) Clears all stacks and returns control to the terminal. No message is given.

Forth Geography

```
low memory  ┌─────────────────────────┐
            │    pre-compiled Forth   │
            ├─────────────────────────┤
            │    system variables     │
            ├─────────────────────────┤
            │    load definitions     │
            ├─────────────────────────┤
            │    user dictionary      │
            │           ↓             │
HERE ─────▶ ├─────────────────────────┤
            │                         │
            ├─────────────────────────┤
            │          PAD            │
            ├─────────────────────────┤
            │           ↑             │
            ├─────────────────────────┤
            │    parameter stack      │
            ├─────────────────────────┤
            │          TIB            │
            │           ↓             │
            ├─────────────────────────┤
            │           ↑             │
            ├─────────────────────────┤
            │      return stack       │
            ├─────────────────────────┤
            │     user variables      │
            ├─────────────────────────┤
            │        buffers          │
high memory └─────────────────────────┘
```
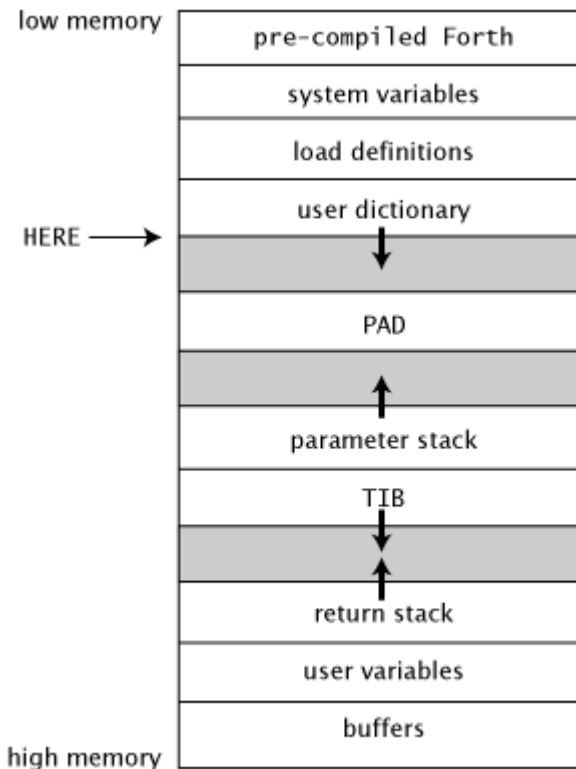
This is the memory map of a typical Forth system:

System Variables

This section of memory contains "system variables" which are created by the basic Forth core and used by the entire system. They are not generally used by the user.

User Dictionary

The dictionary will grow into higher memory as you add your own definitions. The next available cell in the dictionary at any time is pointed to by a variable called CP. During the process of compilation, the pointer CP is adjusted cell-by-cell as the entry is being added to the dictionary. Thus CP is the compiler's bookmark; it points to the place in the dictionary where the compiler can next compile.

CP is also used by the word ALLOT, which advances CP by the number of bytes given. For example, the phrase

```
5 CELLS ALLOT
```

adds twenty to CP so that the compiler will leave room in the dictionary for a five-cell array.

A related word is HERE, which is simply defined as

```
: HERE   CP @ ;
```

to put the value of CP on the stack. The word **,** (comma), which stores a single-length value into the next available cell in the dictionary, is simply defined

```
: ,   HERE !  CELL ALLOT ;
```

that is, it stores a value into HERE and advances the dictionary pointer one cell to leave room for it.

You can use HERE to determine how much memory any part of your application requires, simply by comparing the HERE from before with the HERE after compilation. For example,

```
HERE  S" random.frt" INCLUDED  HERE SWAP - .⏎ 196 ok
```

indicates that the definitions loaded by the file random.frt filled 196 bytes of memory space in the dictionary.

The Pad

At a certain distance from HERE in your dictionary, you will find a small region of memory called the "pad." Like a scratch pad, it is usually used to hold ASCII character strings that are being manipulated prior to being sent out to a terminal. For example, the number formatting words use the pad to hold the ASCII numerals during the conversion process, prior to TYPE.

The size of the pad is indefinite. In most systems there are hundreds of kilobytes between the beginning of the pad and the top of the parameter stack.

Since the pad's beginning address is defined relative to the last dictionary entry, it moves every time you add a new definition or execute FORGET or MARKER. This arrangement proves safe, however, because the pad is never used when any of these events are occurring. The word PAD returns the current address of the beginning of the pad. It is defined simply:

```
: PAD   HERE 340 + ;
```

that is, it returns an address that is a fixed number of bytes beyond HERE. (The actual number varies.)

Parameter Stack

Far above the pad in memory is the area reserved for the parameter stack. Although we like to imagine that values actually move up or down somewhere as we "pop them off" and "push them on," in reality nothing moves. The only thing that changes is a pointer to the "top" of the stack.

As you can see below, when we "put a number on the stack," what really happens is that the pointer is "decremented" (so that it points to the next available location towards low memory), then our number is stored where the pointer is pointing. When we "remove a number from the stack," the number is fetched from the location where the pointer is pointing, then the pointer is incremented. Any numbers above the stack pointer on our map are meaningless.
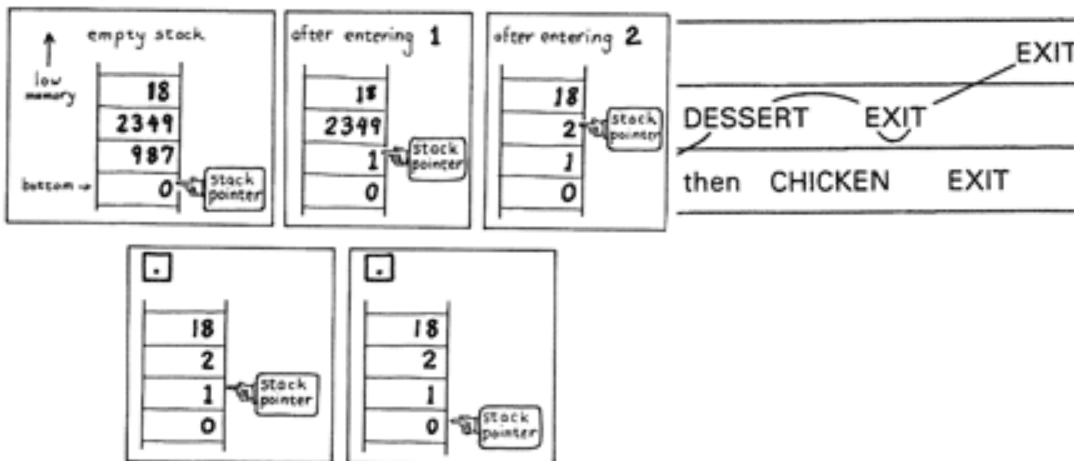
As new values are added to the stack, it "grows towards low memory."

The stack pointer is fetched with the word SP@ (pronounced *s-p-fetch*). Since SP@ provides the address of the top stack location, the phrase

```
SP@ @
```

fetches the contents of the top of stack. This operation, of course, is identical to that of DUP. If we had five values on the stack, we could copy the fifth one down with the phrase

```
SP@ 4 CELLS + @
```

(but this is not considered good programming practice).
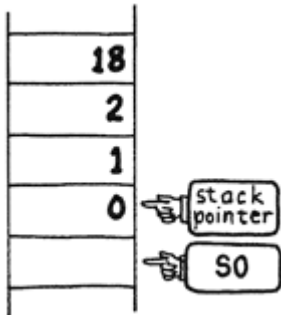
The bottom of the stack is pointed to by a variable called SP0 (*s-p-zero*). SP0 always contains the address of the next cell *below* the "empty stack" cell.

Notice that with double-length numbers, the high-order cell is stored at the lower memory address whether on the stack or in the dictionary. The operators 2@ and 2! keep the order of the cells consistent.
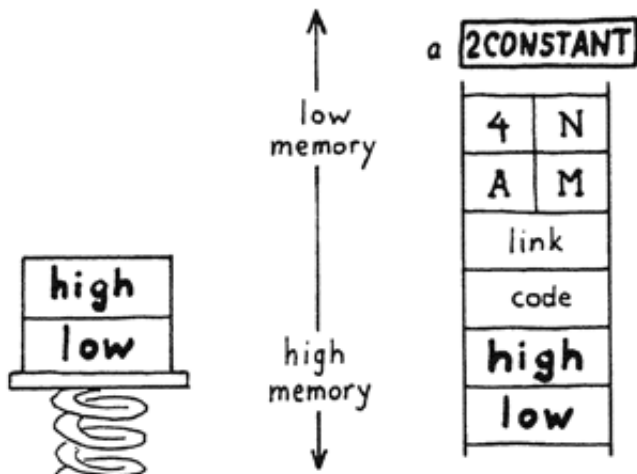
Input Message Buffer
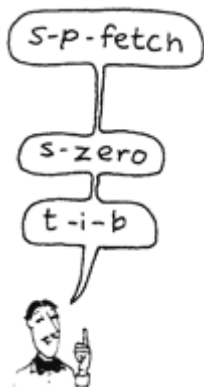
TIB contains the starting address for the "input message buffer," or "Terminal Input Buffer," which grows towards *high* memory (the same direction as the pad). When you enter text from the terminal, it gets stored into this buffer where the text interpreter will scan it.

Return Stack

Above the buffer resides the return stack, which operates identically to the parameter stack. There are no high-level Forth words analogous to SP0 and SP@ that refer to the return stack.

User Variables

The next section of memory contains "user variables." These variables include BASE, SP0, and many others that we'll cover in an upcoming section.

This completes our journey across the memory map of a typical Forth system. Here are the words we've just covered that relate to memory regions in the Forth system:

HERE (-- addr) Returns the next available dictionary location.

PAD (-- addr) Returns the beginning address of a scratchpad area used to hold character strings for intermediate processing.

SP@ (-- addr) User variable. Return the address of the top of the stack before SP@ is executed.

SP0 (-- addr) User variable. Contains the address of the bottom of the parameter stack.

User Variables

The following list shows most of the user variables. Some we won't ever mention again. Don't try to memorize this table. Just remember where you can find it.

TIB   Contains the address of the start of the terminal input buffer.
#TIB Contains the size of the terminal input buffer.
SCR   A pointer to the current block number (set by LIST).
BASE Number conversion base.
CP   Dictionary pointer. Pointer to the next available byte.
>IN   A pointer to the current position in the input stream.
BLK   If non-zero, a pointer to the block being interpreted by LOAD. A zero indicates interpretation from the terminal (via the input message buffer).

User variables are not like ordinary variables. With an ordinary variable (one defined by the word VARIABLE), the value is kept in the body of the dictionary entry. Each user variable, on the other hand, is kept in an array called the "user table." The dictionary entry for each user variable is located elsewhere; its body contains an offset into the user table. When you execute the name of a user variable, such as CP, this offset is added to the beginning address of the user table, allowing you to use @ or ! in the normal way.
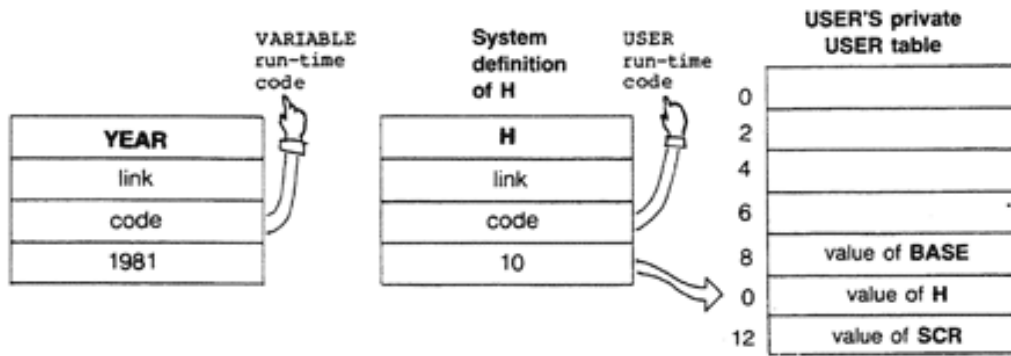
The main advantage of user variables is that any number of tasks can use the same *definition* of a variable and each get its own *value* (because each task has not only its own stacks, but also its own user table). Each task that executes

```
BASE @
```

gets the value for BASE from its own user table. This saves a lot of room in the system while still allowing each task to execute independently.

User variables are defined by the word USER. The sequence of user variables in the table and their offset values vary from one system to another.

To summarize, there are three kinds of variables: System variables contain values used by the entire Forth system. User variables contain values that are unique for each task, even though the definitions can be used by all tasks in the system. Regular variables can be accessible either system-wide or within a single task only.

Forth Vocabularies

Forth systems provide ways to manage the dictionary by organizing words into vocabularies. For example, the programmer can create a vocabulary for all words related to a specific project. The system can be instructed which vocabularies to search for a word, and in which order.

For information about vocabularies, refer to the ANS or ISO Forth standard or to the Forth Programmer's Handbook, which offers discussion of scoping, vocabularies and search orders.

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

' xxx	(-- addr)	Attempts to find the execution token of *xxx* (the word that follows in the input stream) in the dictionary.
[']	compile time: (--) run time: (-- addr)	Used only in a colon definition, compiles the execution token of the next word in the definition as a literal.
EXECUTE	(xt --)	Executes the dictionary entry whose execution token is on the stack.
EXIT	(--)	When compiled within a colon definition, terminates execution at that point.
QUIT	(--)	Clears all stacks and returns control to the terminal. No message is given.
HERE	(-- addr)	Returns the next available dictionary location.
PAD	(-- addr)	Returns the beginning address of a scratchpad area used to hold character strings for intermediate processing.
SCR	(-- addr)	User variable. A pointer to the current block number (set by LIST).
BASE	(-- addr)	User variable. Number conversion base.
SP@	(-- addr)	User variable. Return the address of the top of the stack before SP@ is executed.

TIB	(-- addr)	User variable. Contains the address of the start of the terminal input buffer.
#TIB	(-- addr)	User variable. Contains the size of the terminal input buffer.
SP0	(-- addr)	User variable. Contains the address of the bottom of the parameter stack.
>IN	(-- addr)	User variable. A pointer to the current position in the input stream.
BLK	(-- addr)	User variable. If non-zero, a pointer to the block being interpreted by LOAD. A zero indicates interpretation from the terminal (via the input message buffer).

Review of Terms

Address interpreter
> The second of Forth's two interpreters, the one which executes the data (list of addresses, list of calls, machine code, ...) found in the dictionary entry of a colon definition. The address interpreter also handles the nesting of execution levels for words within words.

Body
> the code and data field of a Forth dictionary entry.

Cfa
> code field address; the address of a dictionary entry's code pointer field.

Code pointer field
> the cell in a dictionary entry which somehow points out the xt of the run-time code for this particular type of definition. For example, in a dictionary entry compiled by **:**, the field would point out the address interpreter.

Defining word
> a Forth word which creates a dictionary entry. Examples include **:**, CONSTANT, VARIABLE, etc.

Head
> the name and link fields of a Forth dictionary entry.

Input message buffer
> the region of memory within a terminal task that is used to store text as it arrives from the terminal. Incoming source text is interpreted here.

Link field
> the cell in a dictionary entry which contains the address of the previous definition, used in searching the dictionary.

Name field
> the area of a dictionary entry which contains the name of the defined word, along with the number of characters in the name.

Pad
> the region of memory within a terminal task that is used as a scratch area to hold character strings for intermediate processing.

Data field
> the area of a dictionary entry which contains the "contents" of a definition: for a CONSTANT, the value of the constant, for a VARIABLE, the value of the variable; for a colon definition, the list of xts of words that are to be executed in turn when the definition is executed.

Run-time code
> a routine, compiled in memory, which specifies what happens when a member of a given class of words is executed. The run-time code for a colon definition is the address interpreter; the run-time code for a variable pushes the address of the variable's body on the stack.

System variable
> one of a set of variables provided by Forth, which are referred to system-wide (by any task). Contrast with "user variables."

Task
> in Forth, a partition in memory that contains at minimum a parameter and a return stack and a set of user variables.

User variable

one of a set of variables provided by Forth, whose values are unique for each task. Contrast with "system variables."

Vectored execution

the method of specifying code to be executed by providing not the address of the code itself, but the address of a location which contains the xt of the code. This location is often called "the vector." As circumstances change within the system, the vector can be reset to point to some other piece of code.

Problems — Chapter 9

1. First review Chap. 2, Prob. 6. Without changing any of those definitions, write a word called COUNTS which will allow the judge to optionally enter the number of counts for any crime. For instance, the entry

2. `CONVICTED-OF  BOOKMAKING  3 COUNTS TAX-EVASION WILL-SERVE↵ 17 years ok`

    will compute the sentence for one count of bookmaking and three counts of tax evasion. [answer]

3. What's the beginning address of your private dictionary? [answer]
4. In your system, how far is the pad from the top of your private dictionary? [answer]
5. Assuming that DATE has been defined by VARIABLE, what is the difference between these two phrases:

6. `DATE .`

    and

    `' DATE .`

7. What is the difference between these two phrases:

8. `BASE .`

    and

    `' BASE .`

    [answer]

9. In this exercise you will create a "vectored execution array," that is, an array which contains xts of Forth words. You will also create an operation word which will execute one word stored in the array when the operation word is executed.

    Define a one-dimensional array of cells which will return the nth element's address when given a subscript n. Define several words which output something at your terminal and take no inputs.

10. Store the xts of these output words in various elements of the array. Store the address of a do-nothing word in any remaining elements of the array. Define a word which will take a valid array index and execute the word whose address is stored in the referenced element.

    For example,

11. `1 DO-SOMETHING↵ Hello, I speak Forth. ok`
12. `2 DO-SOMETHING↵ 1 2 3 4 5 6 7 8 9 10 ok`
13. `3 DO-SOMETHING↵`
14. `**********`

```
15.  **********
16.  **********
17.  **********
18.  ********** ok
19.  4 DO-SOMETHING↵ ok
20.  5 DO-SOMETHING↵ ok
```

[answer]

10. I/O and You

In this chapter, we'll explain how Forth handles I/O of character strings to and from disk and the terminal.🖰

Specifically, we'll discuss disk-access commands, output commands, string-manipulation commands, input commands, and number-input conversion.

Output Operators

The word EMIT takes a single ASCII representation on the stack, using the low-order byte only, and prints the character at your terminal. For example, in decimal:

```
65 EMIT↵ A ok
66 EMIT↵ B ok
```

The word TYPE prints an entire string of characters at your terminal, given the starting address of the string in memory and the count, in this form:

```
( addr u -- )
```

We've already seen TYPE in our number-formatting definitions without worrying about the address and count, because they are automatically supplied by #>.

Let's give TYPE an address that we know contains a character string. Remember that the starting address of the terminal input buffer is returned by TIB? Suppose we enter the following command:

```
TIB #TIB @ TYPE
```

This will type 15 characters from the terminal input buffer, which contains the command we just entered:

```
TIB #TIB @ TYPE↵ TIB #TIB @ TYPE ok
```

Let's digress for a moment to look at the operation of .". At compile time, when the compiler encounters a dot-quote, it compiles the ensuing string right into the dictionary, letter-by-letter, up to the delimiting double-quote. To keep track of things, it also compiles the *count* of characters into the dictionary entry. Given the definition

```
: TEST   ." sample " ;
```

and looking at bytes in the dictionary horizontally rather than vertically, here is what the compiler has compiled:

4	T	E	S	T		link	colon code	."		7	S	A	M	P	L	E	EXIT

If we wanted to, we could type the word "SAMPLE" ourselves (without executing TEST) with the phrase

```
' TEST >BODY CELL+ 1+ 7 TYPE
```

where

```
' TEST >BODY
```

gives us the body address of TEST,

```
CELL+ 1+
```

offsets us past the address and the count, to the beginning of the string (the letter "s"), and

```
7 TYPE
```

types the string "sample."

That little exercise may not seem too useful. But let's go a step further.

Remember how we defined LABEL in our egg-sizing application, using nested IF...THEN statements? We can rework our definition using TYPE. First let's make all the labels the same length and "string them together" within a single definition as a string array. (We can abbreviate the longest label to "XTRA LRG" so that we can make each label eight characters long, including trailing spaces.)

```
: "LABEL"  ." REJECT  SMALL   MEDIUM  LARGE   XTRA LRGERROR   " ;
```

Once we enter

```
' "LABEL" >BODY CELL+ 1+
```

to get the address of the start of the string, we can type any particular label by offsetting into the array. For example, if we want label 2, we simply add sixteen (2 x 8) to the starting address and type the eight characters of the name:

```
16 + 8 TYPE
```

Now let's redefine LABEL so that it takes a category-number from zero through five and uses it to index into the string array, like this:

```
: LABEL   8 *  ['] "LABEL" >BODY CELL+ 1+  +  8 TYPE  SPACE ;
```

Recall that the word ['] is just like ' except that it may only be used inside a definition to compile the address of the next word in the definition (in this case, "LABEL"). Later, when we execute LABEL, bracket-tick-bracket followed by to-body will push the body address of "LABEL" onto the stack. The number corresponding to CELL+ 1+ is added, then the string offset is added to compute the address of the particular label name that we want.

This kind of string array is sometimes called a "superstring." As a naming convention, the name of the superstring usually has quotes around it. Note that this method is in practice never used, as the same result can be had with the completely portable ANS Forth word C", as follows:

```
: "LABEL"  C" REJECT  SMALL   MEDIUM  LARGE   XTRA LRGERROR   " ;
: LABEL   8 *  "LABEL" 1+ +  8 TYPE  SPACE ;
```

Our new version of LABEL will run a little faster because it does not have to perform a series of comparison tests before it hits upon the number that matches the argument. Instead it uses the argument to compute the address of the appropriate string to be typed.

Notice, though, that if the argument to LABEL exceeds the range zero through five, you'll get garbage. If LABEL is only going to be used within EGGSIZE in the application, there's no problem. But if an "end user," meaning a person, is going to use it, you'd better "clip" the index, like this:

```
: LABEL   0 MAX 5 MIN  LABEL ;
```
TYPE (addr u --) Transmits u characters, beginning at address, to the current output device.

Outputting Strings from Disk

We mentioned before that the word BLOCK copies a given block into an available buffer and leaves the address of the buffer on the stack. Using this address as a starting-point, we can index into one of the buffer's 1,024 bytes and type any string we care to. For example, to print line 0 of block 1, we could say (assuming you've executed USE blocks.forth)

```
CR 1 BLOCK 64 TYPE↵
```
                                                                    ok

To print line eight, we could add 512 (8 x 64) to the address, like this:

```
CR 1 BLOCK  512 + 64 TYPE
```

Before we give a more interesting example, it's time to introduce a word that is closely associated with TYPE.

-                (addr u1 --   Eliminates trailing blanks from the string that starts at the address by reducing
TRAILING addr u2)   the count from u1 (original byte count) to u2 (shortened byte count).

Handy Hint:
A Random Number Generator

This simple random number generator can be useful for games, although for more sophisticated applications such as simulations, better versions are available.

```
( Random number generation -- High level )
VARIABLE rnd   HERE rnd !
: RANDOM  rnd @ 31421 *  6927 +  DUP rnd ! ;
: CHOOSE  ( u1 -- u2 )  RANDOM UM*  NIP ;

( where CHOOSE returns a random integer
  within the range 0 = or < u2 < u1. )
```

Here's how to use it:

To choose a random number between zero and ten (but exclusive of ten) simply enter

```
10 CHOOSE
```

and CHOOSE will leave the random number on the stack.

-TRAILING can be used immediately before the TYPE command so that trailing blanks will not be printed. For instance, inserting it into our first example above would give us

```
CR 1  BLOCK 64
-TRAILING TYPE↵
 ok
```

The following example uses TYPE

```
USE blocks.forth
: POOF
  16 CHOOSE 64 *
  2 BLOCK +
  CR 64 -TRAILING
  TYPE ;
```

try it:

```
POOF↵
qualified ok
POOF↵
flexible ok
POOF↵
total ok
```

Internal String Operators

The commands for moving character strings or data arrays are very simple. Each requires three arguments: a source address, a destination address, and a count.

	(addr1 addr2 u --)	
CMOVE	(addr1 addr2 u --)	Copies a region of memory u bytes long, byte-by-byte beginning at addr1, to memory beginning at addr2. The move begins with the contents of addr1 and proceeds toward high memory.
CMOVE>	(addr1 addr2 u --)	If u is greater than zero, copy u consecutive characters from the data space starting at c-addr1 to that starting at c-addr2, proceeding character-by-character from higher addresses to lower addresses.
MOVE	(addr1 addr2 u --)	After this move, the u bytes at addr2 contain exactly what the u bytes at addr1 contained before the move (no "clobbering" occurs).

Notice that these commands follow certain conventions we've seen before:

- When the arguments include a source and a destination, the source precedes the destination.
- When the arguments include an address and a count (as they do with TYPE), the address precedes the count.

And so with these three words the arguments are
(source destination count --)

To move the entire contents of a buffer into the PAD, for example, we would write

```
210 BLOCK  PAD  1024 CMOVE
```

although on cell-address machines the move might be made faster if it were cell-by-cell, like this:

```
210 BLOCK  PAD  1024 MOVE
```

The word CMOVE> lets you move a string to a region that is higher in memory but that overlaps the source region.
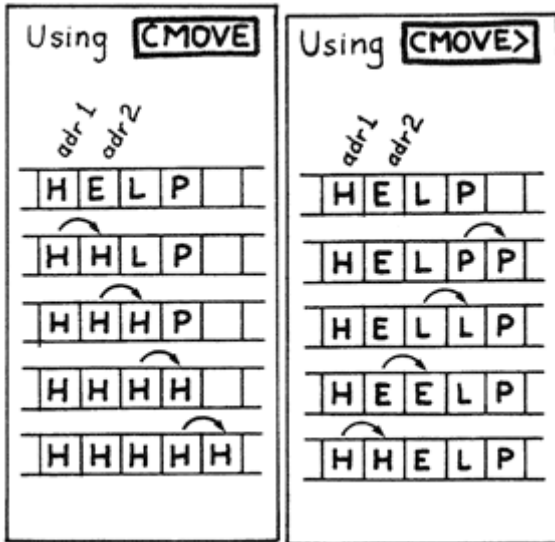
If you were to use CMOVE, the first letter of the string would get copied to the second byte, but that would "clobber" the second letter of the string. The final result would be a string composed of a single character.

Using CMOVE> in this situation keeps the string from clobbering itself during the move.

You probably notice that CMOVE can be used to fill an array with a certain byte. On older systems the word FILL, which we introduced earlier, may have been defined using this trick. On modern Forths it is recommended to explicitly use FILL, if fill is what you want to do. For example, to store blanks into 1024 bytes of the pad, we say

```
PAD  1024 CHAR BL FILL
```

Single-character Input

The word KEY awaits the entry of a single key from your terminal keyboard and leaves the character's ASCII equivalent on the stack in the low-order byte.

To execute it directly, you must follow it with a return, like this:

```
KEY↵
```
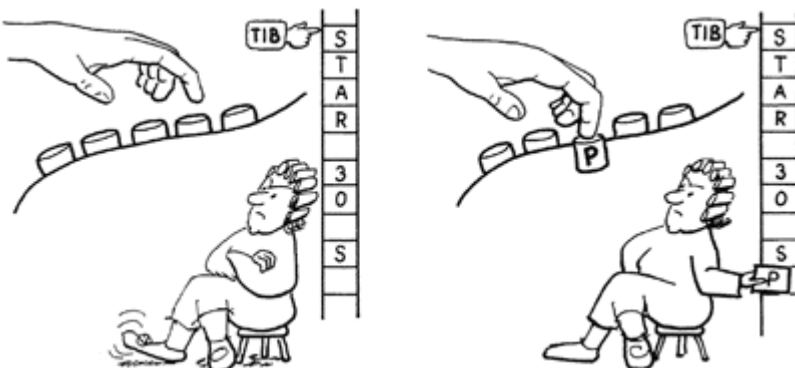

The cursor will advance a space, but the terminal will *not* print the "ok"; it is waiting for your input. Press the letter "A," for example, and the screen will "echo" the letter "A," followed by the "ok." The ASCII value is now on the stack, so enter .:

```
KEY A↵ ok
.↵ 65 ok
```

This saves you from having to look in the table to determine a character's ASCII code.

You can also include KEY inside a definition. Execution of the definition will stop, when KEY is encountered, until an input character is received. For example, the following definition will list a given number of blocks in series, starting with the current block, and wait for you to press any key before it lists the next one:

```
: BLOCKS   ( count -- )
        SCR @ + SCR @ DO  I LIST KEY DROP  LOOP ;
```

In this case we drop the value left by KEY because we don't care what it is.

Or we might add a feature that allows us either to leave the loop at any time by pressing return or to continue by pressing any other key, such as as space. In this case we will perform a conditional test on the value returned by KEY.

```
13 CONSTANT #EOL
: BLOCKS  ( count -- )
        SCR @ +
        SCR @ DO           I LIST
                           KEY #EOL = ( cr) IF  LEAVE  THEN
            LOOP ;
```

Note that in some Forth systems, the carriage-return key is received as a linefeed (10) or as a null (zero).

KEY (-- c) Returns the ASCII value of the next available character from the current input device.

String Input Commands, from the Bottom up

There are several words involved with string input. We'll start with the lower-level of these and proceed to some higher-level words. Here are the words we will cover in this section:

ACCEPT (c-addr u1 -- u2) Receives u characters (or a carriage return) from the terminal keyboard and stores them, starting at the address. The count of received characters is returned.

WORD (c -- addr) Reads one word from the input stream, using the character (usually blank) as a delimiter. moves the string to the address (HERE) with the count in the first byte, leaving the address on the stack.

The word ACCEPT stops execution of the task and waits for input from your keyboard. It expects a given number of keystrokes or a carriage return, whichever comes first. The incoming text is stored beginning at the address given as an argument, the count of received characters is returned on the stack.

For example, the phrase

```
TIB 80 ACCEPT
```

will await up to eighty characters and store them in the Terminal Input Buffer (TIB). (Storing directly in the TIB is not standard, but e.g. iForth has no problem with this tradition.)

This phrase is the one used in the definition of QUIT to get the input for INTERPRET.

Let's move on to the next higher-level string-input operator. We've just explained that QUIT contains the phrase

```
... TIB 80 ACCEPT #TIB !  INTERPRET ...
```

but how does the text interpreter scan the terminal input buffer and pick out each individual word there? With the phrase

```
BL WORD
```

WORD scans the input stream looking for the given delimiter, in this case space, and moves the substring into a different buffer of its own, with the count in the first byte of the buffer. Finally, it leaves the address of the buffer on the stack, so that INTERPRET (or anyone else) knows where to find it. WORD's buffer usually begins at HERE, so the address given is HERE.

WORD looks for the given delimiter in the terminal input buffer, and moves the sub-string to WORD's buffer with the count in the first byte.
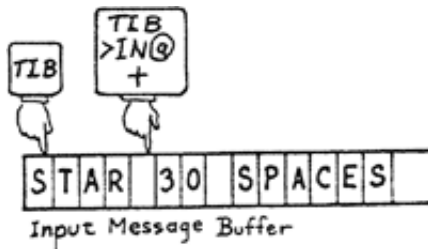
When you are executing words directly from a terminal, WORD will scan the input buffer, starting at TIB. As it goes along, it advances the input buffer *pointer*, called >IN, so that each time you execute WORD, you scan the *next* word in the input stream. WORD knows to stop scanning when >IN @ becomes larger than #TIB @, the count of received characters.

>IN is a "relative pointer"; that is, it does not contain the actual address but rather an offset that is to be added to the actual address, which is is in this case TIB. For example, after WORD has scanned the string "STAR," the value of >IN is five.
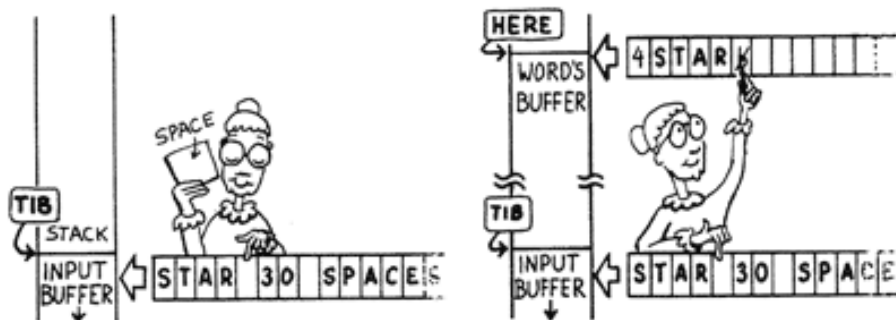
WORD ignores initial occurrences of the delimiter (until any other character is encountered). You could type

```
    STAR
```

(that is, STAR preceded by several spaces) and get exactly the same string in WORD's buffer as shown above.

We'll get back to WORD later on in this chapter. For now, though, let's define a word that uses WORD and that is more useful for handling string input:

```
: TEXT  ( delimiter -- )  PAD 258 BL FILL  WORD COUNT PAD SWAP  MOVE ;
```

TEXT, like WORD, takes a delimiter and scans the input stream until it finds the string delimited by it. It then moves the string to the pad. What is especially nice about TEXT is that before it moves the string, it blanks the pad. This makes it very convenient for use with TYPE. Here's a simple example:

```
CREATE my-name 40 ALLOT
: I'M   BL TEXT  PAD my-name 40 MOVE ;
```

In the first line we define an array called my-name. In the second line we define a word called I'M which will allow us to enter

```
I'M EDWARD↵ ok
```

The definition of I'M breaks down as follows: the phrase

```
BL TEXT
```

scans the remainder of the input stream looking for a space or the end of the line, whichever comes first. (The delimiter that we give to TEXT is actually used by WORD, which is included in the definition of TEXT.) TEXT then moves the phrase to a nice clean "pad."

The phrase

```
PAD my-name 40 MOVE
```

moves forty bytes from the pad into the array called my-name, where it will safely stay for as long as we need it.

We could now define GREET as follows:

```
: GREET   ." Hello, " my-name 40 -TRAILING TYPE  ." , I speak Forth. " ;
```

so that by executing GREET, we get

```
GREET↵ Hello, EDWARD, I speak Forth.  ok
```

Unfortunately, our definition of I'M is looking for a space as its delimiter. This means that a person named Mary Kay will not get her full name into my-name.

To get the complete input stream, we don't want to "see" any delimiter at all, except the end of line. Instead of "BL TEXT," we should use the phrase

```
1 TEXT
```

ASCII 1 is a control character that can't be ever sent from the keyboard and therefore won't ever appear in the input buffer. Thus "1 TEXT" is a convention used to read the entire input buffer, up to the carriage return. By redefining I'M in this way, Mary Kay can get her name into my-name, space and all.

By using other delimiters, such as commas, we can "accept" a series of strings and store each of them into a different array for different purposes. Consider this example, in which the word VITALS uses commas as delimiters to separate three input fields:

```
( Form love letter )

CREATE name 14 ALLOT
CREATE eyes 12 ALLOT
CREATE me   14 ALLOT

: VITALS
        [CHAR] , TEXT  PAD name 14 MOVE
        [CHAR] , TEXT  PAD eyes 12 MOVE
              1 TEXT  PAD me   14 MOVE ;

: LETTER  PAGE
```

```
        ." Dear " name 14 -TRAILING TYPE ." ,"
        CR ." I go to heaven whenever I see your deep "
              eyes 12 -TRAILING TYPE ."  eyes. Can "
        CR ." you go to the movies Friday? "
        CR 30 SPACES ." Love, "
        CR 30 SPACES me 14 -TRAILING TYPE
        CR ." P.S. Wear something " eyes 12 -TRAILING TYPE
              ."  to show off those eyes! " ;
```

Which allows you to enter

```
VITALS Alice,blue,Fred↵ ok
```

then enter

```
LETTER
```

It works every time.

So far all of our input has been "Forth style"; that is, numbers *precede* commands (so that a command will find its number on the stack) and strings *follow* commands (so that a command will find its string in the input stream). This style makes use of one of Forth's unique features: it awaits your commands; it does not prompt you.

But if you want to, you may put ACCEPT inside a definition so that it will request input from you under control of the definition. For example, we could combine the two words I'M and GREET into a single word which "prompts" users to enter their names. For example,

```
GREET↵
What's your name?
```

at which point execution stops so the user can enter a name:

```
GREET↵
What's your name?Travis Mc Gee↵
Hello, Travis Mc Gee, I speak Forth. ok
```

We could do this as follows:

```
: GREET   CR ." What's your name?"
      TIB 40 ACCEPT #TIB !  0 >IN !
      1 TEXT  CR ." Hello, "
      PAD 40 -TRAILING TYPE  ." , I speak Forth. " ;
```

We've explained all the phrases in the above definition except this one:

```
#TIB !  0 >IN !
```

Remember that TEXT, because it uses WORD, always uses >IN as its reference point. But when the user enters the word GREET to execute this definition, the string GREET will be stored in the terminal input buffer and >IN will be pointing beyond "GREET". ACCEPT does *not* use >IN as its reference, so it will store the user's name beginning at TIB, on top of GREET. If you were to execute TEXT *now*, it would miss the first five letters of the user's name. It's necessary to reset >IN to zero so that TEXT will look where ACCEPT has put the name.

Number Input Conversion

When you type a number at your terminal, Forth automatically converts this character string into a binary value and pushes it onto the stack. Forth also provides a command which let you convert a character string that begins at *any* memory location into a binary value.

>NUMBER	(ud1 c-addr1 u1 -- ud2 c-addr2 u2)	ud2 is the unsigned result of converting the characters within the string specified by c-addr1 u1 into digits, using the number in BASE, and adding each into ud1 after multiplying ud1 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any "+" or "-", is encountered or the string is entirely converted. c-addr2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. u2 is the number of unconverted characters in the string.

Here's an example that uses >NUMBER:

```
: PLUS   0. BL WORD COUNT >NUMBER 2DROP DROP  + ." = " . ;
```

PLUS allows us to prove to any skeptic that Forth could use infix notation if it wanted to. We can enter

```
2 PLUS 13↵ = 15 ok
```

When PLUS is executed, the "2" will be put on the stack in binary form, while the "3" will still be in the input stream as a string. The phrase

```
0. BL WORD
```

reads the string and provides the accumulator for >NUMBER; >NUMBER converts it to binary and puts the double-length result plus an unconverted string on the stack. We drop the string and the top half of the double-length result. Now + adds the two single-length values and **.** prints the result.

Note that you can use >NUMBER to create your own specialized number input conversion routines. Since >NUMBER returns the address of the first unconvertible character, you can make decisions based on whether the character is a hyphen, dot, or whatever. You can also make decisions based on the location of the non-convertible character within the number. For instance, you can write a routine that lets you enter a number with a decimal point in it and then scales it accordingly.

To give a good example of the use of >NUMBER, Figure 10-1 shows a definition of NUMBER. This version reads any of the characters

```
: , - . /
```

as valid punctuation marks which cause the value to be returned on the stack as a double-length integer. If none of these characters appear in the string, the value is returned as single-length.

This definition uses the word WITHIN as we defined it in the problems for .

Here we use the variable PUNCT to contain a flag that indicates whether punctuation was encountered. We suggest that you use an available user variable instead.

Figure 10-1. A Definition of NUMBER

VARIABLE punct	Creates a flag that will contain true if the number contains valid punctuation.
: NUMBER (addr u -- n or d)	
0 punct !	Initialize flag, no punctuation has occurred.
OVER C@	Get the first digit.

[CHAR] - =	Is it a minus sign?
DUP >R	Save the flag on the return stack.
IF 1 /STRING THEN	If the first character is "-", adds 1 to the address and decrements the character count. This effectively skips the "-" character, pointing to the real first digit.
0. 2SWAP	provides the double-length zero as an accumulator.
BEGIN	
>NUMBER	Begins conversion; converts until an invalid digit.
DUP	While there are still characters left, fetch the invalid digit.
WHILE	
OVER C@ DUP [CHAR] : =	*a colon, or*
SWAP [CHAR] , [CHAR] / 1+ WITHIN OR	*a comma, hyphen, period or slash.*
DUP punct !	Set punct to indicate whether valid punctuation has occurred.
0= ABORT" ? "	Otherwise issue an error message.
1 /STRING	Skip the punctuation character.
REPEAT	Exits here if a blank is detected; otherwise repeats conversion.
2DROP<	Drop the string from the stack.
R> IF DNEGATE THEN	If the flag on the return stack is true, negates d.
punct @ 0= IF DROP THEN ;	If there was no punctuation, returns a single-length value by dropping the high-order cell.

A Closer Look at WORD

So far we have only talked about using WORD to scan the *terminal* input buffer (which holds the characters that are ACCEPTed from the terminal). But if we recall that the phrase

```
BL WORD
```

is used by the text interpreter, we realize that WORD actually scans the input *stream*, which is either the terminal input buffer, a string being EVALUATEd, or disk memory being LOADed or INCLUDED.

To achieve this flexibility, WORD uses other pointers in addition to >IN. The other pointers make sure WORD looks in memory (when doing EVALUATE), on disk (when doing LOAD or INCLUDED) or in the terminal input buffer.

A useful word to use in conjunction with WORD is COUNT. Recall that WORD leaves the length of the word in the first byte of WORD's buffer and also leaves the *address* of this byte on the stack.

The word COUNT puts the count on the stack and increments the address, like this:

leaving the stack with a string address and a count as appropriate arguments for TYPE, MOVE, etc.

COUNT is used in the definition of TEXT which we gave a few sections back.

COUNT (addr -- addr+1 u) Converts a character string, whose length is contained in its first byte, into the form appropriate for TYPE, by leaving the address of the first character and the length on the stack.

We will further illustrate the use of WORD in one of the examples in Chap. 12.

String Comparisons

Here is a Forth word that you can use to compare character strings:

COMPARE (c-addr1 u1 c-addr2 u2 -- n) Compare the string specified by c-addr1 and u1 to the string specified by c-addr2 and u2. The strings are compared, beginning at the given addresses, character by character up to the length of the shorter string, or until a difference is found. If both strings are the same up to the length of the shorter string, then the longer string is greater than the shorter string. n is -1 if the string specified by c-addr1 and u1 is less than the string specified by c-addr2 and u2. n is zero if the strings are equal. n is 1 if the string specified by c-addr1 and u1 is greater than the string specified by c-addr2 and u2.

COMPARE can be used to test whether two character strings are equal or whether one is alphabetically greater or lesser than the other.

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

TYPE	(addr u --)	Transmits u characters, beginning at address, to the current output device.
-TRAILING	(addr u1 -- addr u2)	Eliminates trailing blanks from the string that starts at the address by reducing the count from u1 (original byte count) to u2 (shortened byte count).
MOVE	(addr1 addr2 u --)	After this move, the u bytes at addr2 contain exactly what the u bytes at addr1 contained before the move (no "clobbering" occurs).
CMOVE	(addr1 addr2 u --)	Copies a region of memory u bytes long, byte-by-byte beginning at addr1, to memory beginning at addr2. The move begins with the contents of addr1 and proceeds toward high memory.
KEY	(-- c)	Returns the ASCII value of the next available character from the current input device.
ACCEPT	(c-addr u1 -- u2)	Receives u characters (or a carriage return) from the terminal keyboard and stores them, starting at the address. The count of received characters is returned.
WORD	(c -- addr)	Reads one word from the input stream, using the character (usually blank) as a delimiter. Moves the string to the address (HERE) with the count in the first byte, leaving the address on the stack.
>NUMBER	(ud1 c-addr1 u1 -- ud2 c-addr2 u2)	ud2 is the unsigned result of converting the characters within the string specified by c-addr1 u1 into digits, using the number in BASE, and adding each into ud1 after multiplying ud1 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any "+" or "-", is encountered or the string is entirely converted. c-addr2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. u2 is the number of unconverted characters in the string.
COUNT	(addr -- addr+1 u)	Converts a character string, whose length is contained in its first byte, into the form appropriate for TYPE, by leaving the address of the first character and the length on the stack.
CMOVE>	(addr1 addr2 u --)	If u is greater than zero, copy u consecutive characters from the data space starting at c-addr1 to that starting at c-addr2, proceeding character-by-character from

	)	higher addresses to lower addresses.
COMPARE	(c-addr1 u1 c-addr2 u2 -- n)	Compare the string specified by c-addr1 and u1 to the string specified by c-addr2 and u2. The strings are compared, beginning at the given addresses, character by character up to the length of the shorter string, or until a difference is found. If both strings are the same up to the length of the shorter string, then the longer string is greater than the shorter string. n is -1 if the string specified by c-addr1 and u1 is less than the string specified by c-addr2 and u2. n is zero if the strings are equal. n is 1 if the string specified by c-addr1 and u1 is greater than the string specified by c-addr2 and u2.
BLANK	(addr u --)	Store ASCII blanks into u bytes of memory, beginning at addr.

Review of Terms

Relative pointer
    A variable which specifies a location in relation to the beginning of an array or string — not the absolute address.
Superstring
    in Forth, a character array which contains a number of strings. Any one string may be accessed by indexing into the array.
Virtual memory
    the treatment of mass storage (such as the disk) as though it were resident memory; also the mechanism of the operating system which makes this treatment possible.

Problems — Chapter 10

1. Enter some famous quotations into an available block, say 3. Now define a word called CHANGE which takes two ASCII values and changes all occurrences within block 3 of the first character into the second character. For example,

2. `  CHAR A CHAR E CHANGE`

    will change all the "A"s into "E"s. [answer]

3. Define a word called FORTUNE which will print a prediction at your terminal, such as "You will receive good news in the mail." The prediction should be chosen at random from a list of sixteen or fewer predictions. Each prediction is sixty-four characters, or less, long. [answer]

4. According to Oriental legend, Buddha endows all persons born in each year with special, helpful characteristics represented by one of twelve animals. A different animal reigns over each year, and every twelve years the cycle repeats itself. For instance, persons born in 1900 are said to be born in the "Year of the Rat." The art of fortune-telling based on these influences of the natal year is called "Juneeshee."

    Here is the order of the cycle:

5. `  Rat Ox Tiger Rabbit Dragon Snake Horse Ram Monkey Cock Dog Boar`

    Write a word called .ANIMAL that types the name of the animal corresponding to its position in the cycle as listed here; e.g.,

    `0 .ANIMAL↵ RAT ok`

Now write a word called (JUNEESHEE) which takes as an argument a year of birth and prints the name of the associated animal. (1900 is the year of the Rat, 1901 is the Ox, etc.)
Finally, write a word called JUNEESHEE which prompts the user for his/her year of birth and prints the name of the person's Juneeshee animal. Define it so that the user won't have to press "return" after entering the year. [answer]

6. Rewrite the definition of LETTER that appears in this chapter so that it uses names and personal descriptions that have been edited into a block, rather than entered into character arrays. In this way, you can keep a file on many "prospects" and produce a letter for any one person with the appropriate descriptions, just by supplying an argument to LETTER, as in

7.   1 LETTER

Now define LETTERS so that it prints one letter for each person in your file.

8. In this exercise you will create and use a virtual array, that is, an array which resides on disk but which is referenced like a memory-resident array (with @ and !).

First select an unused block. Put this block number in a variable. Then define an access word which accepts a cell subscript from the stack, then computes the block number corresponding to this subscript, calls BLOCK and returns the memory address of the subscripted cell. This access word should also call UPDATE.
Test your work so far.

Next use the first cell as a count of how many data items are stored in the array. Define a word PUT which will store a value into the next available cell of the array. Define a display routine which will print the stored elements in the array.

Now use this virtual array facility to define a word ENTER which will accept pairs of numbers and store them in the array.

Finally, define TABLE to print the data entered above, eight members per line. [answer]

11. Extending the Compiler

In comparison with conventional 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 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 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 front, 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
```

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:

- 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?)
- 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.

4	E	X	A	M	P	L	E
link							
execution token							
(body)							

At run time, CREATE pushes the body address 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  0 , ;
```

When we execute VARIABLE as in

```
VARIABLE ORANGES
```

We are indirectly using CREATE to create a dictionary head with the name ORANGES and an xt that points to CREATE's run-time code. Then we are allotting a cell for the variable itself (with "0 ,").

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	Create 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 body address of the word being defined on the stack.
@	Fetches the contents of the constant, using the body address 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.
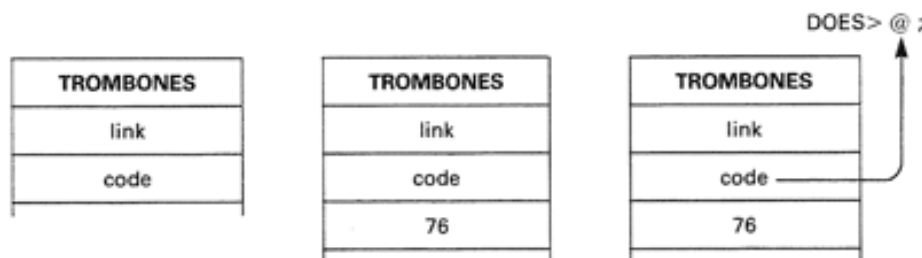
DOES> run time: (-- addr)	Used in creating a defining word; marks the end of its compile-time portion and the beginning of its run-time portion. The run-time operations are stated in higher-level Forth. At run time, the body address of the defined word will be on the stack.

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 characters 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	Create 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 body address of the product-word on the stack at run-time.
DUP	Copies the body address.
CELL+	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 (addr count --).
;	

We have 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 (not CHAR!) 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 body address left by DOES>, 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 ERASE the given number of bytes, starting at HERE, before ALLOTing 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< 0=
                  ABORT" Range error "  + CELL+ ;
```

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< 0=	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 "	Check if the comparison test fails.
+ CELL+	Otherwise adds the argument to the body address, plus an additional cell to skip 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 record on the disk a given byte would be contained in, read the record into a buffer using INCLUDED, and return the address of the desired byte within the buffer. A string defined in this way could span many consecutive records (using the same technique as in Prob. 5, Chap. 10).

    c0 c1 c2 c3

r0

r1

r2   ?
r3

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 DUP , * ALLOT
   DOES> ( member: row col -- addr )
        ROT OVER @ * + +  CELL+ ;
```

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

	c0	c1	c2	c3
r0	0	1	2	3
r1	4	5	6	7
r2	8	9	10	11
r3	12	13	14	15

our real array looks like this

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

If you want the address of the byte in row 2, column 1, it can be computed by multiplying your row number (2) by the number of columns in each row (4) and then adding your column number (1), which indicates that you want the ninth byte in the real array. 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 columns are in each row 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
ROT	col pfa row
OVER @	col pfa row #cols
*	col pfa row-index
+ +	address
CELL+	corrected address

It is necessary to add a cell 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.

```
\ Shapes, using a defining word.

DECIMAL

: star  [CHAR] * EMIT ;

: .row  CR 8 0 DO
    DUP 128 AND IF  star
      ELSE  SPACE
      THEN
    1 LSHIFT
      LOOP  DROP ;

: SHAPE CREATE  8 0 DO  C,  LOOP
  DOES> DUP 7 + DO  I C@ .row  -1 +LOOP  CR ;

  HEX  18 18 3C 5A 99 24 24 24  SHAPE man
  81 42 24 18 18 24 24 81  SHAPE equis
  AA AA FE FE 38 38 38 FE  SHAPE castle
  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
```

The 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 like Fortran or BASIC do not provide this flexibility because these traditional compilers and interpreters 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 with 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 a 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 that 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 HERE 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 BEGIN, DO must provide, at compile time, a HERE for LOOP or +LOOP to return to. But unlike BEGIN, DO also has a run-time behavior: it must push the limit and 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   POSTPONE 2>R   HERE ; IMMEDIATE
```

```
┌──────────────────────────┐
│           ...            │
├──────────────────────────┤
│           2>R            │
├──────────────────────────┤
│           ...            │
├──────────────────────────┤
│   compilee definition    │
└──────────────────────────┘
```

The word POSTPONE 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.

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

1. compile the address of EXIT into the dictionary entry being compiled,
2. expose the new word to the colon compiler, and
3. leave compilation mode.

Here's the definition of semicolon:

```
: ;   POSTPONE EXIT   REVEAL POSTPONE [ ; IMMEDIATE
```

The first phrase compiles EXIT, providing the run-time behavior. The second phrase, which is the compile-time behavior, first exposes the word being compiled and then gets out of the compiler.

What is the reason for REVEAL? When words are in the process of being compiled, they are not yet findable by the colon compiler. This is done to make it possible to redefine existing words in terms of the old definition plus additional code, for example:

```
: CR   CR SPACE ;
```

If during the compilation of the new CR its name were findable, the name of the original CR would be blocked, and we would have had to do, e.g.:

```
: _cr_   CR ;
: CR   _cr_ SPACE ;
```

The word POSTPONE can also 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   POSTPONE
     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 to note the "intelligence" built into POSTPONE. POSTPONE parses the next word in the input stream, decides if it is immediate or not, and proceeds accordingly. If the word was not immediate,

POSTPONE compiles the address of the word into a compilee definition; think of it as deferred compilation. If the word is immediate, POSTPONE compiles the address of this 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 the two 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 being compiled.

POSTPONE *xxx* (--)
- Used in the definition of a compiling word. When the compiling word, in turn, is used in a source definition, the execution token of *xxx* will be compiled into the dictionary entry so that when the new definition is executed, *xxx* will be executed.
- 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.

More Compiler-controlling Words

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 we first need to introduce the word LITERAL.

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 + ;
```

9	F	O	U	R	-	M	O	R	E
link										
code pointer										
(LITERAL)										
4										
+										
EXIT										

The use of a literal in a colon definition requires two cells. The first contains the execution token 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:

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

Here the word LITERAL will compile as a literal the "4" that we put on the stack between the square brackets. 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:

```
VARIABLE LIMITS 4 CELLS ALLOT
: LIMIT ( index -- addr ) CELLS LIMITS + ;
```

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

```
HERE  5 CELLS ALLOT  BASE !
: LIMIT ( index -- addr ) CELLS [ BASE @ ] LITERAL + ;
DECIMAL
```

In the first line we put the address of the beginning of the array (HERE) in the system variable BASE (any other scratch variable will work). In the second line, we compile this address as a literal into the definition of LIMIT.

New version	Old version
5	
CELLS	head for LIMITS
	5
head for LIMIT	CELLS
	head for LIMIT
CELLS	**CELLS**
(LITERAL)	LIMITS
addr	
+	+
EXIT	**EXIT**

Now we know all there is to know about LITERAL, we can also give a better example of [and]. Imagine a colon definition in which we need to type the byte from row 2, column 3, of the array BOARD we defined in the previous section. To get the address of this byte, we could use the phrase

```
BOARD  2 8 ( #cols) * 3 +  CELL+  +
```

but it's time consuming to execute

```
2 8 * 3 +
```

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

```
BOARD  19 CELL+ +
```

but it's unclear to human readers exactly what 19 means, and it is irritating that, for portability, we still have to write CELL+ although 1 CELLS is just a constant.

The best solution is to write

```
BOARD [ 2 8 ( #cols) * 3 +  CELL+ ] 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 let's you peek into the innards of the word itself:

```
: DUMP-THIS  [ HERE ] LITERAL  32 DUMP ." DUMP-THIS" ;
```

...
(LITERAL)
32
...

When you execute DUMP-THIS, you will dump the memory into which DUMP-THIS was defined. You should see how your Forth compiles the literal value of "here," the literal "32," the execution token of DUMP, and then how it inlines the string "DUMP-THIS." (At compile-time, HERE points to the address of the next free code byte. LITERAL compiles this number into the definition as a literal, so that it will serve as the argument for DUMP at run-time.)

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

```
: LITERAL  POSTPONE (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 (--) 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 on the stack.
[	(--)	Leaves compilation mode.
]	(--)	Enters compilation mode.

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 a definition of INTERPRET that will work in most Forth systems:

```
: INTERPRET ( -- )
  BEGIN
        BL FIND IF  EXECUTE ?STACK ABORT" Stack empty"
                ELSE  NUMBER
                THEN
AGAIN ;
```

We've 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 to 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.

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:

```
: ]  ( -- )
  BEGIN
   BL FIND DUP IF -1 = IF  EXECUTE ?STACK ABORT" Stack empty"
        ELSE  ,
      THEN
      ELSE DROP (NUMBER) POSTPONE LITERAL
      THEN
  AGAIN ;
```

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 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 to a number and, if it's a number, compile it as a literal.

If it *is* defined, FIND has tested the word's precedence bit. If the word is immediate, then execute it and check to see whether the stack is empty. If it is *not* immediate, FIND returned an execution token that can be compiled. Then repeat the infinite loop.

Compare this to INTERPRET and you'll see that] could be called an interpreter with the ability to decide whether to execute or compile any given word. It is the simplicity of this design that let's 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:
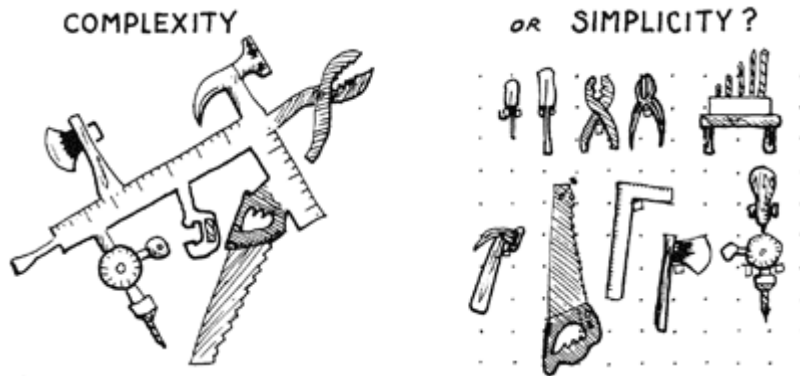
Here is a summary of the words we've covered in this chapter:

DOES>	run time: (-- addr)	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 body address 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 being compiled.
POSTPONE *xxx*	(--)	1. Used in the definition of a compiling word. When the compiling word, in turn, is used in a source definition, the execution token of *xxx* will be compiled into the dictionary entry so that when the new definition is executed, *xxx* will be executed. 2. 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: (--) 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 on the stack.
[	(--)	Leaves compilation mode.
]	(--)	Enters compilation 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.

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 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 include a file when they are executed. Example:

2. ```
   S" mail.forth" LOADED-BY CORRESPONDENCE
   ```

   would define the word CORRESPONDENCE. When CORRESPONDENCE is executed, the file mail.forth is included (Hint: SLITERAL is NOT useful here). [answer]

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

4. ```
   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. .↵ 11 17 ok
   ```

   [answer]

5. 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 execution token of the singular word by using tick. For instance, the phrase

6. ```
   ' CR PLURAL CRS
   ```

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

   ```
   : CRS ( times -- )  0 ?DO  CR  LOOP ;
   ```

   [answer]

7. 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. [answer]

8. 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,

9.   `7 LOOPS CHAR * EMIT SPACE↵ * * * * * * * ok`

[answer]

12. Three Examples

Programming in Forth is more of an "art" than programming in any other language. Like painters drawing brushstrokes, Forth programmers have complete control over where they are going and how they will get there. Charles Moore has written, "A good programmer can do a fantastic job with Forth; a bad programmer can do a disastrous job." A good Forth programmer must be conscious of "style."

Forth style is not easily taught; it's a subject that deserves a book of its own. Some elements of good Forth style include:

- simplicity,
- the use of many short definitions rather than a few longer ones,
- a correspondence between words and easy-to-understand actions or data structures,
- well-chosen names, and
- well laid-out files, clearly commented.

One good way to learn style, aside from trial and error, is to study existing Forth applications, including Forth itself. In this book we've included the definitions of many Forth system words, and we encourage you to continue this study on your own.

This chapter introduces three applications which should serve as examples of good Forth style.

The first example will show you the typical process of programming in Forth: starting out with a problem and working step-by-step towards the solution.

The second example involves a more complex application already written: you will see the use of well-factored definitions and the creation of an application-specific "language."

The third example demonstrates the way to translate a mathematical equation into a Forth definition; you will see that working with fixed-point arithmetic does not necessarily mean sacrificing speed and compactness.

1. WORD game

The example in this section is a refinement of the buzzphrase generator we programmed back in Chap. 10. (You might want to review that version before reading this section.) The previous version did not keep track of its own carriage returns, causing us to force CRs into the definition and creating a very ragged right margin. The job of deciding how many whole words can fit on a line is a reasonable application for a computer and not a trivial one.

The problem is this: to draft a "brief" which consists of four paragraphs, each paragraph consisting of an appropriate introduction and sentence. Each sentence will consist of four randomly-chosen phrases linked together by fillers to create grammatically logical sentences and a period at the end.

The words and phrases have already been edited into the file phrases.forth. Look at this file now, without looking at wordgame.forth. (we're pretending we haven't written the application yet).

File phrases.forth defines the four introductions, compiled into the INTROS string array. The four (or more, INTROS is self-organizing) introductions must be used in sequence. The same file phrases.forth contains four sets of fillers, in FILLER. The four sets are used in sequence, but any of the three versions within a set (organized in columns) is chosen at random. Again, phrases.forth contains the three columns of buzzwords from our previous version, with some added words. We've organized the buzz words in separate 1ST-ADJECTIVE, 2ND-ADJECTIVE and NOUN string arrays.

You might also look at at the sample output that precedes the end of this section, to get a better idea of the desired result.

"Top-down design" is a widely accepted approach to programming that can help to reduce development time. The idea is that you first study your application as a whole, then break the problem into smaller processes, then break these processes into still smaller units. Only when you know what all the units should do, and how they will connect together, do you begin to write code.

The Forth language encourages top-down design. But in Forth you can actually begin to write top-level definitions immediately. Already we can imagine that the "ultimate word" in our application might be called PAPER, and that it will probably be defined something like this:

```
: PAPER  4 0 DO  I INTRO  SENTENCE LOOP ;
```

where INTRO uses the loop index as its argument to select the appropriate introduction. SENTENCE could be defined

```
: SENTENCE  4 0 DO  I PHRASE  LOOP  ENDS ;
```

where PHRASE uses the loop index as its argument to select the appropriate set, then chooses one of the three versions within the set. ENDS takes care of the final '.' and CR at the end of a sentence.

Using our favorite editor, we can enter these top-level definitions into wordgame.forth. Of course we can't INCLUDE this file until we have written our lower-level definitions.

In complicated applications, Forth programmers often test the logic of their top-level definitions by using "stubs" for the lower-level words. A stub is a temporary definition. It might simply print a message to let us know its been executed. Or it may do nothing at all, except resolve the reference to its name in the high-level definition.

While the top-down approach helps to organize the programming process, it isn't always feasible to code in purely top-down fashion. Usually we have to find out how certain low-level mechanisms will work before we can design the higher-level definitions.

The best compromise is to keep a perspective on the problem as a whole while looking out for low-level problems whose solutions may affect the whole application.

In our example application, we can see that it will no longer be possible to force CRs at predictable points. Instead we've got to invent a mechanism whereby the computer will perform carriage returns automatically.

The only way to solve this problem is to count every character that is typed. Before each word is typed, the application must decide whether there is room to type it on the current line or do a carriage return first.

So let's define the variable LINECOUNT to keep the count and the constant RMARGIN with the value 78, to represent the maximum count per line. Each time we type a word we will add its count to LINECOUNT. Before typing each word we will execute this phrase:

```
( length-of-next-word -- ) LINECOUNT @ +  RMARGIN < 0= IF  CR
```

that is, if the length of the next word added to the current length of the line exceeds our right margin, then we'll do a carriage return.

But we have another problem: how do we isolate words with a known count for each word? For now, let's assume we have available a word Split-At-Char. This word breaks strings apart, given a specific delimiter.

Let's write out a "first draft" of this low-level part of our application. It will type a single word, making appropriate calculations for carriage return.

BL Split-At-Char	Break string in two at first BL. Leaves the count on the stack, with the address of the first character underneath.
DUP 1+	Leaves the incremented count and a copy of the original count on the stack.
LINECOUNT @ +	Compute how long the current line would be if a space plus the new word were to be included on it.
RMARGIN >	Decides if it would exceed the margin.
IF CR 0 LINECOUNT !	If so, resets the carriage and the count.
ELSE SPACE THEN	Otherwise, leaves a space between the words.
DUP 1+ LINECOUNT +!	Increases the count by the length of the word to be typed, plus one for the space.
TYPE	Types the word using the count and the address left by Split-At-Char.

Now the problem is getting Split-At-Char to look at the strings in phrases.forth. This is handled by INCLUDED, so if we say

```
S" phrases.forth" INCLUDED
```

then CREATE will make sure all necessary strings are compiled in memory.

To help CREATE do this, we'll define the word $". As you can see from its definition, $" compiles the string (delimited by a second quotation mark) into the dictionary, with the count in the first byte, and leaves its address on the stack for }$, }s$ and }r$. To compile the count and string into the dictionary, we simply have to execute WORD, since WORD's buffer is HERE. We get the string's address as a fillip, since WORD also leaves HERE.

All that remains is to ALLOT the appropriate number of bytes. This number is obtained by fetching the count from the first byte of the string and adding one for the count's byte.

We have written $" to compile the next string into the dictionary, but also to pile the address of this string on the stack, on top of the addresses of other strings that were compiled already just before that. In order to let other words know how many string addresses are on the stack, $" also increments the top of stack:

```
( 'string1 'string2 ... stringN N  new_string_address -- )  SWAP 1+ ;
```

In order to make this work for the first string $" must compile, we have the constant ${ put a 0 on the stack.

We now have ${ and $" compiling our strings for us, but at some point these addresses must be stored in the dictionary. There we can choose one of them to print, when INTRO or PHRASE need to do so. Because there is clearly work to be done both at compile and run-time, this is an ideal job for a defining word. The compile-time work is done in CREATE parts which typically look as follows:

```
( u -- ) DUP , ( first compile count ) 0 ?DO , LOOP ( compile u string addresses )
```

while the run-time part is handled in DOES> parts, doing something like

```
DOES>   ( ix body -- c-addr u ) SWAP CELLS +  CELL+ @ COUNT ;
```

This DOES> part is actually usable for }$, which has the rather simple job to deliver INTRO's string, selected by an index on the stack. Other words that need a string address want more randomness, which is easily provided by using CHOOSE (see the listing for }s$ and }r$).

Now we have a mechanism to present strings to Split-At-Char, the next question is: how do we know when we've gotten to the end of such a string?

Since we are typing word by word what Split-At-Char outputs, we only have to check whether the character count of these strings is larger than zero. Once Split-At-Char gets to the end of its input string, it starts returning empty strings.

For example, the phrase

```
    S" Hello, I speak Forth" .PHRASE
```

should type out the contents of the string, word by word, performing carriage returns where necessary.

How should we structure our definition of .PHRASE? Let's re-examine what it must do:

1. Determine whether there is still a word in the string to be typed.
2. If there is, type the word (with margin checking), then repeat. If there isn't, exit.

The two part nature of this structure suggests that we need a BEGIN...WHILE...REPEAT loop. Let's write our problem this way, if only to understand it better.

```
... BEGIN ANOTHER? WHILE .WORD REPEAT ...
```

ANOTHER? will do step 1; .WORD will do step 2.

How should ANOTHER? determine whether there is still a word to be typed from the string? It simply tests the top of stack to see if the string count is not yet zero, by using the phrase DUP:

```
: ANOTHER? DUP ; ( #chars -- TRUE=string-not-empty )
```

The (not properly formed) flag will serve as the argument for WHILE.

How do we compute the strings for .PHRASE to work on? This is accomplished by executing one of the various children of our compiling word }$, }r$ or }s$. Thus our definition of .PHRASE might be

```
: .PHRASE ( c-addr u -- ) BEGIN  ANOTHER?  WHILE  .WORD  REPEAT  2DROP ;
```

We need the 2DROP because, when we exit the loop, we will have the final address of Split-At-Char and a zero count on the stack, neither of which we need any longer.

How do we define .WORD? Actually, we've defined it already, a few pages back. However, it pays to split .WORD up into a few other useful words, so that it looks like this:

```
: -FITS? linecount @ +  RMARGIN > ;
: SPACE' linecount @ IF  SPACE  1 linecount +!  THEN ;
: CR'    CR  0 linecount ! ;

: .WORD ( addr1 #chars1 -- addr2 #chars2 )
    BL Split-At-Char
    DUP 1+ ( space!) -FITS? IF  CR'  THEN
    SPACE' TYPE' ;
```

Now we have our word-typing mechanism. But let's see if we're overlooking anything. For example, consider that every time we start a new paragraph, we must remember to reset LINECOUNT to zero.

Otherwise our .WORD will think that the current line is full when it isn't. We should ask ourselves this question: is there *ever* a case in this application where we would want to perform a CR without resetting LINECOUNT? The answer is no, by the very nature of the application. For this reason we defined

```
: CR'  CR  0 LINECOUNT ! ;
```

to create a version of CR that is appropriate for this application. We have used this CR in our definition of .WORD.

We should also consider our handling of spaces between words. By using the phrase

```
IF  CR  ELSE  SPACE  THEN
```

before typing each word, we guarantee that there will be a space between each pair of words on the same line but no space at the beginning of successive lines. And since we are typing a space *before* each word rather than after, we can place a period immediately after a word, as we must at the end of a sentence.

But there is a problem with this logic: at the beginning of a new paragraph, we will always get one space before the first word. Our solution: to redefine SPACE so that it will be sensitive to whether or not we're at the beginning of a line, and will *not* space if we *are*:

```
: SPACE  LINECOUNT @  IF  SPACE  THEN ;
```

If LINECOUNT is "0" then we know we are at the beginning of a line, because of the way we have redefined CR.

While we are redefining SPACE, it would be logical to include the phrase

```
1 LINECOUNT +!
```

in the redefinition. Again our reasoning is that we should never perform a space without incrementing the count.

Let's assume that we have edited our definitions into wordgame.forth. Notice that we had very little typing to do, compared with the amount of thinking we've done. Forth source code tends to be concise.

Now we can define our in-between-level words — words like INTRO and PHRASE that we have already used in our highest-level words, but which we didn't define because we didn't have the low-level mechanism.

Let's start with INTRO. The finished definition of INTRO looks like this:

```
: INTRO  ( u -- ) CR' intros .PHRASE ;
```

Our mechanism has given us a very easy way to select strings. We can test this definition by itself, as follows:

```
0 INTRO  ( or 1, 2 or 3 INTRO )↵
In this paper we will demonstrate that ok
```

Notice that we put the argument to INTRO on the stack first.

The way to get a FILLER phrase is a little more complicated. All of it is handled by the DOES> part of }s$. Since we are dealing with sets, not lines, and since the sets all have three strings, we must multiply the loop index for filler by 3. To pick one of the 3 versions *within* the set, we must choose a random number under three, add it to the index so far, convert it to cells, then add this result to the beginning of the set, taking into account the count of strings in front. We can define

```
...
DOES> ( ix -- ) DUP @ 1- ROT -  3 *  3 CHOOSE +   CELLS + CELL+ @ COUNT ;
```

The DUP @ 1- ROT - is there because we compiled the strings in reverse order of their specification in phrases.forth, and therefore need to find the complement of the actual compiled number of strings.

Again we can test this definition by writing

```
3 FILLER↵
to function as ok
```

The remaining words in the application are similar to their previous counterparts, stated in terms of the new mechanism.

Here is a sample of the output. (We've added REDO as an afterthought so that we'd be able to print the same part more than once.)

In this paper we will demonstrate that by using synchronized third generation capability balanced by qualified digital projections it becomes not unfeasible for all but the least stand-alone organizational hardware to function as transient undocumented mobility.

On the one hand, studies have shown that by applying available resources towards synchronized fail-safe mobility coordinated with random context sensitive mobility it is possible for even the most responsive management mobility to avoid partial unilateral engineering.

On the other hand, however, practical experience indicates that with structured deployment of stand-alone fail-safe concepts coordinated with optimal omnirange time phasing it is possible for even the most qualified monitored utilities to avoid optional undocumented utilities.

In summary, then, we propose that by using total incremental programming coordinated with representative policy engineering it is possible for even the most responsive transitional engineering to generate a high level of compatible incremental engineering.

2. File Away!

Our second example consists of a simple filing system. It is a moderately useful application, and a good one to learn Forth from. We have divided this section into four parts:

1. A "How To" for the end user. This will give you an idea of what the application can do.
2. Notes on the way the application is structured and the way certain definitions work.
3. A glossary of all the definitions in the application.
4. A listing of the application, including the data files themselves.

How to Use the Simple File System

This computer filing system lets you store and retrieve information quickly and easily. At the moment, it is set up to handle people's names, occupations, and phone numbers. Not only does it allow you to enter, change, and remove records, it also allows you to search the file for any piece of information. For example, if your have a phone number, you can find the person's name; or, given a name, you can find the person's job, etc.

For each person there is a "record" which contains four "fields." The names which specify each of these four fields are

```
SURNAME    GIVEN    JOB    PHONE
```

("Given," of course, refers to a person's given name, or first name.)

File Retrieval

You can search the file for the contents of any field by using the word FIND, followed by the field-name and the contents, as in

```
FIND JOB newscaster↵Dan Rather ok
```

If any "job" field contains the string "newscaster," then the system prints the person's full name. If no such field exists, it prints "NOT IN FILE."

Once you have found a field, the record in which it was found becomes "current." You can get the contents of any field in the current record using the word GET. For instance, having entered the line above, you can now enter

```
GET phone↵555-9876 ok
```

The FIND command will only find the *first* instance of the field that you are looking for. To find out if there is another instance of the field that you last found, use the command ANOTHER. For example, to find another person whose "job" is "newscaster," enter

```
ANOTHER↵Jessica Savitch ok
```

and

```
ANOTHER↵Frank Reynolds ok
```

When there are no more people whose job is "newscaster" in the file, the ANOTHER command will print "NO OTHER."

To list all the names whose field contains the string that was last found, use the command ALL:

```
ALL↵
Dan Rather
Jessica Savitch
Frank Reynolds
ok
```

Since the surname and given name are stored separately, you can use FIND to search the file on the basis of either one. But if you know the person's *full* name, you can often save time by locating both fields at once, by using the word FULLNAME. FULLNAME expects the full name to be entered with the last name first and the two names separated by a comma, as in

```
FULLNAME Wonder,Stevie↵Stevie Wonder ok
```

(There must not be a space after the comma, because the comma marks the end of the first field and the beginning of the second field.) Like FIND and ANOTHER, FULLNAME repeats the name to indicate that it has been found.

You can actually find *any pair* of fields by using the word PAIR. You must specify both the field names and their contents, separated by a comma. For example, to find a newscaster whose given name is Dan, enter

```
PAIR JOB newscaster,GIVEN Dan↵Dan Rather ok
```

File Maintenance

To enter a new record, use the command ENTER, followed by the surname, given name, job, and phone, each separated by a comma only. For example,

```
ENTER Nureyev,Rudolf,Ballet dancer,555-1234↵ ok
```

To change the contents of a single field within the *current* record, use the command CHANGE followed by the name of the field, then the new string. For example,

```
CHANGE JOB choreographer↵ ok
```

To completely remove the current record, use the command REMOVE:

```
REMOVE↵ ok
```

Comments

This section is meant as a guide, for the novice Forth programmer, to the glossary and listing which follow. We'll describe the structure of this application and cover some of the more complicated definitions. As you read this section, study the glossary and listing on your own, and try to understand as much as you can.

Turn to the listing . Near the end, this file contains the definitions for all nine end-user commands we've just discussed. Notice how simple these definitions are, compared to their power!

This is a characteristic of a well-designed Forth application. Notice that the word -FIND, the elemental file-search word, is factored in such a way that it can be used in the definitions of FIND, ANOTHER, and ALL, as well as in the internal word, (PAIR), which is used by PAIR and by FULLNAME.

We'll examine these definitions shortly, but first let's look at the overall structure of this application.

One of the basic characteristics of this application is that each of the four fields has a name which we can enter in order to specify the particular field. For example, the phrase

```
SURNAME PUT
```

will put the character string that follows in the input stream into the "surname" field of the current record. The phrase

```
SURNAME .FIELD
```

will print the contents of the "surname" field of the current record, etc.

There are two pieces of information that are needed to identify each field: the field's starting address relative to the beginning of a record and the length of the field.

In this application, a record is laid out like this:
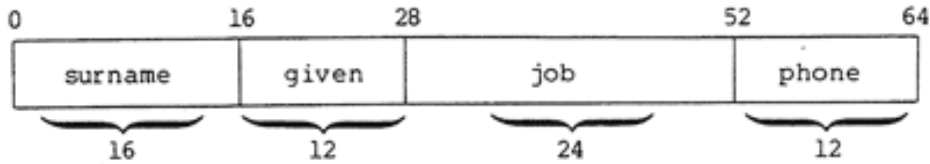
For instance, the "job" field starts thirty-two bytes in from the beginning of every record and continues for twenty-four bytes.

We chose to make a record exactly sixty-four bytes long, but this system can be modified to hold records of any length and any number of fields.

To add more fields, just add lines with the length of the new field, followed by RECORD new-field-name. For example, to add a field FOO which is thirty bytes long, do

```
30 RECORD foo
```

etc. The system automatically computes the values of R-LENGTH and #MAXRECS.

We've taken the two pieces of information for each field and put them into a double-length table associated with each field name. This task is performed by the defining word RECORD, at compile-time. Our definition of JOB, therefore eventually executes CREATE, as in


```
CREATE JOB  32 , 24 ,
```

The literal 32 is computed by the system, which keeps track of the actual offset into a record through updating R-LENGTH.

Thus when we enter the name of a field, we are putting on the stack the address of the table that describes the "job" field. We can fetch either or both pieces of information relative to this address.

Let's call each of these entries a "field specifying table," or a "spec table" for short.

Part of the design for this application is derived from the requirements of FIND, ANOTHER, and ALL; that is, FIND not only has to find a given string within a given type of field, but also needs to "remember" the string and the type of field so that ANOTHER and ALL can search for the same thing.

We can specify the kind of field with just one value, the address of the spec table for that type of field. This means that we can "remember" the type of field by storing this address into KEEP.

KIND was created for this purpose, to indicate the "kind" of field.

To remember the *string*, we have defined a buffer called WHAT to which the string can be moved.

The word KEEP serves the dual purpose of storing the given field type into KIND and the given character string into WHAT. If you look at the definition of the end-user word FIND, you will see that the first thing it does is KEEP the information on what is being searched for. Then FIND executes the internal word -FIND, which uses the information in KIND and WHAT to find a matching string.

ANOTHER and ALL also use -FIND, but they don't use KEEP. Instead they look for fields that match the one most recently "kept" by FIND.

So that we can GET any piece of information from the record we have just "found," we need a pointer to the "current" record. This need is met by the value #RECORD. The operations of the words SET, TOP and DOWN should be fairly obvious to you.

The word RECORD@ uses its stack parameter to compute the absolute address (the computer-memory address, somewhere in the disk buffer) of the beginning of the current record. RECORD@ also makes sure that the record really *is* in the disk buffer.

While a spec table contains the *relative* address of the field and its length, we usually need to know the field's absolute address and length for words such as TYPE, MOVE, and PARSE. Look at the definition of the word FIELD to see how it converts the address of a spec table into an absolute address and length. Then examine how FIELD is applied in the definition of .FIELD.

The word PUT also employs FIELD. Its phrase

```
>R KBD, R> >FLD_
```

leaves on the stack the arguments

```
addr-of-string count-of-string  absolute-addr-of-field size-of-field
```

for MOVE to move the string into the appropriate field of the current record. Before we move the string, we fill the field with spaces, to blank possible old contents. Also, we make sure the length of the moved string is not larger than the size of the field.

There are two things worth noting about the definition of FREE. The first is the method used to determine whether the record is empty. We've made the assumption that if the first byte of a record is empty, then the whole record is empty, because of the way ENTER works. If the first byte contains a character whose ASCII value is less than or equal to BL, then it is not a printing character and the line is empty. As soon as an empty record is found, LEAVE ends the loop. #RECORD will contain the number of the free record.

Another thing worth noting about FREE is that it aborts if the file is full, that is, if it runs through all the records without finding one empty. We can use a DO loop to run through all the records, but how can we tell that the loop has run out before it has found an empty record?

The best way is to leave a TRUE on the stack, to serve as a flag, before beginning the loop. If an empty record is found, we can change the flag to FALSE (with the word INVERT) before we leave the loop. When we come out of the loop, we'll have a TRUE if we never found an empty record, a FALSE if we did. This flag will be the argument for ABORT".

We use a similar technique in the definition of -FIND. -FIND must return a flag to the word that executed it: FIND, ANOTHER, ALL or (PAIR). The flag indicates whether a match was found before the end of the file was reached. Each of these outer words needs to make a different decision based on the state of this flag. This flag is TRUE if a match is not found (hence the name -FIND). The decision to use negative logic was based on the way -FIND is used.

Because the flag needs to be TRUE if a match is not found, the easiest way to design this word is to start with a TRUE on the stack and change it to a FALSE only if a match is found.

Now that you understand the basic design of this application, you should have no trouble understanding the rest of the listing, using the glossary as a guide.

Filer Glossary

/CR	A constant that defines the length in bytes of a newline sequence.
#MAXRECS	A constant that defines the maximum number of records in the data file. To increase this number, add lines containing R-LENGTH spaces, followed by a newline, to the data file.
FILE	A value that holds the handle of the file containing the data.
KIND	A value that contains the address of the field-specifying table for the type of field that was last searched for by FIND.
R-LENGTH	A value that contains the length in bytes of a single record.
#RECORD	A value that points to the current record.
RECORD	A defining word to create field-specifying tables. Takes the field width in bytes as a parameter and updates R-LENGTH. All uses of RECORD should happen before #MAXRECS is defined. Usage: 10 RECORD foo
SURNAME	Returns the address of the field-specifying table for the "surname" (last name) field.
GIVEN	Returns the address of the field-specifying table for the "given" (first name) field.
JOB	Returns the address of the field-specifying table for the "job" field.
PHONE	Returns the address of the field-specifying table for the "phone" field.
WHAT	Returns the address of a buffer that contains the string that is being searched for, or was last searched for, by FIND.
RBUF	Returns the address of a buffer that contains the current record data.
FLUSH	Makes sure all changed data is committed to disk, but does not close the file.
UPDATE	Writes the data for the current record to disk.
RECORD@	Ensures that the specified record is in RBUF.
>FLD_	Given the address of a field-specifying table, returns the address of the associated field in RBUF, along with its assigned length.
>FLD	Given the address of a field-specifying table, returns the address of the associated field in RBUF, along with its actual length.
FIELD	Ensures that the associated field in the current record is in a disk buffer and returns the address of the field in the buffer along with its actual length.
.FIELD	From the current record, types the contents of the field that is associated with the field-specifying table at addr.
SET	Sets the record pointer to the specified record.
TOP	Resets the record pointer to the top of the file.
DOWN	Moves the record pointer down one record.
.NAME	Prints the full name found in the current record.
READ	Moves a character string, delimited by a comma or by a carriage return, from the input stream to a temporary buffer, then returns its address and count.
PUT	Moves a character string, delimited by a comma or by a carriage return, from the input stream into the field whose field-specifying table address is given on the stack.
KEEP	Moves a character string, delimited by a comma or by a carriage return, from the input stream into WHAT, and saves the address of the given field in KIND, for future use by -FIND.
FREE	Starting at the top of the file, finds the first record that is free, that is, whose count is zero. Aborts if the file is full.
-FIND	Beginning at #record and proceeding down, compares the contents of the field indicated by KIND against the contents of WHAT.
(PAIR)	Starting from the top, attempts to find a match on the contents of WHAT, using KIND to indicate the type of field. If a match is made, then attempts to match a second field, whose type is indicated by "field", with the contents {c-addr u}. If both match, prints the name; otherwise repeats until a match is made or until the end of the file is reached, in which case prints an error message.

|	
ENTER	Finds the first free record, then moves four strings delimited by commas into the surname, given, job and phone fields of that record. Usage: ENTER lastname,firstname,job,phone
REMOVE	Erases the current record.
CHANGE	Changes the contents of the given field in the current record. Usage: CHANGE field-name new-contents
GET	Prints the contents of the given type of field from the current record. Usage: GET field-name
FIND	Finds the record in which there is a match between the contents of the given field and the given string. Usage: FIND field-name string
ANOTHER	Beginning with the next record after the current one, and using KIND to determine type of field, attempts to find a match on WHAT. If successful, types the name; otherwise an error message.
ALL	Beginning at the top of the file, uses KIND to determine type of field and finds all matches on WHAT. Types the full name(s).
PAIR	Finds the record in which there is a match between both the contents of the first given field and the first given string, and also the contents of the second given field and the second given string. Comma is delimiter. Usage: PAIR field1 string1,field2 string2
FULLNAME	Finds the record in which there is a match on both the first and last names given. Usage: FULLNAME lastname,firstname

Filer Listing

The listing is here.

3. No Weighting

Our final example is a math problem which many people would assume could only be solved by using floating point. It will illustrate how to handle a fairly complicated equation with fixed-point arithmetic and demonstrate that for all the advantages of using fixed-point, range and precision need not suffer. Of course, when the hardware does have floating point one should preferably use that instead, and we show how to do that, too. Using fixed-point has the slight disadvantage that, in order to correctly compute scale factors, we have to know our Forth's number of bits per cell. For modern Forths the number of bits per cell can be 16, 32, 64, or even higher. In order not to complicate the following description too much, we will assume 16-bit hardware. That is probably the only environment this example will be useful for, anyway. Also, we'll assume 1 CHARS is equivalent to one byte.

In this example we will compute the weight of a cone-shaped pile of material, knowing the height of the pile, the angle of the slope of the pile, and the density of the material.

To make the example more "concrete," let's weigh several huge piles of sand, gravel, and cement. The slope of each pile, called the "angle of repose," depends on the type of material. For example, sand piles itself more steeply than gravel.
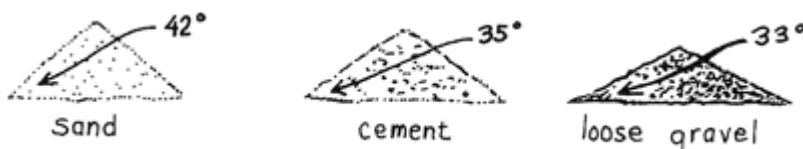
(In reality these values vary widely, depending on many factors; we have chosen approximate angles and densities for purposes of illustration.)

Here is the formula for computing the weight of a conical pile *h* feet tall with an angle of repose of *theta* degrees, where *D* is the density of the material in pounds per cubic foot:

$$W = \frac{\pi h^3 D}{3 \tan^2 \theta}$$

This will be the formula which we must express in Forth.

For Skeptics

The volume of a cone, *V*, is given by

$$V = \frac{1}{3} \pi b^2 h$$

where *b* is the radius of the base and *h* is the height. We can compute the base by knowing the angle or, more specifically, the tangent of the angle. The tangent of an angle is simply the ratio of the segment marked *h* to the segment marked *b* in this drawing:
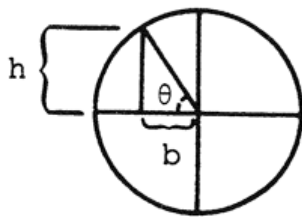
If we call this angle "theta", then

$$\tan \theta = \frac{h}{b}$$

Thus we can compute the radius of the base with

$$b = \frac{h}{\tan \theta}$$

When we substitute this into the expression for *V*, and then multiply the result by the density *D* in pounds per cubic foot, we get the formula shown in the text.

Let's design our application so that we can enter the name of a material first, such as

```
DRY-SAND
```

then enter the height of a pile and get the result for dry sand.

C E M E N T
131
700

Let's assume that for any one type of material the density and angle of repose never vary. We can store both of these values for each type of material into a table. Since we ultimately need each angle's tangent, rather than the number of degrees, we will store the tangent. For instance, the angle of repose for a pile of cement is 35°, for which the tangent is .700. We will store this as the integer 700.

Bear in mind that our goal is not just to get an answer; we are programming a computer or device to get the answer for us in the fastest, most efficient, and most accurate way possible. As we indicated in Chap. 5, to write equations using fixed-point arithmetic requires an extra amount of thought. But on hardware that would have to emulate floating point, the effort pays off in two ways:

1. vastly improved run-time speed, which can be very important when there are millions of steps involved in a single calculation, or when we must perform thousands of calculations every minute. Also,
2. program size, which would be critical if, for instance, we wanted to put this application in a hand-held device specifically designed as a pile-measuring calculator. Forth is often used in this type of instrument.

Let's approach our problem by first considering scale. The height of our piles ranges from 5 to 50 feet. By working out our equation for a pile of cement 50 feet high, we find that the weight will be nearly 3,500,000 pounds.

But because our piles will not be shaped as perfect cones and because our values are averages, we cannot expect better than four or five decimal places of accuracy. If we scale our result to tons, we get about 17,500. This value will comfortably fit within the range of a single-length number, even on 16-bit hardware. For this reason, let's write this application entirely with single-length arithmetic operators. (Although we will assume 16-bit hardware in the following, the code as shown will run unmodified on any ANS Forth.)

Applications which require greater accuracy can be written using double-length arithmetic; to illustrate we've even written a second version of *this* application using double-length math, as you'll see later on. But we intend to show the accuracy that Forth can achieve even with 16-bit math.

By running another test with a pile 40 feet high, we find that a difference of one-tenth of a foot in height can make a difference of 25 tons in weight. So we decide to scale our input to feet and inches rather than merely to whole feet.

We'd like the user to be able to enter

```
15 FOOT  2 INCH  PILE
```

where the words FOOT and INCH will convert the feet and inches into tenths of an inch, and PILE will do the calculation. Here's how we might define FOOT and INCH:

```
: FOOT  10 * ;
: INCH  100 12 */ 5 +  10 / + ;
```

The use of INCH is optional.

(By the way, we could as easily have designed input to be in tenths of an inch with a decimal point, like this:

```
15.2
```

In this case, NUMBER would convert the input as a double-length value. Since we are only doing single-length arithmetic, PILE could simply begin with DROP, to eliminate the high-order cell.)

In writing the definition of PILE, we must try to maintain the maximum number of places of precision without overflowing 15 bits. According to the formula, the first thing we must do is cube the argument. But let's remember that we will have an argument which may be as high as 50 feet, which will be 500 as a scaled integer. Even to *square* 500 produces 250,000, which exceeds the capacity of single-length arithmetic using 16-bit cells.

We might reason that, sooner or later in this calculation, we're going to have to divide by 2000 to yield an answer in tons. Thus the phrase

```
DUP DUP 2000 */
```

will square the argument and convert it to tons at the same time, taking advantage of */'s double-length intermediate result. Using 500 as our test argument, the above phrase will yield 125.

But our pile may be as small as 5 feet, which when squared is only 25. To divide by 2000 would produce a zero in integer arithmetic, which suggests that we are scaling down too much.

To retain the maximum accuracy, we should scale down no more than necessary. 250,000 can be safely accommodated by dividing by 10. Thus we will begin our definition of PILE with the phrase

```
DUP DUP 10 */
```

The integer result at this stage will be scaled to one place to the right of the decimal point (25000 for 2500.0).

Now we must *cube* the argument. Once again, straight multiplication will produce a double-length 32-bits result, so we must use */ to scale down. We find that by using 1000 as our divisor, we can stay just within single-length range. Our result at this stage will be scaled to one place to the *left* of the decimal point (12500 for 125000.) and still be accurate to 5 digits.

According to our formula, we must multiply our argument by pi. We know that we can do this in Forth with the phrase

```
355 113 */
```

which causes no problems with scaling.

Next we must divide our argument by the tangent squared, which we can do by dividing the argument by the tangent *twice*. Because our tangent is scaled to three decimal places, to divide by the tangent we multiply by 1000 and divide by the table value. Thus we will use the phrase

```
1000 TAN(THETA) */
```

Since we must perform this twice, let's make it a definition, called /TAN (for *divide-by-the-tangent*) and use the word /TAN twice in our definition of PILE. Our result at this point will be scaled to one place to the left of the decimal (26711 for 267110, using our maximum test values).

All that remains is to multiply by the density of the material, of which the highest is 131 pounds per cubic foot. To avoid overflowing, let's try scaling down by two decimal places with the phrase

```
DENSITY 100 */
```

But by testing, we find that the result at this point for a 50-foot pile of cement will be 34,991, which just exceeds the 15-bit limit. Now is a good time to take the 2000 into account. Instead of

```
DENSITY 100 */
```

we can say

```
DENSITY 200 */
```

and our answer will now be scaled to whole tons.

You will find this version in the listing. As we mentioned, we have also written this application using double-length arithmetic. In this version you enter the height as a double-length number scaled to tenths of a foot, followed by the word FEET, as in 50.0 feet.

By using double-length integer arithmetic, we are able to compute the weight of the pile to the *nearest* whole pound. The range of double-length 32-bit integer arithmetic compares with that of single-precision floating-point arithmetic. Below is a comparison of the results obtained using a 10-decimal-digit pocket calculator, single-length Forth, double-length (32-bit) Forth, and floating-point Forth. The test assumes a 50-foot pile of cement, using the table values.

	in pounds	in tons
calculator	34,995,633	17,497.816
Forth 16-bit single-length	---	17,495
Forth 16-bit double-length	34,995,634	17,497.817
Forth 32-bit single-length	---	17,495
Forth 32-bit double-length	34,995,634	17,497.817
Forth floating-point	34,995,633	17,497.816

Here's an example of our application's output:

```
S" spiles.forth" INCLUDED↵ ok
cement↵ ok
10 foot pile↵ = 138 tons of cement ok
10 foot 3 inch pile↵ = 151 tons of cement ok
dry-sand↵ ok
10 foot pile↵ = 81 tons of dry sand ok
S" dpiles.forth" INCLUDED cement↵ ok
10.0 feet↵ = 279939 pounds of cement or 139.969 tons ok
S" fpiles.forth" INCLUDED cement↵ ok
10e feet↵ = 279965.06373598 pounds, or 139.98253187 tons of cement ok
```

A note on "

The defining word MATERIAL takes three arguments for each material, one of which is the address of a string. .SUBSTANCE uses this address to type the name of the material.

To put the string in the dictionary and to give an address to MATERIAL, we have defined a word called ". As you can see from its definition, " compiles the string (delimited by a second quotation mark) into the dictionary, with the count in the first byte, and leaves its address on the stack for MATERIAL. To compile the count and string into the dictionary, we simply have to execute WORD, since WORD's buffer is HERE. We get the string's address as a fillip, since WORD also leaves HERE.

All that remains is to ALLOT the appropriate number of bytes. This number is obtained by fetching the count from the first byte of the string and adding one for the count's byte.

A Browser Interface for FPILES

This interface is Forth system dependent. It will work for iForth 2.0, after some preparations:

- Run iForth on the file fsserver.frt
- Execute the word PILE-SERVER.
- Manipulate the below form and press SEND. A new browser window opens with the calculation result.

Height in feet: [   ]

- ◉ cement
- ○ wet sand
- ○ dry sand
- ○ clay
- ○ loose gravel
- ○ packed gravel

Review of Terms

Stub
    in Forth, a temporary definition created solely to allow testing of a higher-level definition.
Top-down Programming
    a programming methodology by which a large application is divided into smaller units, which may be further subdivided as necessary. The design process starts with the overview, or "top," and proceeds down to the lowest level of detail. Coding of the low-level units begins only after the entire structure of the application has been designed.

Notes