

Basic

A computer's a useless hunk of metal & plastic until somebody feeds it a **program**, which is a list of commands teaching the computer how to perform a task. A person who writes a program is called a **programmer**. Now you'll learn how to write programs, so you become a programmer.

You learned to buy & use programs such as Microsoft Office. Now you'll learn how to invent your own programs, so you can become the computer's master and make it do whatever you wish, not limited to the creations of other programmers.

Programming the computer can be easy. You'll write your own programs just a few minutes from now, when you read the next page. Then on your résumé you can brag you're a "programmer." As you read farther, you'll learn how to become a *good* programmer, by writing programs that are more sophisticated.

Learning to program is fun, an adventure that expands your mind and turns you into a brilliant thinker.

You learn the secret of computer life: how to take a computer — that hunk of metal and plastic — and teach it new skills by feeding it programs you invent. Your teaching and programs turn the computer into a thinking organism. You can teach the computer to become as smart as you and even imitate your personality. You become the computer's God, able to make the computer do anything you wish. Ah, the power!

To program the computer, feed it a list of commands written in a **computer language**. Each computer language is a small part of English. The easiest popular computer language is **Basic**.

Basic was invented in New Hampshire by 2 Dartmouth College professors (**John Kemeny & Tom Kurtz**) in 1964 and improved afterwards. Basic consists of words such as "print", "input", and "if". I'll explain how to program the computer by using those Basic words.

Different computers speak different **dialects** of Basic.

For example, a popular dialect was invented in 1975 by a 19-year-old kid, **Bill Gates**. Since he developed software for microcomputers, he called himself **Microsoft** and called his Basic dialect **Microsoft Basic**.

Since Microsoft Basic is so wonderful, all popular computer companies paid him to make their computers understand Microsoft Basic. IBM, Apple, and *hundreds* of other computer companies all had to pay off Bill. Microsoft Basic became so popular that he had to hire hundreds of employees to help him fill all the orders. Microsoft Incorporated became a multi-billion-dollar company, and Bill became a famous billionaire, the wealthiest person in the world.

Over the years, Bill improved Microsoft Basic. Some computers used old versions of Microsoft Basic; other computers used his later improvements.

One of the most popular versions of Microsoft Basic has been **QBasic**. It's been popular because it works well and most people got it at no charge: free!

QBasic expected your operating system to be MS-DOS. You could force QBasic to run with some versions of Windows, and you could even download QBasic free from some Websites (by Microsoft and others), but now Microsoft has taken steps to stop you from doing so, since Microsoft wants you to use Visual Basic instead.

Previous editions of this book explained **QB64** (pronounced "Q B sixty-four"), which imitated QBasic, ran well if you had Windows, and could be downloaded free from QB64.net. It was invented in Sydney, Australia by a guy whose name is "Rob" but whose nickname is "**Galleon Dragon**." Unfortunately, QB.net has stopped allowing easy downloads.

A different dialect of Basic was invented in 1981 in England by the **British Broadcasting Company (BBC)** then greatly improved by **Richard Russell**, who's amazingly brilliant. His newest and best version is called **BBC Basic for Windows**. In many ways, it's better than Microsoft Basic. You can download it from his Website for free (unless you need advanced features that cost extra).

This chapter explains BBC Basic for Windows, because it's the most pleasant programming language for beginners.

After reading this chapter, you can explore **Visual Basic**, which is explained in a later chapter. Visual Basic, invented by Microsoft, is harder than BBC Basic for Windows but more powerful: it lets you teach the computer to create windows & buttons and handle mouse clicks.

The **commands** of BBC Basic for Windows are explained on these pages:

Command	What the computer will do	Page	Similar to
case a\$ of	analyze a\$ to select a case from list below	492	endcase
circle 200,300,50	draw a circle at (200,300) with radius 50	509	ellipse
clg	erase all graphics in the output window	510	cls, plot
cls	erase everything in the output window	490	clg, print
colour 4	print in blue (color #4) instead of black	510	print
data meat,potatoes	use this list of data: meat, potatoes	498	read
def proc_insult	make the lines below define "insult"	524	proc_
dim x\$(7)	make x\$ be a list of 7 strings	520	x =
draw 600,400	draw a line to pixel (600,400)	509	line, plot
ellipse 200,300,50,25	draw ellipse at (200,300), radii 50 & 25	509	circle
else	do indented lines when "if" condition false	492	if, endif
end	skip the rest of the program	491	stop, quit
endcase	make this the bottom of "case" statement	492	case, when
endif	make this the bottom of an "if" statement	492	if, else
endproc	make this the bottom of a "def proc_"	524	def
exit repeat	skip down to the line that's under "until"	495	repeat
for x=1 to 20	repeat the indented lines, 20 times	495	next, repeat
gcol 9	make graphics red (color #9), not black	510	colour, clg
goto 10	skip to line 10 of the program	491	repeat
if age<18 print "kid"	if age is less than 18, print "kid"	492	endif, else
input "what name";n\$	ask "What name?" and get answer n\$	486	x=
line 200,300,500,350	draw a line from (200,300) to (500,350)	509	draw, plot
next	make this the bottom line of a "for" loop	495	for
otherwise print "feel"	if the "when" conditions false, print "feel"	492	case, when
plot 100,200	put a tiny dot at pixel (100,200)	510	line, draw
print 4+2	print the answer to 4+2	478	colour,@%=
proc_insult	do the subprocedure named "insult"	524	def
quit	skip rest of program; close output window	491	end, stop
read a\$	get a string from the data and call it a\$	498	data
rectangle 200,300,8,5	draw rectangle at (200,300), sides 8 & 5	509	line
repeat	do the indented lines below, repeatedly	489	until
restore 1	skip to line 1 of the data	501	read, data
sound 1,-15,4*25,20*3	make a loud sound for 3 seconds	510	print
stop	skip rest of program and just print "STOP"	491	end, quit
until false	make this the bottom line of "repeat" loop	489	repeat
vdu 2	copy the computer's answers onto paper	481	print, cls
wait 600	wait 600 centiseconds	489	print
when "fine" print "g"	if case is "fine", print "g"	492	case
x=47	make x stand for the number 47	484	input
@%=131082+256*2	print 2 digits after each decimal point	510	print, x=
'Zoo program is fishy	ignore this comment	505	print

The **functions** of BBC Basic for Windows are explained on these pages:

Function	Meaning	Value	Page	Similar to
abs(-3.89)	absolute value of -3.89	3.89	512	sgn, int
acs(.5)*180/pi	arccosine of .5, in degrees	60	519	sin, acs
asc("A")	Ascii code number for "A"	65	517	chr\$
asn(.5)*180/pi	arcsine of .5, in degrees	30	519	sin, acs
atn(1)*180/pi	arctangent of 1, in degrees	45	519	tan, asn
chr\$(241)	character whose code# is 241	"ñ"	517	asc
cos(60*pi/180)	cosine of 60 degrees	.5	519	sin, acs
exp(1)	e raised to the first power	2.71828183	511	log, sqr
instr("needed", "ed")	position of "ed" in "needed"	3	518	len, mid\$
int(3.89)	round down to a lower integer	3	512	abs, sgn
left\$("smart")	leftmost character in "smart"	"s"	517	right\$, mid\$
len("smart")	length of "smart"	5	517	left\$
ln(100000)	logarithm base e of 100000	11.5129255	512	log, exp
log(100000)	logarithm base 10 of 100000	5	511	ln, exp, sqr
mid\$("smart",2)	begin at the 2nd character	"mart"	518	right\$
right\$("smart")	rightmost character	"t"	518	left\$, mid\$
rnd(4)	random integer from 1 to 4	1, 2, 3, or 4	513	int
sgn(-3.89)	sign of -3.89	-1	512	abs, int
sin(30*pi/180)	sine of 30 degrees	.5	519	cos, asn
sqr(9)	square root of 9	3	511	exp, log, ln
str\$(81.4)	turn 81.4 into a string	"81.4"	518	val
sum(x())	sum of numbers in array x	varies	523	len
string\$(20, "b")	a string of 5 b's	"bbbbb"	518	str\$
tan(45*pi/180)	tangent of 45 degrees	1	519	sin, atn
val("52.6")	remove the quotation marks	52.6	518	str\$

Those are the best commands & functions for beginners, but BBC Basic for Windows includes many more!

Fun

Let's have fun programming! If you have any difficulty, phone me at 603-666-6644 (day or night) for free help.

Get Basic

Here's how to copy the nicest Basic (**BBC Basic for Windows**, version 6.14a) from the Internet to a Windows 10 or 11 computer, free (using Microsoft Edge):

Go to bbcbasic.co.uk/bbcwin/download.html. Tap "BBCWDEMO.EXE". You see a blue horizontal line get longer. When that line stops getting longer, tap the underlined down-arrow (which is above the blue line) then "Open file" then "Yes" then thrice "Next" then "Install" then "Finish". Close the Microsoft Edge window (by tapping its X). Close any other windows you have open, so you can start fresh.

Start Basic

Double-tap "BBC BASIC for Windows" (which is on the desktop screen). Tap "OK".

You see the **program window**, titled "BBC BASIC for Windows 6.14a (trial)". To make your life easy (and follow the examples in this book), **make sure the computer allows lowercase typing**, by doing this:

Tap "Options". Left of "Lowercase Keywords", do you see a checkmark yet? If yes, just tap "Options" again; if no, tap "Lowercase Keywords".

Type your program

Now you're ready to type your first program!

For example, type this program:

```
print 4+2
```

Here's how:

Type the word "print". Then press the Space bar. Then type 4+2 (and remember to hold down the Shift key to type the symbol +).

At the end of that line, press the Enter key. (You must press the Enter key at the end of each line.) Pressing the Enter key makes the computer indent the line and makes the word "print" become orange.

A **program** is a list of commands that you want the computer to obey. The sample program you typed contains one command, which tells the computer to do some math: it tells the computer to compute 4+2, get the answer (6), and print the answer onto the screen.

Run your program

To make the computer obey the program you wrote, tap the "▶" (which is green). That tells the computer to **run** the program: the computer will run through the program and obey any commands in it. The computer obeys the "print 4+2" command and prints this answer onto the **output window**:

```
6
```

Congratulations! You've written your first program! You've programmed the computer to compute the answer to 4+2! You've become a programmer! Now you can put on your résumé: "programmer!"

When you finish admiring the computer's answer, close the output window, by tapping its X button.

Edit your program

After you've typed your program, try typing another one. For example, create a program that makes the computer print the answer to 79+2. To do that, make this program appear on the screen:

```
print 79+2
```

To make that program appear, just edit the program you typed previously (which said print 4+2). To edit, do this:

Move the cursor (vertical line) to the left edge of the character you want to change (which was the 4), by using the arrow keys (or clicking there with the mouse). Then delete that character (4) by pressing the Delete key. Then type the characters you want instead (79).

If you've edited the program successfully, the screen shows just the new program —

```
print 79+2
```

and you don't see the old program anymore.

When you've finished editing the program, run it by tapping the green triangle. Then the computer will print the answer:

```
81
```

Your program can contain *several* lines. Each line is a separate command. For example, your program can be:

```
print 8+1
print 41+5
```

When you run that program (by tapping the green triangle), the computer will obey *both* commands, so the output window will show *both* answers:

```
9
46
```

While editing, use these tricks...

To delete a character:

move the cursor (vertical line) to that character's left edge by using the arrow keys (or clicking there with the mouse), then press the Delete key.

To delete SEVERAL characters:

move to the first character you want to delete, then hold down the Delete key awhile.

To delete the LINE you typed recently:

tap the Undo icon (which is a curved arrow pointing backwards).

To delete A DIFFERENT LINE:

drag across that line then press the Delete key.

To INSERT A NEW LINE between two lines:

move to the beginning of the lower line, then press the Enter key.

Fix your errors

What happens if you misspell a computer word, such as "print"? For example, what happens if you accidentally say "primpt" instead of "print"?

Here's the result:

When you run the program (by tapping the green triangle), the output window says "Mistake". When you close the output window and see your bad program, the cursor is at the beginning of the bad line. Fix the error.

Abbreviation

Instead of typing the word "print", you can abbreviate: type just the letter "p" and a period. When you press the Enter key at the end of the line, the computer will change the "p." to "print".

So instead of typing —

```
print 4+2
```

you can type just:

```
p. 4+2
```

Immediate mode

To make Basic do math, you've learned to type a program then make the computer run it (by tapping the green triangle), so the answer appears in the output window. But here's a faster way: type directly in the output window! That technique is called **immediate mode**.

Here's how to do it.

Make the output window appear. If it hasn't appeared yet, make it appear by using one of these methods:

Menu method While looking at the program window, tap "Run" then "Immediate Mode".

Keyboard method While looking at the program window, tap the Keyboard icon (which looks like a keyboard and is at the far right).

Triangle method While looking at the program window, tap the green triangle. (That runs any program in the program window and shows you the output window.)

In the output window, you see this symbol:

```
>
```

Next to that symbol, type any command you wish, such as:

```
>print 4+2
```

At the end of that line, press the Enter key. The computer will obey that line *immediately*, so it will show:

```
6
```

Try it!

Math

This program makes the computer add 4+2:

```
print 4+2
```

When you run the program (by tapping the green triangle), the computer will print the answer:

```
6
```

If you want to subtract 3 from 7, type this command instead:

```
print 7-3
```

(When typing the minus sign, do *not* press the Shift key.) The computer will print:

```
4
```

You can use decimal points and negative numbers. For example, if you type this —

```
print -26.3+1
```

the computer will print:

```
-25.3
```

Multiplication To multiply, use an asterisk. So to multiply 2 by 6, type this:

```
print 2*6
```

The computer will print:

```
12
```

Division To divide, use a slash. So to divide 8 by 4, type this:

```
print 8/4
```

The computer will print:

```
2
```

To divide 2 by 3, type this:

```
print 2/3
```

The computer will round the answer and print just:

```
0.66666667
```

If you try to divide by 0, by typing —

```
print 5/0
```

the computer will refuse. Instead, it will print this gripe:

```
Division by zero
```

Avoid commas Do *not* put commas in big numbers. To write four million, do *not* write 4,000,000; instead, write 4000000.

E notation If the computer's answer is huge (more than a million) or tiny (less than .0001), the computer might print an E in the answer. The E means "move the decimal point".

For example, suppose the computer says the answer to a problem is:

```
8.516743297E12
```

The E means, "move the decimal point". The plus sign means, "towards the right". Altogether, **the E12 means "move the decimal point towards the right, 12 places."** So look at 8.516743297 and move the decimal point towards the right, 12 places; you get 8516743297000.

So when the computer says the answer is 8.516743297E12, the computer really means the answer is 8516743297000, approximately. The exact answer might be 8516743297000.2 or 8516743297000.79 or some similar number, but the computer prints just an approximation.

Suppose your computer says the answer to a problem is:

```
9.23E-6
```

After the E, the minus sign means, "towards the *left*". So look at 9.23 and move the decimal point towards the left, 6 places. You get: .00000923

So when the computer says the answer is 9.23E-6, the computer really means the answer is: .00000923

You'll see E notation rarely: the computer uses it just if an answer is at least a billion (1,000,000,000) or tiny (tinier than .0001). But when the computer *does* use E notation, remember to move the decimal point!

If you want to write E notation yourself, you must capitalize the E.

Rounding To print a long number, the computer will round it and print just 9 significant digits.

The highest number The highest number the computer can handle well is about 1E4932, which is 1 followed by 4932 zeros. If an answer is much higher, the computer will say:

```
Number too big
```

The tiniest decimal The tiniest decimal the computer can handle easily is about 4E-4932, which is a decimal point followed by 4932 digits (4931 zeros then a 4). If you try to go much tinier, the computer will say 0.

Order of operations What does “2 plus 3 times 4” mean? The answer depends on whom you ask.

To a clerk, it means “start with 2 plus 3, then multiply by 4”; that makes 5 times 4, which is 20. But to a scientist, “2 plus 3 times 4” means something different: it means “2 plus three fours”, which is 2+4+4+4, which is 14.

Since computers were invented by scientists, computers think like scientists. If you type —

```
print 2+3*4
```

the computer will think you mean “2 plus three fours”, so it will do 2+4+4+4 and print this answer:

```
14
```

The computer will *not* print the clerk’s answer, which is 20. So if you’re a clerk, tough luck!

Scientists and computers follow this rule: **do multiplication and division before addition and subtraction**. So if you type —

```
print 2+3*4
```

the computer begins by hunting for multiplication and division. When it finds the multiplication sign between the 3 and the 4, it multiplies 3 by 4 and gets 12, like this:

```
print 2+3*4
      \ /
       12
```

So the problem becomes 2+12, which is 14, which the computer prints.

For another example, suppose you type:

```
print 10-2*3+72/9*5
```

The computer begins by doing all the multiplications and divisions. So it does 2*3 (which is 6) and does 72/9*5 (which is 8*5, which is 40), like this:

```
print 10-2*3+72/9*5
      \  / \  /
       6   40
```

So the problem becomes 10-6+40, which is 44, which is the answer the computer prints.

You can use parentheses the same way as in algebra. For example, if you type —

```
print 5-(1+1)
```

the computer will compute 5-2 and print:

```
3
```

You can put parentheses inside parentheses. If you type —

```
print 10-(5-(1+1))
```

the computer will compute 10-(5-2), which is 10-3, and will print:

```
7
```

Strings

Let’s make the computer fall in love. Let’s make it say, “I love you”.

Type this program:

```
print "I love you"
```

Here’s how to type the second line:

Begin by typing the word “print”. Then type a blank space (by pressing the Space bar). Then type a quotation mark, but be careful: **to type the quotation mark, you must hold down the Shift key**. Then type these words: *I love you*. Then type another quotation mark.

At the end of that line, press the Enter key. Pressing the Enter key makes the computer analyze that line: the word “print” turns orange, “I love you” turns purple, and the whole line gets indented.

When you run that program (by tapping the green triangle), the computer will obey your command; it will print:

```
I love you
```

You can change the computer’s personality. For example, if you give this command —

```
print "I hate you"
```

the computer will reply:

```
I hate you
```

Notice that **to make the computer print a message, you must put the message between quotation marks**.

The quotation marks make the computer copy the message without worrying about what the message means. For example, if you misspell “I love you”, and type —

```
print "aieee luf ya"
```

the computer will still copy the message (without worrying about what it means); the computer will print:

```
aieee luf ya
```

Jargon The word “joy” consists of 3 characters: j and o and y. Programmers say that the word “joy” is a **string** of 3 characters.

A **string** is any collection of characters, such as “joy” or “I love you” or “aieee luf ya” or “76 trombones” or “GO AWAY!!!” or “xypw exr///746”. The computer will print whatever string you wish, but remember to **put the string in quotation marks**.

Strings versus numbers The computer can handle two types of expressions: **strings** and **numbers**. Put strings (such as “joy” and “I love you”) in quotation marks. Numbers (such as 4+2) do *not* go in quotation marks.

Accidents Suppose you accidentally put the number 2+2 in quotation marks, like this:

```
print "2+2"
```

The quotation marks make the computer think “2+2” is a string instead of a number. Since the computer thinks “2+2” is a string, it copies the string without analyzing what it means; the computer will print:

```
2+2
```

It will *not* print 4.

Suppose you want the computer to print the word “love” but you accidentally forget to put the string “love” in quotation marks, and type this instead:

```
PRINT love
```

Since you forgot the quotation marks, the computer thinks *love* is a number instead of a string but doesn’t know which number, since the computer doesn’t know the meaning of love. Whenever the computer is confused, it gripes at you. In this particular example, when you run the program the computer will print this gripe:

```
No such variable
```

So if you incorrectly tell the computer to proclaim its love, it will say no.

Longer programs You can program the computer to say it’s madly in love with you!

Let’s make the computer say:

```
I love you.
You turned me on.
Let's get married!
```

To make the computer say all that, just run this program:

```
print "I love you."
print "You turned me on."
print "Let's get married!"
```

To run that program, type it then tap the green triangle. Try it!

To have even more fun, run this program:

```
print "I long"
print 2+2
print "U"
```

It makes the computer print “I long”, then print the answer to 2+2 (which is 4), then print “U”. So altogether, the computer prints:

```
I long
4
U
```

Yes, the computer says it longs for you!

Adding strings You can add strings together: "hot"+"dog" is the same as "hotdog". If you say —

```
print "hot"+"dog"
```

the computer will print:

```
hotdog
```

Tricky printing

Printing can be tricky! Here are the tricks.

Indenting Suppose you want the computer to print this letter onto the screen:

```
Dear Joan,  
  Thank you for the beautiful  
  necktie. Just one problem--  
  I don't wear neckties!  
          Love,  
          Fred-the-Hippie
```

This program prints it:

```
print "Dear Joan,"  
print "  Thank you for the beautiful"  
print "necktie. Just one problem--"  
print "I don't wear neckties!"  
print "          Love,"  
print "          Fred-the-Hippie"
```

In the program, each line contains 2 quotation marks. **To make the computer indent a line, put blank spaces AFTER the first quotation mark.**

Blank lines Life consists sometimes of joy, sometimes of sorrow, and sometimes of a numb emptiness. To express those feelings, run this program:

Program	What the computer will do
print "joy"	Print "joy".
print	Print blank empty line, under "joy".
print "sorrow"	Print "sorrow".

Altogether, the computer will print:

```
joy  
  
sorrow
```

Semicolons Run this program:

```
print "fat";  
print "her"
```

The top line, which makes the computer print "fat", ends with a semicolon. **The semicolon makes the computer print the next item on the same line;** so the computer will print "her" on the same line, like this:

```
father
```

The next program shows what happens to an evil king on a boat:

```
print "sin";"king"
```

The computer will print "sin", and will print "king" on the same line, like this:

```
sinking
```

Notice that in a print statement, you can type several items (such as "sin" and "king"). Between the items, type a semicolon.

Spaces around numbers To print a number, the computer wants to print 10 characters. If the number is shorter than 10 characters, the computer puts blank spaces before the number, to make the total number of characters be 10. Exception: if the "print" statement has a semicolon before the number, the computer does *not* put blank spaces before the number.

For example, "print 2+3" makes the computer print 5 with blank spaces before it —

```
5
```

but "print ;2+3" makes the computer print 5 without blank spaces before it:

```
5
```

Suppose you want the computer to print the answer to 2+2 and also the answer to 5+4, both on the same line. If you write —

```
print 2+2;5+4
```

the computer will print the answer to 2+2 (which is 4) and put blank spaces before it, then print the answer to 5+4 (which is 9) without blank spaces, like this:

```
4 9
```

To separate those two answers from each other, you can tell the computer to print a space between the numbers, like this:

```
print 2+2;" ";5+4
```

That makes the computer print:

```
4 9
```

This command is correct:

```
print "eat ";3+2;" apples"
```

The computer will print "eat " (which includes a blank space in its ending), then 3+2 (which is 5), then " apples" (which begins with a blank space), so the computer will print:

```
eat 5 apples
```

If you accidentally put no spaces in the strings, like this —

```
print "eat";3+2;"apples"
```

the computer will print no spaces in the answer, so it prints this jumble:

```
eat5apples
```

Double quotation marks Let's make the computer print this:

```
You really "wow" me
```

To make the computer print quotation marks around "wow", **put DOUBLE quotation marks** around that word:

```
print "You really ""wow"" me"
```

If you accidentally put just single quotation marks, like this —

```
print "You really 'wow' me"
```

the computer will print "You really " then complain it doesn't know how to wow. The computer will print:

```
You really  
No such variable
```

Copy program onto paper To print a **copy of your program** onto paper, make sure the printer is turned on. Then tap the **Printer icon** (which is above your program) then "OK".

For example, if you program says —

```
print 4+2
```

the printer will print this copy onto paper:

```
print 4+2
```

The printer automatically underlines all orange words (such as "print"). It also prints today's date on the paper's bottom-right corner, so you can brag about when you printed it.

Copy answers onto paper To **copy the computer's answers onto paper**, make the program's top line say:

```
vdu 2
```

Make sure the printer is turned on, then run the program (by tapping the green triangle).

The answers will still appear in the output window, as usual; but afterwards, when you close the output window (by clicking its X), the printer will print a copy of the answers.

For example, if the program is —

```
print 4+2
```

the printer will print 6 onto paper (although the 6 won't be quite as far to the right as you expect).

If you want the printer to copy just *some* of the answers, type “vdu 2” above the lines whose answers you want copied and type “vdu 3” below them. The “vdu 2” means “copy to paper”; “vdu 3” means “hide from paper”; but all printing gets delayed until you close the output window (or until the computer encounters a line saying “vdu 12”, which forces the printer to spit out the answers immediately).

Save

If you want the computer to copy the program onto drive C, tap the **Save icon**. (It's above your program. It's left of the Printer icon. It's below the “U” in “Utilities”. It looks like a floppy disk, which is square.)

To make sure things stay simple, do this: tap “This PC” then double-tap “Documents”.

If you haven't invented a name for the program yet, here's what happens:

The computer says “Save As”.

Tap “This PC” then double-tap “Documents” (to emphasize you want the program to be stored in drive C's main Documents folder).

Tap in the “File name” box. Invent a name for your program. Invent any name you wish. For example, the name can be Joe or Sue or Lover or poem4u. Pick a name that reminds you of the program's purpose. When you finish typing the name, press the Enter key.

Then the computer automatically adds “.bbc” to the end of the program's name and copies the program to drive C's main “Documents” folder.

Exception: if the name you invented was already used by another program, the computer asks you, “Do you want to replace it?” Tap the “Yes” button if you want the new program to replace the old program, so the old program disappears. If you do *not* want the new program to replace the old program, tap the “No” button instead of “Yes” then invent a *different* name for your new program.

Suppose you're creating a program that's so long it takes you several hours to type. You'll be upset if, after several hours of typing, your town suddenly has a blackout that makes the computer forget what you typed. To protect yourself against such a calamity, tap Save every 15 minutes. Then if your town has a blackout, you'll lose just a few minutes of work; the rest of your work will have already been saved on the disk. Saving your program every 15 minutes protects you against blackouts and also “computer malfunction” and any careless errors you might make.

New When you've finished inventing and saving a program, **here's how to erase everything in the program window, so you can start writing a different program instead:** tap “File” (which is near the program window's top-left corner) then “New”.

(If you didn't save the program you worked on, the computer asks, “Save current changes?” If you want to save the program you worked on, tap the “Yes” button; if you do *not* want to save the program you worked on, tap the “No” button instead.)

Here's a faster way to erase everything in the program window:

While holding down the Ctrl key, tap the A key. Then press the Delete key.

Load If you saved a program onto the hard disk (drive C), here's how to use it again: tap the **Load icon**. (It's above your program. It looks like a file folder opening up, so it's also called the **Open icon**. It's below the “E” in “Edit”.) If the computer asks “Save current changes?” tap the “No” button (unless you really want “Yes”).

The computer says “Load”.

The first box should say:

```
> This PC > Documents >
```

If it doesn't say that yet, do this:

Tap “This PC” then double-tap “Documents”.

Then you see a list of Basic programs (and other files) in drive C's Documents folder. Double-tap the program you want.

Then the program appears in the program window. To run it, tap the green triangle.

Save As Suppose you've saved a program called Joe. Here's how to create a variant of it, called Joe2, so your hard disk contains *both* versions, Joe and Joe2.

First, create Joe and save it. Then make sure the program window still shows Joe's program lines. Edit those program's lines, to create whatever variant you want. Then name that variant “Joe2” by doing this:

Tap “File” then “Save As”. Type “Joe2” and press the Enter key.

Exit

When you've finished using Basic, close its window (the program window) by tapping its X button.

(If you didn't save the program you worked on, the computer asks, “Save current changes?” If you want to save the program you worked on, tap the “Yes” button; if you do *not* want to save the program you worked on, tap the “No” button instead.)

Then the computer will exit from Basic.

Line numbers

You can number the lines in your program. For example, instead of typing —

```
print "Let's munch"
print "sandwiches"
print "under the trees"
```

you can type:

```
1 print "Let's munch"
2 print "sandwiches"
3 print "under the trees"
```

Then when you're discussing your program with another programmer, you can talk about “line 2” instead of having to talk about “the line that mentions sandwiches”.

Selective numbering You can number just the lines you're planning to discuss.

For example, if you're planning to discuss just lines 1 and 3, you can number just those lines:

```
1 print "Let's munch"
   print "sandwiches"
3 print "under the trees"
```

Or if you prefer, number them like this:

```
1 print "Let's munch"
   print "sandwiches"
2 print "under the trees"
```

(When you type an unnumbered line, the computer automatically indents it to look like that.)

Number by tens Instead of making line numbers be 1, 2, 3, etc., make the line numbers be 10, 20, 30, etc., like this:

```
10 print "Let's munch"
20 print "sandwiches"
30 print "under the trees"
```

Then you can insert an extra numbered line without changing your numbers:

```
10 print "Let's munch"
15 print "delicious"
20 print "sandwiches"
30 print "under the trees"
```

Number automatically Above your program, you see this icon:

```
10
20
30
```

If you click it and then press the Enter key (while letting the “First line” box say 10, the “Increment” box say 10, and the “Remove unused line numbers box” be blank), the computer will automatically number all the lines in your program (10, 20, 30, etc.).

Multi-statement line

In your program, **a line can contain several statements separated by colons**, like this:

```
print "I dream": print "of you"
```

When you run that program, the computer will print “I dream” then print “of you”, like this:

```
I dream
of you
```

If you want to number the line, put the number at the far left, like this:

```
10 print "I dream": print "of you"
```

Become an expert

Congratulations! You’ve learned how to program!
C’mon, write some programs! It’s easy! Try it. You’ll have lots of fun!

A person who writes a program is called a **programmer**. Congratulations: *you’re* a programmer!

Write *several* programs like the ones I’ve shown you already. Then you can put on your résumé that you have “a wide variety of programming experience”, and you can talk your way into a programming job!

The rest of this chapter explains how to become a *good* programmer.

Practice Programming the computer is like driving a car: **the only way to become an expert is to put your hands on that mean machine and try it yourself.**

If you have a computer, put this book next to the computer’s keyboard. At the end of each paragraph, type the examples and look, look, see the computer run! Invent your own variations: try typing different numbers and strings. Invent your own programs: make the computer print your name or a poem; make it solve problems from your other courses and the rest of your life. The computer’s a fantastic toy. Play with it.

If you’re a student, don’t wait for your instructor to give lectures and assign homework. *Act now.* You’ll learn more from handling the computer than from lectures or readings. Experience counts.

Hang around your computer. Communicate with it every day. At first, that will be even harder than talking with a cat or a tree, because the computer belongs to a different species, a different kingdom; but keep trying. Get to know it as well as you know your best friend.

If you’re taking a French course, you might find French hard; and if you’re taking a computer course, you might find computers hard also. But even a stupid 3-year-old French kid can speak French, and even kindergarten kids can program the computer. They have just one advantage over you: practice!

Be bold In science fiction, computers blow up; in real life, they never do. No matter what keys you press, no matter what commands you type, you won’t hurt the computer. The computer is invincible! So go ahead and experiment. If it doesn’t like what you type, it will gripe at you, but so what?

Troubles When you try using the computer, you’ll have trouble — because you’re making a mistake, or the computer is broken, or the computer is weird and works differently from the majority computers discussed in this book. (Each computer has its own “personality”, its own quirks.)

Whenever you have trouble, laugh about it, and say, “Oh, boy! Here we go again!” (If you’re Jewish, you can say all that more briefly, in one word: “Oy!”) Then get some help.

Get help For help with your computer, read this book! For further help, read the manuals that came with your computer or ask the genie who got you the computer (your salesperson or parent or boss or teacher or friend).

If you’re sitting near computers in your office, school, or home, and other people are nearby, ask them for help. They’ll gladly answer your questions because they like to show off and because the way *they* got to know the answers was by asking.

Computer folks like to explain computers, just as priests like to explain religion. You’re joining a cult! Even if you don’t truly believe in “the power and glory of computers”, at least you’ll get a few moments of weird fun. Just play along with the weird computer people, boost their egos, and they’ll help you get through your initiation rite. Assert yourself and **ask questions**. “Shy guys finish last.” To get your money’s worth from a computer course, ask your teacher, classmates, lab assistants, and other programmers lots of questions!

Your town might have a **computer club**. (To find out, ask the local schools and computer stores.) Join the club and tell the members you’d like help with your computer. Probably some computer hobbyist will help you.

Call me anytime at **603-666-6644**: I’ll help you, free!

Variables

A letter can stand for a number. For example, x can stand for the number 47, as in this program:

```
x=47
print x+2
```

The top line says x stands for the number 47; x is a **name** for the number 47.

The bottom line says to print x+2. Since x is 47, the x+2 is 49; so the computer will print 49. That's the only number the computer will print; it will not print 47.

Jargon

A letter that stands for a number is called a **numeric variable**. In that program, x is a numeric variable; it stands for the number 47. The **value** of x is 47.

In that program, the statement "x=47" is called an **assignment statement**, because it **assigns** 47 to x.

A variable is a box

When you run that program, here's what happens inside the computer.

The computer's random-access memory (RAM) consists of electronic boxes. When the computer encounters the line "x=47", the computer puts 47 into box x, like this:

```
box x 
```

Then when the computer encounters the line "print x+2", the computer prints what's in box x, plus 2; so the computer prints 49.

Undefined variable

In that program, if you forget to type "x=47", here's what happens:

```
The computer won't know what x is, so x is undefined. When the computer tries to do "print x+2", it will gripe by saying "No such variable".
```

More examples

Here's another example:

```
y=38
print y-2
```

The top line says y is a numeric variable that stands for the number 38.

The bottom line says to print y-2. Since y is 38, the y-2 is 36; so the computer will print 36.

Example:

```
b=8
print b*3
```

The top line says b is 8. The bottom line says to print b*3, which is 8*3, which is 24; so the computer will print 24.

One variable can define another:

```
n=6
d= n+1
print n*d
```

The top line says n is 6. The next line says d is n+1, which is 6+1, which is 7; so d is 7. The bottom line says to print n*d, which is 6*7, which is 42; so the computer will print 42.

Changing a value

A value can change:

```
k=4
k=9
print k*2
```

The top line says k's value is 4. The next line changes k's value to 9, so the bottom line prints 18.

When you run that program, here's what happens inside the computer's RAM. The top line (k=4) makes the computer put 4 into box k:

```
box k 
```

The next line (k=9) puts 9 into box k. The 9 replaces the 4:

```
box k 
```

That's why the bottom line (print k*2) prints 18.

After that program runs, the computer will keep remembering k is 9. In the output window, if you type —

```
>print k
```

the computer will print 9. The computer will keep remembering k is 9, until you close the output window. **Closing the output window makes the computer forget the values of all variables.**

Hassles

When writing an equation (such as x=47), here's what you must put before the equal sign: the name of just one box (such as x). **So before the equal sign, put one variable:**

```
Allowed:    d=n+1 (d is one variable)
Not allowed: d-n=1 (d-n is two variables)
Not allowed: 1=d-n (1 is not a variable)
```

The variable on the left side of the equation is the only one that changes. For example, the statement d= n+1 changes the value of d but not n. The statement b=c changes the value of b but not c:

```
b=1
c=7
b=c
print b+c
```

The third line changes b, to make it equal c; so b becomes 7. Since both b and c are now 7, the bottom line prints 14.

"b=c" versus "c=b" Saying "b=c" has a different effect from "c=b". That's because "b=c" changes the value of b (but not c); saying "c=b" changes the value of c (but not b).

Compare these programs:

```
b=1      b=1
c=7      c=7
b=c      c=b
print b+c  print b+c
```

In the left program (which you saw before), the third line changes b to 7, so both b and c are 7. The bottom line prints 14.

In the right program, the third line changes c to 1, so both b and c are 1. The bottom line prints 2.

While you run those programs, here's what happens inside the computer's RAM. For both programs, the second and third lines do this:

```
box b 
```

```
box c 
```

In the left program, the fourth line makes the number in box b become 7 (so both boxes contain 7, and the bottom line prints 14). In the right program, the fourth line makes the number in box c become 1 (so both boxes contain 1, and the bottom line prints 2).

Modifying itself

A variable can modify itself:

```
d=5
d=d+2
print d
```

The top line says d is 5. The next line says d's new value is what d was before, plus 2, so d is 5+2, which is 7. The bottom line prints 7.

Instead of typing "d=d+2", you can type just "d+=2", which means "d is increased by 2".

Here are similar statements:

Statement	Meaning	Shortcut	Meaning
d=d+2	new d is old d + 2	d+=2	d is increased by 2
d=d-2	new d is old d - 2	d-=2	d is decreased by 2
d=d*2	new d is old d*2	d*=2	d is multiplied by 2
d=d/2	new d is old d/2	d/=2	d is divided by 2

Example:

```
d=100
d=d/4
print d
```

(or you can type just d/=4)

The top line says d is 100. The next line says d's new value is d's old value divided by 4 (so d is divided by 4), so d becomes 25. The bottom line prints 25.

When to use variables

Here's a practical example of when to use variables.

Suppose you're selling something that costs \$1297.43, and you want to do these calculations:

```
multiply $1297.43 by 2
multiply $1297.43 by .05
add $1297.43 to $483.19
divide $1297.43 by 37
```

To do those four calculations, you could run this program:

```
print 1297.43*2
print 1297.43*.05
print 1297.43+483.19
print 1297.43*37
```

But that program's silly, since it contains the number 1297.43 four times. This program's briefer, because it uses a variable:

```
c=1297.43
print c*2
print c*.05
print c+483.19
print c*37
```

It's also easier to update, if later your boss says you must change 1297.43 to a different number.

So **whenever you need to use a number several times, turn the number into a variable**, which will make your program briefer and easier to update.

String variables

A string is any collection of characters, such as "I love you". Each string must be in quotation marks.

A letter can stand for a string — if you put a dollar sign after the letter, like this:

```
g$="down"
print g$
```

The top line says g\$ stands for the string "down". The bottom line prints:

```
down
```

In that program, g\$ is a variable. Since it stands for a string, it's called a **string variable**.

Every string variable must end with a dollar sign. The dollar sign is supposed to remind you of a fancy S, which stands for String. The second line is pronounced, "g String is down".

If you're paranoid, you'll love this program:

```
t$="They're laughing at you!"
print t$
print t$
print t$
```

The top line says t\$ stands for the string "They're laughing at you!" The later lines make the computer print:

```
They're laughing at you!
They're laughing at you!
They're laughing at you!
```

Spaces between strings

Examine this program:

```
s$="sin"
k$="king"
print s$;k$
```

The bottom line says to print "sin" and then "king", so the computer will print:

```
sinking
```

Let's make the computer leave a space between "sin" and "king", so the computer prints:

```
sin king
```

To make the computer leave that space, choose one of these methods....

Method 1 Instead of saying s\$ = "sin", make s\$ include a space:
s\$="sin "

Method 2 Instead of saying k\$ = "king", make k\$ include a space:
k\$=" king"

Method 3 Instead of saying —
print s\$;k\$
say to print s\$ then a space then k\$:
print s\$;" ";k\$

Nursery rhymes

The computer can recite nursery rhymes:

```
p$="Peas porridge "  
print p$;"hot!"  
print p$;"cold!"  
print p$;"in the pot,"  
print "Nine days old!"
```

The top line says p\$ stands for “Peas porridge”. The later lines make the computer print:

```
Peas porridge hot!  
Peas porridge cold!  
Peas porridge in the pot,  
Nine days old!
```

This program prints a fancier rhyme:

```
h$="Hickory, dickory, dock!"  
m$="THE MOUSE (squeak! squeak!) "  
c$="THE CLOCK (tick! tock!) "  
print h$  
print m$;"ran up ";c$  
print c$;"struck one"  
print m$;"ran down"  
print h$
```

Lines 1-3 define h\$, m\$, and c\$. The later lines make the computer print:

```
Hickory, dickory, dock!  
THE MOUSE (squeak! squeak!) ran up THE CLOCK (tick! tock!)  
THE CLOCK (tick! tock!) struck one  
THE MOUSE (squeak! squeak!) ran down  
Hickory, dickory, dock!
```

Long variable names

A numeric variable’s name can be a letter (such as x) or a longer combination of characters, such as:

```
money_earned_in_November_2020_before_promotion
```

The variable’s name must begin with a letter. It can include small letters, capital letters, digits, and underlines. It can be as long as you wish!

It must not begin with a word that has a special meaning to the computer. For example, it cannot begin with the letters “print”.

If the variable stands for a string, it must end in a dollar sign.

Beginners are usually too lazy to type long variable names, so beginners use variable names that are short. But when you become a pro and write a long, fancy program containing hundreds of lines and hundreds of variables, you should use long variable names to help you remember each variable’s purpose.

In this book, I’ll use short variable names in short programs (so you can type those programs quickly) but long variable names in long programs (so you can keep track of which variable is which).

Programmers employed at Microsoft capitalize each word’s first letter and can’t use underlines. So instead of writing —

```
money_earned_in_November_2020_before_promotion
```

they write:

```
MoneyEarnedInNovember2020BeforePromotion
```

That’s harder to read; but if you’re programming for Microsoft, you have no choice.

Input

Humans ask questions; so to turn the computer into a human, you must make it ask questions too. **To make the computer ask a question, use the word “input”.**

This program makes the computer ask for your name:

```
input "what is your name";n$  
print "I adore anyone whose name is ";n$
```

When the computer sees that input line, the computer asks “What is your name?” then waits for you to answer the question. Your answer will be called n\$. For example, if you answer Maria, then n\$ is Maria. The bottom line makes the computer print:

```
I adore anyone whose name is Maria
```

When you run that program, here’s the whole conversation that occurs between the computer and you; I’ve underlined the part typed by you....

```
Computer asks for your name: What is your name? Maria  
Computer praises your name: I adore anyone whose name is Maria
```

Try that example. Be careful! When you type the input line, make sure you type the two quotation marks and the semicolon. You don’t have to type a question mark: when the computer runs your program, it will automatically put a question mark at the end of the question.

Just for fun, run that program again and pretend you’re somebody else....

```
Computer asks for your name: What is your name? Bud  
Computer praises your name: I adore anyone whose name is Bud
```

When the computer asks for your name, if you say something weird, the computer will give you a weird reply....

```
Computer asks: What is your name? none of your business!  
Computer replies: I adore anyone whose name is none of your business!
```

Abbreviation

Instead of typing the word “input”, you can abbreviate: type just the letter “i” and a period. When you press the Enter key at the end of the line, the computer will change the “i.” to “input”.

So instead of typing —

```
input "what is your name";n$
```

you can type just:

```
i. "what is your name";n$
```

College admissions

This program prints a letter, admitting you to the college of your choice:

```
input "What college would you like to enter";c$
print "Congratulations!"
print "You have just been admitted to ";c$
print "because it fits your personality."
print "I hope you go to ";c$;"."
print "      Respectfully yours,"
print "      The Dean of Admissions"
```

When the computer sees the INPUT line, the computer asks "What college would you like to enter?" and waits for you to answer. Your answer will be called c\$. If you'd like to be admitted to Harvard, you'll be pleased...

```
Computer asks you:  what college would you like to enter? Harvard
Computer admits you: Congratulations!
                   You have just been admitted to Harvard
                   because it fits your personality.
                   I hope you go to Harvard.
                   Respectfully yours,
                   The Dean of Admissions
```

You can choose any college you wish:

```
Computer asks you:  what college would you like to enter? Hell
Computer admits you: Congratulations!
                   You have just been admitted to Hell
                   because it fits your personality.
                   I hope you go to Hell.
                   Respectfully yours,
                   The Dean of Admissions
```

That program consists of three parts:

1. The computer begins by asking you a question ("What college would you like to enter?"). The computer's question is called the **prompt**, because it prompts you to answer.
2. Your answer (the college's name) is called **your input**, because it's information that you're *putting into* the computer.
3. The computer's reply (the admission letter) is called the **computer's output**, because it's the final answer that the computer puts out.

Input versus print

The word "input" is the opposite of the word "print".

The word "print" makes the computer print information out. The word "input" makes the computer take information in.

What the computer prints out is called the **output**. What the computer takes in is called **your input**.

Input and Output are collectively called **I/O**, so the input and print statements are called **I/O statements**.

Once upon a time

Let's make the computer write a story, by filling in the blanks:

```
Once upon a time, there was a youngster named _____
                                     your name

who had a friend named _____
                             friend's name

_____ wanted to _____
your name          verb (such as "pat") friend's name

but _____ didn't want to _____!
friend's name          verb (such as "pat") your name

Will _____?
your name          verb (such as "pat") friend's name

Will _____?
friend's name          verb (such as "pat") your name

To find out, come back and see the next exciting episode
of _____ and _____!
your name          friend's name
```

To write the story, the computer must ask for your name, your friend's name, and a verb. To make the computer ask, your program must say "input":

```
input "what is your name";y$
input "what's your friend's name";f$
input "In 1 word, say something you can do to your friend";v$
```

Then make the computer print the story:

```
print "Here's my story..."
print "Once upon a time, there was a youngster named ";y$
print "who had a friend named ";f$;"."
print y$;" wanted to ";v$;" ";f$;"."
print "but ";f$;" didn't want to ";v$;" ";y$;"!"
print "Will ";y$;" ";v$;" ";f$;"?"
print "Will ";f$;" ";v$;" ";y$;"?"
print "To find out, come back and see the next exciting episode"
print "of ";y$;" and ";f$;"!"
```

Here's a sample run:

```
what's your name? Dracula
what's your friend's name? Madonna
In 1 word, say something you can do to your friend? bite
Here's my story...
Once upon a time, there was a youngster named Dracula
who had a friend named Madonna.
Dracula wanted to bite Madonna,
but Madonna didn't want to bite Dracula!
Will Dracula bite Madonna?
Will Madonna bite Dracula?
To find out, come back and see the next exciting episode
of Dracula and Madonna!
```

Here's another run:

```
what's your name? Superman
what's your friend's name? King Kong
In 1 word, say something you can do to your friend? tickle
Here's my story...
Once upon a time, there was a youngster named Superman
who had a friend named King Kong.
Superman wanted to tickle King Kong,
but King Kong didn't want to tickle Superman!
Will Superman tickle King Kong?
Will King Kong tickle Superman?
To find out, come back and see the next exciting episode
of Superman and King Kong!
```

Try it: put in your own name, the name of your friend, and something you'd like to do to your friend.

Contest

The following program prints a certificate saying you won a contest. Since the program contains many variables, it uses long variable names to help you remember which variable is which:

```
input "what's your name";you$
input "what's your friend's name";friend$
input "what's the name of another friend";friend2$
input "Name a color";color$
input "Name a place";place$
input "Name a food";food$
input "Name an object";object$
input "Name a part of the body";part$
input "Name a style of cooking (such as baked or fried)";style$
print
print "Congratulations, ";you$;"!"
print "You've won the beauty contest, because of your gorgeous ";part$;"."
print "Your prize is a ";color$;" ";object$
print "plus a trip to ";place$;" with your friend ";friend$
print "plus--and this is the best part of all--"
print "dinner for the two of you at ";friend2$;"'s new restaurant,"
print "where ";friend2$;" will give you ";
print "all the ";style$;" ";food$;" you can eat."
print "Congratulations, ";you$;" , today's your lucky day!"
print "Now everyone wants to kiss your award-winning ";part$;"."
```

Here's a sample run:

```
What's your name? Long John Silver
What's your friend's name? the parrot
What's the name of another friend? Jim
Name a color? gold
Name a place? Treasure Island
Name a food? rum-soaked coconuts
Name an object? chest of jewels
Name a part of the body? missing leg
Name a style of cooking (such as baked or fried)? barbecued

Congratulations, Long John Silver!
You've won the beauty contest, because of your gorgeous missing leg.
Your prize is a gold chest of jewels
plus a trip to Treasure Island with your friend the parrot
plus--and this is the best part of all--
dinner for the two of you at Jim's new restaurant,
where Jim will give you all the barbecued rum-soaked coconuts you can eat.
Congratulations, Long John Silver, today's your lucky day!
Now everyone wants to kiss your award-winning missing leg.
```

Numeric input

This program makes the computer predict your future:

```
print "I predict what'll happen to you in the year 2030!"
input "In what year were you born";y
print "In the year 2030, you'll turn ";2030-y;" years old."
```

Here's a sample run:

```
I predict what'll happen to you in the year 2030!
In what year were you born? 1972
In the year 2030, you'll turn 58 years old.
```

Suppose you're selling tickets to a play. Each ticket costs \$2.79. (You decided \$2.79 would be a nifty price, because the cast has 279 people.) This program finds the price of multiple tickets:

```
input "How many tickets";t
print "The total price is $";t*2.79
```

This program tells you how much the "oil crisis" costs you when you drive your car:

```
input "How many miles do you want to drive"; m
input "How many pennies does a gallon of gas cost"; p
input "How many miles-per-gallon does your car get"; r
print "The gas for your trip will cost you $";m*p/(r*100)
```

Here's a sample run:

```
How many miles do you want to drive? 400
How many pennies does a gallon of gas cost? 264.9
How many miles-per-gallon does your car get? 31
The gas for your trip will cost you $34.1806452
```

Conversion

This program converts feet to inches:

```
input "How many feet";f
print f;" feet = ";f*12;" inches"
```

Here's a sample run:

```
How many feet? 3
3 feet = 36 inches
```

Trying to convert to the metric system? This program converts inches to centimeters:

```
input "How many inches"; i
print i;" inches = ";i*2.54;" centimeters"
```

Nice day today, isn't it? This program converts the temperature from Celsius to Fahrenheit:

```
input "How many degrees Celsius"; c
print c;" degrees celsius = ";c*1.8+32;" degrees Fahrenheit"
```

Here's a sample run:

```
How many degrees Celsius? 20
20 degrees Celsius = 68 degrees Fahrenheit
```

See, you can write the *Guide* yourself! Just hunt through any old math or science book, find any old formula (such as $f=c*1.8+32$), and turn it into a program.

Going & stopping

You can control how your computer goes and stops.

Wait

A **centisecond** is a very short period of time: just a hundredth of a second!

If you say —

```
wait 600
```

the computer will wait for 600 centiseconds, which is 6 seconds. That's how to make the computer pause for 6 seconds.

Give the 6-second pause before you reveal a joke's punch line:

```
print "Human, your intelligence is amazing! You must be an M.D.";
wait 600
print "--Mentally Deficient!"
```

That program makes the computer print the joke's setup ("Human, your intelligence is amazing! You must be an M.D."), then pause for 6 seconds, then reveal the joke's punch line, so the screen finally shows:

```
Human, your intelligence is amazing! You must be an M.D.--Mentally Deficient!
```

If you want the computer to pause for 9 seconds instead of 6, say "wait 900" instead of "wait 600".

This program makes the computer brag, then confess:

```
print "we computers are smart for three reasons."
print "The first is our VERY GOOD MEMORY."
print "The other two reasons...";
wait 900
print "I forgot."
```

The computer begins by bragging:

```
We computers are smart for three reasons.
The first is our VERY GOOD MEMORY.
The other two reasons...
```

But then the computer pauses for 10 seconds and finally admits:

```
I forgot.
```

This program makes the computer change its feelings, in surprising ways:

```
print "I'm up";
wait 300
print "set! I want to pee";
wait 400
print "k at you";
wait 500
print "r ma";
wait 600
PRINT "nua1."
```

The computer will print —

```
I'm up
```

then pause 3 seconds and change it to —

```
I'm upset! I want to pee
```

then pause 4 seconds and change it to —

```
I'm upset! I want to peek at you
```

then pause 5 seconds and change it to —

```
I'm upset! I want to peek at your ma
```

then pause 6 seconds and change it to:

```
I'm upset! I want to peek at your manual.
```

Experiment: invent your *own* jokes, and make the computer pause before printing the punch lines.

Repeat

This program makes the computer print the word "love" once:

```
print "love"
```

This fancier program makes the computer print the word "love" *three* times:

```
print "love"
print "love"
print "love"
```

When you run that program, the computer will print:

```
love
love
love
```

Let's make the computer print the word "love" *many* times. To do that, we must make the computer obey this line repeatedly:

```
print "love"
```

To make the computer obey the line repeatedly, say "repeat" above the line, so the program looks like this:

```
repeat
  print "love"
```

The computer automatically indents the print line for you, when you press the Enter key at that line's end.

How often do you want the computer to repeat? Underneath all that, say "until false" (which means "forever"), so the program looks like this:

```
repeat
  print "love"
until false
```

The computer automatically unindents the "until" for you, when you press the Enter key at that line's end.

When you run that program, the computer will print "love" many times, so it will print:

```
love
love
love
love
love
love
love
love
love
etc.
```

The computer will print "love" on every line of the output window.

But even when the output window is full of "love", the computer won't stop: the computer will try to print even more loves onto your screen! The computer will lose control of itself and try to devote its entire life to making love! The computer's mind will spin round and round, always circling back to the thought of making love again!

Since the computer's thinking keeps circling back to the same thought, the computer is said to be in a **loop**. In that program, "repeat" means "repeat what's underneath & indented"; "until false" means "do it forever". The lines that say "repeat" and "until false" — and the lines between them — form a loop, which is called a **repeat loop**.

To stop the computer's lovemaking madness, you must give the computer a "jolt" that will put it out of its misery and get it out of the loop. To jolt the computer out of the program, **abort** the program by using one of these methods:

Window method Close the output window (by tapping its X).

Keyboard method Press the Escape key. (It's the first key on the keyboard. It says "Esc" on it.) That stops the program from running and makes the output window show the word "Escape".

In that program, since the computer tries to go round and round the loop forever, the loop is called **infinite**. The only way to stop an infinite loop is to abort it.

Semicolon For more lovely fun, put a semicolon after “love”, so the program looks like this:

```
repeat
  print "love";
until false
```

The semicolon makes the computer print “love” *next to* “love”, so the output window looks like this:

```
love love
love love love love love love love love love love love love love love love love love love
love love love love love love love love love love love love love love love love love love
etc.
```

If you put a space after love, like this —

```
repeat
  print "love ";
until false
```

the computer will put a space after each love:

```
love l
ove love lo
ve love lov
e love love
etc.
```

Bigger loop Run this program:

```
repeat
  print "dog";
  print "cat";
until false
```

Lines 2 & 3 (which say print “dog” and print “cat”) make the computer print “dog” and then print “cat” next to it. Since those lines are between the words “repeat” and “until”, the computer does them repeatedly — print “dog”, then print “cat”, then print “dog” again, then print “cat” again — so the screen looks like this:

```
dogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcat
dogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcat
dogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcatdogcat
etc.
```

The computer will keep printing “dog” and “cat” until you abort the program.

Cls

To make the computer erase everything that was printed in the output window, type type “cls” (which means “CLear the Screen”).

You can type “cls” in the output window or make “cls” be a line in your program (so the computer will erase what’s in the output window). Typing “cls” lets you hide whatever naughty thing you put in the output window.

Blinking Let’s make the screen say “Stop pollution!” and make that message blink.

To do that, flash “Stop pollution!” onto the screen for 2 seconds, then turn that message off for 1 second (so the screen is blank), then flash that message on again. Here’s the program:

```
repeat
  print "Stop pollution!"
  wait 200
  cls
  wait 100
until false
```

Lines 2 & 3 (which say to print “Stop pollution!” and wait for 2 seconds) flash the message “Stop pollution!” onto the output window and keep it there for 2 seconds. The next pair of lines (cls and wait for 1 second) make the output window become blank for 1 second. Since those lines are all in a repeat loop, the computer does them repeatedly — flash message then blank, flash message then blank, flash message then blank — so your output window becomes a continually flashing sign. It will keep flashing until you abort the program.

Instead of saying “Stop pollution!”, edit that program so it flashes your favorite phrase instead, such as “Save the whales!” or “Marry me!” or “Keepa youse hands offa my computer!” or “Jesus saves — America spends!” or “In God we trust — all others pay cash” or “Please wait — Dr. Doom will be with you shortly” or “Let’s rock!” or whatever else turns you on. Make the computer say whatever you feel emotional about. Like a dog, the computer imitates its master’s personality. If your computer acts “cold and heartless”, it’s because *you* are!

In the program, you typed just a few lines; but since the top line says “repeat” (and the bottom line says “until false”), the computer does an infinite loop. By saying “repeat”, you can make the computer do an infinite amount of work. Moral: **the computer can turn a finite amount of human energy into an infinite amount of good**. Putting it another way: **the computer can multiply your abilities by infinity**.

Conditions

Here's how to restrict the computer, so it performs certain lines only under certain conditions....

If

Let's write a program so that if the human is less than 18 years old, the computer will say:

```
You are still a minor.
```

Here's the program:

```
input "How old are you";age
if age<18 print "You are still a minor"
```

The top line makes the computer ask "How old are you" and wait for the human to type an age. Since **the symbol for "less than" is "<"**, the bottom line says: if the age is less than 18, then print "You are still a minor".

Go ahead! Run that program! The computer begins the conversation by asking:

```
How old are you?
```

Try saying you're 12 years old, by typing a 12, so the screen looks like this:

```
How old are you? 12
```

When you finish typing the 12 and press the Enter key at the end of it, the computer will reply:

```
You are still a minor
```

Try running that program again, but this time try saying you're 50 years old instead of 12, so the screen looks like this:

```
How old are you? 50
```

When you finish typing the 50 and press the Enter key at the end of it, the computer will *not* say "You are still a minor". Instead, the computer will say nothing — since we didn't teach the computer how to respond to adults yet!

In that program, the most important line says:

```
if age<18 print "You are still a minor"
```

In that line, "age<18" is the **condition**. The whole line is called a **conditional statement**. If the condition is true (if age is really less than 18), the computer does the **action**, which comes afterwards:

```
print "You are still a minor"
```

Else

Let's teach the computer how to respond to adults.

Here's how to program the computer so that if the age is less than 18, the computer will say "You are still a minor", but if the age is *not* less than 18 the computer will say "You are an adult" instead:

```
input "How old are you";age
if age<18 print "You are still a minor" else print "You are an adult"
```

In programs, **the word "else" means "otherwise"**. That program's bottom line means: if the age is less than 18, print "You are still a minor"; otherwise (if the age is *not* less than 18), print "You are an adult". So the computer will print "You are still a minor" or else print "You are an adult", depending on whether the age is less than 18.

Try running that program! If you say you're 50 years old, so the screen looks like this —

```
How old are you? 50
```

the computer will reply by saying:

```
You are an adult
```

Daddy's always right A Daddy wrote a program for his 5-year-old son, John. When John runs the program and types his name, the computer asks "What's 2 and 2?" If John answers 4, the computer says "No, 2 and 2 is 22". If he runs the program again and answers 22, the computer says "No, 2 and 2 is 4". No matter how many times he runs the program and how he answers the question, the computer says he's wrong. But when Daddy runs the program, the computer replies, "Yes, Daddy is always right".

Here's how Daddy programmed the computer:

```
input "what's your name";n$
input "what's 2 and 2";a
if n$="Daddy" print "Yes, Daddy is always right": end
if a=4 print "No, 2 and 2 is 22" else print "No, 2 and 2 is 4"
```

Multi-line if

If the age is less than 18, here's how to make the computer print "You are still a minor" and also print "Ah, the joys of youth":

```
if age<18 print "You are still a minor": print "Ah, the joys of youth"
```

Here's a more sophisticated way to say the same thing:

```
if age<18 then
  print "You are still a minor"
  print "Ah, the joys of youth"
endif
```

That sophisticated way (in which you type 4 short lines instead of a single long line) is called a **multi-line if** (or a **block if**).

In a multiline if:

The top line must say "if" and **"then"** (with nothing after "then").

The computer automatically indents the middle lines, which are called the **block** and typically say "print".

The bottom line must say **"endif"** (which means "end the multi-line if"). The computer automatically unindents it (when you've typed it and press the Enter key).

In the middle of a multi-line if, you can say "else":

```
if age<18 then
  print "You are still a minor"
  print "Ah, the joys of youth"
else
  print "You are an adult"
  print "We can have adult fun"
endif
```

That means: if the age is less than 18, then print "You are still a minor" and "Ah, the joys of youth"; otherwise (if age *not* under 18) print "You are an adult" and "We can have adult fun".

Case

Let's turn your computer into a therapist!

To make the computer ask the patient, "How are you?", begin the program like this:

```
input "How are you";a$
```

Make the computer continue the conversation by responding this way:

```
If the patient says "fine", print "That's good!"
If the patient says "lousy" instead, print "Too bad!"
If the patient says anything else instead, print "I feel the same way!"
```

To accomplish all that, type this:

```
case a$ of
  when "fine" print "That's good!"
  when "lousy" print "Too bad!"
  otherwise print "I feel the same way!"
endcase
```


Fancy conditions

You can make the “if” clause very fancy:

“If” clause	Meaning
if b\$="male"	If b\$ is “male”
if b=4	If b is 4
if b<4	If b is less than 4
if b>4	If b is greater than 4
if b<=4	If b is less than or equal to 4
if b>=4	If b is greater than or equal to 4
if b<>4	If b is not 4
if b\$<"male"	If b\$ is a word that comes before “male” in dictionary
if b\$>"male"	If b\$ is a word that comes after “male” in dictionary

In the IF statement, the symbols =, <, >, <=, >=, and <> are called **relations**.

When writing a relation, mathematicians and computerists habitually **put the equal sign last**:

Right	Wrong (but the computer understands it anyway)
<=	=<
>=	=>

To say “not equal to”, say “less than or greater than”, like this: <>.

Or The computer understands the word “or”. For example, here’s how to say, “If x is either 7 or 8, print the word *wonderful*”:

```
if x=7 or x=8 print "wonderful"
```

That example is composed of two conditions: the first condition is “x=7”; the second condition is “x=8”. Those two conditions combine, to form “x=7 or x=8”, which is called a **compound condition**.

If you use the word “or”, put it between two conditions.

Right: if x=7 or x=8 print "wonderful"
Right because “x=7” and “x=8” are conditions

Wrong: if x=7 or 8 print "wonderful"
Wrong because “8” is not a condition

And The computer understands the word “and”. Here’s how to say, “If p is more than 5 and less than 10, print *tuna fish*”:

```
if p>5 and p<10 print "tuna fish"
```

Here’s how to say, “If s is at least 60 and less than 65, print *you almost failed*”:

```
if s>=60 and s<65 print "you almost failed"
```

Here’s how to say, “If n is a number from 1 to 10, print *that’s good*”:

```
if n>=1 and n<=10 print "that's good"
```

Can a computer be President?

To become President of the United States, you need 4 basic skills:

First, you must be a **good talker**, so you can give effective speeches saying “Vote for me!”, express your views, and make folks do what you want.

But even if you’re a good talker, you’re useless unless you’re also a **good listener**. You must be able to listen to people’s needs and ask, “What can I do to make you happy and get you to vote for me?”

But even if you’re a good talker and listener, you’re still useless unless you can **make decisions**. Should you give more money to poor people? Should you bomb the enemy? Which actions should you take, and under what conditions?

But even if you’re a good talker and listener and decision maker, you still need one more trait to become President: you must be able to take the daily grind of politics. You must, again and again, shake hands, make compromises, and raise funds. You must have the **patience to put up with the repetitive monotony** of those chores.

So altogether, to become President you need to be a good talker and listener and decision maker and also have the patience to put up with monotonous repetition.

Those are exactly the 4 qualities the computer has!

The word **“print”** turns the computer into a good speech-maker. By using the word “print”, you can make the computer write whatever speech you wish.

The word **“input”** turns the computer into a good listener. By using the word “input”, you can make the computer ask humans lots of questions, to find out who the humans are and what they want.

The word **“if”** turns the computer into a decision maker. The computer can analyze the “if” condition, determine whether that condition is true, and act accordingly.

Finally, the word **“goto”** enables the computer to perform loops, which the computer will repeat patiently.

So by using the words “print”, “input”, “if”, and “goto”, you can make the computer imitate any intellectual human activity. Those 4 magic words — “print”, “input”, “if”, and “goto” — are the only concepts you need, to write whatever program you wish!

Yes, you can make the computer imitate the President of the United States, do your company’s payroll, compose a beautiful poem, play a perfect game of chess, contemplate the meaning of life, act as if it’s falling in love, or do whatever other intellectual or emotional task you wish, by using those 4 magic words. The only question is: how? This book teaches you how, by showing you many examples of programs that do those remarkable things.

What programmers believe Yes, we programmers believe that all of life can be explained and programmed. We believe all of life can be reduced to just those 4 concepts: “print”, “input”, “if”, and “goto”. Programming is the ultimate act of scientific reductionism: programmers reduce all of life scientifically to just 4 concepts.

The words that the computer understands are called **keywords**. The 4 essential keywords are “print”, “input”, “if”, and “goto”.

The computer also understands extra keywords, such as:

```
wait
repeat, until, false
cls
goto, end
if, else, then, endif
case, of, when, otherwise, endcase
or, and
```

Those extra keywords aren’t necessary: if they hadn’t been invented, you could still write programs without them. But they make programming easier.

A Basic programmer is a person who translates an ordinary English sentence (such as “act like the President” or “do the payroll”) into a series of Basic statements, using keywords such as “print”, “input”, “if”, “goto”, “wait”, “repeat”, etc.

The mysteries of life Let’s dig deeper into the mysteries of “print”, “input”, “if”, “goto”, and the extra keywords. The deeper we dig, the more you’ll wonder: are *you* just a computer, made of flesh instead of wires? Can everything *you* do be explained in terms of “print”, “input”, “if”, and “goto”?

By the time you finish this book, you’ll know!

Exiting a loop

This program plays a guessing game, where the human tries to guess the computer's favorite color, which is pink:

```
1 input "what's my favorite color";guess$
  if guess$="pink" then
    print "Congratulations! You discovered my favorite color."
  else
    print "No, that's not my favorite color. Try again!"
    goto 1
  endif
```

The "input" line asks the human to guess the computer's favorite color; the guess is called guess\$.

If the guess is "pink", the computer prints:

```
Congratulations! You discovered my favorite color.
```

But if the guess is *not* "pink", the computer will instead print "No, that's not my favorite color" and then go back to line 1, which asks the human again to try guessing the computer's favorite color.

End Here's how to write that program without saying "goto":

```
repeat
  input "what's my favorite color";guess$
  if guess$="pink" then
    print "Congratulations! You discovered my favorite color."
  end
  endif
  print "No, that's not my favorite color. Try again!"
until false
```

That new version of the program contains a "repeat" loop. That loop makes the computer repeat this repeatedly: ask "What's my favorite color?" and then print "No, that's not my favorite color."

The only way to stop the loop is to guess "pink", which makes the computer print "Congratulations!" and end.

Exit repeat Here's another way to write that program without saying "goto":

```
repeat
  input "what's my favorite color";guess$
  if guess$="pink" exit repeat
  print "No, that's not my favorite color. Try again!"
until false
print "Congratulations! You discovered my favorite color."
```

That program's "repeat" loop makes the computer do this repeatedly: ask "What's my favorite color?" and then print "No, that's not my favorite color."

The only way to stop the loop is to guess "pink", which makes the computer exit from the "repeat" loop; then the computer proceeds to the line underneath the "repeat" loop. That line prints:

```
Congratulations! You discovered my favorite color.
```

Until condition Here's a shorter way to program the guessing game:

```
repeat
  print "You haven't guessed my favorite color yet!"
  input "what's my favorite color";guess$
until guess$="pink"
print "Congratulations! You discovered my favorite color."
```

That program's "repeat" loop makes the computer do this repeatedly: say "You haven't guessed my favorite color yet!" and then ask "What's my favorite color?"

The "until" line makes the computer repeat the indented lines again and again, until the guess is "pink". When the guess is "pink", the computer proceeds to the line underneath the loop and prints "Congratulations!".

The "until" condition (guess\$="pink") is called the **loop's goal**. The computer does the loop repeatedly, until the loop's goal is achieved. Here's how:

The computer does the indented lines, then checks whether the goal is achieved yet. If the goal is *not* achieved yet, the computer does the indented lines again, then checks again whether the goal is achieved. The computer does the loop again and again, until the goal is achieved. Then the computer, proud at achieving the goal, does the program's **finale**, which consists of any lines under the "until" line.

Saying —

```
until guess$="pink"
```

is just a briefer way of saying this pair of lines:

```
if guess$="pink" exit repeat
until false
```

For...next

Let's make the computer print every number from 1 to 20, like this:

```
1
2
3
4
5
6
7
etc.
20
```

Here's the program:

```
for x=1 to 20
  print x
next
```

The top line (for x=1 to 20) says x will be every number from 1 to 20; so x will be 1, then 2, then 3, etc. The line underneath, which is indented, says what to do about each x; it says to print each x.

Whenever you write a program that contains the word "for", you must say "next"; so the bottom line says "next".

The indented line, which is between the "for" line and the "next" line, is the line that the computer will do repeatedly; so the computer will repeatedly print x. The first time the computer prints x, the x will be 1, so the computer will print:

```
1
```

The next time the computer prints x, the x will be 2, so the computer will print:

```
2
```

The computer will print every number from 1 up to 20.

When men meet women

Let's make the computer print these lyrics:

```
I saw 2 men
meet 2 women.
Tra-la-la!
```

```
I saw 3 men
meet 3 women.
Tra-la-la!
```

```
I saw 4 men
meet 4 women.
Tra-la-la!
```

```
I saw 5 men
meet 5 women.
Tra-la-la!
```

```
They all had a party!
Ha-ha-ha!
```

To do that, type these lines —

```
The first line of each verse: print "I saw ";x;" men"
The second line of each verse: print "meet ";x;" women."
The third line of each verse: print "Tra-la-la!"
Blank line under each verse: print
```

and make x be every number from 2 up to 5:

```
for x=2 to 5
  print "I saw ";x;" men"
  print "meet ";x;" women."
  print "Tra-la-la!"
  print
next
```

At the end of the song, print the closing couplet:

```
for x=2 to 5
  print "I saw ";x;" men"
  print "meet ";x;" women."
  print "Tra-la-la!"
  print
next
print "They all had a party!"
print "Ha-ha-ha!"
```

That program makes the computer print the entire song.

Here's an analysis:

```
for x=2 to 5
  print "I saw ";x;" men"
  print "meet ";x;" women."
  print "Tra-la-la!"
  print
next
print "They all had a party!"
print "Ha-ha-ha!"
```

The computer will do the indented lines repeatedly, for x=2, x=3, x=4, and x=5.

Then the computer will print this couplet once.

Since the computer does the indented lines repeatedly, those lines form a loop. Here's the general rule: **the statements between "for" and "next" form a loop**. The computer goes round and round the loop, for x=2, x=3, x=4, and x=5. Altogether, it goes around the loop 4 times, which is a finite number. Therefore, the loop is **finite**.

If you don't like the letter x, choose a different letter. For example, you can choose the letter i:

```
for i=2 TO 5
  print "I saw ";i;" men"
  print "meet ";i;" women."
  print "Tra-la-la!"
  print
next
print "They all had a party!"
print "Ha-ha-ha!"
```

When using the word "for", most programmers prefer the letter i; most programmers say "for i" instead of "for x". Saying "for i" is an "old tradition". Following that tradition, the rest of this book says "for i" (instead of "for x"), except in situations where some other letter feels more natural.

Print the squares

To find the **square** of a number, multiply the number by itself. The square of 3 is "3 times 3", which is 9. The square of 4 is "4 times 4", which is 16.

Let's make the computer print the square of 3, 4, 5, etc., up to 20, like this:

```
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
etc.
The square of 20 is 400
```

To do that, type this line —

```
print "The square of ";i;" is";i*i
```

and make i be every number from 3 up to 20, like this:

```
for i=3 to 20
  print "The square of ";i;" is ";i*i
next
```

Count how many copies

This program, which you saw before, prints "love" on every line of your screen:

```
repeat
  print "love"
until false
```

That program prints "love" again and again, until you abort the program (by closing the output window or pressing the Esc key).

But what if you want to print "love" just 20 times? This program prints "love" just 20 times:

```
for i=1 to 20
  print "love"
next
```

As you can see, "for...next" resembles "repeat...until false" but is smarter: while doing for...next, the computer counts!

Count to midnight

This program makes the computer count to midnight:

```
for i=1 to 11
  print i
next
print "midnight"
```

The computer will print:

```
1
2
3
4
5
6
7
8
9
10
11
midnight
```

Print horizontally

Let's make the computer print those numbers horizontally, like this:

```
1 2 3 4 5 6 7 8 9 10 11 midnight
```

Here's the program:

```
for i=1 to 11
  print ;i;" ";
next
print "midnight"
```

In the "print" line, the first semicolon prevents the computer from putting lots of spaces before each number. The " " makes the computer put a single space after each number. The final semicolon makes the computer print the output all on the same line instead of pressing the Enter key after each number.

If you want the computer to press the Enter key before "midnight", insert a "print" line:

```
for i=1 to 11
  print ;i;" ";
next
print
print "midnight"
```

That extra "print" line makes the computer press the Enter key just before "midnight", so the computer will print "midnight" on a separate line, like this:

```
1 2 3 4 5 6 7 8 9 10 11
midnight
```

Nested loops

Let's make the computer count to midnight 3 times, like this:

```
1 2 3 4 5 6 7 8 9 10 11
midnight
1 2 3 4 5 6 7 8 9 10 11
midnight
1 2 3 4 5 6 7 8 9 10 11
midnight
```

To do that, put the entire program between the words "for" and "next":

```
for j=1 to 3
  for i=1 to 11
    print ;i;" ";
  next
  print
  print "midnight"
next
```

That version contains a loop inside a loop: the loop that says "for i" is inside the loop that says "for j". The j loop is called the **outer loop**; the i loop is called the **inner loop**. The inner loop's variable must differ from the outer loop's. Since we called the inner loop's variable "i", the outer loop's variable must *not* be called "i"; so I picked the letter j instead.

Programmers often think of the outer loop as a bird's nest, and the inner loop as an egg *inside the nest*. So programmers say the inner loop is **nested in** the outer loop; the inner loop is a **nested loop**.

Abnormal exit

Earlier, we programmed a game where the human tries to guess the computer's favorite color, pink. Here's a fancier version of the game, in which the human gets just 5 guesses:

```
print "I'll give you 5 guesses..."
for i=1 to 5
  input "what's my favorite color";guess$
  if guess$="pink" goto 1
  print "No, that's not my favorite color."
next
print "Sorry, your 5 guesses are up! You lose."
end
1 print "Congratulations! You discovered my favorite color."
print "It took you ";i;" guesses."
```

The top line warns the human that just 5 guesses are allowed. The "for" line makes the computer count from 1 to 5; to begin, i is 1. The "input" line asks the human to guess the computer's favorite color; the guess is called guess\$.

If the guess is "pink", the computer jumps down to the line numbered 1, prints "Congratulations!", and tells how many guesses the human took. But if the guess is *not* "pink", the computer will print "No, that's not my favorite color" and go on to the next guess.

If the human guesses 5 times without success, the computer proceeds to the line that prints "Sorry, ... You lose."

For example, if the human's third guess is "pink", the computer prints:

```
Congratulations! You discovered my favorite color.
It took you 3 guesses.
```

If the human's very first guess is "pink", the computer prints:

```
Congratulations! You discovered my favorite color.
It took you 1 guesses.
```

Saying "1 guesses" is bad grammar but understandable.

That program contains a "for...next" loop. The "for" line says the loop will normally be done 5 times. The line below the loop (which says to print "Sorry") is the loop's **normal exit**. But if the human happens to input "pink", the computer jumps out of the loop early, to line numbered 1, which is the loop's **abnormal exit**.

Here's how to write that program without saying "goto":

```
print "I'll give you 5 guesses..."
for i=1 to 5
  input "what's my favorite color";guess$
  if guess$="pink" then
    print "Congratulations! You discovered my favorite color."
    print "It took you ";i;" guesses."
  end
endif
print "No, that's not my favorite color."
next
print "Sorry, your 5 guesses are up! You lose."
```

Step

The "for" statement can be varied:

Statement	Meaning
for i=5 to 17 step .1	The i will go from 5 to 17, counting by tenths. So i will be 5, then 5.1, then 5.2, etc., up to 17.
for i=5 to 17 step 3	The i will be every 3 rd number from 5 to 17. So i will be 5, then 8, then 11, then 14, then 17.
for i=17 to 5 step -3	The i will be every 3 rd number from 17 down to 5. So i will be 17, then 14, then 11, then 8, then 5.

To count down, you *must* use the word "step". To count from 17 down to 5, give this instruction:

```
FOR i = 17 TO 5 STEP -1
```

This program prints a rocket countdown:

```
for i=10 to 1 step -1
  print i
next
print "Blast off!"
```

The computer will print:

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```

This statement is tricky:

```
for i=5 to 16 step 3
```

It says to start *i* at 5, and keep adding 3 until it gets past 16. So *i* will be 5, then 8, then 11, then 14. The *i* won't be 17, since 17 is past 16. The first value of *i* is 5; the last value is 14.

In the statement "for *i*=5 to 16 step 3, the **first value** or **initial value** of *i* is 5, the **limit value** is 16, and the **step size** or **increment** is 3. The *i* is called the **counter** or **index** or **loop-control variable**. Although the limit value is 16, the **last value** or **terminal value** is 14.

Programmers usually say "for *i*" instead of "for *x*", because the letter *i* reminds them of the word **index**.

Round-off errors

If the step size is a decimal, the computer might make small errors (called round-off errors), which can add up to a result that's very wrong.

For example, suppose you say:

```
for i=5 to 17 step .1
```

That means you want the last few values of *i* to be 16.8, 16.9, and 17; but the computer will accidentally make the step size be slightly *more* than .1, so the computer's last few values of *i* will be about 16.800000000000000003 and 16.900000000000000003. The computer will refuse to do the next number (which would be about 17.000000000000000003), since you said not to go past 17; so the last *i* will be 16.900000000000000003, which isn't at all what you wanted for the last value!

To make the last *i* be about 17, make the limit value be slightly *more* than 17, like this —

```
for i = 5 to 17.01 step .1
```

or, better yet, avoid a decimal step size by using this pair of lines instead:

```
for j=50 to 170
  i=j/10
```

That makes *i* indeed be 5 then 5.1 then 5.2, etc., up to 17.

Data...read

Let's make the computer print this message:

```
I love meat
I love potatoes
I love lettuce
I love tomatoes
I love honey
I love cheese
I love onions
I love peas
```

That message concerns this list of food: meat, potatoes, lettuce, tomatoes, honey, cheese, onions, peas. That list doesn't change: the computer continues to love those foods throughout the entire program.

A list that doesn't change is called **data**. So in the message about food, the data is meat, potatoes, lettuce, tomatoes, honey, cheese, onions, peas.

Whenever a problem involves data, put the data at the program's top, like this:

```
data meat,potatoes,lettuce,tomatoes,honey,cheese,onions,peas
```

You must tell the computer to read the data:

```
data meat,potatoes,lettuce,tomatoes,honey,cheese,onions,peas
read a$
```

That "read" line makes the computer read the first datum ("meat") and call it *a\$*. So *a\$* is "meat".

Since *a\$* is "meat", this shaded line makes the computer print "I love meat":

```
data meat,potatoes,lettuce,tomatoes,honey,cheese,onions,peas
read a$
print "I love ";a$
```

Hooray! We made the computer handle the first datum correctly: we made the computer print "I love meat".

To make the computer handle the rest of the data (potatoes, lettuce, etc.), tell the computer to read and print the rest of the data, by putting the "read" and "print" lines in a loop. Since we want the computer to read and print all 8 data items (meat, potatoes, lettuce, tomatoes, honey, cheese, onions, peas), put the read and print lines in a loop that gets done 8 times, by making the loop say "for *i*=1 to 8":

```
data meat,potatoes,lettuce,tomatoes,honey,cheese,onions,peas
for i=1 to 8
  read a$
  print "I love ";a$
next
```

Since that loop's main purpose is to read the data, it's called a **read loop**.

When writing that program, make sure the "for" line's last number (8) is the number of data items. If the "for" line accidentally says 7 instead of 8, the computer won't read or print the 8th data item. If the "for" line accidentally says 9 instead of 8, the computer will try to read a 9th data item, realize no 9th data item exists, and gripe by saying:

Out of data

Let's make the computer end by printing "Those are the foods I love", like this:

```
I love meat
I love potatoes
I love lettuce
I love tomatoes
I love honey
I love cheese
I love onions
I love peas
Those are the foods I love
```

To make the computer print that ending, put a PRINT line at the end of the program:

```
data meat,potatoes,lettuce,tomatoes,honey,cheese,onions,peas
for i=1 TO 8
  read a$
  print "I love ";a$
next
print "Those are the foods I love"
```

End mark

When writing that program, we had to count the data items and put that number (8) at the end of the “for” line.

Here’s a better way to write the program, so you don’t have to count the data items:

```

data meat,potatoes,lettuce,tomatoes,honey,cheese,onions,peas
data end
repeat
  read a$: if a$="end" exit repeat
  print "I love ";a$
until false
print "Those are the foods I love"

```

The second line (data end) is called the **end mark**, since it marks the end of the data. The “read” line means:

```

read a$ from the data;
but if a$ is the “end” of the data, exit from the “repeat” loop.

```

When the computer exits from the “repeat” loop, the computer prints “Those are the foods I love”. So altogether, the entire program makes the computer print:

```

I love meat
I love potatoes
I love lettuce
I love tomatoes
I love honey
I love cheese
I love onions
I love peas
Those are the foods I love

```

The routine that says:

```

  if a$="end" exit repeat

```

is called the **end routine**, because the computer does that routine when it reaches the data’s end.

Henry the Eighth

Let’s make the computer print this nursery rhyme:

```

I love ice cream
I love red
I love ocean
I love bed
I love tall grass
I love to wed

I love candles
I love divorce
I love kingdom
I love my horse
I love you
Of course, of course,
For I am Henry the Eighth!

```

If you own a jump rope, have fun: try to recite that poem while skipping rope!

This program makes the computer recite the poem:

```

data ice cream,red,ocean,bed,tall grass,to wed
data candles,divorce,my kingdom,my horse,you
data end
repeat
  read a$: if a$="end" exit repeat
  print "I love ";a$
  if a$="to wed" print
until false
print "Of course, of course,"
print "For I am Henry the Eighth!"

```

Since the data’s too long to fit on a single line, I’ve put part of the data in the top line and the rest in line 2. Each line of data must begin with the word data. In each line, put commas between the items. Do *not* put a comma at the end of the line.

The program resembles the previous one. The new line (if a\$=“to wed” print) makes the computer leave a blank line underneath “to wed”, to mark the first verse’s bottom.

Pairs of data

Let’s throw a party! To make the party yummy, let’s ask each guest to bring a kind of food that resembles the guest’s name. For example, let’s have Sal bring salad, Russ bring Russian dressing, Sue bring soup, Tom bring turkey, Winnie bring wine, Kay bring cake, and Al bring Alka-Seltzer.

Let’s send all those people invitations, in this form:

```

Dear _____,
    person's name

```

Let's party in the clubhouse at midnight!

```

Please bring _____
    food

```

Here’s the program:

```

data Sal,salad,Russ,Russian dressing,Sue,soup,Tom,turkey
data winnie,wine,Kay,cake,Al,Alka-Seltzer
data end,end
vdu 2
repeat
  read person$,food$: if person$="end" exit repeat
  print "Dear ";person$;","
  print "    Let's party in the clubhouse at midnight!"
  print "Please bring ";food$;","
  vdu 12
until false
vdu 3
print "I've finished writing the letters."

```

The data comes in pairs. For example, the first pair consists of “Sal” and “salad”; the next pair consists of “Russ” and “Russian dressing”. Since the data comes in pairs, you must make the end mark also be a pair (data end,end).

To make the output appear on paper, say “vdu 2”.

Since the data comes in pairs, the “read” line says to read a pair of data (person\$ and food\$). The first time that the computer encounters the “read” line, person\$ is “Sal”; food\$ is “salad”. Then the print lines print this message onto the screen and paper:

```

Dear Sal,
    Let's party in the clubhouse at midnight!
Please bring salad.

```

The “vdu 12” makes the computer eject the paper from the printer.

Then the computer comes to “until false”, which sends the computer back to the word “repeat”, which sends the computer to the “read” line again, which reads the next pair of data, so person\$ becomes “Russ” and food\$ becomes “Russian dressing”. The “print” lines print onto paper:

```

Dear Russ,
    Let's party in the clubhouse at midnight!
Please bring Russian dressing.

```

The computer prints similar letters to all the people.

After all people have been handled, the “read” statement comes to the end mark (data end,end), so that person\$ and food\$ both become “end”. Since person\$ is “end”, the “if” statement makes the computer exit the “repeat” loop, so the computer does “vdu 3” (which turns off the printer’s output) and prints this message onto just the screen:

```

I've finished writing the letters.

```

In that program, you need *two* ends to mark the data’s ending, because the “read” statment says to read two strings (person\$ and food\$).

Debts Suppose these people owe you things:

Person	What the person owes
Bob	\$537.29
Mike	a dime
Sue	2 golf balls
Harry	a steak dinner at Mario's
Mommy	a kiss

Let's remind those people of their debt, by writing them letters, in this form:

```
Dear _____,
    person's name

    I just want to remind you...

that you still owe me _____.
                    debt
```

To start writing the program, begin by feeding the computer the data. The final program is the same as the previous program, except for the part I've shaded:

```
data Bob,$537.29,Mike,a dime,Sue,2 golf balls
data Harry,a steak dinner at Mario's,Mommy,a kiss
data end,end
vdu 2
repeat
    read person$,debt$: if person$="end" exit repeat
    print "Dear ";person$;","
    print "    I just want to remind you..."
    print "that you still owe me ";debt$;". "
    vdu 12
until false
vdu 3
print "I've finished writing the letters."
```

Triplets of data

Suppose you're running a diet clinic and get these results:

Person	Weight before	Weight after
Joe	273 pounds	219 pounds
Mary	412 pounds	371 pounds
Bill	241 pounds	173 pounds
Sam	309 pounds	198 pounds

This program makes the computer print a nice report on your screen:

```
data Joe,273,219,Mary,412,371,Bill,241,173,Sam,309,198
data end,0,0
repeat
    read person$,weight_before,weight_after
    if person$="end" exit repeat
    print person$;" weighed ";weight_before;
    print " pounds before attending the diet clinic"
    print "but weighed just ";weight_after;" pounds afterwards."
    print "That's a loss of ";weight_before-weight_after;" pounds."
    print
until false
print "Come to the diet clinic!"
```

The top line contains the data, which comes in triplets. The first triplet consists of Joe, 273, and 219. Each triplet includes a string (such as Joe) and two numbers (such as 273 and 219), so the second line's end mark also includes a string and two numbers: it's the word "end" and two zeros. (If you hate zeros, you can use other numbers instead; but most programmers prefer zeros.)

The "read" line says to read a triplet: a string (person\$) and two numbers (weight_before and weight_after). The first time the computer comes to the "read" statement, the computer makes person\$ be "Joe", weight_before be 273, and weight_after be 219. The "print" lines print this:

```
Joe weighed 273 pounds before attending the diet clinic
but weighed just 219 pounds afterwards.
That's a loss of 54 pounds.

Mary weighed 412 pounds before attending the diet clinic
but weighed just 371 pounds afterwards.
That's a loss of 41 pounds.

Bill weighed 241 pounds before attending the diet clinic
but weighed just 173 pounds afterwards.
That's a loss of 68 pounds.

Sam weighed 309 pounds before attending the diet clinic
but weighed just 198 pounds afterwards.
That's a loss of 111 pounds.

Come to the diet clinic!
```

Restore

Examine this program:

```
data love,death,war
1 data chocolate,strawberry
read a$
print a$
restore 1
read a$
print a$
```

The first “read” makes the computer read the first datum (love), so the first “print” makes the computer print:

```
love
```

The next “read” would normally make the computer read the next datum (death); but the **“restore 1” tells the “read” to skip ahead to the data line numbered 1**, so the “read” line reads “chocolate” instead. The entire program prints:

```
love
chocolate
```

So saying “restore 1” makes the next “read” skip ahead to the data line numbered 1. If you write a new program, saying “restore 2” makes the next “read” skip ahead to data line numbered 2. Saying just “restore” makes the next “read” skip back to the beginning of the *first* data line.

Continents This program prints the names of the continents:

```
data Europe,Asia,Africa,Australia,Antarctica,North America,South America
data end
repeat
  read a$: if a$="end" exit repeat
  print a$
until false
print "Those are the continents."
```

That program makes the computer print this message:

```
Europe
Asia
Africa
Australia
Antarctica
North America
South America
Those are the continents.
```

Let’s make the computer print that message *twice*, so the computer prints:

```
Europe
Asia
Africa
Australia
Antarctica
North America
South America
Those are the continents.
```

```
Europe
Asia
Africa
Australia
Antarctica
North America
South Ameruca
Those are the continents.
```

To do that, put the program in a loop saying “for i=1 to 2”, like this:

```
data Europe,Asia,Africa,Australia,Antarctica,North America,South America
data end
for i=1 to 2
  repeat
    read a$: if a$="end" exit repeat
    print a$
  until false
  print "Those are the continents."
  print
  restore
NEXT
```

After that program says to print “Those are the continents”, the program says to print a blank line and then restore. The word “restore” makes the “read” go back to the data’s beginning, so the computer can read and print the data a second time without saying “Out of data”.

Search loop

Let's make the computer translate colors into French. For example, if the human says "red", we'll make the computer say the French equivalent, which is:

```
rouge
```

Let's make the computer begin by asking "Which color interests you?", then wait for the human to type a color (such as "red"), then reply:

```
In French, it's rouge
```

The program begins simply:

```
input "which color interests you";request$
```

Next, we must make the computer translate the requested color into French. To do so, feed the computer this English-French dictionary:

English French

white	blanc
yellow	jaune
orange	orange
red	rouge
green	vert
blue	bleu
brown	brun
black	noir

That dictionary becomes the data:

```
data white,blanc,yellow,jaune,orange,orange,red,rouge
data green,vert,blue,bleu,brown,brun,black,noir
input "which color interests you";request$
```

The data comes in pairs; each pair consists of an English word (such as "white") followed by its French equivalent ("blanc"). To make the computer read a pair, say:

```
read english$,french$
```

To let the computer look at *all* the pairs, put that "read" statement in a "repeat" loop. Here's the complete program:

```
data white,blanc,yellow,jaune,orange,orange,red,rouge
data green,vert,blue,bleu,brown,brun,black,noir
input "which color interests you";request$
repeat
  read english$,french$
until english$=request$
print "In French, it's ";french$
```

Since the "read" line is in a "repeat" loop, the computer does the "read" line repeatedly. So the computer keeps reading pairs of data, until the computer find the pair of data the human requested. For example, if the human requested "red", the computer keeps reading pairs of data until it finds a pair whose English word matches the requested word ("red"). When the computer finds that match, the english\$ is equal to the request\$, so the "until" line says to stop repeating. The computer proceeds to the next line, which makes the computer print:

```
In French, it's rouge
```

So altogether, when you run the program the chat can look like this:

```
which color interests you? red
In French, it's rouge
```

Here's another sample run:

```
which color interests you? brown
In French, it's brun
```

Here's another:

```
which color interests you? pink
```

```
Out of data
```

The computer says "Out of data" because it can't find "pink" in the data.

Avoid "Out of data" Instead of saying "Out of data", let's make the computer say "I wasn't taught that color". To do that, put an end mark at the data's end; and when the computer reaches the end mark, make the computer say "I wasn't taught that color":

```
data white,blanc,yellow,jaune,orange,orange,red,rouge
data green,vert,blue,bleu,brown,brun,black,noir
data end,end
input "which color interests you";request$
repeat
  read english$,french$
  if english$="end" print "I wasn't taught that color": end
until english$=request$
print "In French, it's ";french$
```

In that program, the "repeat" loop's purpose is to *search* through the data, to find data that matches the input. Since the "repeat" loop's purpose is to search, it's called a **search loop**.

The typical search loop has these characteristics:

It starts with "**repeat**" and ends with "**until**" and a **match condition**. It says to **read a pair** of data.

It includes an **error trap** saying what to do if you reach the data's **end** because no match found.

Below the loop, say what to **print** when the match is found.

Above the loop, put the **data** and tell the human to **input** a search request.

Auto rerun At the end of the program, let's make the computer automatically rerun the program and translate another color.

To do that, make the program's bottom say to go back to the "input" line:

```
data white,blanc,yellow,jaune,orange,orange,red,rouge
data green,vert,blue,bleu,brown,brun,black,noir
data end,end
1 input "which color interests you";request$
  restore
  repeat
    read english$,french$
    if english$="end" print "I wasn't taught that color": goto 1
  until english$=request$
  print "In French, it's ";french$
  goto 1
```

The word "restore", which is above the search loop, makes sure the computer's search through the data always starts at the data's beginning.

Press Q to quit That program repeatedly asks "Which color interests you" until the human aborts the program. But what if the human's a beginner who hasn't learned how to abort?

Let the human stop the program more easily by pressing just the Q key to quit:

```
data white,blanc,yellow,jaune,orange,orange,red,rouge
data green,vert,blue,bleu,brown,brun,black,noir
data end,end
1 input "which color interests you (press q to quit)";request$
  if request$="q" quit
  restore
  repeat
    read english$,french$
    if english$="end" print "I wasn't taught that color": goto 1
  until english$=request$
  print "In French, it's ";french$
  goto 1
```

Helpful hints

Here are some hints to help you master programming.

Constants & beyond

A **numeric constant** is a simple number, such as:

```
0      1      2      8      43.7      -524.6      .003
```

Another example of a numeric constant is 1.3E5, which means, “take 1.3, and move its decimal point 5 places to the right”.

A numeric constant does not include any arithmetic. For example, since 7+1 includes arithmetic (+), it's *not* a numeric constant. 8 is a numeric constant, even though 7+1 isn't.

A **string constant** is a simple string, in quotation marks:

```
"I love you"      "76 trombones"      "Go away!!!"      "xypw exr///746"
```

A **constant** is a numeric constant or a string constant:

```
0      8      -524.6      1.3E5      "I love you"      "xypw exr///746"
```

Variables A **variable** is something that stands for something else. If it stands for a string, it's called a **string variable** and ends with a dollar sign, like this:

```
a$      b$      y$      z$      my_job_before_promotion$
```

If the variable stands for a number, it's called a **numeric variable** and lacks a dollar sign, like this:

```
a      b      y      z      profit_before_promotion
```

So all these are variables:

```
a$ b$ y$ z$ my_job_before.promotion$ a b y z profit_before_promotion
```

Expressions A **numeric expression** is a numeric constant (such as 8) or a numeric variable (such as b) or a combination of them, such as 8+z, or 8*a, or z*a, or 8*2, or 7+1, or even z*a-(7+z)/8+1.3E5*(-524.6+b). A **string expression** is a string constant (such as “I love you”) or a string variable (such as a\$) or a combination (such as “hot”+“dog”).

An **expression** is a numeric expression or a string expression.

Statements At the end of a “goto” statement, the line number must be a numeric constant or numeric variable.

```
Right: goto 100      (100 is a numeric constant.)
Right: goto n      (n is a numeric variable.)
Wrong: goto 2+3      (2+3 is not a numeric constant.)
```

The “input” statement's prompt must be a string constant.

```
Right: input "what is your name;n$ ("What is your name" is a constant.)
Wrong: input q$;n$      (q$ is not a constant.)
```

In a data statement, you must have constants or numeric expressions.

```
Right: data 8,"Joe",2+4      (8 and "Joe" are constants. 2+4 is a numeric expression.)
Wrong: data "hot"+"dog"      ("hot"+"dog" is not a constant or numeric expression.)
```

In the data statement, if the constant is a string, you can omit the quotation marks (unless the string contains a comma).

```
Right: DATA "Joe", "Mary"
Also right: DATA Joe, Mary
```

Debugging

If you write and run your own program, it probably won't work.

Your first reaction will be to blame the computer. Don't!

The probability is 99.99% that the fault is yours. Your program contains an error. An error is called a **bug**. Your next task is to **debug** the program, which means get the bugs out.

Bugs are common; top-notch programmers make errors all the time. If you write a program that works perfectly on the first run and doesn't need debugging, it's called a **gold-star program** and means you should have tried writing a harder one instead!

It's easy to write a program that's nearly correct but hard to find the little bug fouling it up. Most time you spend at the computer will be devoted to debugging.

Debugging can be fun. Hunting for the bug is like going on a treasure hunt – or solving a murder mystery. Pretend you're Sherlock Holmes. Your mission: to find the bug and squish it! When you squish it, have fun: yell out, “Squish!”

How can you tell when a roomful of programmers is happy? Answer: when you hear continual cries of “Squish!”

To find a bug, use three techniques:

```
Inspect the program.
Trace the computer's thinking.
Shorten the program.
```

Here are the details...

Inspect the program Take a good, hard look at the program. If you stare hard enough, maybe you'll see the bug.

Usually, the bug will turn out to be just a typing error, a **typo**. For example...

Maybe you typed the letter O instead of zero? Zero instead of the letter O?

Typed I instead of 1? Typed 1 instead of I?

Pressed the Shift key when you weren't supposed to? Forgot to press it?

Typed an extra letter? Omitted a letter?

Typed a line you thought you hadn't? Omitted a line?

You must **put quotation marks around each string, and a dollar sign after each string variable**:

```
Right: a$="jerk"
Wrong: a$=jerk
Wrong: a="jerk"
```

Here are 2 reasons why the computer might **print too much**:

1. You forgot to insert the word “end” or “exit repeat” into your program.

2. Into a “repeat” loop or “for” loop, you inserted a “print” line that should be *outside* the loop.

Trace the computer's thinking

If you've inspected the program thoroughly and *still* haven't found the bug, the next step is to **trace** the computer's thinking. **Pretend you're the computer. Do what your program says.** Do you find yourself printing the same wrong answers the computer printed? If so, why? To help your analysis, **make the computer print everything it's thinking** while it's running your program. For example, suppose your program uses the variables b, c, and x\$. Insert lines such as these into your program:

```
10 print "I'm at line 10. Values are ";b;" ";c;" ";x$
20 print "I'm at line 20. Values are ";b;" ";c;" ";x$
```

Then run the program. Those extra lines tell you what the computer is thinking about b, c, and x\$ and also tell you how many times the computer reached lines 10 and 20. For example, if the computer prints what you expect in line 10 but prints strange values in line 20 (or doesn't even get to line 20), you know the bug occurs after line 10 but before line 20.

Here's a good strategy. Halfway down your program, insert a line that says to print all the values. Then run your program. If the line you inserted prints the correct values, you know the bug lies underneath that line; but if the line prints *wrong* values (or if the computer never reaches that line), you know the bug lies *above* that line. In either case, you know which half of your program contains the bug. In that half of the program, insert more lines, until you finally zero in on the line containing the bug.

Shorten the program When all else fails, shorten the program.

Hunting for a bug in a program is like hunting for a needle in a haystack: the job is easier if the haystack is smaller. So make your program shorter: delete the last half of your program. Then run the shortened version. That way, you'll find out whether the first half of your program is working the way it's supposed to. When you've perfected the first half of your program, tack the second half back on.

Does your program contain **a statement whose meaning you're not completely sure of**? Check the meaning by reading a book or asking a friend; or **write a tiny experimental program that contains the statement**, and see what happens when you run it.

Hint: before you shorten your program (or write tiny experimental ones), **save the original version** (by tapping the Save icon), even though the version contains a bug. After you've played with the shorter versions, retrieve the original (by tapping the Load icon) and fix it.

To write a long, correct program easily, write a short program first and debug it, then add a few more lines and debug them, add a few more lines and debug them, etc. So start with a small program, perfect it, then gradually add perfected extras so you *gradually* build a perfected masterpiece. If you try to compose a long program all at once – instead of building it from perfected pieces – you'll have nothing more than a *mastermess* – full of bugs.

Moral: to build a large masterpiece, start with a *small* masterpiece. To build a program so big that it's a skyscraper, begin by laying a good foundation; double-check the foundation before you start adding the program's walls and roof.

Error messages

If the computer can't obey your command, the computer will print an **error message**. The following error messages are the most common...

If you say "prind" instead of "print", the computer will say:

Mistake

That means the computer hasn't the faintest idea of what you're talking about!

If the computer says you have a "Mistake", it's usually because you spelled a word wrong, or forgot a word, or used a word the computer doesn't understand. It can also result from wrong punctuation: check your commas, semicolons, and colons.

If you type a left parenthesis but forget to type the right parenthesis that matches it, the computer will say:

Missing)

If you try to say "print 5+2" but forget to type the 2, the computer will say:

Syntax error

Numeric errors If the answer to a calculation is a bigger number than the computer can handle, the computer will say:

Number too big

If you try to divide by zero, the computer will say:

Division by zero

Logic errors If you say "goto 10", the computer tries to find a line numbered 10. If there's no line numbered 10, the computer will say:

No such line

If a line says to use x (such as "print x" or "print 3+x" "y=3+x" or "goto x") but you haven't said how much x is yet, the computer will say:

No such variable

If you say "read" but the computer can't find any more data to read (because the computer has read all the data already), the computer will say:

Out of data

The computer handles 2 major **types** of info: numbers & strings. If you feed the computer the wrong type of info – if you feed it a number when you should have fed it a string, or you feed it a string when you should have fed it a number – the computer will say:

Type mismatch

When you feed the computer a string, you must put the string in quotation marks, and put a dollar sign after the string's variable. If you forget to type the string's quotation marks or dollar sign, the computer won't realize it's a string; the computer will think you're trying to type a number instead; and if a number would be inappropriate, the computer will give that gripe. So when the computer gives that gripe, it usually means you forgot a quotation mark or a dollar sign.

Pause

Magicians often say, "The hand is quicker than the eye." The computer's the ultimate magician: the computer can print info on the output window much faster than you can read it.

When the computer is printing faster than you can read, make it pause. Here's how...

Tap the program window (which is partly hidden behind the output window). Then tap the **Pause icon** (which looks like "||").

The computer will pause, to let you read what's on the screen.

Tap the output window. Read what's in it.

When you've finished reading what's in the output window and want the computer to stop pausing, tap the Pause icon again. Then the computer will continue printing rapidly, where it left off.

If your eyes are as slow as mine, you'll need to use the Pause icon often! You'll want the computer to pause while you're running a program containing many "print" statements (or a "print" statement in a loop).

Apostrophe

Occasionally, jot a note to remind yourself what your program does and what the variables stand for. Slip the note into your program by putting an apostrophe before it:

```
'This program is a dumb example, written by Russ.  
'It was written on Halloween, under a full moon.  
c=40 'because Russ has 40 computers  
h=23 'because 23 of his computers are haunted  
print "Russ has ";c-h;" unhaunted computers"
```

When you run that program, **the computer ignores everything that's to the right of an apostrophe**. So the computer ignores lines 1 & 2; in lines 3 & 4, the computer ignores the "because...". Since c is 40, and h is 23, the bottom line makes the computer print:

```
Russ has 17 unhaunted computers
```

Everything to the right of an apostrophe is called a **comment** (or **remark**). While the computer runs the program, it ignores the comments. But the comments remain part of the program; they appear in the program window the rest of the program. Though the comments appear in the program, they don't affect the run.

When the computer sees an apostrophe, it thinks the apostrophe marks the beginning of a comment (unless the apostrophe is in a string or "data" statement or "print" statement).

Loop techniques

Here's a strange program:

```
x=9  
x=4+x  
print x
```

The second line ($x=4+x$) means: the new x is 4 plus the old x. So the new x is $4+9$, which is 13. The bottom line prints:

```
13
```

Let's look at that program more closely. The top line ($x=9$) puts 9 into box x:

```
box x 
```

When the computer sees the next line ($x=4+x$), it examines the equation's right side and sees the $4+x$. Since x is 9, the $4+x$ is $4+9$, which is 13. So the line " $x=4+x$ " means $x=13$. The computer puts 13 into box x:

```
box x 
```

The program's bottom line prints 13.

Here's another weirdo:

```
b=6  
b=b+1  
print b*2
```

The second line ($b=b+1$) says the new b is "the old b plus 1". So the new b is $6+1$, which is 7. The bottom line prints:

```
14
```

In that program, the top line says b is 6; but the next line increases b, by adding 1 to b; so b becomes 7. Programmers say that b has been **increased** or **incremented**. In the third line, the "1" is called the **increase** or the **increment**.

The opposite of "increment" is **decrement**:

```
j=500  
j=j-1  
print j
```

The top line says j starts at 500; but the next line says the new j is "the old j minus 1", so the new j is $500-1$, which is 499. The bottom line prints:

```
499
```

In that program, j was **decreased** (or **decremented**). In the third line, the "1" is called the **decrease** (or **decrement**).

Counting Suppose you want the computer to count, starting at 3, like this:

```
3  
4  
5  
6  
7  
8  
etc.
```

This program does it, by a special technique:

```
c=3  
repeat  
  print c  
  c=c+1  
until false
```

In that program, c is called the **counter**, because it helps the computer count.

The top line says c starts at 3. The "print" line makes the computer print c, so the computer prints:

```
3
```

The next line ($c=c+1$) increases c by adding 1 to it, so c becomes 4. The "until" line sends the computer back to the "print" line, which prints the new value of c:

```
4
```

Then the computer comes to the " $c=c+1$ " again, which increases c again, so c becomes 5. The "until" line sends the computer back again to the "print" line, which prints:

```
5
```

The program's an infinite loop: the computer will print 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on, forever, unless you abort it.

Here's the general procedure to make the computer count:

Start c at some value (such as 3).

Then write a "repeat" loop.

In the "repeat" loop, make the computer use c (such as by saying "print c") and increase c (by saying " $c=c+1$ ").

This program makes the computer count, starting at 1:

```
c=1  
repeat  
  print c  
  c=c+1  
until false
```

The computer will print 1, 2, 3, 4, etc.

This program makes the computer count, starting at 0:

```
c=0  
repeat  
  print c  
  c=c+1  
until false
```

The computer will print 0, 1, 2, 3, 4, etc.

Quiz Let's make the computer give this quiz:

What's the capital of Nevada?
What's the chemical symbol for iron?
What word means 'brother or sister'?
What was Beethoven's first name?
How many cups are in a quart?

To make the computer score the quiz, we must tell it the correct answers:

Question	Correct answer
What's the capital of Nevada?	Carson City
What's the chemical symbol for iron?	Fe
What word means 'brother or sister'?	sibling
What was Beethoven's first name?	Ludwig
How many cups are in a quart?	4

So feed the computer this data:

```
data what's the capital of Nevada,Carson City
data what's the chemical symbol for iron,Fe
data what word means 'brother or sister',sibling
data what was Beethoven's first name,Ludwig
data How many cups are in a quart,4
```

In the data, each pair consists of a question and an answer. To make the computer read the data, tell the computer to read a question and an answer, repeatedly:

```
repeat
  read question$,answer$
until false
```

Here's the complete program:

```
data what's the capital of Nevada,Carson City
data what's the chemical symbol for iron,Fe
data what word means 'brother or sister',sibling
data what was Beethoven's first name,Ludwig
data How many cups are in a quart,4
data end,end
repeat
  read question$,answer$: if question$="end" exit repeat
  print question$;
  input response$
  if response$=answer$ then
    print "Correct!"
  else
    print "No, the answer is: ";answer$
  endif
until false
print "I hope you enjoyed the quiz!"
```

The lines underneath "read" make the computer print the question, wait for the human to input a response, and check if the human's response matches the correct answer. Then the computer will either print "Correct!" or print "No" and reveal the correct answer. When the computer reaches the data's end, the computer exits the "repeat: loop and prints "I hope you enjoyed the quiz!"

Here's a sample run, where I've underlined the parts typed by the human:

```
What's the capital of Nevada? Las Vegas
No, the answer is: Carson City
What's the chemical symbol for iron??? Fe
Correct!
What word means 'brother or sister'? I give up
No, the answer is: sibling
What was Beethoven's first name? Ludvig
No, the answer is: Ludwig
How many cups are in a quart? 4
Correct!
I hope you enjoyed the quiz!
```

To give a quiz about different topics, change the data.

Let's make the computer count how many questions the human answered correctly. To do that, we need a counter.

As usual, let's call it c:

```
data what's the capital of Nevada,Carson City
data what's the chemical symbol for iron,Fe
data what word means 'brother or sister',sibling
data what was Beethoven's first name,Ludwig
data How many cups are in a quart,4
data end,end
c=0
repeat
  read question$,answer$: if question$="end" exit repeat
  print question$;
  input response$
  if response$=answer$ then
    print "Correct!"
    c=c+1
  else
    print "No, the answer is: ";answer$
  endif
until false
print "I hope you enjoyed the quiz!"
print "You answered ";c;" of the questions correctly."
```

At the program's beginning, the human hasn't answered any questions correctly yet, so the counter begins at 0 (by saying "c=0"). Each time the human answers a question correctly, the computer does "c=c+1", which increases the counter. The program's bottom line prints the counter, by printing a message such as:

```
You answered 2 of the questions correctly.
```

It would be nicer to print —

```
You answered 2 of the 5 questions correctly.
Your score is 40 %
```

or, if the quiz were changed to include 8 questions:

```
You answered 2 of the 8 questions correctly.
Your score is 25 %
```

To make the computer print such a message, we must make the computer count how many questions were asked. So we need another counter. Since we already used c to count the number of correct answers, let's use q to count the number of questions asked. Like c, q must start at 0; and we must increase q, by adding 1 each time another question is asked:

```
data what's the capital of Nevada,Carson City
data what's the chemical symbol for iron,Fe
data what word means 'brother or sister',sibling
data what was Beethoven's first name,Ludwig
data How many cups are in a quart,4
data end,end
q=0
c=0
repeat
  read question$,answer$: if question$="end" exit repeat
  print question$;
  q=q+1
  input response$
  if response$=answer$ then
    print "Correct!"
    c=c+1
  else
    print "No, the answer is: ";answer$
  endif
until false
print "I hope you enjoyed the quiz!"
print "You answered ";c;" of the ";q;" questions correctly."
print "Your score is ";c/q*100;"%"
```

Summing Let's make the computer imitate an adding machine, so a run looks like this:

```
Now the sum is 0
What number do you want to add to the sum? 5
Now the sum is 5
What number do you want to add to the sum? 3
Now the sum is 8
What number do you want to add to the sum? 6.1
Now the sum is 14.1
What number do you want to add to the sum? -10
Now the sum is 4.1
etc.
```

Here's the program:

```
s=0
repeat
  print "Now the sum is";s
  input "What number do you want to add to the sum";x
  s=s+x
until false
```

The top line starts the sum at 0. The "print" line prints the sum. The "input" line asks the human what number to add to the sum; the human's number is called x. The next line ($s=s+x$) adds x to the sum, so the sum changes. The "until" line sends the computer back to the "print" line, which prints the new sum. The program's an infinite loop, which you must abort.

Here's the general procedure to make the computer find a sum:

Start s at 0.

Then write a "repeat" loop.

In the "repeat" loop, make the computer use s (such as by saying "print s") and increase s (by saying $s=s+\text{the number to be added}$).

Checking account If your bank's nasty, it charges you 20¢ to process each good check you write, a \$25 penalty for each check that bounces, and pays no interest on money you've deposited.

This program makes the computer imitate such a bank....

```
s=0
repeat
  print "Your checking account contains";s
1  input "Press d (to make a deposit) or c (to write a check)";a$
   case a$ of
     when "d"
       input "How much money do you want to deposit";d
       s=s+d
     when "c"
       input "How much money do you want the check for";c
       c=c+.2
       if c<=s then
         print "Okay"
         s=s-c
       else
         print "That check bounced!"
         s=s-25
       endif
     otherwise
       print "Please press d or c"
       goto 1
   endcase
until false
```

In that program, the total amount of money in the checking account is called the sum, s. The top line ($s=0$) starts that sum at 0. The first "print" line prints the sum. The next line asks the human to press "d" (to make a deposit) or "c" (to write a check).

If the human presses "d" (to make a deposit), the computer asks "How much money do you want to deposit?" and waits for the human to type an amount to deposit. The computer adds that amount to the sum in the account ($s=s+d$).

If the human presses "c" (to write a check), the computer asks "How much money do you want the check for?" and waits for the human to type the amount on the check. The computer adds the 20¢ check-processing fee to that amount ($c=c+.2$). Then the computer reaches the line saying "if $c<=s$ ", which checks whether the sum s in the account is big enough to cover the check (c). If $c<=s$, the computer says "Okay" and processes the check, by subtracting c from the sum in the account. If the check is too big, the computer says "That check bounced!" and decreases the sum in the account by the \$25 penalty.

That program is nasty to customers:

For example, suppose you have \$1 in your account, and you try to write a check for 85¢. Since $85¢ + \text{the } 20¢ \text{ service charge} = \1.05 , which is more than you have in your account, your check will bounce, and you'll be penalized \$25. That makes your balance become *negative* \$24, and the bank will demand you pay the bank \$24 — just because you wrote a check for 85¢!

Another nuisance is when you leave town permanently and want to close your account. If your account contains \$1, you can't get your dollar back! The most you can withdraw is 80¢, because $80¢ + \text{the } 20¢ \text{ service charge} = \1 .

That nasty program makes customers hate the bank — and hate the computer! The bank should make the program friendlier. Here's how:

To stop accusing the customer of owing money, the bank should change any negative sum to 0, by inserting this line just under the word "repeat":

```
if s<0 s=0
```

Also, to be friendly, the bank should ignore the 20¢ service charge when deciding whether a check will clear. So the bank should eliminate the line saying " $c = c+.2$ ". On the other hand, if the check *does* clear, the bank should impose the 20¢ service charge afterwards, by changing the " $s=s-c$ " to " $s = s-c-.2$ ".

So if the bank is kind, it will make all those changes. But some banks complain that those changes are *too* kind! For example, if a customer whose account contains just 1¢ writes a million-dollar check (which bounces), the new program charges him just 1¢ for the bad check; \$25 might be more reasonable.

Moral: **the hardest thing about programming is choosing your goal — deciding what you WANT the computer to do.**

Series Let's make the computer add together all the numbers from 7 to 100, so that the computer finds the sum of this series: $7 + 8 + 9 + \dots + 100$. Here's how.

```
Start the sum at 0:      s=0
Make i go from 7 to 100:  for i=7 TO 100
Increase sum, by adding each i to it:  s=s+i
Print the final sum (which is 5029):  next
                                       print s
```

Let's make the computer add together the *squares* of all the numbers from 7 to 100, so that the computer finds the sum of this series: $(7 \text{ squared}) + (8 \text{ squared}) + (9 \text{ squared}) + \dots + (100 \text{ squared})$. Here's how:

```
s=0
for i=7 to 100
  s=s+i*i
next
print s
```

It's the same as the previous program, except that indented line says to add $i*i$ instead of i. The bottom line prints the final sum, which is 338259.

Data sums This program adds together the numbers in the data:

```
data 5,3,6.1,etc.
data 0
s=0
repeat
  read x: if x=0 exit repeat
  s=s+x
until false
print s
```

The “data” line contains the numbers to be added. The “data 0” is an end mark. The line saying “s=0” starts the sum at 0. The “read” statement reads an x from the data. The next line (s=s+x) adds x to the sum. The “until” line makes the computer repeat that procedure for every x. When the computer has read all the data and reaches the end mark (0), the x becomes 0, so the computer will exit the “repeat” loop and print the final sum, s.

Pretty output

Here’s how to make your output prettier.

Zones

The output window is divided into zones. **Each zone is 10 characters wide. A comma makes the computer hop to the next zone:**

```
print "sin","king"
```

That makes the computer print "sin" then hop to the next zone, where it prints “king”, like this:

```
sin      king
```

Here are the words of a poet who drank too much and is feeling spaced out:

```
print "love","cries","out"
```

The computer will print “love” in the first zone, “cries” in the second zone, and “out” in the third zone, so the words are spaced out like this:

```
love     cries     out
```

This program tells a bad joke:

```
print "You're ugly!","I'm joking!"
```

The computer will print “You’re ugly!” That string has 12 characters. Its first 10 characters are in the first zone; its remaining 2 characters are in the second zone. Then the comma makes the computer hop to the next zone (the third zone), where the computer prints “I’m joking!”, like this:

```
You're ugly!      I'm joking!
```

When you combine commas with semicolons, you can get weird results:

```
print "eat","me";"at";"ball","no";"w"
```

That line contains commas and semicolons. A comma makes the computer hop to a new zone, but a semicolon does *not* make the computer hop. The computer will print “eat”, then hop to the second zone, then print “me” and “at” and “ball”, then hop to the third zone, then print “no” and “w”. Altogether, the computer will print:

```
eat      meatball now
```

Here’s a strange example:

```
print "underwater","ing"
```

The computer prints “underwater”, which contains 10 characters, so it fills up the first zone. Then the computer is supposed to print “ing” at the beginning of a new zone, but the computer’s *already* at the beginning of the second zone, so the computer prints:

```
Underwatering
```

Skip a zone You can make the computer skip over a zone:

```
print "Joe"," ", "loves Sue"
```

The computer will print “Joe” in the first zone, a blank space in the second zone, and “loves Sue” in the third zone, like this:

```
Joe              loves Sue
```

Numbers Compare these 4 “print” statements:

```
print 53
print 53;986
print 53;" ";986
print 53,986
```

The first “print” statement makes the computer print the number 53. **The computer wants to make every number be 10 characters wide**, so it puts 8 spaces before the digits, like this:

```
53
```

In the second “print” statement, **the semicolon prevents the computer from putting blank spaces** before the 986, so the computer prints:

```
53986
```

In the third “print” statement, **the " " forces the computer to put a blank space** before the 986, so the computer prints:

```
53 986
```

In the fourth “print” statement, **the comma makes the computer go to the beginning of a new zone**; but after printing the 53, the computer’s *already* at the beginning of the second zone, so the computer prints 986 without hopping to another zone. Since there’s no semicolon, the computer prints 986 as 10 characters, by putting 7 spaces before it, like this:

```
57      986
```

Tables This program prints a list of words and their opposites:

```
print "good","bad"
print "black","white"
print "parent","child"
print "he","she"
```

The top line makes the computer print “good”, then hop to the next zone, then print “bad”. Altogether, the computer will print:

```
good      bad
black     white
parent    child
he        she
```

The first zone contains a column of words; the second zone contains the opposites. Altogether, the computer’s printing looks like a table. So **to make a table easily, use zones, by putting commas in your program.**

Let's make the computer print this table:

Number	Square
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Here's the program:

```
print "      Number   Square"
for i=3 to 10
  print i,i*i
next
```

In the "for...next" loop, the "print" line makes the computer print each number in the first zone and its square in the second zone. (The computer puts enough spaces before the digits to make each number consume 10 characters.) The top line makes the computer print the headings ("Number" and "Square") with enough spaces so the headings are centered over the numbers below.

Tab

When the computer puts a line of information on your screen, the leftmost character in the line is said to be at **position 0**. The next character in the line is said to be at **position 1**. Then comes **position 2**.

This program makes the computer skip to position 6 and then print "hot":

```
print tab(6);"hot"
```

The computer will print:

```
hot
```

Here's a fancier example:

```
print tab(6);"hot";tab(13);"buns"
```

The computer will skip to position 6, then print "hot", then skip to position 13, then print "buns":

```
hot    buns
```

Diagonal This program prints a diagonal line:

```
for i=0 to 8
  print tab(i);"*"
next
```

The "for" line says to do the indented line many times. The first time the computer does the indented line, the i is 0, so the computer prints an asterisk at position 0:

```
*
```

The next time, the i is 1, so the computer skips to position 1 and prints an asterisk:

```
 *
```

The next time, the i is 2, so the computer skips to position 2 and prints an asterisk:

```
  *
```

Altogether, the program makes the computer print this picture:

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

2-dimensional tab In the output window, the top line is called **line 0**; underneath it is **line 1**; then comes **line 2**; etc.

To make the computer print the word "drown" so that "drown" begins at position 6 of line 2, type this:

```
print tab(6,2);"drown"
```

The computer will print the word's first letter (d) at the 6th position of line 2. The computer will print the rest of the word afterwards, still on line 2.

If words were already printed on the output screen there, "drown" will cover them up. Covering up can be done by a 2-dimensional tab, such as tab(6,2), but not by a plain tab, such as tab(6), which refuses to erase previous typing.

Pixels

The image on screen is called the **picture**. If you stare at the picture closely, you'll see the picture's composed of thousands of tiny dots. Each dot, which is a tiny rectangle, is called a **picture's element**, or **pic's el**, or **pixel**, or **pel**.

Coordinates The dot in the output window's bottom-left corner is called **pixel (0,0)**. Just to the right of it is pixel (1,0). Then comes pixel (2,0), etc.

Above pixel (0,0) is pixel (0,1). Farther up is pixel (0,2).

Each pixel's name consists of two numbers in parentheses. The first number is the **X coordinate**; the second number is the **Y coordinate**. For example, if you're talking about pixel (4,3), its X coordinate is 4; its Y coordinate is 3.

The X coordinate tells how far to the right the pixel is. The Y coordinate tells how far up. So **pixel (4,3) is the pixel that's 4 to the right and 3 up**.

Circle This command draws a **circle** whose center is pixel (200,300) and radius is 50:

```
circle 200,300,50
```

Ellipse This command draws an ellipse (oval) whose center is pixel (200,300), horizontal radius is 50, and vertical radius is 25:

```
ellipse 200,300,50,25
```

Rectangle This command draws a **rectangle** (box) whose bottom-left corner is pixel (200,300), width is 80, and height is 50:

```
rectangle 200,300,80,50
```

To draw a **square**, give the rectangle command but don't mention the rectangle's height:

```
rectangle 200,300,80
```

The computer will assume you want the height to be the same as the width.

Fill If you say "**circle fill**" instead of "circle" or say "**ellipse fill**" instead of "ellipse" or say "**rectangle fill**" instead of "rectangle", the computer will draw the shape but also fill in the shape's middle (the shape's interior).

For example, this command draws a circle whose center is pixel (200,300) and radius is 50 and will also fill the circle's middle:

```
circle fill 200,300,50
```

Line This command draws a diagonal **line** from pixel (200,300) to pixel (500,350):

```
line 200,300,500,350
```

This command draws a diagonal line from the previous line's end to pixel (600,400):

```
draw 600,400
```

Plot This command puts a tiny dot at pixel (100,200):

```
plot 100,200
```

Colors

Normally, the computer prints & graphs everything in dark black, but you can switch to a different color. You can use 16 colors:

0. dark black (pure black)	8. bright black (dark gray)
1. dark red	9. bright red
2. dark green	10. bright green
3. dark yellow (gold)	11. bright yellow (pure yellow)
4. dark blue	12. bright blue
5. dark magenta (dark purplish red)	13. bright magenta (bright purplish red)
6. dark cyan (dark greenish blue)	14. bright cyan (bright greenish blue)
7. dark white (light gray)	15. bright white (pure white)

This program prints "I'm sad" in dark blue (which is color #4):

```
colour 4
print "I'm sad"
```

Since BBC Basic was invented by the British, it wants you to write "colour", not "color" (which is American). If you accidentally write "color", BBC Basic will automatically change it to "colour" when you press the Enter key.

Once you pick a color, the computer will keep printing in that color until you switch to a different color (or close the output window).

This program prints "I love you" in bright red (which is color #9):

```
colour 9
print "I love you"
```

Background color Normally, the computer prints on a pure white background, but you can change the background color by adding 128. This program prints "I love you" in bright red on a bright yellow background:

```
colour 9: colour 11+128
print "I love you"
```

Graphics colors The "colour" command alters the colors of text, not graphics. To alter the colors of *graphics* (pixel pictures), say "gcol" instead of "colour". To change the graphics' background color, you must also say "clg" (which means "clear the graphics screen"). This program creates a bright red circle on a bright yellow background:

```
gcol 9: gcol 11+128: clg
circle 200,300,50
```

Sounds

This program produces a normal sound:

```
sound 1, -15, 4*25, 20*3
```

The sound is loud, middle C (on the piano), and lasts for 3 seconds. Try it: run that program!

Here's what those shaded numbers mean...

The "15" is how loud the sound will be.

If you switch the 15 to 1, the sound will be much softer. You can pick any integer from 0 (which means "silent") to 15 (which means "as loud as possible"). You must put a minus sign before your number.

The "3" is how long the sound will last: how many seconds.

If you want the sound to last longer than 3 seconds, switch to a number bigger than 3.

For example, if you want the sound to last for 4 seconds, switch the 3 to a 4. If you want the sound to last for 1 second, switch the 3 to a 1.

If you want the sound to last for half a second, switch the 3 to .5. (The 20 is because BBC Basic measures the sound length in 20ths of a second.)

The "25" is the sound's pitch (how high the note will be).

25 is the piano's middle C.
26 is the piano's next note (C sharp).
27 is the piano's note after that (D).
24 is the piano's note below middle C; it's B.
23 is the piano's note below that (B flat).
You can pick any number between .25 and 63.75.
(The 4 is because BBC Basic measures the pitch in quarters of a semitone.)

The "1" is the sound's voice number. BBC Basic can produce 3-part harmony. This program makes the first voice sing middle C (pitch #25) while the second voice sings E (pitch #29) and the third voice sings G (pitch # 32), to produce a chord:

```
sound 1, -15, 4*25, 20*3
sound 2, -15, 4*29, 20*3
sound 3, -15, 4*32, 20*3
```

Decimal places

Suppose you want to add \$12.47 to \$1.03. The correct answer is \$13.50. This almost works:

```
print 12.47+1.03
```

It makes the computer print:

```
13.5
```

But instead of 13.5, let's make the computer print 13.50.

This **Print Format** command makes the computer print 2 digits after each decimal point —

```
@%=131082+256*2
```

so this program makes the computer print 13.50:

```
@%=131082+256*2
print 12.47+1.03
```

The computer will print:

```
13.50
```

Here's a contradiction:

```
@%=131082+256*2
print 86.239
```

The bottom line says to print 86.239, but the top line says to print just 2 digits after the decimal point, so the computer will round 86.239 and print:

```
86.24
```

In the Print Format command, the last digit (the 2) says how many digits to print after the decimal point. If you want 3 digits after the decimal point instead just 2, change the 2 to 3, like this:

```
@%=131082+256*3
```

The Print Format command affects all future printing, until you close the output window or give a different Print Format command or give this **Cancel Print Format** command —

```
@%=2314
```

which returns the printing to normal.

Fancy calculations

You can do fancy calculations — easily!

Pi

A circle's circumference divided by its diameter is called "pi" (which Americans pronounce the same as the word "pie"). It's approximately 3.14159265. This program makes the computer print 3.14159265:

```
print pi
```

Exponents

Try typing this program:

```
print 4^3
```

To type the symbol ^, do this: while holding down the Shift key, tap this key:

```
^
6
```

That symbol (^) is called a **caret**.

In that program, **the "4^3" makes the computer use the number 4, three times**. The computer will multiply together those three 4's, like this: 4 times 4 times 4. Since "4 times 4 times 4" is 64, the computer will print:

```
64
```

In the expression "4^3", the 4 is called the **base**; the 3 is called the **exponent**.

Here's another example:

```
print 10^6
```

The "10^6" makes the computer use the number 10, six times. The computer will multiply together those six 10's (like this: 10 times 10 times 10 times 10 times 10 times 10) and print the answer:

```
1000000
```

Here's another example:

```
PRINT 3^2
```

The "3^2" makes the computer use the number 3, two times. The computer will multiply together those two 3's (like this: 3 times 3) and print the answer:

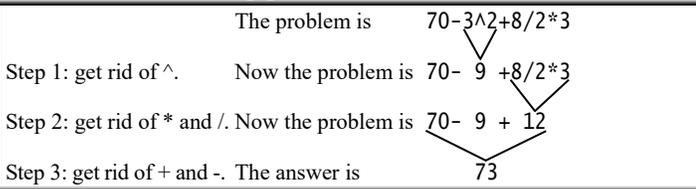
```
9
```

Order of operations The symbols +, -, *, /, and ^ are all called **operations**.

To solve a problem, the computer uses the three-step process taught in algebra and the "new math". For example, suppose you say:

```
print 70-3^2+8/2*3
```

The computer will *not* begin by subtracting 3 from 70; instead, it will use the three-step process:



In each step, it looks from left to right. For example, in step 2, it sees / and gets rid of it before it sees *.

Speed Though exponents are fun, the computer handles them slowly. For example, the computer handles 3^2 slower than 3*3. So for fast calculations, say 3*3 instead of 3^2.

Square roots What positive number, when multiplied by itself, gives 9? The answer is 3, because 3 times itself is 9.

3 **squared** is 9. 3 is called the **square root** of 9.

To make the computer deduce the square root of 9, type this:

```
print sqrt(9)
```

The computer will print 3.

The symbol "sqrt" is called a **function**. The number in parentheses (9) is called the function's **input** (or **argument** or **parameter**). The answer, which is 3, is called the function's **output** (or **value**).

The sqrt(9) gives the same answer as 9^.5. The computer handles sqrt(9) faster than 9^.5.

BBC Basic lets you omit the parentheses: instead of typing "sqrt(9)", you can type just "sqrt9" or "sqrt 9", and the computer will still print the correct answer (which is 3).

Cube roots What number, when multiplied by itself and then multiplied by itself *again*, gives 64? The answer is 4, because 4 times 4 times 4 is 64. The answer (4) is called the **cube root** of 64.

Here's how to make the computer find the cube root of 64:

```
print 64^(1/3)
```

The computer will print 4.

Exp The letter "e" stands for a special number, which is approximately 2.718281828459045. You can memorize that number easily, if you pair the digits:

```
2.7 18 28 18 28 45 90 45
```

That weird number is important in calculus, radioactivity, biological growth, and other areas of science. It's calculated by this formula:

$$e = 1 + \frac{1}{1} + \frac{1}{1*2} + \frac{1}{1*2*3} + \frac{1}{1*2*3*4} + \frac{1}{1*2*3*4*5} + \dots$$

Therefore:

$$e = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \dots$$

Exp(x) means e^x. For example, exp(3) means e³, which is e*e*e, which is:

```
2.718281828459045 * 2.718281828459045 * 2.718281828459045
```

Exp(4) means e⁴, which is e*e*e*e. Exp(3.1) means e^{3.1}, which is more than e³ but less than e⁴.

Here's a practical application. Suppose you put \$800 in a savings account, and the bank promises to give you 5% annual interest "compounded continuously". How much money will you have at the end of the year? The answer is 800*exp(.05).

Logarithms Suppose you write 1 with a lot of zeros afterwards, like this:

```
100000
```

The **logarithm** of that number is how many zeros you wrote. So if you say —

```
print log(100000)
```

the computer will count the zeros and print how many there are; the computer will print:

```
5
```

Here are more examples:

```
log(10) is 1
log(100) is 2
log(1000) is 3
log(10000) is 4
log(100000) is 5
```

How much is $\log(7852)$? Since 7852 is between 1000 and 10000, its log is between 3 and 4. If you say —

```
print log(7852)
```

the computer will print a number between 3 and 4; it will print:

```
3.89498029
```

I've been explaining the **common log**, which is used in chemistry. It's also called the **log base 10**. Mathematicians write it as " \log_{10} ". For example, they say $\log_{10} 100000$ is 5.

A different kind of log, used in calculus, is called the **natural log**. It's also called the **log base e**. Mathematicians write it as " \log_e " or "LN" or "ln". It's about 2.3 times the common log. For example, if you say —

```
print ln(100000)
```

the computer will print a number that's about 2.3 times $\log(100000)$, so it will print about "2.3 times 5", which is 11.5. It will print:

```
11.5129255
```

Here are some powers of 2:

x	2 ^x
1	2
2	4
3	8
4	16
5	32
6	64

To compute the logarithm-base-2 of a number, find the number in the right-hand column; the answer's in the left column. For example, the logarithm-base-2 of 32 is 5. The logarithm-base-2 of 15 is slightly less than 4.

The logarithm-base-2 of 64 is 6. That fact is written:

```
log2 64 is 6
```

It's also written:

```
log 64 / log 2 is 6
```

To make the computer find the logarithm-base-2 of 64, say:

```
print log(64)/log(2)
```

The computer will print 6.

Contrasts

The computer's notation resembles that of arithmetic and algebra, but beware of these contrasts....

Multiplication To make the computer multiply, you must type an asterisk:

Traditional notation	Computer notation
2n	2*n
5(n+m)	5*(n+m)
nm	n*m

Exponents Put an exponent in parentheses, if it contains an operation:

Traditional notation	Computer notation
x ⁿ⁺²	x^(n+2)
x ³ⁿ	x^(3*n)
5 ^{2/3}	5^(2/3)
2 ^{3/4}	2^(3/4)

Fractions Put a fraction's numerator in parentheses, if it contains addition or subtraction:

Traditional notation	Computer notation
$\frac{a+b}{c}$	(a+b)/c
$\frac{k-20}{6}$	(k-20)/6

Put a denominator in parentheses, if it contains addition, subtraction, multiplication, or division:

Traditional notation	Computer notation
$\frac{5}{3+x}$	5/(3+x)
$\frac{5a^3}{4b}$	5*a^3/(4*b)

Mixed numbers A **mixed number** is a number that contains a fraction. For example, $9\frac{1}{2}$ is a mixed number. When you write a mixed number, put a plus sign before its fraction:

Traditional notation	Computer notation
$9\frac{1}{2}$	9+1/2

If you're using the mixed number in a further calculation, put the mixed number in parentheses:

Traditional notation	Computer notation
$7 - 2\frac{1}{4}$	7-(2+1/4)

Stripping

Sometimes the computer prints *too* much info: you wish the computer would print less, to save yourself the agony of reading excess info irrelevant to your needs. Whenever the computer prints too much info about a numerical answer, say "abs", "int", or "sgn".

Abs removes any minus sign. For example, the abs of -3.89 is 3.89. So if you say `print abs(-3.89)`, the computer will print just 3.89.

Int rounds the number DOWN to an integer that's LOWER. For example, the int of 3.89 is 3 (because 3 is an integer that's lower than 3.89); the int of -3.89 is -4 (because -4 is lower than -3.89).

Sgn removes ALL the digits and replaces them with a 1 — unless the number is 0. For example, the sgn of 3.89 is 1. The sgn of -3.89 is -1. The sgn of 0 is just 0.

Abs, int, and sgn are called **stripping functions** or **strippers** or **diet functions** or **diet pills**, because they strip away the number's excess fat and reveal just the fundamentals that interest you.

Here are more details about those three functions....

Abs To find the **absolute value** of a negative number, just omit the number's minus sign. For example, the absolute value of -7 is 7.

The absolute value of a positive number is the number itself. For example, the absolute value of 7 is 7. The absolute value of 0 is 0.

To make the computer find the absolute value of -7, type this:

```
print abs(-7)
```

The computer will print:

```
7
```

Like `sqrt`, `abs` is a function: you're supposed to put parentheses after the "abs", though in BBC Basic the parentheses are optional.

Since `abs` omits the minus sign, `abs` turns negative numbers into positive numbers. Use `abs` whenever you insist that an answer be positive.

For example, abs helps solve math & physics problems about "distance", since the "distance" between two points is always a positive number and cannot be negative.

This program computes the distance between two numbers:

```
print "I will find the distance between two numbers."  
input "what's the first number";x  
input "what's the second number";y  
print "The distance between those numbers is";abs(x-y)
```

When you run that program, suppose you say that the first number is 4 and the second number is 7. Since x is 4, and y is 7, the distance between those two numbers is $\text{abs}(4-7)$, which is $\text{abs}(-3)$, which is 3.

If you reverse those two numbers, so x is 7 and y is 4, the distance between them is $\text{abs}(7-4)$, which is $\text{abs}(3)$, which is still 3.

Int turns a number into an **integer** (a number without decimal digits).

Int rounds a number *down* to an integer that's *lower*. For example:

```
The int of 3.9 is 3 (because 3 is an integer that's lower than 3.9).  
The int of -3.9 is -4 (because a temperature of -4 is lower and colder than a temperature of -3.9).  
The int of 7 is simply 7.
```

To explore further the mysteries of rounding, run this program:

```
input "what's your favorite number";x  
print "Your number rounded down is";int(x)  
print "Your number rounded up is";-int(-x)  
print "Your number rounded to the nearest integer is";int(x+.5)
```

The top line asks you to type a number x .

The next line prints your number rounded *down*. For example, if you input 3.9, the computer prints 3.

The next line, `PRINT -INT(-x)`, prints your number rounded *up*. For example if you input 3.9, the computer prints 4.

The bottom line prints your number rounded to the *nearest* integer. For example, if you input 3.9, the computer will print 4.

Here's the rule: if x is a number, **int(x) rounds x down; -int(-x) rounds x up; int(x+.5) rounds x to the nearest integer.**

Rounding down and rounding up are **useful in the supermarket:**

```
Suppose some items are marked "30¢ each", and you have just two dollars. How many can you buy?  
Two dollars divided by 30¢ is 6.66667; rounding down to an integer, you can buy 6.
```

```
Suppose some items are marked "3 for a dollar", and you want to buy just one of them. How much  
will the supermarket charge you? One dollar divided by 3 is 33.3333¢; rounding up to an integer, you  
will be charged 34¢.
```

By using `int`, **you can do fancier kinds of rounding:**

```
to round x to the nearest thousand, ask for int(x/1000+.5)*1000  
to round x to the nearest thousandth, ask for int(x/.001+.5)*.001
```

This program rounds a number, so that it will have just a *few* digits after the decimal point:

```
input "what's your favorite number";x  
input "How many digits would you like after its decimal point";d  
b=10^d  
print "Your number rounded is ";int(x/b+.5)*b
```

Here's a sample run:

```
what's your favorite number? 4.28631  
How many digits would you like after its decimal point? 2  
Your number rounded is 4.29
```

Sgn If a number is negative, its **sign** is -1. For example, the sign of -546 is -1.

If a number is positive, its **sign** is +1. For example the sign of 8231 is +1.

The **sign** of 0 is 0.

The computer's abbreviation for "sign" is "sgn". So if you say —

```
print sgn(-546)
```

the computer will print the sign of -546; it will print -1.

If you say —

```
PRINT sgn(8231)
```

the computer will print the sign of 8231; it will print 1.

If you say —

```
print sgn(0)
```

the computer will print the sign of 0; it will print 0.

Sgn is the opposite of **abs**. Let's see what both functions do to -7.2. **Abs** removes the minus sign, but leaves the digits:

```
abs(-7.2) is 7.2
```

Sgn removes the digits, but leaves the minus sign:

```
sgn(-7.2) is -1
```

The Latin word for *sign* is **signum**. Most mathematicians prefer to talk in Latin — they say "signum" instead of "sign" — because the English word "sign" sounds too much like the trigonometry word "sine". So mathematicians call `sgn` the **signum function**.

Random numbers

Usually, the computer is predictable: it does exactly what you say. But sometimes, you want the computer to be *unpredictable*.

For example, if you're going to play a game of cards with the computer and tell the computer to deal, you want the cards dealt to be unpredictable. If the cards were predictable — if you could figure out exactly which cards you and the computer would be dealt — the game would be boring.

In many other games too, you want the computer to be unpredictable, to "surprise" you. Without an element of surprise, the game would be boring.

Being unpredictable increases the pleasure you derive from games — and from art. To make the computer act artistic, and create a new *original* masterpiece that's a "work of art", you need a way to make the computer get a "flash of inspiration". Flashes of inspiration aren't predictable: they're surprises.

Here's how to make the computer act unpredictably...

Random integers This program makes the computer print an unpredictable number from 1 to 5:

```
print rnd(5)
```

Unpredictable numbers are called **random** numbers. That program makes the computer print a random number from 1 to 5, so the computer will print 1, 2, 3, 4, or 5. The computer's choice will be a surprise.

For example, when you run that program, the computer might print 3. If you run the program a second time, the computer might print a different number (1 or 2 or 4 or 5), or it might print the same number (3). You can't predict which number the computer will print. The only thing you can be sure of is: the number will be from 1 to 5.

To make the computer print *many* such random numbers, say “goto”:

```
1 print rnd(5)
  goto 1
```

The computer will print *many* numbers, like this:

```
3
2
4
4
1
3
5
2
2
5
etc.
```

Each number will be 1 or 2 or 3 or 4 or 5. The order in which the computer prints them is unpredictable. The program’s an infinite loop: it won’t stop until you abort it. If you run the program again, the pattern will be different; for example, it might be:

```
1
4
3
3
2
5
1
1
2
etc.
```

When you run that program, the numbers will fly up the screen faster than you can read. To make the numbers easier to read, make the computer pause after each number, by telling the computer to wait a second (100 centiseconds):

```
1 print rnd(5)
  wait 100
  goto 1
```

Guessing game This program plays a guessing game:

```
print "I'm thinking of a number from 1 to 10."
computer_number=rnd(10)
1 input "what do you think my number is";guess
  if guess<computer_number print "Your guess is too low.": goto 1
  if guess>computer_number print "Your guess is too high.": goto 1
  print "Congratulations! You found my number!"
```

The top line makes the computer say:

```
I'm thinking of a number from 1 to 10.
```

The next line makes the computer think of a random number from 1 to 10; the computer’s number is called “computer_number”. The “input” line asks the human to guess the number.

If the guess is less than the computer’s number, the first “if” line makes the computer say “Your guess is too low” and then goto 1, which lets the human guess again. If the guess is *greater* than the computer’s number, the bottom “if” line makes the computer say “Your guess is too high” and then goto 1.

When the human guesses correctly, the computer arrives at the bottom line, which prints:

```
Congratulations! You found my number!
```

Here’s a sample run:

```
I'm thinking of a number from 1 to 10.
what do you think my number is? 3
Your guess is too low.
what do you think my number is? 8
Your guess is too high.
what do you think my number is? 5
Your guess is too low.
what do you think my number is? 6
Congratulations! You found my number!
```

Dice This program makes the computer roll a pair of dice:

```
print "I'm rolling a pair of dice"
a=rnd(6)
print "One of the dice says ";a
b=rnd(6)
print "The other says ";b
print "The total is ";a+b
```

Line 2 makes the computer say:

```
I'm rolling a pair of dice
```

Each of the dice has 6 sides. The next line, a=rnd(6), rolls one of the dice, by picking a number from 1 to 6. The line saying “b=rnd(6)” rolls the other. The bottom line prints the total.

Here’s a sample run:

```
I'm rolling a pair of dice
One of the dice says 3
The other says 5
The total is 8
```

Here’s another run:

```
I'm rolling a pair of dice
One of the dice says 6
The other says 4
The total is 10
```

Coin flipping This program makes the computer flip a coin:

```
if rnd(2)=1 print "heads" else print "tails"
```

Rnd(2) is a random number from 1 to 2, so it's either 1 or 2. If it's 1, the program makes the computer say "heads"; if it's 2 instead, the program makes the computer say "tails".

Until you run the program, you won't know which way the coin will flip; the choice is random. Each time you run the program, the computer will flip the coin again; each time, the outcome is unpredictable.

Here's how to let the human bet on whether the computer will say "heads" or "tails":

```
1 input "Do you want to bet on heads or tails";bet$
  if bet$<>"heads" and bet$<>"tails" print "Please say heads or tails": goto 1
  if rnd(2)=1 coin$="heads" else coin$="tails"
  print "The coin says ";coin$
  if coin$=bet$ print "You win" else print "You lose"
```

The top line makes the computer ask:

```
Do you want to bet on heads or tails?
```

The next line makes sure the human says "heads" or "tails": if the human's answer isn't "heads" and isn't "tails", the computer gripes. The bottom three lines make the computer flip a coin and determine whether the human won or lost the bet.

Here's a sample run:

```
Do you want to bet on heads or tails? heads
The coin says tails
You lose
```

Here's another:

```
Do you want to bet on heads or tails? tails
The coin says tails
You win
```

Here's another:

```
Do you want to bet on heads or tails? tails
The coin says heads
You lose
```

Here's how to let the human use money when betting:

```
bankroll=100
repeat
  print "You have ";bankroll;" dollars"
1  input "How many dollars do you want to bet";stake
   if stake>bankroll print "You don't have that much! Bet less!": goto 1
   if stake<0 print "You can't bet less than nothing!": goto 1
   if stake=0 print "I guess you don't want to bet anymore": goto 3
2  input "Do you want to bet on heads or tails";bet$
   if bet$<>"heads" and bet$<>"tails" print "Please say heads or tails": goto 2
   if rnd(2)=1 coin$="heads" else coin$="tails"
   print "The coin says ";coin$
   if coin$=bet$ then
     print "You win ";stake;" dollars"
     bankroll=bankroll+stake
   else
     print "You lose ";stake;" dollars"
     bankroll=bankroll-stake
   endif
until bankroll=0
print "You're broke! Too bad!"
3 print "Thanks for playing with me! You were fun to play with!"
print "I hope you play again sometime!"
```

The top line gives the human a \$100 *bankroll*, so the human starts with \$100. The third line makes the computer say:

```
You have 100 dollars
```

The next line makes the computer ask:

```
How many dollars do you want to bet?
```

The number that the human inputs (the number of dollars that the human bets) is called the human's *stake*. The next three lines (which say "if stake") make sure the stake is reasonable.

The line numbered 2 gets the human to bet on heads or tails. The next few lines flip the coin, determine whether the human won or lost the bet, and then send the computer back for another round (if the human isn't broke yet). The bottom three lines say goodbye to the human.

Here's a sample run:

```
You have 100 dollars
How many dollars do you want to bet? 120
You don't have that much! Bet less!
How many dollars do you want to bet? 75
Do you want to bet on heads or tails? heads
The coin says tails
You lose 75 dollars
You have 25 dollars
How many dollars do you want to bet? 10
Do you want to bet on heads or tails? tails
The coin says tails
You win 10 dollars
You have 35 dollars
How many dollars do you want to bet? 35
Do you want to bet on heads or tails? tails
The coin says heads
You lose 35 dollars
You're broke! Too bad!
Thanks for playing with me! You were fun to play with!
I hope you play again sometime!
```

Daily horoscope This program predicts what will happen to you today:

```
print "You will have a ";
case rnd(5) of
  when 1 print "wonderful";
  when 2 print "fairly good";
  when 3 print "so-so";
  when 4 print "fairly bad";
  when 5 print "terrible";
endcase
print " day today!"
```

The computer will say —

```
You will have a wonderful day today!
```

or —

```
You will have a terrible day today!
```

or some in-between comment. That's because the "case" line makes the computer pick a random integer from 1 to 5.

For inspiration, run that program when you get up in the morning. Then notice whether your day turns out the way the computer predicts!

Random decimals You've seen that `rnd(5)` is a random number from 1 to 5: it's 1 or 2 or 3 or 4 or 5. To get a random *decimal* between 0 and 1, say "`rnd(1)`" instead of "`rnd(5)`".

The decimal that `rnd(1)` produces is at least 0 and is less than 1, so it can be any decimal from 0.00000000 to 0.99999999. For example, the decimal might be 0.28459187.

Suppose you want the computer to *maybe* print "love". Here's how to make the probability of printing "love" be 37 percent:

```
if rnd(1)<.37 print "love"
```

Character codes

You can use these code numbers:

33 !	64 @	96 `	160	192 À	224 à
34 "	65 A	97 a	161 ¡	193 Á	225 á
35 #	66 B	98 b	162 ¢	194 Â	226 â
36 \$	67 C	99 c	163 £	195 Ã	227 ã
37 %	68 D	100 d	164 ¤	196 Ä	228 ä
38 &	69 E	101 e	165 ¥	197 Å	229 å
39 '	70 F	102 f	166 ¦	198 Æ	230 æ
40 (71 G	103 g	167 §	199 Ç	231 ç
41)	72 H	104 h	168 ¨	200 È	232 è
42 *	73 I	105 i	169 ©	201 É	233 é
43 +	74 J	106 j	170 ª	202 Ê	234 ê
44 ,	75 K	107 k	171 «	203 Ë	235 ë
45 -	76 L	108 l	172 ¬	204 Ì	236 ì
46 .	77 M	109 m	173 ¯	205 Í	237 í
47 /	78 N	110 n	174 ®	206 Î	238 î
48 0	80 P	112 p	175 ¯	207 Ï	239 ï
49 1	81 Q	113 q	176 °	208 Ð	240 ð
50 2	82 R	114 r	177 ±	209 Ñ	241 ñ
51 3	83 S	115 s	178 ²	210 Ò	242 ò
52 4	84 T	116 t	179 ³	211 Ó	243 ó
53 5	85 U	117 u	180 ´	212 Ì	244 ô
54 6	86 V	118 v	181 µ	213 Õ	245 õ
55 7	87 W	119 w	182 ¶	214 Ö	246 ö
56 8	88 X	120 x	183 ·	215 ×	247 ÷
57 9	89 Y	121 y	184 ¸	216 Ø	248 ø
58 :	90 Z	122 z	185 ¹	217 Ù	249 ù
59 ;	91 [123 {	186 º	218 Ú	250 ú
60 <	92 \	124	187 »	219 Û	251 û
61 =	93]	125 }	188 ¼	220 Ü	252 ü
62 >	94 ^	126 ~	189 ½	221 Ý	253 ý
63 ?	95 _	127	190 ¾	222 Þ	254 þ
			191 ¿	223 ß	255 ÿ

Alt key Here's how to type the symbol ñ, whose code number is 241. Hold down the Alt key; and while you keep holding down the Alt key, type 0241 *by using the numeric keypad* (the number keys on the far right side of the keyboard). When you finish typing 0241, lift your finger from the Alt key, and you'll see ñ on your screen!

The Alt key works for all numbers in that chart. Remember to type 0 before the number.

You can use the Alt key in your program. For example, try typing this program:

```
print "In Spanish, tomorrow is mañana"
```

While typing that program, make the symbol ñ by typing 0241 on the numeric keypad while holding down the Alt key. When you run that program, the computer will print:

```
In Spanish, tomorrow is mañana
```

The Alt key works well in Windows 10, such as when you're typing in the program window, but *not* if you're typing weirdly (such typing in the output window).

Chr\$ Here's another way to type the symbol ñ:

```
print chr$(241)
```

When you run that program, the computer will print the **character** whose code number is 241. The computer will print:

```
ñ
```

This program makes the computer print "In Spanish, tomorrow is mañana":

```
print "In Spanish, tomorrow is ma";chr$(164);"ana"
```

That makes the computer print "In Spanish, tomorrow is ma", then print character 164 (which is ñ), then print "ana".

This program prints, on your screen, all the symbols in the chart:

```
for i= 33 TO 255
  print chr$(i);
next
```

Low codes Codes below 33 do special things.

In a "print" statement, **chr\$(7)** makes the computer play a 3-note melody.

Try it! Run this program:

```
print chr$(7)
```

Chr\$(32) is a blank space. It's the same as " ".

Chr\$(12) erases the entire screen.

Saying —

```
print chr$(12);
```

has the same effect as saying:

```
cls
```

Chr\$(10) make the computer move down on the screen. Saying —

```
print "hot";chr$(10);"dog"
```

makes the computer print "hot" then move down then print "dog", so you see:

```
hot
```

```
  dog
```

Asc The code numbers from 33 to 126 are for characters that you can type on the keyboard easily. Established by a national committee, those code numbers are called the **American Standard Code for Information Interchange**, which is abbreviated **Ascii**, which is pronounced "ass key".

Programmers say, "the Ascii code number for A is 65". If you say —

```
print asc("A")
```

the computer will print the Ascii code number for "A". It will print:

```
65
```

If you say to print asc("B"), the computer will print 66. If you say to print asc("b"), the computer will print 98.

If you say to print asc("ñ"), the computer will print 241 (which is the code number for ñ), even though ñ isn't an Ascii character.

String analysis

Let's analyze the word "smart".

Length Since "smart" has 5 characters in it, the **length** of "smart" is 5. If you say —

```
print len("smart")
```

the computer will print the **length** of "smart"; it will print:

```
5
```

Left, right, middle The leftmost character in "smart" is "s". If you say —

```
print left$("smart")
```

the computer will print:

```
s
```

Try this program:

```
a$="smart"
print left$(a$)
```

The top line says a\$ is "smart". The bottom line says to print the leftmost character in a\$, which is "s". The computer will print:

```
s
```

If a\$ is “smart”, here are the consequences....

```
len(a$) is the length of a$. It is 5.  
left$(a$) is the leftmost character in a$. It is “s”.  
left$(a$,2) is the left 2 characters in a$. It is “sm”.  
right$(a$) is the rightmost character in a$. It is “t”.  
right$(a$,2) is the right 2 characters in a$. It is “rt”.  
mid$(a$,2) begins in the middle of a$, at the 2nd character. It is “mart”.  
mid$(a$,2,3) begins at 2nd character and includes 3 characters. It is “mar”.
```

You can change a string’s left part, like this:

```
a$="bunkers"  
left$(a$)="hoo"  
print a$
```

The top line says a\$ is “bunkers”. The left\$ line changes the left part of a\$ to “hoo”. The “hoo” covers up the “bun”, so the bottom line prints:

```
hookers
```

Here’s a variation:

```
a$="bunkers"  
left$(a$,2)="hoo"  
print a$
```

The top line says a\$ is “bunkers”. The left\$ line steals the left 2 characters from “hoo”, so it steals “ho”. The “ho” covers up “bu”, so the bottom line prints:

```
honkers
```

You can change a string’s right part, like this:

```
a$="bunkers"  
right$(a$,4)="tingly"  
print a$
```

The top line says a\$ is “bunkers”. The right\$ line changes the right 4 characters of “bunkers”, by stealing them from the beginning of “tingly”, which is “ting”. The “ting” covers up “kers”, so the bottom line prints:

```
bunting
```

You can change a string’s middle, like this:

```
a$="bunkers"  
mid$(a$,2)="owl"  
print a$
```

The top line says a\$ is “bunkers”. The mid\$ line changes the middle of a\$ to “owl”; the change begins at the 2nd character of a\$. The bottom line prints:

```
bowlers
```

Here’s a variation:

```
a$="bunkers"  
mid$(a$,2)="ad agency"  
print a$
```

The top line says a\$ is “bunkers”. The mid\$ line says to change the middle of a\$, beginning at the 2nd character of a\$. But “ad agency” is too long to become part of “bunkers”. The computer uses as much of “ad agency” as will fit in “bunkers”. The computer will print:

```
bad age
```

Another variation:

```
a$="bunkers"  
mid$(a$,2,1)="owl"  
print a$
```

The top line says a\$ is “bunkers”. The mid\$ line says to change the middle of a\$, beginning at the 2nd character of a\$. But the “,1” makes the computer use just 1 letter from “owl”. The bottom line prints:

```
bonkers
```

Adding strings You can add strings together, to form a longer string:

```
a$="fat"+"her"  
print a$
```

The top line says a\$ is “father”. The bottom line makes the computer print:

```
father
```

Searching in a string You can make the computer search in a string to find another string. To make the computer search in the **string** “needed” to find “ed”, say:

```
print instr("needed","ed")
```

Since “ed” begins at the third character of “needed”, the computer will print:

```
3
```

If you say —

```
print instr("needed","ey")
```

the computer will search in the string “needed” for “ey”. Since “ey” is *not* in “needed”, the computer will print:

```
0
```

If you say —

```
print instr("needed","ed",4)
```

the computer will hunt in the string “needed” for “ed”; but the hunt will begin at the 4th character of “needed”. The computer finds the “ed” that begins at the 5th character of “needed”. The computer will print:

```
5
```

String-number conversion This program converts a string to a number:

```
a$="52.6"  
b=val(a$)  
print b+1
```

The top line says a\$ is the string “52.6”. The next line says b is the numeric **value** of a\$, so b is the number 52.6. The bottom line prints:

```
53.6
```

Val converts a string to a number. The opposite of val is **str\$**, which converts a number to a string. For example, str\$(-7.2) is the string “-7.2”. Str\$(81.4) is the string “81.4”.

Repeating characters Suppose you love the letter b (because it stands for big, bold, and beautiful) and want to print “bbbbbbbbbbbbbbbbbb”. Here’s a short-cut:

```
print string$(20,"b")
```

That tells the computer to print a string of 20 b’s.

This makes the computer print “fat” 10 times:

```
print string$(10,"fat")
```

The computer will print:

```
fatfatfatfatfatfatfatfatfatfat
```

Let’s make the computer draw a dashed line containing 50 dashes, like this:

```
-----
```

Here’s how: just say print string\$(50,“-”).

Let's make the computer print this triangle:

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

To do that, we want the computer to print 1 asterisk on the first line, then 2 asterisks on the next line, then 3 asterisks on the next line, and so on, until it finally prints 12 asterisks on the bottom line. Here's the program:

```
for i=1 to 12
  print string$(i,"*")
next
```

The "for" line makes *i* be 1, then 2, then 3, and so on, up to 12. When *i* is 1, the "print" line makes the computer print one asterisk, like this:

```
*
```

When *i* is 2, the "print" line makes the computer print a line of 2 asterisks, like this:

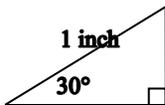
```
**
```

The "for" line makes *i* be every number from 1 up to 20, so computer will print 1 asterisk, then underneath print a line of 2 asterisks, then underneath print a line of 3 asterisks, and so on, until the entire triangle is printed.

Trigonometry

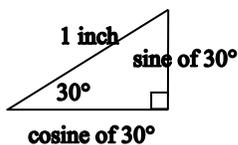
The study of triangles is called **trigonometry** — and the computer can do it for you!

For example, look at this triangle:



In that triangle, the left angle is 30°, the bottom-right angle is 90°, and the longest side (the hypotenuse) is 1 inch long.

The side opposite the 30° angle is called the **sine** of 30°; the remaining side is called the **cosine** of 30°:



How long is the sine of 30°? How long is the cosine of 30°?

Since the longest side (the hypotenuse) is 1 inch long, and since the sine and the cosine are shorter sides, the sine and the cosine must each be shorter than 1 inch. So the lengths of the sine and cosine are each less than 1. But which decimals are they?

To find out, you can use a ruler. You'll discover that the sine is half an inch long, and the cosine is nearly seven-eighths of an inch long. But a faster and more accurate way to measure the sine and cosine is to let the computer do it! Yes, the computer can calculate triangles in its mind!

This program makes the computer measure the sine and cosine of 30°:

```
d=pi/180
print sin(30*d)
print cos(30*d)
```

The top line is a special formula that defines *d* to mean *degrees*. The first "print" line prints the sine of 30 degrees:

```
.5
```

The bottom line prints the cosine of 30°, which is a decimal that's slightly less than .87.

The computer can measure the sine and cosine of *any* size angle. Try it! For example, to make the computer print the sine and cosine of a 33° angle, say:

```
d=pi/280
print sin(33*d)
print cos(33*d)
```

If you choose an angle of -33° instead of 33°, the triangle will dip down instead of rising up, and so the sine will be a negative number instead of positive.

In those "print" lines, the "*d" is important: it tells the computer that you want the sine of 33 **degrees**. If you accidentally omit the "*d", the computer will print the sine of 33 **radians** instead. (A radian is larger than a degree. A radian is about 57.3 degrees. More precisely, a radian is 180/π degrees.)

Tangent The sine divided by the cosine is called the **tangent**. For example, to find the tangent of 33°, divide the sine of 33° by the cosine of 33°.

To make the computer print the tangent of 33°, you could tell the computer to print sin(33*d)/cos(33*d). But to find the tangent more quickly and easily, say just print tan(33*d).

Arc functions The opposite of the sine is called the **arcsine**:

```
the sine of 30° is .5
the arcsine of .5 is 30°
```

Similarly, the opposite of the cosine is called the **arccosine**, and the opposite of the tangent is called the **arctangent**.

This program prints the the arcsine of .5, the arccosine of .87, and the arctangent of .58:

```
d=pi/180
print asn(.5)/d
print acs(.87)/d
print atn(.58)/d
```

Line 2 prints the arcsine of .5, in degrees. (If you omit the "/" degrees", the computer will print the answer in radians instead of degrees.) Line 3 prints the arccosine of .87. Line 4 prints the arctangent of .58. The answer to each of the three problems is about 30 degrees.

Subscripts

Instead of being a single string, x\$ can be a whole *list* of strings, like this:

```
x$ = ( "love"  
      "hate"  
      "kiss"  
      "kill"  
      "peace"  
      "war"  
      "why" )
```

Here's how to make x\$ be that list of strings....

Begin your program by saying:

```
dim x$(7)
```

That line says x\$ will be a list of 7 strings. DIM means **dimension**; the line says the dimension of x\$ is 7.

Next, tell the computer what strings are in x\$. Type these lines:

```
x$(1)="love"  
x$(2)="hate"  
x$(3)="kiss"  
x$(4)="kill"  
x$(5)="peace"  
x$(6)="war"  
x$(7)="why"
```

That says x\$'s first string is "love", x\$'s second string is "hate", etc.

If you want the computer to print all those strings, type this:

```
for i=1 to 7  
  print x$(i)  
next
```

That means: print all the strings in x\$. The computer will print:

```
love  
hate  
kiss  
kill  
peace  
war  
why
```

That program includes a line saying x\$(1)="love". Instead of saying x\$(1), math books say:

```
x1
```

The "1" is called a **subscript**.

Similarly, in the line saying x\$(2)="hate", the number 2 is a subscript. Some programmers pronounce that line as follows: "x string, subscripted by 2, is hate". Hurried programmers just say: "x string 2 is hate".

In that program, x\$ is called an **array** (or **matrix**). Definition: an **array** (or **matrix**) is a variable that has subscripts.

Each item in an array is called an **element** of the array. For example, "love" is an element of that x\$, and so is "hate".

Subscripted data

That program said x\$(1) is "love", and x\$(2) is "hate", and so on. This program does the same thing, more briefly:

```
dim x$(7)  
data love,hate,kiss,kill,peace,war,why  
for i=1 to 7  
  read x$(i)  
next  
for i=1 to 7  
  print x$(i)  
next
```

The "dim" line says x\$ will be a list of 7 strings. The "data" line contains a list of 7 strings. The first "for...next" loop makes the computer read those strings and call them x\$. The bottom "for...next" loop makes the computer print those 7 strings.

In that program, the first 3 lines say:

```
dim  
data  
for i
```

Most practical programs begin with those 3 lines.

Let's lengthen the program, so the computer prints all this:

```
love  
hate  
kiss  
kill  
peace  
war  
why  
  
why love  
why hate  
why kiss  
why kill  
why peace  
why war  
why why
```

That consists of two verses. The second verse resembles the first, except each line of the second verse begins with "why".

To make the computer print all that, just add the shaded lines to the program:

```
dim x$(7)  
data love,hate,kiss,kill,peace,war,why  
for i=1 to 7  
  read x$(i)  
next  
for i = 1 TO 7  
  print x$(i)  
next  
print  
for i=1 to 7  
  print "why ";x$(i)  
next
```

The shaded "print" line leaves a blank line between the first verse and the second. The shaded "for...next" loop, which prints the second verse, resembles the "for...next" loop that printed the first verse but prints "why" before each x\$(i).

Let's add a third verse, which prints the words in reverse order:

```
why  
war  
peace  
kill  
kiss  
hate  
love
```

Before printing that third verse, print a blank line:

```
print
```

Then print the verse itself. To print the verse, you must print x\$(7), then print x\$(6), then print x\$(5), etc. To do that, you could say:

```
print x$(7)  
print x$(6)  
print x$(5)  
etc.
```

But this way is shorter:

```
for i=7 to 1 step -1  
  print x$(i)  
next
```

Numeric arrays

Let's make y be this list of five numbers: 100, 94, 201, 8.3, and -7. To begin, tell the computer that y will consist of five numbers:

```
dim y(5)
```

Next, tell the computer what the six numbers are:

```
data 100,94,201,8.3,-7
```

Make the computer read all that data:

```
for i=1 to 5
  read y(i)
next
```

To make the computer print all that data, type this:

```
for i=1 to 5
  print y(i)
next
```

If you want the computer to add those 5 numbers together and print their sum, say:

```
print y(1)+y(2)+y(3)+y(4)+y(5)
```

Strange example

Getting tired of x and y? Then pick another letter! For example, you can play with z:

Silly, useless program	What the program means
<pre>dim z(5)</pre>	z will be a list of 5 numbers
<pre>for i=2 to 5 z(i)=i*100 next</pre>	z(2)=200; z(3)=300; z(4)=400; z(5)=500
<pre>z(1)=z(2)-3</pre>	z(1) is 200 - 3, so z(1) is 197
<pre>z(3)=z(1)-2</pre>	z(3) changes to 197 - 2, which is 195
<pre>for i=1 to 5 print z(i) next</pre>	print z(1), z(2), z(3), z(4), and z(5)

The computer will print:

```
197
200
195
400
500
```

Problems and solutions

Suppose you want to analyze 20 numbers. Begin your program by saying:

```
dim x(20)
```

Then type the 20 numbers as data:

```
data etc.
```

Tell the computer to read the data:

```
for i=1 to 20
  read x(i)
next
```

Afterwards, do one of the following, depending on which problem you want to solve....

Print all x values Solution:

```
for i=1 to 20
  print x(i)
next
```

Print all x values, in reverse order Solution:

```
for i=20 to 1 step -1
  print x(i)
next
```

Print the sum of all x values In other words, print $x(1)+x(2)+x(3)+\dots+x(20)$. Solution: start the sum at 0 —

```
s=0
```

and then increase the sum, by adding each $x(i)$ to it:

```
for i=1 to 20
  s=s+x(i)
next
```

Finally, print the sum:

```
print "The sum of all the numbers is";s
```

Find the average of x In other words, find the average of the 20 numbers. Solution: begin by finding the sum —

```
s=0
for i=1 to 20
  s=s+x(i)
next
```

then divide the sum by 20:

```
print "The average is";s/20
```

Find whether any x value is 79.4 In other words, find out whether 79.4 is a number in the list. Solution: if $x(i)$ is 79.4, print "Yes" —

```
for i=1 to 20
  if x(i)=79.4 print "Yes, 79.4 is in the list": end
next
```

otherwise, print "No":

```
print "No, 79.4 is not in the list"
```

In x's list, count how often 79.4 appears Solution: start the counter at zero —

```
counter=0
```

and increase the counter each time you see the number 79.4:

```
for i=1 to 20
  if x(i)=79.4 counter=counter+1
next
```

Finally, print the counter:

```
print "The number 79.4 appears ";counter;" times"
```

Print all x values that are negative In other words, print all the numbers that have minus signs. Solution: begin by announcing your purpose —

```
print "Here are the values that are negative:"
```

then print the values that are negative; in other words, print each $x(i)$ that's less than 0:

```
for i=1 to 20
  if x(i)<0 print x(i)
next
```

Print all x values that are above average Solution: find the average —

```
s=0
for i=1 to 20
  s=s+x(i)
next
average=s/20
```

then announce your purpose:

```
print "The following values are above average:"
```

Finally, print the values that are above average; in other words, print each $x(i)$ that's greater than average:

```
for i=1 to 20
  if x(i)>average print x(i)
next
```

Find x's biggest value In other words, find which of the 20 numbers is the biggest. Solution: begin by assuming that the biggest is the first number —

```
biggest=x(1)
```

but if you find another number that's even bigger, change your idea of what the biggest is:

```
for i=2 to 20
  if x(i)>biggest biggest=x(i)
next
```

Afterwards, print the biggest:

```
print "The biggest number in the list is"; biggest
```

Find x's smallest value In other words, find which of the 20 numbers is the smallest. Solution: begin by assuming that the smallest is the first number —

```
smallest=x(1)
```

but if you find another number that's even smaller, change your idea of what the smallest is:

```
for i=2 to 20
  if x(i)<smallest smallest=x(i)
next
```

Afterwards, print the smallest:

```
print "The smallest number in the list is";smallest
```

Check whether x's list is in strictly increasing order In other words, find out whether the following statement is true: x(1) is a smaller number than x(2), which is a smaller number than x(3), which is a smaller number than x(4), etc. Solution: if x(i) is *not* smaller than x(i + 1), print "No" —

```
for i=1 to 19
  if x(i)>=x(i+1) then
    print "No, the list is not in strictly increasing order"
  end
endif
next
```

otherwise, print "Yes":

```
print "Yes, the list is in strictly increasing order"
```

Test yourself: look at those problems again, and see whether you can figure out the solutions *without peeking at the answers*.

Multiple arrays

Suppose your program involves three lists. Suppose the first list is called a\$ and consists of 18 strings; the second list is called b and consists of 57 numbers; and the third list is called c\$ and consists of just 3 strings. To say all that, begin your program with this statement:

```
dim a$(18),b(57),c$(3)
```

Double subscripts

You can make x\$ be a **table** of strings, like this:

```
x$= ( "dog"      "cat"      "mouse"
      "hotdog"  "catsup"   "mousetard" )
```

Here's how to make x\$ be that table....

Begin by saying:

```
dim x$(2,3)
```

That says x\$ will be a table having 2 rows and 3 columns.

Then tell the computer what strings are in x\$. Type these lines:

```
x$(1,1)="dog"
x$(1,2)="cat"
x$(1,3)="mouse"
x$(2,1)="hotdog"
x$(2,2)="catsup"
x$(2,3)="mousetard"
```

That says the string in x\$'s first row and first column is "dog", the string in x\$'s first row and second column is "cat", etc.

If you'd like the computer to print all those strings, type this:

```
for i=1 to 2
  for j=1 to 3
    print x$(i,j),;
  next
print
next
```

That means: print all the strings in x\$. The computer will print:

dog	cat	mouse
hotdog	catsup	mousetard

Most programmers follow this tradition: **the row's number is called i, and the column's number is called j**. That program obeys that tradition. The "for i=1 to 2" means "for both rows"; the "for j=1 to 3" means "for all 3 columns".

Notice i comes before j in the alphabet; i comes before j in x(i,j); and "for i" comes before "for j". If you follow the i-before-j tradition, you'll make fewer errors.

At the end of the first "print" line, the comma makes the computer print each column in a separate zone; the semicolon makes the computer keep printing on the same line. The other "print" line makes the computer press the Enter key at the end of each row. The x\$ is called a **table** or **2-dimensional array** or **doubly subscripted array**.

The sum of all numbers in the table is sum(x()).

Multiplication table

This program prints a multiplication table:

```
dim x(10,5)
for i=1 to 10
  for j=1 to 5
    x(i,j)=i*j
  next
next
for i=1 to 10
  for j=1 to 5
    print x(i,j);
  next
print
next
```

The top line says x will be a table having 10 rows and 4 columns.

The line saying "x(i,j)=i*j" means the number in row i and column j is i*j. For example, the number in row 3 and column 4 is 12. Above that line, the program says "for i=1 to 10" and "for j=1 to 4", so that x(i,j)=i*j for *every* i and j, so *every* entry in the table is defined by multiplication.

The computer prints the whole table:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25
6	12	18	24	30
7	14	21	28	35
8	16	24	32	40
9	18	27	36	45
10	20	30	40	50

Instead of multiplication, you can have addition, subtraction, or division: just change the line saying "x(i,j)=i*j".

Summing a table

Suppose you want to analyze this table:

32.7	19.4	31.6	85.1
-8	402	-61	0
5106	-.2	0	-1.1
36.9	.04	1	11
777	666	55.44	2
1.99	2.99	3.99	4.99
50	40	30	20
12	21	12	21
0	1000	2	500

Since the table has 9 rows and 4 columns, begin your program by saying:

```
dim x(9,4)
```

Each row of the table becomes a row of the data:

```
data 32.7,19.4,31.6,85.1
data -8,402,-61,0
data 5106,-.2,0,-1.1
data 36.9,.04,1,11
data 777,666,55.44,2
data 1.99,2.99,3.99,4.99
data 50,40,30,20
data 12,21,12,21
data 0,1000,2,500
```

Make the computer read the data:

```
for i=1 to 9
  for j=1 to 4
    read x(i,j)
  next
next
```

To make the computer print the table, say this:

```
for i=1 to 9
  for j=1 to 4
    print x(i,j);
  next
  print
next
```

Here are some problems, with solutions....

Find the sum of all the numbers in the table

Solution: start the sum at 0 —

```
s=0
```

and then increase the sum, by adding each $x(i,j)$ to it:

```
for i=1 to 9
  for j=1 to 4
    s=s+x(i,j)
  next
next
```

Finally, print the sum:

```
print "The sum of all the numbers is ";s
```

The computer will print:

```
The sum of all the numbers is 8877.84
```

Find the sum of each row In other words, make the computer print the sum of the numbers in the first row, then the sum of the numbers in the second row, then the sum of the numbers in the third row, etc. Solution: the general idea is —

```
for i=1 to 9
  print the sum of row i
next
```

Here are the details:

```
for i=1 to 9
  s=0
  for j=1 to 4
    s=s+x(i,j)
  next
  print "The sum of row ";i;" is ";s
next
```

The computer will print:

```
The sum of row 1 is 168.8
The sum of row 2 is 333
The sum of row 3 is 5104.7
etc.
```

Find the sum of each column In other words, make the computer print the sum of the numbers in the first column, then the sum of the numbers in the second column, then the sum of the numbers in the third column, etc. Solution: the general idea is —

```
for j=1 to 4
  print the sum of column j
next
```

Here are the details:

```
for j =1 to 4
  s=0
  for i=1 to 9
    s=s+x(i,j)
  next
  print "The sum of column ";j;" is ";s
next
```

The computer will print:

```
The sum of column 1 is 6008.59
The sum of column 2 is 2151.23
The sum of column 3 is 75.03
The sum of column 4 is 642.99
```

In all the other examples, “for i” came before “for j”; but in this unusual example, “for i” comes *after* “for j”.

Secrets about subscripts

When you say “dim x(4)”, the computer creates $x(1)$, $x(2)$, $x(3)$, and $x(4)$ but also **secretly creates $x(0)$** , which you can use the same way as the other elements, so altogether the array has 5 elements.

The computer secretly makes all 5 of those elements be temporarily 0, until you change them, by giving commands such as $x(2)=9.4$.

You can change all 5 of them fast by giving a command such as:

```
x(0)=138,2049,9.4,-4997,120
```

That makes $x(0)$ be 138, $x(1)$ by 2049, $x(2)$ be 9.4, $x(3)$ be -4997, and $x(4)$ be 120.

To make all 5 of them be 9.4, you can say just:

```
x(0)=9.4
```

The sum of all 5 numbers is automatically called $\text{sum}(x())$, so to print their sum you can say just:

```
print sum(x())
```

That’s much easier than computing the sum by writing a “for...next” loop.

Similarly, saying “dim x\$(4)” creates an array of 5 strings, beginning with $x\$(0)$. The computer makes all 5 of those strings be temporarily “” (which is an empty string), until you give commands such as $x\$(2) = \text{“grape”}$. You can change all 5 of them fast by giving a command such as”

```
x$(0) = "lemon", "lime", "grape", "cherry", "banana"
```

To make all 5 of them be “grape”, say $x\$(0) = \text{“grape”}$. The sum of all 5 strings is called $\text{sum}(x\$(0))$, which you can print by saying:

```
print sum(x$(0))
```

The sum is computed using string arithmetic: “hot”+”dog” is “hotdog”.

Saying “dim x(2,3)” creates a table that includes 2 rows and 3 columns but also secretly includes a row #0 and column #0, so altogether the table has 3 rows and 4 columns, a total of 12 elements. All numbers in the table are temporarily 0, until you change them. You can change all of them fast by giving a command such as:

```
x(,)=1,2,3,4,5,6,7,8,9,10,11,12
```

That makes:

```
x(0,0) be 1, x(0,1) be 2, x(0,2) be 3, x(0,3) be 4
x(1,0) be 5, x(1,1) be 6, x(1,2) be 7, x(1,3) be 8
x(2,0) be 9, x(2,1) be 10, x(2,2) be 11, x(2,3) be 12
```

Proc

Here’s a sick program:

```
print "we all know..."
print "You are stupid!"
print "You are ugly!"
print "...and yet we love you."
```

It makes the computer print this message:

```
we all know...
You are stupid!
You are ugly!
...and yet we love you.
```

So the computer prints “We all know...”, then insults the human (“You are stupid! You are ugly!”), then prints “...and yet we love you.”

Here’s a more sophisticated way to write that program:

```
print "we all know..."
proc_insult
print "...and yet we love you."
end

def proc_insult
print "You are stupid!"
print "You are ugly!"
endproc
```

In the sophisticated version, the top 3 lines tell the computer to print “We all know...”, then do a **procedure** to insult the human, then print “...and yet we love you.” But the computer doesn’t know how to insult yet.

The bottom 4 lines teach the computer how to insult: they say “insult” means to print “You are stupid!” and “You are ugly!” Those bottom 4 lines define the word insult; they’re the **definition** of insult.

That program is divided into two **procedures**. The top 4 lines are called the **main procedure** (or **main routine** or **main module**). The bottom 4 lines (which just **define** “proc_insult”) are called the **subprocedure** (or **subroutine** or **submodule**).

The subprocedure’s first line (“**def proc_insult**”) means “here’s the definition of proc_insult”. The subprocedure’s bottom line (“endproc”) means “this is the end of the subprocedure”.

The main procedure’s bottom line (“end”) means “this is the end of the main procedure”.

When you run the program, the computer will say:

```
we all know...
You are stupid!
You are ugly!
...and yet we love you.
```

Refrains

This is chanted by boys playing tag — and protesters fearing dictators:

```
The lion is a-coming near.
    He'll growl and sneer
    And drink our beer.
The lion never brings us cheer.
    He'll growl and sneer
    And drink our beer.
The lion is the one we fear.
    He'll growl and sneer
    And drink our beer.
Gotta stop the lion!
```

In that chant, this refrain is repeated:

```
He'll growl and sneer
And drink our beer.
```

This program prints the entire chant:

```
print "The lion is a-coming near."
proc_refrain
print "The lion never brings us cheer."
proc_refrain
print "The lion is the one we fear."
proc_refrain
print "Gotta stop the lion!"
end

def proc_refrain
print "    He'll growl and sneer"
print "    And drink our beer."
endproc
```

Big love

This program prints a love poem:

```
print "The most beautiful thing in the world is"
print "LOVE"
print "The opposite of war is"
print "LOVE"
print "And when I look at you, I feel lots of"
print "LOVE"
```

In that program, many of the lines make the computer print the word LOVE. Let’s make those lines print the word LOVE bigger, like this:

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

To make LOVE be that big, run this version of the program:

```
print "The most beautiful thing in the world is"
proc_big_love
print "The opposite of war is"
proc_big_love
print "And when I look at you, I feel lots of"
proc_big_love
end

def proc_big_love
print "*"
print "*           *           *           *           * * * * *"
print "*           *   *           *           *           *"
print "*           *           *           *           * * *"
print "*           *   *           * *           *"
print "* * * * *           *           *           * * * * *"
endproc
```

In that version, the lines say proc_big_love instead of print “LOVE”. The subprocedure teaches the computer how to make big_love.

Changed variable

Suppose you run this program:

```
x=4
proc_joe
print x
end

def proc_joe
x=x+1
endproc
```

The computer begins by doing the main procedure, which says x is 4. Then the computer does procedure `joe`, which adds 1 to x , so x becomes 5. Then the computer finishes the main procedure, which says to print x , so the computer prints:

5

Style

To become a *good* programmer, write your programs using a good style. Here's how....

Design a program

First, decide on your ultimate goal. Be optimistic. Maybe you'd like the computer to play the perfect game of chess? or translate every English sentence into French?

Research the past Whatever you want the computer to do, someone else probably thought of the same idea already and wrote a program for it.

Find out. Ask your friends. Ask folks in nearby schools, computer stores, computer centers, companies, libraries, and bookstores. Look through books and magazines. There are even books that list what programs have been written. Ask the company you bought your computer from.

Even if you don't find exactly the program you're looking for, you may find one that's close enough to be okay, or that will work with just a little fixing or serve as *part* of your program or at least give you a *clue* as to where to begin. In a textbooks or magazines, you'll probably find a discussion of the problem you're trying to solve and the pros and cons of various solutions to it — some methods are faster than others.

If you keep your head in the sand and don't look at what other programmers have done already, your programming effort may turn out to be a mere exercise, useless to the rest of the world.

Simplify Too often, programmers embark on huge projects and never get them done. Once you have an idea of what's been done before and how hard your project is, simplify it.

Instead of making the computer play a perfect game of chess, how about settling for a game in which the computer plays unremarkably but at least doesn't cheat? Instead of translating every English sentence into French, how about translating just English colors? (We wrote that program already.)

So **pick a less ambitious, more realistic goal**, which if achieved will please you and be a steppingstone to your ultimate goal.

Finding a bug in a program is like finding a needle in a haystack: removing the needle is easier if the haystack is small than if you wait until more hay's been piled on.

Specify the I/O Make your new, simple goal more precise. That's called **specification**. One way to be specific is to **draw a picture, showing what your screen will look like if your program's running successfully**.

In that picture, find the lines typed by the computer. They become your program's "print" statements. Find the lines typed by the human: they become the "input" statements. Now you can start writing your program: **write the "print" and "input" statements** on paper, with a pencil, and leave blank lines between them. You'll fill in the blanks later.

Suppose you want the computer to find the average of two numbers. Your picture will look like this:

```
what's the first number? 7
what's the second number? 9
The average is 8
```

Your program at this stage will be:

```
input "what's the first number";a
input "what's the second number";b
etc.
print "The average is ";c
```

All you have left to do is figure out what the "etc." is. Here's the general method....

Choose your statements Suppose you didn't have a computer. Then how would you get the answer?

Would you have to use a mathematical formula? If so, put the formula into your program, but remember that the equation's left side must have just one variable. For example, if you're trying to solve a problem about right triangles, you might have to use the Pythagorean formula $a^2+b^2=c^2$; but the left side of the equation must have just one variable, so your program must say $a=\text{sqr}(c^2-b^2)$, or $b=\text{sqr}(c^2-a^2)$, or $c=\text{sqr}(a^2+b^2)$, depending on whether you're trying to

compute a , b , or c .

Would you have to use a memorized list, such as an English-French dictionary or the population of each state or the weight of each chemical element? If so, that list becomes your data, and you must read it. If it would be helpful to have the data numbered — so the first piece of data is called $x(1)$, the next piece of data is called $x(2)$, etc. — use the "dim" statement.

Subscripts are particularly useful if one long list of information will be referred to *several* times in the program.

Does your reasoning repeat? That means your program should have a loop. If you know how many times to repeat, say "for...next". If you're not sure how often, say "repeat...until". If the thing to be repeated isn't repeated immediately, but just after several other things have happened, make the repeated part be a subprocedure.

At some point in your reasoning, do you have to make a *decision*? Do you have to choose among several alternatives? To choose between 2 alternatives, say "if". To choose among 3 or more alternatives, say "case". If you want the computer to make the choice arbitrarily, "by chance" instead of for a reason, say "if rnd".

Do you have to compare two things? The way to say "compare x with y " is: "if $x=y$ ".

Write pseudocode Some English teachers say that before you write a paper, you should make an outline. Some computer teachers give similar advice about writing programs.

The "outline" can look like a program in which some of the lines are written in plain English instead of computerese. For example, one statement in your outline might be:

```
a = the average of the 12 values of x
```

Such a statement, written in English instead of in computerese, is called **pseudocode**. Later, when you fill in the details, expand that pseudocode to this:

```
s=0
for i=1 to 12
  s=sum+x(i)
next
average=s/12
```

Organize yourself Keep the program's over-all organization simple. That will make it easier for you to expand the program and find bugs. Here's some folklore, handed down from generation to generation of programmers, that will simplify your organization....

Use top-down programming. That means write a one-sentence description of your program; then expand that sentence to several sentences; then expand each of those sentences to several more sentences; and so on, until you can't expand any more. Then turn each of those new sentences into lines of program. Then your program will be in the same order as the English sentences, therefore organized the same way as an English-speaking mind.

A variation is to **use subprocedures**. That means writing the essence of the program as a very short main procedure; instead of filling in the grubby details immediately, replace each piece of grubbiness by a subprocedure. Your program will be like a good book: your main procedure will move swiftly, and the annoying details will be relegated to the appendices at the back; the appendices are the subprocedures. Make each procedure brief — no more than 30 lines — so the entire procedure can fit on the screen; if it starts getting longer and grubbier, replace each piece of grubbiness by *another* subprocedure.

Avoid "goto". It's hard for a human to understand a program that's a morass of "goto" statements. It's like trying to read a book where each paragraph says to turn to a different page! When you *must* say "goto", try to go forward instead of backwards and not go too far.

Use variables After you've written some lines of your program, you may notice that your reasoning "almost repeats": several lines bear a strong resemblance to each other. You can't use "repeat...until" or "for...next" unless the lines repeat exactly. To make the repetition complete, use a variable to represent the parts that are different.

For example, suppose your program contains these lines:

```
print 29.34281+9.876237*sqr(5)
print 29.34281+9.876237*sqr(7)
print 29.34281+9.876237*sqr(9)
print 29.34281+9.876237*sqr(11)
print 29.34281+9.876237*sqr(13)
print 29.34281+9.876237*sqr(15)
print 29.34281+9.876237*sqr(17)
print 29.34281+9.876237*sqr(19)
print 29.34281+9.876237*sqr(21)
```

Each of those lines says to print $29.3428+9.87627*\text{sqr}(\text{a number})$. The number keeps changing, so call it *x*. All those "print" lines can be replaced by this loop:

```
for x=5 to 21 step 2
  print 29.34281+9.876237*sqr(x)
next
```

Here's a harder example to fix:

```
print 29.34281+9.876237*sqr(5)
print 29.34281+9.876237*sqr(97.3)
print 29.34281+9.876237*sqr(8.62)
print 29.34281+9.876237*sqr(.4)
print 29.34281+9.876237*sqr(200)
print 29.34281+9.876237*sqr(12)
print 29.34281+9.876237*sqr(591)
print 29.34281+9.876237*sqr(.2)
print 29.24281+9.876237*sqr(100076)
```

Again, let's use *x*. All those "print" lines can be combined like this:

```
data 5,97.3,8.62,.4,200,12,591,.2,100076
for i=1 to 9
  read x
  print 29.34281+9.876237*sqr(x)
next
```

This one's even tougher:

```
print 29.34281+9.876237*sqr(a)
print 29.34281+9.876237*sqr(b)
print 29.34281+9.876237*sqr(c)
print 29.34281+9.876237*sqr(d)
print 29.34281+9.876237*sqr(e)
print 29.34281+9.876237*sqr(f)
print 29.34281+9.876237*sqr(g)
print 29.34281+9.876237*sqr(h)
print 29.34281+9.876237*sqr(i)
```

Let's assume *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, and *i* have been computed earlier in the program. The trick to shortening those lines is to change the names of the variables. Throughout the program, say *x*(1) instead of *a*, say *x*(2) instead of *b*, say *x*(3) instead of *c*, etc. Say *dim x*(9) at the beginning of your program. Then replace all those "print" lines by this loop:

```
for i=1 to 9
  print 29.34281+9.876237*sqr(x(i))
next
```

Make it efficient

Your program should be **efficient**. That means it should use as little of the computer's time and memory as possible.

To use less of the computer's memory, make your **dimensions** as small as possible. Try writing the program without any arrays at all; if that turns out to be terribly inconvenient, use the smallest and fewest arrays possible.

To use less of the computer's time, avoid having the computer do the same thing more than once.

These lines force the computer to compute $\text{sqr}(8.2*n+7)$ three times:

```
print sqr(8.3*n+7)+2
print sqr(8.3*n+7)/9.1
print 5-sqr(8.3*n+7)
```

You should change them to:

```
k=sqr(8.3*n+ 7)
print k+2
print k/9.1
print 5-k
```

These lines force the computer to compute x^9+2 a hundred times:

```
for i=1 to 100
  print (x^9+2)/i
next
```

You should change them to:

```
k=x^9+2
for i=1 to 100
  print k/i
next
```

These lines force the computer to count to 100 twice:

```
s=0
for i=1 to 100
  s=s+x(i)
next
print "The sum of the x's is ";s
product=1
for i=1 to 100
  product=product*x(i)
next
print "The product of the x's is ";product
```

You should combine the two "for...next" loops into a single "for...next" loop, so the computer counts to 100 just once. Here's how:

```
s=0
product=1
for i=1 to 100
  s=s+x(i)
  product=product*x(i)
next
print "The sum of the x's is ";sum
print "The product of the x's is"; product
```

Instead of exponents, use multiplication, which is faster:

```
slow: y=x^2
faster: y=x*x
```

Test it

When you've written a program, **test** it: run it and see whether it works.

If the computer does *not* gripe, your tendency will be to say "Whoopee!" Don't cheer too loudly. **The answers the computer prints might be wrong.** Even if its answers look reasonable, don't assume they're right: the computer's errors can be subtle. Check some of its answers by computing them with a pencil.

Even if the answers the computer prints are correct, don't cheer. Maybe you were just lucky. Type different input, and see whether your program still works. Probably you can input something that will make your program go crazy or print a wrong answer. Your mission: to find input that will reveal the existence of a bug.

Try 6 kinds of input....

Try simple input Type in simple integers, like 2 and 10, so the computation is simple, and you can check the computer's answers easily.

Try input that increases See how the computer's answer changes when the input changes from 2 to 1000.

Does the change in the computer's answer look reasonable? Does the computer's answer go up when it should go up, and down when it should go down?... and by a reasonable amount?

Try input testing each "if" For a program that says —

```
if x<7 goto 10
```

input an x less than 7 (to see whether line 10 works), then an x greater than 7 (to see whether the line underneath the "if" line works), then an x equal to 7 (to see whether you really want "<" instead of "<="), then an x very close to 7, to check round-off error.

For a program that says —

```
if x^2+y<z goto 10
```

input an x, y, and z that make x^2+y less than z. Then try inputs that make x^2+y very close to z.

Try extreme input What happens if you input:

```
a huge number, like 45392000000 or 1E35?  
a tiny number, like .00000003954 or 1E-35?  
a trivial number, like 0 or 1?  
a typical number, like 45.13?  
a negative number, like -52?
```

Find out.

If the input is supposed to be a string, what happens if you input aaaaa or zzzzz? What happens if you capitalize the input? If there are supposed to be two inputs, what happens if you input the same thing for each?

Try input making a line act strange If your program contains

division, try input that will make the divisor be zero or a tiny decimal close to zero. If your program contains the square root of a quantity, try input that will make the quantity be negative. If your program says "for i=x to y", try input that will make y be less than x, then equal to x. If your program mentions x(i), try input that will make i be zero or negative or greater than the dim.

Try input that causes round-off error: for a program that says "x-y" or says "if x=y", try input that will make x almost equal y.

Try garbage Computers often print wrong answers. A computer can print a wrong answer because its circuitry is broken or because a program has a bug. But **the main reason why computers print wrong answers is incorrect input.** Incorrect input is called **garbage** and has several causes....

The user's finger slips. Instead of 400, he inputs 4000. Instead of 27, he inputs 72. Trying to type .753, he leaves out the decimal point.

The user got wrong info. He tries to input the temperature, but his thermometer is leaking. He tries to input the results of a questionnaire, but everybody who filled out his questionnaire lied.

The instructions aren't clear, so the user isn't sure what to input.

```
If the program asks "How far did the ball fall?" should the user type the distance in feet or in meters? Is time to be given in seconds or minutes? Are angles to be measured in degrees or radians?
```

```
Can the user input "y" instead of "yes"?
```

```
Maybe the user isn't clear about whether to insert commas, quotation marks, and periods. If several items are to be typed, should they be typed on the same line or on separate lines? If your program asks "How many brothers and sisters do you have?" and the user has 2 brothers & 3 sisters, should he type "5" or "2,3" or "2 brothers and 3 sisters"?
```

```
If the program asks "What is your name?" should the user type "Joe Smith" or "Smith,Joe" or just "Joe"? For a quiz that asks "Who was the first U.S. President?" what if the user answers "George Washington" or simply "Washington" or "washington" or "G. Washington" or "General George Washington" or "President Washington" or "Martha's husband"? Make the instructions clearer: who was the first U.S. President (give just his last name)?
```

The user tries to joke or sabotage. Instead of inputting his name, he types an obscene comment. When asked how many brothers and sisters he has, he says 275.

Responsibility! As a programmer, it's your duty to include clear directions for using your program, and you must make the program reject ridiculous input.

For example, if your program is supposed to print weekly paychecks, it should refuse to print checks for more than \$10000. Your program should contain these lines:

```
1 input "How much money did the employee earn";e  
  if e>10000 then  
    print e; " is quite a big paycheck! I don't believe you."  
    print "Please retype your request."  
    goto 1  
  endif
```

That "if" line is called an **error trap** (or **error-handling routine**). Your program should contain several, to prevent printing checks that are too small (2¢?) or negative or otherwise ridiculous (\$200.73145?)

To see how your program reacts to input that's either garbage or unusual, **ask a friend to run your program.** That person might input something you never thought of.

Document it

Write an explanation that helps other people understand your program.

An explanation is called **documentation**. When you write an explanation, you're **documenting** the program.

You can write the documentation on a separate sheet of paper, or you can make the computer print the documentation when the user runs or lists the program.

A popular device is to begin the program by making the computer ask the user:

```
Do you need instructions?
```

You need two kinds of documentation: how to use the program, and how the program was written.

How to use the program Your explanation of using the program should include:

```
the program's name  
how to get the program from the disk  
the program's purpose  
a list of other programs that must be combined with this program, to make a workable combination  
the correct way to type the input and data (show an example)  
the correct way to interpret the output  
the program's limitations (input it can't handle, a list of error messages that might be printed, round-off error)  
a list of bugs you haven't fixed yet
```

How the program was written An explanation of how you wrote the program will help other programmers borrow your ideas, and help them expand your program to meet new situations. It should include:

```
your name  
the date you finished it  
the computer you wrote it for  
the language you wrote it in (probably BBC Basic)  
the name of the method you used ("solves quadratic equations by using the quadratic formula")  
the name of the book or magazine where you found the method  
the name of any program you borrowed ideas from  
an informal explanation of how program works ("It loops until x>y, then computes the weather forecast.")  
the purpose of each subprocedure  
the meaning of each variable  
the significance of reaching a line (for a program saying "if x<60 goto 1000", say "Reaching line 1000 means the student flunked.")
```

Python

Python is a computer language that resembles Basic. It tries to be even easier to learn than Basic, though in some ways it's harder.

Python is considered to be modern; Basic is considered to be old-fashioned. Many colleges teach students to program in Python instead of Basic.

Python was invented by a Dutchman, Guido van Rossum, in December 1989, as a hobby, to keep himself busy while his office was closed for Christmas vacation. He called it "Python" to honor the British comedy group **Monty Python's Flying Circus**. In October 2000 he invented an improved version, **Python 2**. In December 2008, he invented a further improvement, **Python 3**.

This chapter explains the current version, Python 3.10.2, which is a slight improvement on Python 3.

In your Python program, you can put these **commands** —

Command	What the computer will do	Page
break	break out of the "while" loop, stop repeating that loop	541
elif age<100:	do the indented lines when earlier conditions false and age<100	537
else:	do the indented lines when "if" conditions are false	536
for x in range(20):	repeat the indented lines, for x being 0 through 19 (less than 20)	539
if age<18:	do the indented lines if age<18	536
if age==18:	do the indented lines if age is 18	537
if age!=18:	do the indented lines if age is not 18	537
while True:	do the indented lines repeatedly	538
x=2	make x be 2	532
x+=2	make x increase by 2	533
x-=2	make x decrease by 2	533
#Zoo is fishy	ignore this comment	532

and these **functions** (which have parentheses):

Function	Value	Page
eval(input('What number? '))	whatever number the human will input	535
input('What is your name? ')	whatever name the human will input	534
int(input('What number? '))	whatever integer the human will input	535
float(input('What number? '))	the number the human will input, with decimal	535
print('I love you')	'I love you' will print onto the computer's screen	530
print(2+2)	4 will print onto the computer's screen	530
range(20)	the numbers less than 20 (0 through 19)	539

Fun

Let's have fun programming in Python! If you have any difficulty, phone me at 603-666-6644 (day or night) for free help.

Get Python

Here's how to copy Python (version 3.10.2) from the Internet to a Windows 10 or 11 computer, free (using Microsoft Edge).

Go to [Python.org](https://python.org). You see this menu bar:

About	Downloads	Documentation	Community	Success Stories	News	Events
-------	-----------	---------------	-----------	-----------------	------	--------

Tap "Downloads" then "Python 3.10.2" (or "Download Python 3.10.2") then "Open file" then "Install Now" then "Yes". The computer will say "Installing" then "Setup was successful." Tap "Close". Close the Microsoft Edge window (by clicking its X button).

Start Python

To start Python (version 3.10.2), do this:

Windows 10 In the search box (which is next to the Windows Start button), type "idle" then press Enter.
Windows 11 While you're viewing the Start menu, type "idle" then press Enter.

You see the **Python Shell window**, which is also called Python's **Integrated Development Environment (IDLE)**.

Math

In the Python Shell window, you see this Python prompt:

```
>>>
```

That prompt imitates 3 right-arrows. It points to the **writing box**. The writing box is a big box, right of that prompt, and consumes most of the Python Shell window.

In the writing box, you can type any Python command. For example, you can type 4+2, so the writing box shows this:

```
4+2
```

Try doing that: type 4+2.

After typing 4+2, press the Enter key. The Enter key makes the computer read what you typed and reply to it. The computer will reply by typing the answer, 6, like this:

```
6
```

So the writing box looks like this:

```
4+2
6
```

Below that, the computer shows the Python prompt again, so you can give another command.

If you want to subtract 3 from 7, type 7-3, so the writing box looks like this:

```
7-3
```

When you press the Enter key at the end of that line, the computer will reply:

```
4
```

You can use decimal points and negative numbers. For example, if you type -26.3+1, the computer will say:

```
25.3
```

Multiplication To multiply, use an asterisk. So to multiply 2 by 6, type this:

```
2*6
```

The computer will say:

```
12
```

Division To divide, use a slash. So to divide 8 by 4, type this:

```
8/4
```

Instead of saying the answer is 2, the computer will say —

```
2.0
```

because **whenever the computer divides, its answer includes a decimal point**.

If you try to divide by 0 (by giving a command such as 3/0 or 0/0), the computer will refuse: it will say "ZeroDivisionError".

Last digit might be wrong To divide 5 by 3, type this:

```
5/3
```

The computer will say:

```
1.6666666666666667
```

In that example, the computer gave the right answer, rounded to 17 digits. But in other calculations that have a decimal answer, the computer might accidentally say the last digit wrong. For example, suppose you say to divide 7 by 3, like this:

```
7/3
```

The computer will accidentally say:

```
2.3333333333333335
```

In that answer, the 5 should be 3 instead. Moral: **when Python makes the computer give a long decimal answer, don't trust its last digit!** (I hope Python's future versions hide that error, by showing just the 16 reliable digits and hiding the 17th digit.)

To see an even scarier example, type this:

```
.1+.2
```

The answer should be simply .3, but Python makes the computer say this:

```
.30000000000000004
```

The last digit, the 17th, should be 0, not 4.

Avoid commas Do not put commas in big numbers. To write four million, do not write 4,000,000; instead, write 4000000.

E notation If the computer's answer is tiny (less than .0001) or "a huge number containing a decimal point" (at least 1000000000000000.0), the computer will put an e in the answer. The e means "move the decimal point".

For example, suppose the computer says the answer to a problem is:

```
1.3586281902638497e+18
```

The e means "move the decimal point". The plus sign means, "toward the right". Altogether, the **e+18 means "move the decimal point toward the right, 18 places."** So look at 1.3586281902638497 and move the decimal point toward the right, 18 places; you get —

```
1358628190263849700.
```

which has the same meaning as:

```
1358628190263849700.0
```

So when the computer says the answer is 1.3586281902638497e+18, the computer really means the answer is 1358628190263849700.0, approximately. Since you can't trust the computer's last digit (the 7) and the zeros that belong after it, the exact answer might be 1358628190263849700.2 or 1358628190263849700.29 or 1358628190263849800.0 or some similar

number, but the computer says just an approximation.

Suppose your computer says the answer to a problem is:

```
9.23e-06
```

After the e, the minus sign means, "towards the left". So look at 9.23 and move the decimal point towards the left, 6 places. You get: .00000923

So when the computer says the answer is 9.23e-06, the computer really means the answer is: .00000923

You'll see e notation rarely: the computer uses it just if an answer involves decimals and tinier than .0001 or huge (at least 10 quadrillion). But when the computer *does* use e notation, remember to move the decimal point!

The highest number The highest number the computer can handle well is about 1e308, which is 1 followed by 308 zeros then a decimal point. If you try to go much higher, the computer will gripe by saying —

```
inf
```

which means "infinity".

The tiniest decimal The tiniest decimal the computer can handle well is about 1e-323, which is a decimal point followed by 323 digits (322 zeros then 1). If you try to go much tinier, the computer will give up and say just:

```
0.0
```

Order of operations What does "2 plus 3 times 4" mean? The answer depends on whom you ask.

To a clerk, it means "start with 2 plus 3, then multiply by 4"; that makes 5 times 4, which is 20. But to a scientist, "2 plus 3 times 4" means something different: it means "2 plus three fours", which is 2+4+4+4, which is 14.

Since computers were invented by scientists, computers think like scientists. If you type —

```
2+3*4
```

the computer will think you mean "2 plus three fours", so it will do 2+4+4+4 and say this answer:

```
14
```

The computer will *not* print the clerk's answer, which is 20. So if you're a clerk, tough luck!

Scientists and computers follow this rule: **do multiplication and division before addition and subtraction.** So if you type —

```
2+3*4
```

the computer begins by hunting for multiplication and division. When it finds the multiplication sign between the 3 and the 4, it multiplies 3 by 4 and gets 12, like this:

```
2+3*4
  √
  12
```

So the problem becomes 2+12, which is 14, which the computer prints.

For another example, suppose you type:

```
10-2*3+72/9*5
```

The computer begins by doing all the multiplications and divisions. So it does 2*3 (which is 6) and does 72/9*5 (which is 8.0*5, which is 40.0), like this:

```
10-2*3+72/9*5
  √      √
  6      40.0
```

So the problem becomes 10-6+40.0, which is 44.0, which is the answer the computer says:

```
44.0
```

You can use parentheses the same way as in algebra. For example, if you type —

```
5-(1+1)
```

the computer will compute 5-2 and say:

```
3
```

You can put parentheses inside parentheses. If you type —

```
10-(5-(1+1))
```

the computer will compute 10-(5-2), which is 10-3, and will say:

```
7
```

Strings

Let's make the computer fall in love. Let's make it say, 'I love you'.

To do that, type 'I love you', beginning and ending with a **single-quote mark** (which is the same mark as an apostrophe), so your screen looks like this:

```
'I love you'
```

At the end of that typing, when you press the Enter key, the computer will obey your command: it will say:

```
'I love you'
```

You can change the computer's personality. For example, if you give this command —

```
'I hate you'
```

the computer will reply:

```
'I hate you'
```

Notice that **to make the computer say a message, you must put the message between single-quote marks.** The single-quote marks make the computer copy the message without worrying about what the message means. For example, if you misspell 'I love you' and type —

```
'aieee luf ya'
```

the computer will still copy the message (without worrying about what it means); the computer will say:

```
'aieee luf ya'
```

Jargon The word ‘joy’ consists of 3 characters: j and o and y. Programmers say that the word ‘joy’ is a **string** of 3 characters.

A **string** is any collection of characters, such as ‘joy’ or ‘I love you’ or ‘aieee luf ya’ or ‘76 trombones’ or ‘GO AWAY!!!’ or ‘xypw exr//746’. The computer will say whatever string you wish, but remember to **put the string in single-quote marks**.

Strings versus numbers The computer can handle two types of expressions: **strings** and **numbers**. Put strings (such as ‘joy’ and ‘I love you’) in single-quote marks. Numbers (such as 4+2) do *not* go in single-quote marks.

Accidents Suppose you accidentally put the number 2+2 in single-quote marks, like this:

```
'2+2'
```

The single-quote marks make the computer think ‘2+2’ is a string instead of a number. Since the computer thinks ‘2+2’ is a string, it copies the string without analyzing what it means; the computer will say:

```
'2+2'
```

It will *not* say 4.

Suppose you want the computer to say the word ‘love’ but you accidentally forget to put the string ‘love’ in single-quote marks. You accidentally type this instead:

```
love
```

Since you forget to type the single-quote marks, the computer will try to figure out what you mean but will get confused, since it doesn’t know the meaning of love. Whenever the computer gets confused, it gripes by saying you have a “NameError” or “SyntaxError”.

String arithmetic You can add strings. For example, ‘fat’+‘her’ is ‘father’. So if you type —

```
'fat'+her'
```

the computer will say:

```
'father'
```

You can multiply a string by a number. For example, ‘fat’ multiplied by 3 is ‘fatfatfat’. So if you type —

```
'fat'*3
```

the computer will say:

```
'fatfatfat'
```

If you prefer, write the number *before* the string, like this:

```
3*'fat'
```

The computer will still say:

```
'fatfatfat'
```

Print

If you say **print**, the computer will print onto your screen more briefly.

For example, if you say —

```
print('I love you')
```

The computer will print this onto your screen:

```
I love you
```

The computer will *not* print single-quote marks around that reply.

After the word **print**, you must type a **parenthesis**. If you forget to type the parenthesis and put a blank space instead, the computer will say “SyntaxError: Missing parentheses”.

If you say —

```
print('love',2+2,'you')
```

the computer prints the results of ‘love’ and 2+2 and ‘you’, all on the same line of your screen but separated by spaces, like this:

```
love 4 you
```

Yes, the computer produces love 4 you.

This command makes the computer do 6+2, 6-2, 6*2, and 6/2, all at once:

```
print(6+2,6-2,6*2,6/2)
```

That makes the computer print the four answers, all on the same line:

```
8 4 12 3.0
```

The computer puts spaces between the answers.

Create a program

Here’s how to create a Python program.

At the top of the Python Shell window, you see this menu:

```
File Edit Shell Debug Options Window Help
```

Click “File” then “New File”.

You see the **program window**, called “untitled”. The program window is empty: it doesn’t contain any >>> prompt.

The program window partly covers up the Python Shell window. To make programming easier, **drag the word “untitled” toward the right**; as you do so, the entire program window moves toward the right. Keep dragging toward the right until the program window no longer overlaps the Python Shell window. (If your screen isn’t wide enough to accomplish that, just drag as far as possible.)

In the program window, type your program. For example, let’s type a program that makes the computer say:

```
make your nose
touch your toes
```

To do that, type this program:

```
print('make your nose')
print('touch your toes')
```

At the end of each line, press the Enter key.

When you’ve done all that, tap “File” (at the top of the program window) then “Save”. Invent a name for your program, such as Joe; type the name then press the Enter key. That makes the computer copy the program to drive C. (If you’re using Python 3.10.2, the program will be in drive C’s Python310 folder. If you named the program Joe, the program’s name will actually be Joe.py, because the computer automatically puts “.py” at the end of the program’s name. The “.py” means “written in Python”.)

To run the program, tap the F5 key.

(Exception: if the “F5” is blue or tiny or on a new computer by Lenovo, HP, Microsoft, or Toshiba, tap that key *while holding down the Fn key*, which is left of the Space bar.) Then, in the Python Shell window, you see the result of the program running, so you see:

```
make your nose
touch you toes
```

That writing is called the program’s **output**, since it’s what the program puts out.

Above the output, you see “==== RESTART”. Above and below the output, you see the >>> prompt, so you can give another Python command.

If you want to edit the program you wrote, **tap in the program window** (which is to the right of the Python Shell window or at least peeks out behind the the Python Shell window) or do this in the Program Shell window:

```
Tap “File” then your program’s name (such as “Joe”) then “Open”.
```

You see your program again. Make whatever changes you wish, then save the program again (by tapping “File” then “Save”), then run the program (by tapping the F5 key).

To see an old program you created, go to the Python Shell window then tap “File” then “Open” then double-tap the program’s name. You see the program’s lines. To run the program, tap the F5 key.

Warning: **in a normal program, you must say print**. For example, to make a program say the answer to 2+2, you can’t type just 2+2; instead the program must say:

```
print(2+2)
```

Saying just 2+2 is okay next to the >>> prompt, which means you’re in **interactive mode**, not in a program.

Polite versus fast To run a Python program, you must save it first. I showed you the **polite** way to do Python: save the program (by clicking “File” then “Save”) then run the program (by tapping the F5 key).

Here’s the **faster** way to run a Python program: tap the F5 key (which means you want to run the program), then watch the computer yell at you (for not saving the program first), then press the Enter key (which means you agree to save it). That’s impolite (so you get yelled at), but it’s faster than tapping “File” then “Save”.

Finish

When you finish using Python, close all windows (by clicking their X buttons).

Tricky printing

Printing can be tricky! Here are the tricks.

Indenting Suppose you want the computer to print this letter onto your screen:

```
Dear Joan,  
    Thank you for the beautiful  
necktie. Just one problem--  
I do not wear neckties!  
        Love,  
        Fred-the-Hippie
```

This program prints it:

```
print('Dear Joan,')  
print('    Thank you for the beautiful')  
print('necktie. Just one problem--')  
print('I do not wear neckties!')  
print('        Love,')  
print('        Fred-the-Hippie')
```

In the program, each line contains 2 single-quote marks. **To make the computer indent a line, put blank spaces AFTER the first single-quote mark.**

Blank lines Life consists sometimes of joy, sometimes of sorrow, and sometimes of a numb emptiness. To express those feelings, run this program:

Program	What the computer will do
print('joy')	print 'joy'
print()	print a blank empty line, under 'joy'
print('sorrow')	print "sorrow"

Altogether, the computer will print:

```
joy  
  
sorrow
```

Apostrophe An apostrophe is this symbol:

```
'
```

Many words contain apostrophes:

```
can't don't won't ain't I'll I'd I've I'm  
Jack's O'Doole gov't '60s it's let's Qur'an
```

To put an apostrophe in a string’s middle, use one of these tricks:

Backslash trick Type a backslash before the apostrophe.

Double-quote trick Enclose the string in double-quote marks instead of single-quote marks.

For example, suppose you want the computer to say:

```
We've gone to Jack's house
```

This does *not* work:

```
print('We've gone to Jack's house')
```

Instead, you must use the backslash trick (putting a backslash before each apostrophe) —

```
print('We\'ve gone to Jack\'s house')
```

or the double-quote trick (putting the string in double-quote marks instead of single-quote marks):

```
print("We've gone to Jack's house")
```

If you use the backslash trick, make sure you type a backslash (\), not a forward slash (/). The backslash key is *above* the Enter key.

New line In a string, \n means “create a **new line**”. For example, if you type —

```
print('Love\nDeath')
```

the computer will print the word Love, then create a new line (by pressing the Enter key), then print the word Death, so you see this:

```
Love  
Death
```

That’s how to make one print statement print 2 lines.

Separator If you say —

```
print('he', 'art', 'be', 'at')
```

the computer will print the 4 words and put blank spaces between them, like this:

```
he art be at
```

If instead you say —

```
print('he', 'art', 'be', 'at', sep='!')
```

the computer will print the 4 words and **separate** them with exclamation points instead of spaces, so you see this:

```
he!art!be!at
```

That’s because sep='!' means “the **separator** is an exclamation point”.

If instead you say —

```
print('he', 'art', 'be', 'at', sep='')
```

The computer will print those 4 words and separate them with nothing, so you see this:

```
heartbeat
```

If instead you say —

```
print('the boy', 'the dog', 'the car', sep=' who chased ')
```

the computer will print:

```
the boy who chased the dog who chased the car
```

End In your program, if you say —

```
print('fat')  
print('her')
```

the computer will print ‘fat’ and ‘her’ on separate lines, like this:

```
fat  
her
```

That’s because, at the end of printing ‘fat’, the computer presses the Enter key.

Suppose you say this instead:

```
print('fat', end='!')  
print('her')
```

The end='!' means:

At the **end** of printing the line, print an exclamation point instead of pressing the Enter key.

So after printing ‘fat’, the computer will print an exclamation point instead of pressing the Enter key. The computer will print:

```
fat!her
```

Suppose you say this instead:

```
print('fat', end=' ')  
print('her')
```

The end=' ' means:

At the **end** of printing the line, press the space bar instead of the Enter key.

So after printing 'fat', the computer will press the space bar instead of the Enter key. The computer will print:

```
fat her
```

Suppose you say this instead:

```
print('fat',end=' ')
print('her')
```

The end=' ' means:

At the **end** of printing the line, do *nothing* — don't press the Enter key.

So after printing 'fat', the computer *won't* press the Enter key; instead, the computer will just obey the next command, which makes the computer print 'her', so 'her' appears next to 'fat', like this:

```
father
```

Lines that aren't commands

Usually, each line you type is a command. Here's how to change that.

Semicolon To type two commands on one line, **put a semicolon between the commands**:

```
2+3; 7+1
```

The computer will say:

```
5
8
```

Backslash To type just *part* of a command on one line, put a backslash at the end of that part. Type the rest of the command on the line below.

For example, instead of typing —

```
3+6+200
```

you can type:

```
3+6+\
    200
```

(The computer automatically indents the second line for you.) The computer will say the answer:

```
209
```

Instead of typing —

```
print('I love you')
```

you can type:

```
print('I lo\
ve you')
```

(Since you put the backslash in the middle of a string, the computer does *not* indent the second line.) The computer will say:

```
I love you
```

If you want to type a command that's too long to fit on your screen, put a backslash at the end of the command's first part; underneath, type the rest of the command.

Don't put a backslash in the middle of a computer word (such as "print"). Don't put a backslash in the middle of a number (such as 57).

Comment Occasionally, jot a note to remind yourself what your program does. Slip the note into your program by putting a **hashtag** (the symbol "#") before it:

```
#This program is another dumb example, written by Russ.
#It was written on Halloween, under a full moon.
print('I love you') #because I want to date someone
```

When you run the program, **the computer ignores everything that's to the right of a hashtag**. So the computer ignores the top two lines; in the bottom line, the computer ignores the "because...". The program makes the computer print just this:

```
I love you
```

Everything to the right of a hashtag is called a **comment**. While the computer runs the program, it ignores the comments. But the comments remain part of the program; they appear in the right-hand window with the rest of the program. Though the comments appear in the program, they don't affect the run.

Variables

You can name a number. For example, you can make Joan be the name for the number 7, by typing this:

```
Joan=7
```

Then Joan+2 is 9, so if you type —

```
Joan+2
```

the computer will reply:

```
9
```

The name can be short (like Joan) or long (like PopeFrancisTheGreat) or *very* short (like x) or technical (like TemperatureOfBasementFloor) or include digits (like LeaderOfThe3Musketeers) or disgusting (like number_of_times_we_vomited).

When you invent a name, you face these restrictions:

The name must consist of just **letters, digits, and underscores**. (So no periods, apostrophes, special characters, or blank spaces.)

The name must not begin with a digit.

The name must not be one of these **keywords** (which are also called **reserved words**): and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield.

To avoid confusion, the name shouldn't be a **Python function** such as "print".

Capitalization makes a difference.

If you say x=5, the computer will know x is the name for 5 but won't know what X is yet. If you say Y=8, the computer will know Y is the name for 8 but won't know what y is yet.

Some companies (such as Google)

prohibit employees from using capital letters in names, except in special circumstances. If you work at one of those companies, make a name be:

```
x (not X)
joan (not Joan)
pope_francis_the_great (not PopeFrancisTheGreat)
leader_of_the_3_musketeers (not LeaderOfThe3Musketeers)
```

You can name *any* number. For example, you can say:

```
x=-34.1
```

Then x is -34.1, so if you type —

```
x*2
```

the computer will multiply -34.1 by 2 and say:

```
-68.2
```

You can name any string. For example, you can say:

```
y='go'
y*3
```

Then the computer will multiply 'go' by 3 and say:

```
'gogogo'
```

Beginners are usually too lazy to type long names, so beginners use names that are short (such as x). But when you become a pro and write a long, fancy program containing hundreds of lines and hundreds of names, you should use long names to help you remember each name's purpose. A name can be as long as you wish. In this book, I'll use short names in short programs (so you can type those programs quickly) but long names in long programs (so you can keep track of which name is which).

Jargon

A **name** (for a number or a string) is also called an **identifier**. It's also called a **variable**.

For example, suppose you say:

```
Joan=7
```

That line makes Joan be a **name**, an **identifier**, a **variable**, whose **value** is 7. Since that line **assigns 7 to Joan**, that line is called an **assignment statement**. That line **defines Joan** to be 7.

If a variable (such as Joan) stands for a number, it's called a **numeric variable**. If a variable stands for a string instead, it's called a **string variable**.

Restart

When you run a program, the computer begins by doing a **restart**, which makes it forget any names you invented previously, so the program can start fresh.

After the program has run, the computer still remembers the names in the program. For example, if your program said Joan=7, the computer knows Joan is 7 even after the program

has finished running, so if you say —

```
Joan
```

The computer will say:

```
7
```

A variable is a box

When you say Joan=7, here's what happens inside the computer.

The computer's random-access memory (RAM) consists of electronic boxes. The line Joan=7 makes the computer create a box named Joan and put 7 into it, like this:

```
box Joan [ 7 ]
```

Then when the computer encounters print(x+2), the computer prints what's in box Joan, plus 2; so the computer prints 9.

Variable from variables

One variable can define another. For example, suppose you type:

```
n=6
d=n+1
n*d
```

The top line says n is 6. The next line says d is n+1, which is 6+1, which is 7; so d is 7. The bottom line says to print n*d, which is 6*7, which is 42; so the computer will print:

```
42
```

Change a value

A value can change:

```
k=4
k=9
k*2
```

The top line says k's value is 4. The next line changes k's value to 9, so the bottom line prints 18.

When you run that program, here's what happens inside the computer's RAM. The top line (k=4) makes the computer put 4 into box k:

```
box k [ 4 ]
```

The next line (k = 9) puts 9 into box k. The 9 replaces the 4:

```
box k [ 9 ]
```

That's why k*2 prints 18.

Self-changing variable A variable can change itself. Look at this program:

```
x=7
x=x+2
print(x)
```

The top line (x=7) says x starts by being 7. The next line (x=x+2) means: the new x is "what x was before, plus 2". So the new x is 7+2, which is 9. The bottom line prints:

```
9
```

Let's examine that program more closely. The top line (x=7) puts 7 into box x:

```
box x [ 7 ]
```

When the computer sees the next line (x=x+2), it examines the equation's right side and sees the x+2. Since x is 7, the x+2 is 7+2, which is 9. So the line "x=x+2" means x=9. The computer puts 9 into box x:

```
box x [ 9 ]
```

The program's bottom line prints 9.

Instead of typing —

```
x=x+2
```

you can type this shortcut:

```
x+=2
```

You can pronounce "x+=2" this way:

```
x gets added this amount: 2
```

Here's another weirdo:

```
b=6
b+=1
print(b*2)
```

The second line (b+=1) says the new b gets added this amount: 1. So the new b is 6+1, which is 7. The bottom line prints:

```
14
```

In that program, the top line says b is 6; but the next line increases b, by adding 1; so b becomes 7. Programmers say that b has been **increased** or **incremented**. In the second line, the "1" is called the **increase** or the **increment**.

The opposite of "increment" is **decrement**:

```
j=500
j-=1
print(j)
```

The top line says j starts at 500; but the next line decreases j by subtracting 1, so the new j is 500-1, which is 499. The bottom line prints:

```
499
```

In that program, j was **decreased** (or **decremented**). In the second line, the "1" is called the **decrease** (or **decrement**).

Hassles

Variables can cause hassles.

Undefined variable If you type —

```
Joan
```

the computer tries to say Joan's value (a number or string). If the computer fails (because you forgot to write a line such as Joan=7), the computer says "NameError".

What's before the equal sign?

When writing an equation (such as x=47), put this before the equal sign: the name of just one box (such as x). So **before the equal sign, put one variable**:

```
Allowed:    d=n+1 (d is one variable)
Not allowed: d-n=1 (d-n is two variables)
Not allowed: 1=d-n (1 is not a variable)
```

The variable on the equation's *left* side is the only one that changes. For example, the statement d=n+1 changes the value of d but not n. The statement b=c changes the value of b but not c:

```
b=1
c=7
b=c
b+c
```

The third line changes b, to make it equal c; so b becomes 7. Since both b and c are now 7, the bottom line prints 14.

"b=c" versus "c=b" Saying "b=c" has a different effect from "c=b". That's because "b=c" changes the value of b (but not c); saying "c=b" changes the value of c (but not b).

Compare these programs:

```
b=1          b=1
c=7          c=7
b=c          c=b
print(b+c)  print(b+c)
```

In the left program, the third line changes b to 7, so both b and c are 7. The bottom line prints 14.

In the right program, the third line changes c to 1, so both b and c are 1. The bottom line prints 2.

While you run those programs, here's what happens inside the computer's RAM. For both programs, the second and third lines do this:

```
box b [ 1 ]
box c [ 7 ]
```

In the left program, the third line makes the number in box b become 7 (so both boxes contain 7, and the bottom line prints 14). In the right program, the third line makes the number in box c become 1 (so both boxes contain 1, and the bottom line prints 2).

When to use variables

Here's a practical example of when to use variables.

Suppose you're selling something that costs \$1297.43, and you want to do these calculations:

```
multiply $1297.43 by 2
multiply $1297.43 by .05
add $1297.43 to $483.19
divide $1297.43 by 37
```

To do those four calculations, you could run this program:

```
print(1297.43*2,1297.43*.05,1297.43+483.19,1297.43/37)
```

But that program's silly, since it contains the number 1297.43 four times. This program's briefer, because it uses a variable:

```
c=1297.43
print(c*2,c*.05,c+483.19,c/37)
```

So whenever you need to use a number several times, turn the number into a variable, which will make your program briefer.

Paranoid If you're paranoid, you'll love this program:

```
t='They're laughing at you!'
print(t)
print(t)
print(t)
```

The top line says t stands for the string 'They're laughing at you!' The later lines make the computer print:

```
They're laughing at you!
They're laughing at you!
They're laughing at you!
```

Nursery rhymes The computer can recite nursery rhymes:

```
p='Peas porridge'
print(p, 'hot!')
print(p, 'cold!')
print(p, 'in the pot,')
print('Nine days old!')
```

The top line says p stands for 'Peas porridge'. The later lines make the computer print:

```
Peas porridge hot!
Peas porridge cold!
Peas porridge in the pot,
Nine days old!
```

This program prints a fancier rhyme:

```
h='Hickory, dickory, dock!'
m='THE MOUSE (squeak! squeak!)'
c='THE CLOCK (tick! tock!)'
print(h)
print(m, 'ran up',c)
print(c, 'struck one')
print(m, 'ran down')
print(h)
```

Lines 1-3 define h, m, and c. The later lines make the computer print:

```
Hickory, dickory, dock!
THE MOUSE (squeak! squeak!) ran up THE CLOCK (tick! tock!)
THE CLOCK (tick! tock!) struck one
THE MOUSE (squeak! squeak!) ran down
Hickory, dickory, dock!
```

Input

Humans ask questions; so to turn the computer into a human, you must make it ask questions too. **To make the computer ask a question, use the word "input"**.

This program makes the computer ask for your name:

```
n=input('What is your name? ')
print('I adore anyone whose name is',n)
```

The top line says n is the answer to the question 'What is your name?' When you run the program and the computer sees that line, the computer asks 'What is your name?' then waits for you to answer the question; your answer will be called n. For example, if you answer Maria, then n is 'Maria'. The bottom line makes the computer print:

```
I adore anyone whose name is Maria
```

When you run that program, here's the whole conversation that occurs between the computer and you; I've underlined the part typed by you....

```
Computer asks for your name: What is your name? Maria
Computer praises your name: I adore anyone whose name is Maria
```

Go ahead, type that program and run it, but be careful: **when you type the input line, leave a space after the question mark.**

Just for fun, run that program again and pretend you're somebody else....

```
Computer asks for your name: What is your name? Bud
Computer praises your name: I adore anyone whose name is Bud
```

When the computer asks for your name, if you say something weird, the computer gives you a weird reply....

```
Computer asks: What is your name? none of your business!
Computer replies: I adore anyone whose name is none of your business!
```

That program begins by making the computer ask:

```
What is your name?
```

You can make the computer say this instead:

```
Enter your name:
```

To do so, change the program's top line to this:

```
n=input('Enter your name: ')
```

The program's bottom line makes the computer reply like this:

```
I adore anyone whose name is Maria
```

You can make the computer add an exclamation point, like this:

```
I adore anyone whose name is Maria!
```

To do that, change the program's bottom line to this:

```
print('I adore anyone whose name is',n+'!')
```

The '+' means:

```
add an exclamation point, with no space before the exclamation point
```

College admissions

This program prints a letter, admitting you to the college of your choice:

```
c=input('What college would you like to enter? ')
print('Congratulations!')
print('You have just been admitted to',c)
print('because it fits your personality.')
print('I hope you go to',c+'.')
print('      Respectfully yours,')
print('      The Dean of Admissions')
```

When the computer sees the input line, the computer asks 'What college would you like to enter?' and waits for you to answer. Your answer will be called c. If you'd like to be admitted to Harvard, you'll be pleased....

```
Computer asks you: What college would you like to enter? Harvard
Computer admits you: Congratulations!
You have just been admitted to Harvard
because it fits your personality.
I hope you go to Harvard.
      Respectfully yours,
      The Dean of Admissions
```

The program's 5th line includes these symbols:

```
+'.'
```

Those symbols mean:

```
add a period, with no space before the period
```

You can choose any college you wish:

```
Computer asks you: What college would you like to enter? Hell
Computer admits you: Congratulations!
You have just been admitted to Hell
because it fits your personality.
I hope you go to Hell.
      Respectfully yours,
      The Dean of Admissions
```

That program consists of three parts:

1. The computer begins by asking you a question ('What college would you like to enter?'). The computer's question is called the **prompt**, because it prompts you to answer.
2. Your answer (the college's name) is called **your input**, because it's information that you're *putting into* the computer.
3. The computer's reply (the admission letter) is called the **computer's output**, because it's the final answer that the computer puts out.

Input versus print

The word **input** is the opposite of the word **print**.

The word **print** makes the computer print information out. The word **input** makes the computer take information in.

What the computer prints out is called the **output**. What the computer takes in is called **your input**.

Input and Output are collectively called **I/O**, so the input and print statements are called **I/O statements**.

Once upon a time

Let's make the computer write a story, by filling in the blanks:

Once upon a time, there was a youngster named _____
your name

who had a friend named _____.
friend's name

_____ wanted to _____,
your name verb (such as "pat") friend's name

but _____ didn't want to _____!
friend's name verb (such as "pat") your name

Will _____ ?
your name verb (such as "pat") friend's name

Will _____ ?
friend's name verb (such as "pat") your name

To find out, come back and see the next exciting episode
of _____ and _____!
your name friend's name

To write the story, the computer must ask for your name, your friend's name, and a verb. To make the computer ask, your program must say **input**:

```
y=input('What is your name? ')
f=input('What is the name of your friend? ')
v=input('In 1 word, say something you can do to your friend? ')
```

Then make the computer print the story:

```
print('Here is my story...')
print('Once upon a time, there was a youngster named',y)
print('who had a friend named',f+'.')
print(y,'wanted to',v,f+',')
print('but',f,'did not want to',v,y+'!')
print('Will',y,v,f+'?')
print('Will',f,v,y+'?')
print('To find out, come back and see the next exciting episode')
print('of',y,'and',f+'!')
```

Here's a sample run:

```
What is your name? Dracula
What is the name of your friend? Madonna
In 1 word, say something you can do to your friend? bite
Here is my story...
Once upon a time, there was a youngster named Dracula
who had a friend named Madonna.
Dracula wanted to bite Madonna,
but Madonna did not want to bite Dracula!
Will Dracula bite Madonna?
Will Madonna bite Dracula?
To find out, come back and see the next exciting episode
of Dracula and Madonna!
```

Here's another run:

```
What is your name? Superman
What is the name of your friend? King Kong
In 1 word, say something you can do to your friend? tickle
Here is my story...
Once upon a time, there was a youngster named Superman
Who had a friend named King Kong.
Superman wanted to tickle King Kong,
but King Kong did not want to tickle Superman!
Will Superman tickle King Kong?
Will King Kong tickle Superman?
To find out, come back and see the next exciting episode
of Superman and King Kong!
```

Try it: put in your own name, the name of your friend, and something you'd like to do to your friend.

Numeric input

To let you input a **number** (instead of a string), your program should say "**eval(input)**" instead of just "input".

For example, this program lets you input a number and makes the computer double it:

```
n=eval(input('What number will you give me? '))
print('That number doubled is',n*2)
```

The top line says *f* is the answer, **evaluated** as a number, to the question 'What number will you give me?' When you run the program and the computer sees that line, the computer asks 'What number will you give me?' then waits for you to answer the question; your number will be called *n*. For example, if you say 3, then *n* is 3. The bottom line makes the computer print:

```
That number doubled is 6
```

When you run that program, here's the whole conversation that occurs between the computer and you; I've underlined the part typed by you...

```
Computer asks for a number: What number will you give me? 3
Computer doubles it: That number doubled is 6
```

Go ahead, type that program and run it, but be careful: **at the end of the eval line, type TWO parentheses.**

In that program, the **eval** tells the computer you'll input a **number**.

If you leave out the **eval**, the computer will assume you'll input a string instead of a number. Then if you input 3, the computer will assume you mean the string '3', so the computer will double it (by repeating it) and say 33.

If you say **int** instead of **eval**, the computer will assume you'll input an **integer** (a number that has no decimal point). Then if you input a number containing a decimal point, the computer will say "ValueError".

If you say **float** instead of **eval**, the computer will assume you'll input a **floating-point number** (a number that has a decimal point). Then if you input 3, the computer will assume you mean 3.0, so the computer will double it and say 6.0 (instead of just 6).

This program makes the computer predict your future:

```
print('I predict what will happen to you in the year 2030!')
y=eval(input('In what year were you born? '))
print('In the year 2030, you will turn',2030-y,'years old.')
```

Here's a sample run:

```
I predict what will happen to you in the year 2030!
In what year were you born? 1972
In the year 2030, you will turn 58 years old.
```

Suppose you're selling tickets to a play. Each ticket costs \$2.79. (You decided \$2.79 would be a nifty price, because the cast has 279 people.) This program finds the price of multiple tickets:

```
t=eval(input('How many tickets? '))
print('The total price is $',t*2.79)
```

This program tells you how much the "oil crisis" costs you, when you drive your car:

```
m=eval(input('How many miles do you want to drive? '))
p=eval(input('How many pennies does a gallon of gas cost? '))
r=eval(input('How many miles-per-gallon does your car get? '))
print('The gas for your trip will cost $',m*p/(r*100))
```

Here's a sample run:

```
How many miles do you want to drive? 400
How many pennies does a gallon of gas cost? 257.9
How many miles-per-gallon does your car get? 31
The gas for your trip will cost $ 33.277419354838706
```

So the gas will cost a hair less than \$33.28.

Conversion

This program converts feet to inches:

```
f=eval(input('How many feet? '))
print(f, 'feet =', f*12, 'inches')
```

Here's a sample run:

```
How many feet? 2
2 feet = 24 inches
```

Trying to convert to the metric system? This program converts inches to centimeters:

```
i=eval(input('How many inches? '))
print(i, 'inches =', i*2.54, 'centimeters')
```

Nice day today, isn't it? This program converts the temperature from Celsius to Fahrenheit:

```
c=eval(input('How many degrees Celsius? '))
print(c, 'degrees Celsius =', c*1.8+32, 'degrees Fahrenheit')
```

Here's a sample run:

```
How many degrees Celsius? 20
20 degrees Celsius = 68.0 degrees Fahrenheit
```

See, you can write the *Guide* yourself! Just hunt through any old math or science book, find any old formula (such as $f=c*1.8+32$), and turn it into a program.

If

Let's write a program so if the human is less than 18 years old, the computer will say:

```
You are still a minor.
```

Here's the program:

```
age=eval(input('How old are you? '))
if age<18: print('You are still a minor')
```

The top line makes the computer ask "How old are you?" and wait for the human to type an age. Since **the symbol for "less than" is "<"**, the bottom line says: if the age is less than 18, print "You are still a minor".

Go ahead! Run that program! The computer begins the conversation by asking:

```
How old are you?
```

Try saying you're 12 years old, by typing a 12, so the screen looks like this:

```
How old are you? 12
```

When you finish typing the 12 and press the Enter key at the end of it, the computer will reply:

```
You are still a minor
```

Try running that program again, but this time try saying you're 50 years old instead of 12, so the screen looks like this:

```
How old are you? 50
```

When you finish typing the 50 and press the Enter key at the end of it, the computer will *not* say "You are still a minor". Instead, the computer will say *nothing* — since we didn't teach the computer how to respond to adults yet!

In that program, the bottom line says:

```
if age<18: print('You are still a minor')
```

That line begins with the word "if". **Whenever you say "if", you must also write a colon** (the symbol ":").

What comes between "if" and the colon is called the **condition**. In that example, the condition is "age<18". If the condition is true (if age is really less than 18), the computer does the **action**, which comes after the colon and is:

```
print('You are still a minor')
```

Else

Let's teach the computer how to respond to adults.

Here's how to program the computer so if the age is less than 18, the computer will say "You are still a minor", but if the age is *not* less than 18 the computer will say "You are an adult" instead:

```
age=eval(input('How old are you? '))
if age<18: print('You are still a minor')
else: print('You are an adult')
```

In programs, **the word "else" means "otherwise"**. That program's 2nd and 3rd lines mean: if the age is less than 18, then print "You are still a minor"; otherwise (if the age is *not* less than 18), print "You are an adult". So the computer will print "You are still a minor" or else print "You are an adult", depending on whether the age is less than 18.

Try running that program! If you say you're 50 years old, so the screen looks like this —

```
How old are you? 50
```

the computer will reply by saying:

```
You are an adult
```

Multi-line

If the age is less than 18, here's how to make the computer print "You are still a minor" and also print "Ah, the joys of youth":

```
if age<18: print('You are still a minor'); print('Ah, the joys of youth')
```

Type that correctly: put a **colon** after "if age<18" but a **semicolon** between the two print statements.

Here's a more sophisticated way to say the same thing:

```
if age<18:
    print('You are still a minor')
    print('Ah, the joys of youth')
```

That sophisticated way (in which you type 3 short lines instead of a single long line) is called a **multi-line "if"** (or a **block "if"**).

Here's how to type that **multi-line "if"**:

Type the top line (which begins with "if" and ends with a colon). After you type the colon, press the Enter key.

When the computer sees you typed a colon and then pressed the Enter key, the computer knows you're trying to create a multi-line, so the computer automatically indents the next line for you. When you finish typing that line (which says to print 'You are still a minor'), press the Enter key again. The computer automatically indents the next line for you.

The computer will indent every line you type, until you begin a line by pressing the Backspace key, which tells the computer to stop indenting. Pressing the Backspace key makes the computer unindent that line.

The indented lines are called the **block**. The line above them, which ends in a colon, is called the block's **header**.

You can also create a **multi-line "else"**, so your program looks like this:

```
age=eval(input('How old are you? '))
if age<18:
    print('You are still a minor')
    print('Ah, the joys of youth')
else:
    print('You are an adult')
    print('We can have adult fun')
```

That means: if the age is less than 18, print 'You are still a minor' and 'Ah, the joys of youth'; otherwise (if age *not* under 18) print 'You are an adult' and 'We can have adult fun'.

Elif

Let's say this:

```
If age is under 18, print "You are a minor".
If age is not under 18 but is under 100, print "You are a typical adult".
If age is not under 100 but is under 120, print "You are a centenarian".
If age is not under 120, print "You are a liar".
```

Here's another way to say the same thing, in English:

```
If age is under 18, print "You are a minor".
Otherwise, if age is under 100, print "You are a typical adult".
Otherwise, if age is under 120, print "You are a centenarian".
Otherwise, print "You are a liar".
```

The Python word for "otherwise" is "else", and the Python word for "otherwise if" is **"elif" (which is short for "else if")**, so the Python program is:

```
if age<18: print('You are a minor')
elif age<100: print('You are a typical adult')
elif age<120: print('You are a centenarian')
else: print('You are a liar')
```

Double equal sign

Suppose you want to say:

```
If age is 25
```

Here's how to say that:

```
if age==25:
```

Python doesn't let the "if" condition have a simple equal sign (=); instead you must type a **double equal sign (==)**. If you accidentally type a single equal sign there, the computer will say "SyntaxError".

Therapist Let's turn your computer into a therapist!

To make the computer ask the patient, "How are you?", begin the program like this:

```
feeling=input('How are you? ')
```

Make the computer continue the conversation by responding this way:

```
If the patient says "fine", print "That's good!"
If the patient says "lousy" instead, print "Too bad!"
If the patient says anything else instead, print "I feel the same way!"
```

Here's how:

```
if feeling=='fine': print('That\'s good!')
elif feeling=='lousy': print('Too bad!')
else: print('I feel the same way!')
```

Here's a complete program:

```
feeling=input('How are you? ')
if feeling=='fine': print('That\'s good!')
elif feeling=='lousy': print('Too bad!')
else: print('I feel the same way!')
print('I hope you enjoyed your therapy. Now you owe $50.')
```

The top line makes the computer ask the patient, "How are you?" The next several lines makes the computer analyze the patient's answer and print 'That's good!' or 'Too bad!' or else 'I feel the same way!' Regardless of what the patient and computer said, that program's bottom line always makes the computer end the conversation by printing:

```
I hope you enjoyed your therapy. Now you owe $50.
```

In that program, try changing the strings to make the computer print smarter remarks, become a better therapist, and charge even more money.

Fancy "if" conditions

Different relations You can make the "if" clause very fancy:

"if" clause	Meaning
if b<4	If b is less than 4
if b>4	If b is greater than 4
if b<=4	If b is less than or equal to 4
if b>=4	If b is greater than or equal to 4
if b==4	If b is 4
if b=='fine'	If b is the word 'fine'
if b!=4	If b does not equal 4
if b!='fine'	If b does not equal the word 'fine'
if b<'fine'	If b is a word that comes before 'fine' in dictionary
if b>'fine'	If b is a word that comes after 'fine' in dictionary

In the "if" clause, the symbols <, >, <=, >=, ==, and != are called **relations**.

Or The computer understands the word "or". For example, here's how to say, "If x is either 7 or 8, print the word *wonderful*":

```
if x==7 or x==8: print('wonderful')
```

That example is composed of two conditions: the first condition is "x==7"; the second condition is "x==8". Those two conditions combine, to form "x==7 or x==8", which is called a **compound condition**.

If you use the word "or", put it between two conditions.

```
Right: if x==7 or x==8: print('wonderful')
Right because "x==7" and "x==8" are conditions.
```

```
Wrong: if x==7 or 8: print('wonderful')
Wrong because "8" is not a condition.
```

And The computer understands the word "and". Here's how to say, "If p is more than 5 and less than 10, print *tuna fish*":

```
if p>5 and p<10: print('tuna fish')
```

Here's how to say, "If s is at least 60 and less than 65, print *you almost failed*":

```
if s>=60 and s<65: print('you almost failed')
```

Here's how to say, "If n is a number from 1 to 10, print *that's good*":

```
if n>=1 AND n<=10: print('that's good')
```

Loops

You can make the computer repeat, again and again. Something repeated is called a **loop**. Here's how to create a loop.

While True

This program makes the computer print the word "love" once:

```
print('love')
```

This fancier program makes the computer print the word "love" *three* times:

```
print('love')
print('love')
print('love')
```

When you run that program, the computer will print:

```
love
love
love
```

Let's make the computer print the word "love" *many* times. To do that, we must make the computer do this line many times:

```
print('love')
```

To make the computer do the line many times, say "while True" above the line, so the program looks like this:

```
while True:
    print('love')
```

Here's how to type that program:

Delete any lines you typed previously, so you can start fresh.

Type the top line (which begins with "while" and ends with a colon). After you type the colon, press the Enter key.

When the computer sees you typed a colon and then pressed the Enter key, the computer knows you're trying to create a multi-line, so the computer automatically indents the next line for you. When you finish typing that line (which says to print 'love'), press the Enter key again. (The computer will automatically indent any extra lines you type, until you begin a line by pressing the Backspace key, which tells the computer to stop indenting.)

When you run that program, the computer will print "love" many times, so it will print:

```
love
love
love
love
love
love
etc.
```

The computer will print "love" on every line of the Python Shell window.

But even when that window is full of "love", the computer won't stop: the computer will try to print even more loves onto your window! The computer will lose control of itself and try to devote its entire life to making love! The computer's mind will spin round and round, always circling back to the thought of making love again!

Since the computer's thinking keeps circling back to the same thought, the computer is said to be in a **loop**. In that program, the "while True" means "do repeatedly what's indented". The "while True" and the indented lines underneath form a loop, called a "**while loop**".

To stop the computer's lovemaking madness, you must give the computer a "jolt" that will put it out of its misery and get it out of the loop. To jolt the computer out of the program, **abort** the program. To abort the program, do this: while holding down the Control key (which says "Ctrl" on it), tap the C key. That makes the computer stop running your program; it will **break out of your program**; it will **abort your program** and say "KeyboardInterrupt".

In that program, since the computer tries to go round and round the loop forever, the loop is called **infinite**. The only way to stop an infinite loop is to abort it.

Cats and dogs Run this program:

```
while True:
    print('cat')
    print('dog')
```

The computer will repeatedly print 'cat' and 'dog', so the screen will look like this:

```
cat
dog
cat
dog
cat
dog
cat
dog
etc.
```

Yes, on the screen it will be raining cats and dogs! The computer will keep printing "cat" and "dog" until you abort the program.

Interactive mode Instead of creating that program (which requires you to press the F5 key to run), you can create the cats and dogs by typing in **interactive mode**, at the >>> prompt, like this:

```
while True:
    print('cat')
    print('dog')
```

When you create the blank line under 'dog' (by pressing the Enter key again), the computer performs the while loop and prints lots of cats and dogs, until you abort. Here's why:

In interactive mode, a blank line means "perform the loop now".

Conversions This program, which you saw before, converts feet to inches:

```
f=eval(input('How many feet? '))
print(f,'feet =',f*12,'inches')
```

Here's a sample run:

```
How many feet? 2
2 feet = 24 inches
```

Suppose you want to:

```
convert 2 feet to inches
and also convert 7 feet to inches
and also convert 1000 feet to inches
and also convert 59.2 feet to inches
and also convert 3 feet to inches
and also convert 5280 feet to inches
and also convert other quantities of feet to inches
```

To do all that, you can run that conversion program many times: each time you want to run that conversion program, say "run" (by pressing the F5 key). But instead of pressing the F5 key so many times, you can make the computer rerun the program for you automatically! To do that, begin your program by typing:

```
while True:
```

That means: automatically do, repeatedly, the indented lines underneath. Type those indented lines, so the program becomes like this:

```
while True:
    f=eval(input('How many feet? '))
    print(f,'feet =',f*12,'inches')
```

When you run that program (by pressing the F5 key once), the computer will repeatedly convert feet to inches, each time asking you 'How many feet?' The computer will keep converting feet to inches until you abort the program.

Counting Suppose you want the computer to count, starting at 3, like this:

```
3
4
5
6
7
8
etc.
```

This program does it, by a special technique:

```
c=3
while True:
    print(c)
    c+=1
```

In that program, c is called the **counter**, because it helps the computer count.

The top line says c starts at 3. The "while" loop says to repeatedly do this:

```
print c, then increase c by adding 1 to it
```

So the computer prints *c* (which is 3), then increases *c* (so *c* becomes 4), then repeats the indented lines again, so the computer prints the new *c* (which is 4), then increases *c* again (so *c* becomes 5), then repeats the indented lines again, so the computer prints the new *c* (which is 5), then increases *c* again (so *c* becomes 6), etc. The computer repeatedly prints *c* and increases it. Altogether, the computer prints:

```
3
4
5
6
7
8
9
10
11
etc.
```

The program's an infinite loop: the computer will print 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on, forever, unless you abort it.

Here's the general procedure to make the computer count:

- Start *c* at some value (such as 3).
- Then write a "while" loop.
- In the "while" loop, make the computer use *c* (such as by saying to print *c*), and increase *c* (by saying *c*+1).

To read the printing more easily, say `end=' '`:

```
c=3
while True:
    print(c,end=' ')
    c+=1
```

That makes the computer print horizontally:

```
3 4 5 6 7 8 etc.
```

This program makes the computer count, starting at 1:

```
c=1
while True:
    print(c,end=' ')
    c+=1
```

The computer will print 1, 2, 3, 4, etc.

This program makes the computer count, starting at 0:

```
c=0
while True:
    print(c,end=' ')
    c+=1
```

The computer will print 0, 1, 2, 3, 4, etc.

For

Let's make the computer print every number from 0 to 19, like this:

```
0
1
2
3
4
5
6
7
etc.
19
```

Here's the program:

```
for x in range(20):
    print(x)
```

The top line says *x* will be every number under 20; so *x* will be 0, then 1, then 2, etc. The line underneath, which is indented, says what to do about each *x*; it says to print each *x*.

The computer will do the indented line repeatedly, so the computer will repeatedly print(*x*). To begin, *x* will be 0, so the computer will print:

```
0
```

The next time the computer prints *x*, the *x* will be 1, so the computer will print:

```
1
```

The computer will print every number from 0 up to 19. It will not print 20.

How to start at 1 If you want to print every number from 1 to 20, say this instead:

```
for x in range(1,21):
    print(x)
```

That makes the computer start at 1 (instead of 0) and print the numbers under 21, like this:

```
1
2
3
4
5
etc.
20
```

Interactive mode Instead of creating that program (which requires you to press the F5 key to run), you can type in **interactive mode**, at the `>>>` prompt, like this:

```
>>> for x in range(1,21):
    print(x)
```

When you create the blank line under print(*x*) (by pressing the Enter key again), the computer performs the loop and prints the numbers under 21, starting at 1.

When men meet women Let's make the computer print these lyrics:

```
I saw 2 men
meet 2 women.
Tra-la-la!

I saw 3 men
meet 3 women.
Tra-la-la!

I saw 4 men
meet 4 women.
Tra-la-la!

I saw 5 men
meet 5 women.
Tra-la-la!

They all had a party!
Ha-ha-ha!
```

To do that, type these lines —

```
The first line of each verse: print('I saw',x,'men')
The second line of each verse: print('meet',x,'women.')
The third line of each verse: print('Tra-la-la!')
Blank line under each verse: print()
```

after making *x* be every number from 2 up to 5 (so *x* starts at 2 but stays less than 6):

```
for x in range(2,6):
    print('I saw',x,'men')
    print('meet',x,'women.')
    print('Tra-la-la!')
    print()
```

At the end of the song, print the closing couplet:

```
for x in range(2,6):
    print('I saw',x,'men')
    print('meet',x,'women.')
    print('Tra-la-la!')
    print()
print('They all had a party!')
print('Ha-ha-ha!')
```

(The computer automatically indents every line under "for", until you begin a line by pressing the Backspace key, which tells the computer to stop indenting.) That program makes the computer print the entire song.

Here's an analysis:

```
for x in range(2,6):
The computer will do the      print('I saw',x,'men')
indented lines repeatedly,    print('meet',x,'women.')
for x=2, x=3, x=4, and x=5.   print('Tra-la-la!')
                                print()
Then the computer will        print('They all had a party!')
print this couplet once.      print('Ha-ha-ha!')
```

Since the computer does the indented lines repeatedly, those lines form a loop. Here's the general rule: **the statements indented under "for" form a loop**. The computer goes round and round the loop, for *x*=2, *x*=3, *x*=4, and *x*=5. Altogether, it goes around the loop 4 times, which is a finite number. Therefore, the loop is **finite**.

If you don't like the letter x, choose a different letter. For example, you can choose the letter i:

```
for i in range(2,6):
    print('I saw',i,'men')
    print('meet',i,'women.')
    print('Tra-la-la!')
    print()
print('They all had a party!')
print('Ha-ha-ha!')
```

When using the word “for”, most programmers prefer the letter i; most programmers say “for i” instead of “for x”. Saying “for i” is an “old tradition”. Following that tradition, the rest of this book says “for i” (instead of “for x”), except in situations where some other letter feels more natural.

Print the squares To find the **square** of a number, multiply the number by itself. The square of 3 is “3 times 3”, which is 9. The square of 4 is “4 times 4”, which is 16.

Let's make the computer print the square of 3, 4, 5, etc., up to 20, like this:

```
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
etc.
The square of 20 is 400
```

To do that, type this line —

```
print('The square of',i,'is',i*i)
```

and make i be every number from 3 up to 20 (so below 21), like this:

```
for i in range(3,21):
    print('The square of',i,'is',i*i)
```

Count how many copies This program, which you saw before, prints “love” on every line of your screen:

```
while True:
    print('love')
```

That program prints “love” again and again, until you abort the program by pressing Ctrl with C.

But what if you want to print “love” just 20 times? This program prints “love” 20 times —

```
for i in range(20):
    print('love')
```

because it makes i be 0 then 1 then 2 then 3, etc., up to 19.

As you can see, “for” resembles “while” but is more powerful: “for” makes the computer count!

Count to midnight This program makes the computer count to midnight:

```
for i in range(1,12):
    print(i)
print('midnight')
```

The computer will print:

```
1
2
3
4
5
6
7
8
9
10
11
midnight
```

At the end of the indented line, let's say end=' ', like this:

```
for i in range(1,12):
    print(i,end=' ')
print('midnight')
```

That makes the computer print each item on the same line and separated by spaces, like this:

```
1 2 3 4 5 6 7 8 9 10 11 midnight
```

If you want the computer to press the Enter key before “midnight”, say \n:

```
for i in range(1,12):
    print(i,end=' \n')
print('\nmidnight')
```

That extra print line makes the computer press the Enter key just before “midnight”, so the computer will print “midnight” on a separate line, like this:

```
1 2 3 4 5 6 7 8 9 10 11
midnight
```

Let's make the computer count to midnight 3 times, like this:

```
1 2 3 4 5 6 7 8 9 10 11
midnight
1 2 3 4 5 6 7 8 9 10 11
midnight
1 2 3 4 5 6 7 8 9 10 11
midnight
```

To do that, indent the entire program under the word “for”:

```
for j in range(3):
    for i in range(1,12):
        print(i,end=' ')
    print('\nmidnight')
```

That version contains a loop inside a loop: the loop that says “for i” is inside the loop that says “for j”. The j loop is called the **outer loop**; the i loop is called the **inner loop**. The inner loop's variable must differ from the outer loop's. Since we called the inner loop's variable “i”, the outer loop's variable must *not* be called “i”; so I picked the letter j instead.

Programmers often think of the outer loop as a bird's nest, and the inner loop as an egg *inside the nest*. So programmers say the inner loop is **nested in** the outer loop; the inner loop is a **nested loop**.

Step size The “for” statement can be varied.

If you say —

```
for i in range(5,18,3):
```

the i will start at 5, stay under 18, and keep increasing by 3.

```
So i will be 5,
then 8 (because i increased by 3),
then 11 (because i increased by 3 again),
then 14 (because i increased by 3 again),
then 17 (because i increased by 3 again),
then stop (because i must stay under 18).
```

In that example:

The 5 is called the **start value**.

The 18 is called the **stop value**.

The 3 is called the **increase** (or **increment** or **step size**).

The i is called the **counter** (or **index** or **loop-control variable**).

Although 18 is the stop value, 17 is the **last value** (or **terminal value**).

Programmers usually say “for i”, instead of “for x”, because the letter i reminds them of the word **index**.

If you say —

```
for i in range(17,4,-3):
```

the i will start at 17, not get to 4, and keep decreasing by 3.

```
So i will be 17,
then 14 (because i decreased by 3),
then 11 (because i decreased by 3 again),
then 8 (because i decreased by 3 again),
then 5 (because i decreased by 3 again),
then stop (because i must not get to 4 or beyond).
```

In that example, 3 is called the **decrease** (or **decrement**); -3 is the **step size**.

To count *down*, you must say a *negative* step size, such as -3 or -1. So to count every number from 17 down to 5, give this instruction:

```
for i in range(17,4,-1):
```

This program prints a rocket countdown:

```
for i in range(10,0,-1):
    print(i)
print('Blast off!')
```

The computer will start at 10, count *down* (because the step size is -1), and stop the loop before saying 0; so the computer will print:

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```

Suppose you want *i* to be 6.0, then 6.1, then 6.2, etc., up to 8.0. **Python doesn't let the range contain decimal points**, so use this trick: make *j* be 60, then 61, then 62, etc., up to 80, then make *i* be a tenth of *j*, like this:

```
for j in range(60,81):
    i=j/10
    print(i)
```

Break

If you're stuck in jail, you hope to break out. Similarly, if a computer is stuck in a loop (doing the same thing again and again), the computer hopes to break out.

To let the computer break out of a loop, say "break". Saying "break" lets the computer stop looping; it lets the computer break free from the loop and skip ahead to the rest of your program.

For example, suppose you say:

```
while True:
    print('eat')
    break
    print('sandwiches under')
print('the trees')
```

The "while" tells the computer to obey the indented lines repeatedly, to form a loop. The first indented line makes the computer print:

```
eat
```

But the next line says "break", which makes the computer break out of the loop, stop looping, do no more indented lines, and so not print 'sandwiches under'; the computer will skip ahead to the next *unindented* line, which prints:

```
the trees
```

So the program makes the computer print just this:

```
eat
the trees
```

Guessing game This program plays a guessing game, where the human tries to guess the computer's favorite color, which is pink:

```
while True:
    if 'pink'==input('What is my favorite color? '): break
    print('No, that is not my favorite color. Try again!')
print('Congratulations! You discovered my favorite color.')
```

Here's what the 2nd line means. If 'pink' matches the human's reply to the question 'What is my favorite color?', break out of the loop, so the computer skips ahead to unindented line, which prints:

```
Congratulations! You discovered my favorite color.
```

But if 'pink' does *not* match the human's reply, the computer will continue looping, by saying:

```
No, that is not my favorite color. Try again!
```

Here's another way to program the guessing game:

```
while True:
    print('You have not guessed my favorite color yet!')
    if 'pink'==input('What is my favorite color? '): break
print('Congratulations! You discovered my favorite color.')
```

That program's loop makes the computer do this repeatedly: say 'You have not guessed my favorite color yet!' and then ask 'What is my favorite color?' The computer will repeat the indented lines again and again, until the guess is 'pink'. When the guess is "pink", the computer breaks out of the loop and proceeds to the bottom line, which prints 'Congratulations!'.

Sex This program makes the computer discuss human sexuality:

```
while True:
    sex=input('Are you male or female? ')
    if sex=='male':
        print('So is Frankenstein!')
        break
    if sex=='female':
        print('So is Mary Poppins!')
        break
    print('Please say male or female!')
```

The 2nd line makes the computer ask, 'Are you male or female?' If the human claims to be "male", the computer prints 'So is Frankenstein!' and stops looping. If the human says "female" instead, the computer prints 'So is Mary Poppins!' and stops looping. If the human says anything else (such as "not sure" or "super-male" or "macho" or "none of your business"), the computer prints 'Please say male or female!' and then does the loop again, so the computer asks again, 'Are you male or female?'

That program's bottom line is called an **error handler** (or **error-handling routine** or **error trap**), since its only purpose is to handle human error (a human who says neither "male" nor "female"). The error handler prints a gripe message ('Please say male or female!') and then lets the human try again (by having the human do the loop again).

Here's how to accomplish the same goal without indenting so much:

```
while True:
    sex=input('Are you male or female? ')
    if sex=='male' or sex=='female': break
    print('Please say male or female!')
if sex=='male': print('So is Frankenstein!')
else: print('So is Mary Poppins!')
```

That "while" loop says:

Ask the human 'Are you male or female?' and call the answer "sex".

If the sex is male or female, that's fine, so break out of the loop; otherwise, print 'Please say male or female!' and do the loop again.

Let's extend that program's conversation. If the human says "female", let's make the computer say "So is Mary Poppins?", then ask "Do you like her?", then continue the conversation this way:

If human says "yes", make computer say "I like her too. She is my mother."
If human says "no", make computer say "I hate her too. She owes me a dime."
If human says neither "yes" nor "no", make computer handle that error.

To accomplish all that, put the shaded lines into the program:

```
while True:
    sex=input('Are you male or female? ')
    if sex=='male' or sex=='female': break
    print('Please say male or female!')
if sex=='male': print('So is Frankenstein!')
else:
    print('So is Mary Poppins!')
    while True:
        opinion=input('Do you like her? ')
        if opinion=='yes' or opinion=='no': break
        print('Please say yes or no!')
        if opinion=='yes': print('I like her too. She is my mother.')
        else: print('Neither do I. She still owes me a dime.')
```

Rules Here are the rules about saying "break":

The "break" command is legal just if the computer's in a loop.

You can say "break" if the computer's in a "while" loop or a "for" loop.

If the computer's in nested loops (a loop inside a loop), the "break" command makes the computer break out of the inner loop but not the outer loop.

Data structures

You can combine numbers and strings, to build a **data structure**. Here's how.

Lists

Here's a **list**:

```
['love','death',666,'giggle',3.14]
```

That list contains 5 **items**: 'love' and 'death' and 666 and 'giggle' and 3.14.

In a list, put commas between the items.

Begin the list with an **open bracket** (the symbol "[").
End the list with a **closed bracket** (the symbol "]").

So the entire list is enclosed in **brackets** (the symbols "[" and "]").

If you say —

```
['love','death',666,'giggle',3.14]
```

or say —

```
print(['love','death',666,'giggle',3.14])
```

the computer will say the list:

```
['love', 'death', 666, 'giggle', 3.14]
```

If you say —

```
[5+3,70+20]
```

The computer will do the math and say:

```
[8, 90]
```

A **list** is also called an **array**.

Variable A variable can be a list. For example, you can say:

```
x=['love','death',666,'giggle',3.14]
```

Adding You can add lists together. If you say —

```
['dog','cat']+['mouse','cheese']
```

The computer will add the list ['dog','cat'] to the list ['mouse','cheese'] and say:

```
['dog', 'cat', 'mouse', 'cheese']
```

For The "for" statement can use a list. If you say —

```
for i in [18,21,100]:
```

the i will be 18 then 21 then 100. If you say —

```
for i in ['Joe','Fred','Mary']:
```

the i will be 'Joe' then 'Fred' then 'Mary'.

Let's make the computer print this message:

```
We love meat
We love potatoes
We love lettuce
We love tomatoes
```

This program does that:

```
for i in ['meat','potatoes','lettuce','tomatoes']:
    print('We love',i)
```

You can also write the program this way:

```
x=['meat','potatoes','lettuce','tomatoes']
for i in x:
    print('We love',i)
```

Subscripts Suppose you say:

```
x=['Joe','Fred','Mary']
```

That list contains 3 items: 'Joe', 'Fred', and 'Mary'.

In x's list, **the starting item** (which is 'Joe') is called **x₀** (which is pronounced "x subscripted by zero" or "x sub 0" or just "x 0"). The next item (which is 'Fred') is called **x₁** (which is pronounced "x subscripted by one" or "x sub 1" or just "x 1"). The next item is called **x₂**. So **the 3 numbers in the list are called x₀, x₁, and x₂**.

To make the computer say what x₂ is, type this:

```
x[2]
```

The computer will say:

```
'Mary'
```

Notice this jargon:

In a symbol such as x₂, the lowered number (the 2) is called the **subscript**. To create a subscript, use brackets. For example, to create x₂, type x[2].

You can change what's in a list. For example, if you want to change x₂ to 'Alice', say:

```
x[2]='Alice'
```

Suppose you say:

```
x=['Joe','Fred','Mary']
x[2]='Alice'
```

The x starts as ['Joe','Fred','Mary'], but 'Mary' changes to 'Alice'; so if you say —

```
x
```

the computer will say:

```
['Joe', 'Fred', 'Alice']
```

Too long If you want to type a list that's too long to fit on one line, do this:

Type *part* of the list on one line. Type a backslash at the end of that part. Type the rest of the list below. Type a backslash at the end of each line (except the list's bottom line). The backslash means: the rest of the list continues below.

List in a list A list can contain another list. For example, look at this list:

```
x=['Joe','Fred',['dog','cat']]
```

In that list, x[0] is 'Joe', x[1] is 'Fred', and x[2] is the list ['dog', 'cat']. Since 'dog' is the starting item of the list x[2], 'dog' is x[2][0]. Since 'cat' is the next item of the list x[2], 'cat' is x[2][1].

Blanks in a list You can create a list that's full of blanks, then fill the blanks later.

To create a list that has 100 blank items, say:

```
x=[None]*100
```

The "None" means "blank". Those 100 blank items are called x₀, x₁, x₂, etc., up through x₉₉.

After creating that x, you can give a command such as:

```
x[57]='fun'
```

That command is legal just *after* you've created x.

If you try giving that command without creating x previously, the computer will say "NameError".

If you try talking about x₂₀₀ even though you created up through just x₉₉, the computer will say "IndexError".

Dictionaries

Suppose Jack is great, Jim is jolly, Sue is sweet, and Mary is smart.

To store that info, create a **dictionary** called `d`, like this:

```
d={'Jack':'great', 'Jim':'jolly', 'Sue':'sweet', 'Mary':'smart'}
```

When you type that, put the dictionary in **braces**, which are the symbols “{}” and require you to press the Shift key.

Then if you want to use that dictionary to look up Sue, type:

```
d['Sue']
```

The computer will use the dictionary, look up Sue, discover Sue is sweet, and say:

```
'sweet'
```

Instead of the letter `d`, you can use any variable name you wish.

Here’s how to put a dictionary into a program:

```
d={'Jack':'great', 'Jim':'jolly', 'Sue':'sweet', 'Mary':'smart'}
name=input('What name should I look up? ')
if name in d: print(name,'is',d[name])
else: print('Sorry, I do not know about',name)
```

The top line creates the dictionary. The next line makes the computer ask “What name should I look up?” and wait for the human to type a name. If the human typed a name (such as “Sue”) that’s in the dictionary, the program’s third line makes the computer print a message such as:

```
Sue is sweet
```

But if the human typed a name (such as “Alice”) that’s *not* in the dictionary, the program’s bottom line makes the computer print a message such as:

```
Sorry, I do not know about Alice
```

Besides storing comments such as “sweet”, you can make the dictionary store people’s addresses, phone number, social-security numbers, birthdays, debts, sexual orientations, and methods by which they’d like to kill you when they discover you’ve stored that private data.

A dictionary is also called a **lookup table**.

Let’s make the computer translate English colors to French, by using this lookup table:

English **French**

white	blanc
yellow	jaune
red	rouge
blue	bleu
black	noir

That lookup table becomes our multilingual dictionary. Here’s the program:

```
d={'white':'blanc', 'yellow':'jaune', 'red':'rouge', 'blue':'bleu', 'black':'noir'}
EnglishColor=input('What color should I translate? ')
if EnglishColor in d: print('In French it is',d[EnglishColor])
else: print('Sorry, I do not know the French for',EnglishColor)
```

Too long If you want to type a dictionary (lookup table) that’s too long to fit on one line, do this:

Type *part* of the dictionary on one line. Type a backslash at the end of that part. Type the rest of the dictionary below. Type a backslash at the end of each line (except the dictionary’s bottom line). The backslash means: the rest of the dictionary continues below.

Web-page design

When using the Internet's World Wide Web, don't be just a looker; be a creator! Create your *own* Web pages and let everybody else in the world see them!

Angelfire

The easiest way to create your own Web pages is to use a Web site called **Angelfire**. It's free!

Angelfire is at www.angelfire.com. It used to be an independent company, but now it's owned by **Lycos** (which also owns a similar site, **Tripod**, at www.tripod.com).

Restrictions

Angelfire lets you create any Web pages you wish, as long as you keep them "clean", so they don't contain content or links to anything that's:

unlawful, harmful, hateful, harassing, stalking, or containing viruses
defamatory, libelous, ethnically objectionable, or pirated (copied without permission)
privacy-invading (or vulgar or pornographic), especially if involving minors or viewable by minors
required to stay private (by copyright laws or an employer's nondisclosure agreement)
selling explosives, weapons, securities, or non-existent goods
selling alcohol, tobacco, controlled drugs (or pharmaceuticals), or unpackaged food
advertising gambling (or raffles requiring a fee) or pyramid schemes

Unfortunately, Angelfire will automatically put an ad on your Web page and restrict you (to 20 megabytes of Angelfire's disk space, with a limit of 1 gigabyte of transfers per month between Angelfire's disk and people viewing your Web page), unless you pay extra (to get a fancier plan):

Plan's name	Cost	Disk space	Bandwidth	Forced ad?	Get your own .com, such as joe.com?
Free	free	20M	1G/month	yes	no
Entry	\$1/month	40M	2G/month	yes	no
Basic	\$3/month	100M	5G/month	no	no
THE Plan	\$10/month	5G	1T/month	no	yes

Create an account

Using your Web browser (such as Microsoft Internet Explorer), go to **angelfire.com**. Click "Try It Now for Free!" then "Sign up for one of these plans today!" (which you see when you scroll down).

Click in the "Your New Website" box. Your Website will be named "http://____.angelfire.com". Invent a name to put in the blank. The name cannot contain capital letters or spaces: it must be made of just lower-case letters, digits, dashes, and periods. Type the name you invented (such as "secretguide") then click in the Password box. If the name you invented is okay, the computer says "User name Available"; otherwise the computer says "User name Unavailable" (probably because somebody else picked that name) and you must try again to pick a name.

Next, invent a password (which must be at least 6 characters long, with no spaces). Type the password in the Password box, press the Tab key, type the password again, click "Next", and click "Next" again.

The computer will say "Welcome". Answer the questions about your birthday, security, usage, gender, and newsletters, then click "Confirm".

Angelfire Customer Service will send you an email whose subject is "Welcome to Angelfire". Find it in your email's Inbox. Click the first link in the email.

The computer will say "Your account has been activated". Click "Start Building Now" then "Create a New Website".

Change the text

You're using a Website-creation program called **Webon** (pronounced "web on"). It shows a Web page containing 5 blocks of text. In each block of text, switch those words to *your* words. Here's how to do that:

Drag across the first block of text, which says "MY WEBSITE". Type whatever **heading** you want instead, such as "JOAN'S HOME PAGE". Your typing will be automatically capitalized.

Drag across the second block of text, which says "My website's **subheading**". Type whatever subheading you want instead, such as "Made with love".

Drag across the third block of text, which begins "This is your **main content section**. You should delete..." Type the main message you want instead, such as "I was born yesterday. I want to die."

Drag across the fourth block of text, which begins "This is your **sidebar**. This sidebar is..." Type the sidebar message you want instead, such as "We won't reply to emails addressed to my goldfish."

Scroll down to see the fifth block of text, which begins "This is your **footer**. You can delete..." Drag across that block of text, then type the footer you want instead, such as "Copyright by a wronged woman."

Format the text

You can easily format the text in blocks 3, 4, and 5. To do that, drag across the phrase you want to change (so the phrase temporarily appears in white letters on a blue background), then do one of these things:

Click one of the formatting buttons: Bold, Italic, Underline, Align Left, Align Center, Align Justify, Align Right, Ordered List (which means a numbered list), or Unordered List (which means a bulleted list).

For the Font Size box (which normally says "12px", which means 12 pixels high), click its down-arrow then click a different number of pixels instead. Your choices are 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 32, 36, 48, and 72.

For the Font box (which normally says "Arial"), click its down-arrow then click a different font instead. Your choices are Arial, Times New Roman, Verdana, Georgia, Trebuchet, Courier New, Tahoma, Palatino Linotype, Impact, and Comic Sans.

For the Color box (which is normally black), click its down-arrow then click a different font color instead. The phrase will appear in the new color when you finish highlighting the phrase: click elsewhere.

For the Background Color Box (which says "BG" and is normally white), click its down-arrow then click a different background color instead. The phrase will have the new background color when you finish highlighting the phrase: click elsewhere.

Notice that to format a phrase, you must drag across the phrase *beforehand*. For example, if you want a phrase to be bold, you must drag across the phrase *before* you click the Bold button.

Undo

If you make a typing mistake, here's how to undo it: while holding down the Ctrl key, tap the Z key.

That method undoes your last typing mistake, but it can't undo your last *two* typing mistakes, and it can't undo formatting. If you format a phrase wrong, format it again correctly.

Change the style

You've been using a style called **Working Comp**. To use a different style instead, click "Styles" (which is at the screen's top). You start seeing a list of 152 styles, in alphabetical order. To see the rest of the list, click "Next" several times. To go back toward the list's beginning, click "Previous" several times. If a style interests you, put the mouse pointer on it without clicking; then you see a slightly larger picture of the style.

Click whichever style you like. Then you see the words you wrote, reformatted to fit in that style.

If you don't like that style, try clicking "Revert", which takes you back to a style you used before (such as Working Comp).

When you've finished picking a style, click "Text" (which is near the screen's top-left corner).

Add links

Here's how to make the phrase "house hunting" be underlined and link to www.realtor.com:

Type "house hunting". Drag across that phrase (so it's highlighted). Click "[Link](#)". Click after the "http://". Type "www.realtor.com". Click the green "Create" button.

This book was written by Russ Walter, whose email address is Russ@SecretFun.com. Here's how to make your Web page let people send an email to Russ Walter, by clicking "write to Russ":

Type "write to Russ". Drag across that phrase (so it's highlighted). Click "[Link](#)" then the down-arrow then "an email address". Press the Tab key. Type the email address "Russ@SecretFun.com". Click the green "Create" button.

Then when a person accesses your Web page, "write to Russ" will be underlined. If the person clicks "write to Russ", the computer will automatically run the person's email client program (such as Outlook Express), automatically click "Create Mail", automatically type "Russ@SecretFun.com" in the "To" box, and wait for the person to type an email message to Russ.

Final steps

When you finish editing your Website, click "Save" (which is near the screen's top-right corner) then "Publish" (which is next to "Save") then "Publish to the main page of the site" then the green "Publish" button. The computer says, "Congratulations! Your site was published..."

If you want to edit further, click "Keep Working"; otherwise, exit by doing this:

Click "Back to Angelfire" then "logout" (which is near the screen's top-right corner).

Edit your site

To edit a Website you created before, do this:

Go to www.angelfire.com. Click "Login" (which is at the top). Type your user name (such as "secretguide"), press the Tab key, type your password, then click the green "Log In" button. Click "Edit" (which is below "Create a New Website").

You see your Website. Edit it, then do the "Final steps" procedure again.

Extra pages

Here's how to put extra pages onto your Website.

While you're editing the first page you created, click "Pages" (which is at the top) then the "Create a new page" icon. (That icon is near the screen's top-left corner, under "Text", and shows a single sheet of paper with a green plus sign.)

Invent a title for the page (such as "My Family"); type it and press Enter.

Now the screen's left edge shows you have 2 pages. The first page (which you created before) was automatically called "Home". If the new page you're creating is called "My Family", the screen's left edge shows this list of pages:

1. Home
2. My Family

If you want to create a 3rd page, click the "Create a new page" icon again, type the new page's title (such as "Our Friends") and press Enter again, so the screen's left edge looks like this:

1. Home
2. My Family
3. Our Friends

The screen's bottom is still devoted to showing what you typed on page 1. To switch your view to page 2 or page 3, click the page's name in that list of pages, then click the "Jump to a page" icon (which is a sheet of paper with a green arrow). That makes the screen's bottom show the page you requested. The heading you requested is at the screen's top and has the page name you invented. There's no subheading. The sidebar is at the left instead of the right. The main content section, sidebar, and footer are all blank, waiting for you to type your words there. Below the heading you also see this **navigation bar** —

HOME MY FAMILY OUR FRIENDS

which your readers can click on to jump from page to page, after you've finished creating your Website.

To see normal editing tools again (such as the Bold button), click "Text" (which is near the screen's top-left corner).

HTML

Every page on the Internet's Web is written in a computer language called the **HyperText Markup Language (HTML)**.

Warning: if someone claims to know "HTML", it could mean "HyperText Markup Language" or "How To Make Love". Ask which!

HTML uses these commands:

HTML command	Ending	Page
		549
		549
		547
<big>	</big>	547
<body>	</body>	548
<body bgcolor=red>	</body>	550
<body link=green>	</body>	550
<body text=red>	</body>	550
<body vlink=blue>	</body>	550
 		548
		550
		547
		548
<form>	</form>	559
<form method=...>	</form>	551
<h1>	</h1>	547
<head>	</head>	548
<html>	</html>	548
<i>	</i>	546
<input name=flavor>		551
<input type=button ...>		559
<input type=checkbox ...>		552
<input type=radio ...>		552
<input type=reset>		552
<input type=submit>		551
		548
<link rel=...>		553
<noscript>	</noscript>	559
		549
<p>		547
<pre>	</pre>	549
<script>	</script>	554
<script language=...>	</script>	559
<small>	</small>	547
<style>	</style>	553
_		547
[]	547
<table border=1>	</table>	549
<td>		549
<th>		549
<title>	</title>	547
<tr>		549
<tt>	</tt>	547
		548
<!DOCTYPE HTML ...>		548
<!--I was drunk-->		551

Simple example

Suppose you want to create a Web page that says:

```
We love you
```

Notice that the word “love” is italicized (slanted).

To create that Web page, write this HTML program:

```
we <i>love</i> you
```

Here’s what that program means:

The <i> is an **HTML tag** that means “italics”. Each HTML tag is enclosed in the symbols <>, which are called **angle brackets**.

The </i> is an HTML tag that means “end the italics”, because the symbol / means “end the”. The <i> shows where to begin the italics; the </i> shows where to end the italics.

To type that program, you can use two free methods: **Notepad** or **Angelfire**. I’ll explain both.

Typing in Notepad

If your computer contains Windows, the easiest way to type that HTML program is to use **Notepad**, as follows....

Launch Notepad. Here’s how:

Windows 10 In the Windows Search box, type “notep”. You see a list of things that contain “notep”. Tap “Notepad: Desktop app”.

Windows 8&&8.1 Go to the Apps screen (or Windows 8’s Start screen) and type “no”. You see a list of programs whose names have “no”. Tap “Notepad”.

Windows 7 Click Start then Programs then Accessories then Notepad.

You see the Notepad window. Make it consume the screen’s bottom right quarter. Here’s how:

If the Notepad window is maximized (consumes the whole screen), make the window smaller by clicking the restore button (which is next to the X button).

Drag the window’s bottom-right corner to the screen’s bottom right corner. Drag the window’s top-left corner to the screen’s center. Then the window consumes the screen’s bottom right quarter.

Make Notepad do word wrapping. Here’s how:

Click “Format”. You see “Word Wrap”. If there’s no check mark in front of “Word Wrap”, put a check mark there (by clicking “Word Wrap”).

Click in the middle of the Notepad window. Type the HTML program, like this:

```
we <i>love</i> you
```

Save the program Save the program onto your desktop, and give it a name that ends in “.html”. Here’s how:

Click “File” then “Save”.
Click the word “Desktop”, which you’ll see at the screen’s left edge (after scrolling up, if necessary). Click in the “File name” box.
Type any name ending in “.html” (such as “joan.html”); to be safe, use just small letters (no capitals, no spaces). Press Enter.

Your program’s icon arrives on the desktop but might be covered by the Notepad window. Move the Notepad window (by dragging its blue title bar) until you see your program’s icon.

The icon has an “e” on it. The “e” means the program works with Microsoft Edge (or Microsoft Internet Explorer). It works with Microsoft Edge (or Microsoft Internet Explorer) because the program is written in HTML and ends in .html.

Run the program Double-click your program’s icon. That makes the computer run your program. (If a “Connect To” window appears, click “Cancel” once or twice, to make the “Connect To” window go away.)

If your computer’s been set up properly (to use Microsoft Edge or Microsoft Internet Explorer as the main Web browser), you’ll see a window that shows the result of running your program; it shows a Web page that says:

```
We love you
```

Edit the program If you typed and ran the program

recently, here’s how to edit it.

At the screen’s bottom, to the right of the Start button, you see a wide button for Notebook. (In Windows 10, the button is underlined.) Click it. That makes the Notebook window appear and be the active window. Then make any changes you wish to your HTML program. For example, if you want the Web page to say “We *tickle* you”, change “love” to “tickle” (by dragging across “love” and then typing “tickle”).

Go ahead: make that change and any other changes you wish! Experiment! Go wild!

After you’ve edited the program, save the edited version (by clicking “File” then “Save”).

To run that program, click in the Microsoft Internet Explorer (or Microsoft Edge) window (which hides behind the Notepad window), so that Microsoft window becomes the active window. Then click the Refresh button (a circling arrow), which makes Microsoft re-examine your program and run your program’s new version.

Edit old programs Here’s how to edit an old program that you haven’t typed or run recently.

Find the program’s icon on the desktop. Right-click that icon. Click “Open with”.

If you see “Notepad”, click it. Otherwise, do the following:

Windows 10 Tap “Choose another app” then “More apps” then “Notepad”. (Don’t put a checkmark before “Always use this app to open .html files”.) Tap “OK”.

Windows 7&&8&8.1 Click “Choose default program”. Remove the check mark from “Always use the selected program” (by clicking). Click the down-arrow to the right of “Other Programs”. Scroll down until you see “Notepad” then double-click “Notepad”.

Make the Notepad window consume the screen’s bottom right quarter. Make Notepad do word wrapping.

Then you see your HTML program. Edit it. When you finish editing, click “File” then “Save”.

Typing in Angelfire

Here’s how to type an HTML program by using **Angelfire** instead of Notepad....

Go to www.angelfire.com. Start creating a Web page, using the methods I described on pages 544-545.

While you’re typing & editing the main content section, click “Add-ons” (which is at the top). You see 27 add-ons. Drag “Custom HTML” (which is the last add-on) until it’s below the typing in the main content section and is in the middle of a big black box that suddenly appears.

After several seconds, the computer says this sentence:

```
Click the “Edit” Link to add HTML to this space.
```

Move the mouse pointer to that sentence, without clicking. Then above that sentence, you see “Edit”. Click that “Edit”.

The computer says this sentence:

```
Paste or write your custom HTML in the box below.
```

In the box below that sentence, drag across the sentence that’s already there (which begins with “<p>”) then type your HTML program instead. For example, type:

```
we <i>love</i> you
```

When you finish typing your program in the box, click the green “Save” button (which you must scroll down to see).

Then you see a box showing the result of your programming:

```
We love you
```

If you want to edit your program further, move the mouse pointer to that box (without clicking), then click the “Edit” above that box, then edit your program further, then click the green “Save” button again (after scrolling down to see it).

Simple HTML commands

Here are simple HTML commands you can give.

Italicize To make a phrase be italicized (*like this*), type `<i>` before the phrase; type `</i>` after the phrase.

Bold To make a phrase be bold (**like this**), type `` before the phrase; type `` after the phrase. To make a phrase be bold italic (*like this*), type `<i>` before the phrase; type `</i>` after the phrase.

Paragraph If your document contains more than one paragraph, put `<p>` at the beginning of each paragraph. For example, if you want a paragraph to begin by saying “Motherhood is maddening!”, begin the paragraph by typing this:

```
<p>Motherhood is maddening!
```

At the end of each paragraph, press the Enter key twice. Then when you run the program, the computer will put a blank line below each paragraph.

Title To create a title, type `<title>` before it and `</title>` after it. For example, to make your title say “Joan’s Home Page”, type this:

```
<title>Joan's Home Page</title>
```

When you run the program, the computer will put the title in the blue **title bar** at the Web page’s top.

Fonts

You learned to italicize by saying `<i>` and `</i>`, and to create bold by saying `` and ``. Here are other ways to change the text’s font...

Size You can make the computer can produce text in 7 sizes. Font size 1 is the smallest; Font size 7 is the biggest. Here’s how big they are:

```
This is font size 1. It is 8 points tall.
```

```
This is font size 2. It is 10 points tall.
```

```
This is font size 3. It is 12 points tall.
```

```
This is font size 4. It is 14 points tall.
```

```
This is font size 5. It is 18 points tall.
```

```
This is font size 6, 24 points tall.
```

```
Font size 7, 36 points.
```

That’s how big the font sizes and point size look on paper and a 14-inch monitor. (On a slightly bigger monitor, the font sizes and point sizes look slightly bigger.)

Normally, the computer makes your Web page have font size 3. To make a phrase be font size 5, say `` before the phrase; say `` after the phrase.

You can make part of a phrase be one size bigger, by saying `<big>` before that part and saying `</big>` after that part. For example, if the computer is making a phrase be font size 5 because you said ``, you can make part of that phrase be slightly bigger (font size 6) by saying `<big>` before that part and saying `</big>` after that part. To make part of a phrase be one size smaller, say `<small>` before that part and say `</small>` after that part.

If you said `` and then try to say `<big>`, the computer will ignore the `<big>`, since the computer can’t go bigger than font size 7. If you said `` and then try to say `<small>`, the computer will ignore the `<small>`, since the computer can’t go smaller than font size 1.

To make a phrase be subscript (*like this*), say `_{` before the phrase, `}` afterwards. To make a phrase be superscript (*like this*), say `^{` before the phrase, `}` afterwards.

Heading To create a heading (such as a chapter title or a newspaper headline), say `<h1>` at the heading’s beginning and `</h1>` at the heading’s end, like this —

```
<h1>Chapter 2: Laura giggles at death</h1>
```

or like this:

```
<h1>USA declares war on hangnails</h1>
```

To do that, make sure you type the symbol “<”, then the letter “h”, then the number one, then the symbol “>”, then the rest.

The `<h1>` makes the computer automatically create a new paragraph in font size 6 and bold. Saying `<h1>` is similar to saying `<p>`.

If you want the heading to be centered (instead of at the screen’s left edge), say `<h1 align=center>` instead of just `<h1>`.

If you say `<h2>` instead of `<h1>`, the computer will make the font size slightly smaller (5 instead of 6), so you’ll be creating a “less dramatic heading”, a subheading. If you say `<h2>` at the heading’s beginning, say `</h2>` at the heading’s end.

You can create headings in 6 sizes:

```
<h1> produces font size 6
<h2> produces font size 5
<h3> produces font size 4
<h4> produces font size 3
<h5> produces font size 2
<h6> produces font size 1
```

To create a heading in font size 7, say `<h1><big>` at the heading’s beginning, and say `</big></h1>` at the heading’s end.

Tt Normally, the computer makes the text’s typeface be Times New Roman (which looks like this). If you want a phrase’s typeface to be Courier New (which looks like this and imitates a typewriter and a Teletype), say `<tt>` before the phrase, `</tt>` afterwards.

Arial If you want a phrase’s typeface to be Arial (which looks like this), say `` before the phrase, `` afterwards.

If you want a phrase to be font size 7 and Arial, say `` before the phrase; `` afterwards.

Compliance

If you want to create a Web page whose title is “Joan’s Home Page” and whose body says “We love you”, you can write this HTML program:

```
<title>Joan's Home Page</title>
we <i>love</i> you
```

Although that program works with *most* versions of *most* Web browsers, you’re supposed to add some extra lines, for 3 reasons:

to help other programmers understand your program to make sure the program works with *all* browsers to prevent getting fired from your programming job

Head & body You're supposed to divide the program into 2 parts. The first part, called the **head**, begins with <head> and ends with </head> and includes the title. The second part, called the **body**, begins with <body> and ends with </body> and includes the paragraphs. So your program should look like this:

```
<head>
<title>Joan's Home Page</title>
</head>

<body>
we <i>love</i> you
</body>
```

(If you're typing into the middle of an Angelfire page, do *not* type a head, since Angelfire gave the page a head already.)

The <html> warning To make your program even better, you're supposed to begin the whole program by saying <html> and end the whole program by saying </html>, to emphasize that the whole program is written in HTML rather than a different computer language. So your program should look like this:

```
<html>
<head>
<title>Joan's Home Page</title>
</head>

<body>
we <i>love</i> you
</body>
</html>
```

Is code compliant? You should include all those extra lines — <head>, </head>, <body>, </body>, <html>, and </html> — to make your program comply with the standards that people expect. Those lines help make your program be **compliant**.

I usually don't bother including those lines, since I'm my own boss; but if you're employed, you should include those lines to keep your job.

Title To be compliant, your program must include a <title> line.

DOCTYPE To be compliant, you're supposed to also put this line at your program's top, above the <html> line:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

That line brags that your program document is of this type: it obeys the HTML standard, which is publicly available from the **World Wide Web Consortium (W3C)**, using the **document type definition (DTD)** for HTML version 4.01 — except that you're letting yourself include some older HTML commands also, to be transitional and help older Web browsers understand your Web page; and you're doing all this in **English (EN)**.

Actually, you're supposed to include another line below that, saying what Web site reveals the HTML 4 and 4.01 specifications; so your program is supposed to begin like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

Hardly anybody bothers to include those two lines about DOCTYPE. For example, Yahoo's Web site (www.yahoo.com) and Microsoft's Web site (www.microsoft.com) omit both lines.

Even the inventors of HTML 4.01 don't bother including the second line in their own Web pages.

XHTML HTML 4.01 was invented in 1999. Now committees are working to develop fancier programming language, called the **eXtensible HTML (XHTML)**, which is influenced by the **eXtensible Markup Language (XML)**.

If you want to be compliant with XHTML, adopt these habits:

In commands, use small letters (such as <p>) instead of capitals (such as <P>).

At the end of each paragraph, say </p>.

After each equal sign, put quotation marks; so instead of saying , say .

In XHTML, the top lines say:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"DTD/xhtml1/1-transitional.dtd">
```

Lists

If you want your Web page to contain a simple list, say
 at the beginning of each list item.

For example, suppose you want your Web page to say:

```
Here are the favorite flavors:
chocolate
vanilla
strawberry
```

To do that, say <p> at the beginning of the paragraph,
 at the beginning of each list item, like this:

```
<p>Here are the favorite flavors:
<br>chocolate
<br>vanilla
<br>strawberry
```

The
 stands for "break out a new line".

In XHTML, instead of saying
 you must say
 and make sure to put a blank space before the slash.

Bullets Suppose you want your list to show bullets, like this:

```
Here are the favorite flavors:
• chocolate
• vanilla
• strawberry
```

To do that, say <p> at the beginning of the paragraph, at the beginning of each list item, like this:

```
<p>Here are the favorite flavors:
<li>chocolate
<li>vanilla
<li>strawberry
```

Suppose you want the bullets to be indented, with a space above the list, like this:

```
Here are the favorite flavors:

• chocolate
• vanilla
• strawberry
```

To do that, say above the list, below the list, like this:

```
<p>Here are the favorite flavors:
<ul>
<li>chocolate
<li>vanilla
<li>strawberry
</ul>
```

The stands for "unordered list". When you type , make sure you press the U and L keys on your keyboard (not the number one).

The accomplishes two goals:

It makes the list be indented.

It makes your program be compliant.
(A list without is *not* compliant.)

In XHTML, the end of each list item must say .

Numbers Suppose you want your list to be numbered, like this:

Here are the favorite flavors:

1. chocolate
2. vanilla
3. strawberry

To do that, say `` above the list, `` below the list, like this:

```
<p>Here are the favorite flavors:
<ol>
<li>chocolate
<li>vanilla
<li>strawberry
</ol>
```

The `` stands for “ordered list”. When you type ``, make sure you press the O and L keys on your keyboard.

Tables

You can create a table that’s simple or fancy.

Simple table Here’s how to create a simple table:

Tell the computer the text is preformatted, by saying `<pre>`. Then type the table, by using the Space bar and Enter key to line up the columns. Below the table, say `</pre>`.

For example, suppose you want your Web page to say:

Here are the bowling scores:

NAME	SCORE
Jacqueline	200
Ann	137
Ed	75

To do that, say `<pre>` above the table and `</pre>` below the table, like this:

```
<p>Here are the bowling scores:
<pre>
NAME          SCORE
Jacqueline    200
Ann           137
Ed            75
</pre>
```

The `<pre>` makes the computer use Courier New, as if you had typed `<tt>`. Courier New is a convenient font for tables, because it makes every character and space have the same width, so you can easily align the table’s columns by pressing the Space bar several times.

The `<pre>` also makes the computer copy each Enter and Space onto the Web page, unedited. (If you don’t say `<pre>`, the computer turns each Enter into a Space and turns each pair of Spaces into a single Space.)

Fancy table Let’s create a fancy table, so the Web page says:

Here are the bowling scores:

NAME	SCORE
Jacqueline	200
Ann	137
Ed	75

To do that, say `<table border=1>` above the table, `<tr>` at the beginning of each table row, `<th>` at the beginning of each column heading, `<td>` at the beginning of each data item, and `</table>` below the table, like this:

```
<p>Here are the bowling scores:
<table border=1>
<tr><th>NAME<th>SCORE
<tr><td>Jacqueline<td>200
<tr><td>Ann<td>137
<tr><td>Ed<td>75
</table>
```

The computer automatically makes the columns wide enough to hold their headings and data. The computer automatically makes the column headings be bold and centered.

The `border=1` makes the table have a normal border. If you say `border=12` instead, the table’s outer border will be much thicker and shaded, forming a beautiful 3-D picture frame that makes the table seem to pop out from the screen and into the human’s face. Try it! You’ll impress your friends!

Links

You learned:

To make a phrase be italicized, say `<i>` before the phrase, `</i>` afterwards.
To make a phrase be bold, say `` before the phrase, `` afterwards.

Link to other Web sites To make the phrase “house hunting” be underlined and link to www.realtor.com, say this:

```
<a href=http://www.realtor.com>house hunting</a>
```

Notice that before “house hunting”, you say what “house hunting” is linked to, by saying ``, which means “anchor (link) with a hypertext reference to <http://www.realtor.com>”. After “house hunting”, say ``.

Make sure you include the “`http://`”. If you omit that, the link doesn’t work.

Don’t insert extra spaces. For example, don’t insert a space after “`http:`”; don’t insert a space after “`//`”.

Most text on your Web page is black (on a white background). Links are underlined and typically blue; but that if the link refers to a Web page that was visited recently, the link turns purple (instead of blue).

Link to your own Web pages To make the phrase “funny jokes” be underlined and link to Web page you created and called “`jokes.html`”, say this:

```
<a href=jokes.html>funny jokes</a>
```

That works just if you earlier invented a Web page named `jokes.html` and put it on the same disk and in the same folder as the new Web page you’re inventing. For example, if you’re inventing a new Web page on your hard disk’s Desktop, `jokes.html` must also be a Web page on your hard disk’s Desktop.

Suppose you create a Web page that’s too long to fit on the screen, so when a human tries to read the page the human must scroll down. To let the human avoid scrolling, do this: near the page’s beginning, put a table of contents that links to later parts of the page. Here’s how. To make the phrase “my joys” be underlined and link to a part of the page that discusses your joys, say this:

```
<a href=#joys>my joys</a>
```

Tell the computer which part of your page discusses your joys, by beginning that part like this:

```
<a name=joys></a>
```

Link to email This book was written by Russ Walter, whose email address is Russ@SecretFun.com. On your Web page, you can let people write to Russ Walter. Here’s how...

To make the phrase “write to Russ” be underlined and link to creating an email to “Russ@SecretFun.com”, say this:

```
<a href=mailto:Russ@SecretFun.com>write to Russ</a>
```

Then when a person accesses your Web page, “write to Russ” will be underlined. If the person clicks “write to Russ”, the computer will automatically run the person’s email client program (such as Outlook Express), automatically click “Create Mail”, automatically type “Russ@SecretFun.com” in the “To” box, and then wait for the person to type an email message to Russ.

Try it! Go put that in your Web page, run your Web page, and write an email to Russ! If you don’t know what to say in the email, just say, “Hi, Russ, I’m testing my Web page’s email link to you.”

Be brave! Let people accessing your Web page write an email to you. For example, if your name is Joan Smith and your email address is “jSmith@SecretFun.com”, say this:

```
<a href=mailto:jSmith@SecretFun.com>write to Joan Smith</a>
```

Custom colors

You can change colors.

Change a phrase Normally, the computer makes your text be black. To make a phrase be red, say `` before the phrase; say `` after the phrase.

The computer knows the names of 16 colors:

Light color	Dark color
red	maroon (dark red)
blue	navy (dark blue)
aqua (greenish blue)	teal (dark aqua)
lime (bright green)	green (darker than lime)
fuchsia (bright purple)	purple (darker than fuchsia)
silver (light gray)	gray (darker than silver)
yellow	olive (dark yellow, looks greenish brown)
white	black

If you want a phrase to be font size 7 and red, say `` before the phrase; say `` after the phrase.

Change the whole Web page On a normal Web page, the background is white, the text is black, each typical link is blue, and each viewed link is purple.

Your program is supposed to be divided into 2 parts, called `<head>` and `<body>`. Instead of saying just `<body>`, you can say:

```
<body bgcolor=yellow text=red link=green vlink=fuchsia>
```

That makes the background be yellow, the text be red, each typical link be green, and each viewed link be fuchsia. Choose any colors you wish, from the list of 16 colors.

Usually, the background should be a light color (such as white or yellow) and the text should be a dark color (such as black). If you want to be shockingly different, do the reverse: make the background be a dark color (such as black or navy) and make the text be a light color (such as white or yellow). For example, try this:

```
<body bgcolor=navy text=white link=lime vlink=yellow>
```

Warning: the human eye gets dizzy when it sees red next to blue, so don’t choose “red text on a blue background” or “blue text on a red background”. If you make the mistake of choosing those combinations, the people viewing your Web site will get dizzy and fall asleep without reading your words. Your Web site will put them into a hypnotic trance. The human eye also has difficulty reading red (or maroon) next to black.

Upload

To let the public use your Web page, make sure your Web page is on the Internet. Here’s how.

If you created the Web page by using Angelfire’s Webon, make sure you published your page (by clicking the blue “Save” and “Publish” buttons at Webon’s top-right corner).

If you created the Web page by using Notepad, you must **upload** your page (copy it from your computer’s hard disk to an Internet-connected hard disk). To upload your page, you need

permission from a Web host, which will usually charge you a monthly fee. For example, you can upload to Angelfire if you pay Angelfire \$1 per month to get Angelfire’s **Entry service** (which is better than Angelfire’s Free service). If you’ve upgraded to Angelfire’s Entry service, here’s how to copy your Web page to Angelfire’s hard disk:

Using your Web browser, go to “www.angelfire.com”. Click “Login” (which is at the top). Type your user name (such as “secretguide”), press the Tab key, type your password, then click the green “Log In” button. Click “Upload Files” (which is at the screen’s center and isn’t included in the Free version). Click the first “Browse” button.

You see a list of files that are on your computer’s hard disk. Double-click the file you want to upload, such as “joan”. (If you don’t see that file in the list, try making it appear by clicking the word “Desktop” and using the scroll arrows.) Click the green “Upload” button (which you see when you scroll down).

That file will now be on Angelfire’s disk. For example, if your user name is “secretguide” and the file was joan.html, it’s now available on the Internet as “http://secretguide.angelfire.com/joan.html”. If the file was index.html, it’s now available on the Internet as “http://secretguide.angelfire.com/index.html” or more simply as “http://secretguide.angelfire.com” (since “index.html” is your main page).

If your Web page includes links to other files (such as other Web pages you created and style sheets), make sure you upload those files also.

If you’re too cheap to pay Angelfire \$1 per month, try this crude trick:

Create a free Web page by using Angelfire’s Webon. On that page, start the processing of typing your own HTML, as I explained in the section called “Typing in Angelfire”. But instead of typing lots of HTML there, copy your typing from Notepad to there (by dragging across the HTML you typed in Notepad, then pressing Ctrl with C, then clicking in Angelfire’s HTML editing box, then pressing Ctrl with V).

FTP The typical ISP lets you also copy your Web page to your ISP’s hard disk by using **File Transfer Protocol (FTP)**. For example, if your ISP is Galaxy Internet Services (GIS), here’s how to copy your Web page to GIS’s hard disk:

Click “Start” then “Programs”.
Click “MS-DOS Prompt”. The computer will say:

```
C: \WINDOWS>
That should be in a black window that does not consume the whole screen. (If the window consumes the whole screen, so you see no colors, make the window smaller by tapping the Enter key while holding down the Alt key.)
```

Type “ftp” and then the name of your ISP’s FTP site. For example, GIS’s FPT site is called “ftp.gis.net”, so you’d type “ftp ftp.gis.net”, to make your screen look like this:

```
C: \WINDOWS>ftp ftp.gis.net
```

At the end of that line, press Enter.

If you’re not connected to the Internet at the moment, the computer might ask you to type your password. Do so and press Enter.

The ISP’s computer will say “FTP server” and then “User”. Type the user name that the ISP assigned you (such as “poo”) and press Enter.

The ISP’s computer will say “Password”. Type the password that the ISP assigned you and press Enter.

The ISP’s computer will say “logged in” and then say:

```
ftp>
```

Now you’re using the ISP’s operating system, which is Unix. You can type Unix commands. At the end of each Unix command, press the Enter key. (For example, just for fun, type the Unix command “dir” and press Enter: you’ll see a list of files about you on the ISP’s hard disk; each file’s name is in the rightmost column. If you’d like to see a list of other Unix commands, type “help” or a question mark and press Enter. To see a command’s purpose, type “help” then a space then the command’s name, then press Enter.)

If you haven’t done so already, make a directory (folder) called “public_html” on the ISP’s hard disk by typing “mkdir public_html” (and press Enter).

Next, tell the computer to send (copy) the Desktop’s joan.html file to the ISP’s public_html folder. To accomplish that, type so your screen looks like this:

```
ftp> send Desktop/joan.html public_html/joan.html
```

The computer typed the “ftp>”, but you must type the rest. Type it very carefully! Type forward slashes (/) not backslashes (\), since Unix understands just forward slashes. Type a space after “send” and a space before “public”; those are the only spaces you type. After “public”, type an underline (by holding down the Shift key while you tap the key that’s right of the zero key).

The computer will copy the file and say "Transfer complete".

When you finish using FTP, type "quit" or "bye" and press Enter. The computer will quit using FTP and quit using Unix and say "C:WINDOWS>". Then close the black window by clicking its X box.

That file will now be on the ISP's disk. For example, if the file was joan.html, it's now available on the Internet as "www.gis.net/~poo/joan.html. If the file was index.html, it's now available on the Internet as www.gis.net/~poo/index.html or simply as "www.gis.net/~poo" or even more simply as "gis.net/~poo". (The symbol "~" is at your keyboard's top-left corner, above the Tab key, and requires you to hold down the Shift key.)

If your Web page includes links to other files (such as other Web pages you created and style sheets), make sure you upload those files also.

Special symbols

To put special symbols onto your Web page, type these codes:

Symbol	Symbol's name	Code you type
©	copyright	©
®	registered	®
™	trademark	™
¢	cent	¢
£	British pound	£
¥	Japanese yen	¥
¼	fraction 1/4	¼
½	fraction 1/2	½
¾	fraction 3/4	¾
¿	inverted question	¿
¡	inverted exclamation	¡
¹	superscript 1	¹
²	superscript 2	²
³	superscript 3	³
<	less than	<
>	greater than	>
&	ampersand	&
é	e acute	é
É	E acute	É

(similar for á, Á, í, Í, ó, Ó, ú, Ú, ý, Ý)

Comments

In the middle of your program, you can write a comment such as:

```
<!--I wrote this program while drunk-->
```

The computer will ignore the comment. The comment won't affect what appears on the Web page.

To write a comment, begin with this symbol —

```
<!--
```

and end with this symbol:

```
-->
```

The computer ignores whatever appears between those symbols. Whatever appears between those symbols is a comment. The comment can be short (part of a line) or long (many lines), but make sure you begin it with "<!--" and end with "-->".

Write comments to help other programmers deal with your program! For example, give your name, the date you wrote the program, and your address or phone number. Also include any technical comments you wish to make about how your program works and what further improvements you hope to make.

Forms

You can make your Web page display a form and let the human fill it in. For example, you can make your Web page say this:

```
I'm doing a survey of people who view this Web page.  
  
Tell me about yourself.  
  
What's your favorite ice cream flavor?   
  
What's your favorite animal? 
```

This program makes it happen (if your email address is jSmith@SecretFun.com):

```
<p>I'm doing a survey of people who view this web page.  
  
<p>Tell me about yourself.  
<form method=post action=mailto:jSmith@SecretFun.com>  
<pre>  
what's your favorite ice cream flavor? <input name=flavor>  
what's your favorite animal? </pre>  
<input type=submit>  
</form>
```

If you examine that program, you'll notice these rules:

```
Above the form, say <form method=post> and give your email address. Below the form, say </form>.  
To create each box, say <input> and give a one-word name for the box.  
To make the boxes line up, say <pre> above them and </pre> below them.  
At form's bottom, say <input type=submit>.
```

When a person runs your Web page, here's what happens. The computer shows the form and waits for the person to fill in the form. Each box is wide enough to show 20 typical characters. If the person types more than the box can show, the writing in the box automatically scrolls to the left, to let the person type more.

Below the form, the computer puts a button labeled "Submit Query". The person is supposed to click that button after filling in the form. When the person clicks that button, the computer emails the box's contents to the email address mentioned in the <form> command. Here's how:

```
First, the computer gives the person this warning: "This form is being submitted using email. Submitting this form will reveal your email address to the recipient, and will send the form data without encrypting it for privacy. You may continue or cancel this submission."
```

The computer waits for the person to click "OK". (If the person clicks "Cancel" instead, the process is stopped.)

The computer automatically runs the person's email program (such as Outlook Express) and automatically creates a new email.

The email's "To" is the email address mentioned in the <form> command.

The email's "Subject" is "Form posted from Microsoft Internet Explorer" (if the person used Microsoft Internet Explorer) or "Form posted from Mozilla" if the person used Netscape Navigator).

The email's "Message" is blank, but the email includes an Attachment, which is a Notepad document. That document is called "POSTDATA.ATT" (if the person used Microsoft Internet Explorer) or "Form posted from Mozilla.dat" (if the person used Netscape Navigator). For example, if the person said the favorite ice cream flavor is strawberry and the favorite animal is guinea pig, the document says this:
flavor=strawberry&animal=guinea+pig

The computer tries to send the email. If the computer is not attached to the Internet at the moment, the computer either asks the person to connect or else just puts the email in the Outbox (which is a holding area for email that will be sent automatically when Internet connection is reestablished).

Then the computer stops running the email program and returns to showing the Web page that contained the form.

When you receive the email, try to open the attachment. The computer will ask whether you want to open it or save it.

For example, if you're using Windows Me, do this:

Click "Open it" then "OK".

If the attachment is called "POSTDATA.ATT", it opens immediately.

If the attachment is called "Form posted from Mozilla.dat", the computer warns "You are attempting to open a file of type .dat". To respond to the warning, press Enter then click "Notepad" (from the scrolling list of programs) then press Enter again.

Customize Here's how to customize the form.

If you want a box to be 30 characters wide instead of 20, say "size=30" like this:

```
what's your favorite ice cream flavor? <input name=flavor size=30>
```

If you want the "Submit Query" button to say instead "Click here to transmit", say so in the type=submit line, like this:

```
<input type=submit value="Click here to transmit">
```

You can put a Reset button to the right of the "Submit Query" button:

```
<input type=submit><input type=reset>
```

You can make the Reset button say "Click here to erase and start over":

```
<input type=submit><input type=reset value="Click here to erase and start over">
```

Check boxes Your form can include check boxes, so your Web page says:

Check all that apply:

- You have a pet dog.
- You have a pet cat.
- You can bark like a dog.
- You got arrested for being sneaky as a cat.

To do that, say this below the </pre>:

```
<p>Check all that apply:  
<br><input type=checkbox name=dog>You have a pet dog.  
<br><input type=checkbox name=cat>You have a pet cat.  
<br><input type=checkbox name=bark>You can bark like a dog.  
<br><input type=checkbox name=purr>You can purr like a cat.  
<p>
```

If the person clicks the "You have a pet dog" and "You can bark like a dog" boxes, check marks appear on those boxes and the email will say:

```
dog=on&bark=on
```

Radio buttons Your form can include radio buttons, so your Web page says:

Choose just one:

- You are male.
- You are female.

Each radio button acts like a check box, except that the button is round (instead of square), clicking it makes the middle get a dot (instead of a check mark), and just one button can be selected (since clicking a button makes all other buttons get unselected).

To create those radio buttons, say this below the </pre>:

```
<p>Choose just one:  
<br><input type=radio name=sex value=male>You are male.  
<br><input type=radio name=sex value=female>You are female.  
<p>
```

If the person clicks the "You are male" button, a dot appears in that button and the email will say:

```
sex=male
```

Create your own .com

I invented my own .com and called it "SecretFun.com", so you can access my Web page by typing just "SecretFun.com".

You can invent your own .com! Here are two cheap ways to do it...

Method 1: pay Angelfire

Create a Web site on Angelfire, but pay Angelfire extra to get a .com name or switch to Angelfire's fanciest plan (called "THE Plan") which includes a domain name at no extra charge.

Method 2: pay Go Daddy

Create a Web site cheaply anywhere (such as a free Angelfire site or a cheap Neon Angelfire site or a free site anywhere else), then buy a **domain name** (such as a .com name) from a **domain registrar** such as **Go Daddy**, which charges:

```
$15 per year for .co or .ws  
$12 per year for .com  
$10 per year for .org or .net  
$6 per year for .biz  
$5 per year for .us  
$2 per year for .info
```

Go Daddy was started by a famous nice guy (**Bob Parsons**, whose previous venture was called **Parsons Technology**). You can reach Go Daddy at **GoDaddy.com**.

Tell Go Daddy to charge you for the domain name (at \$15/year or less) and set up a **parked site** (which is free because it just says "under construction — coming soon"). Then tell Go Daddy to do **domain forwarding** (which is free) from the parked site to the site you created at Angelfire. You can also tell Go Daddy to do **email forwarding** (which is free) to your current email address.

For example, I told Go Daddy to do this:

```
create SecretFun.com (now $12/year)
```

```
forward SecretFun.com to  
angelfire.com/nh/secret (free)
```

```
forward Russ@SecretFun.com to  
SecretGuide@comcast.net (free)
```

So now you can see my Web site (angelfire.com/nh/secret) by typing just "SecretFun.com" (which is easier to remember), and you can send me email by typing "Russ@SecretFun.com" (which you might remember easier than "SecretGuide@comcast.net").

If you wish, Go Daddy can also **host** your site (so you don't need to involve Angelfire), but Go Daddy charges more for hosting than Angelfire does. Go Daddy will try to sell you many extra services (for surcharges), but you can decline them all: **pay just \$15 per year or less for a domain name** (with free domain forwarding and email forwarding to your current addresses elsewhere).

CSS

To change the appearance of your whole Web page, create a **cascading style sheet (CSS)**.

For example, suppose you want red characters on a yellow background. You learned you can do that by changing `<body>` to this:

```
<body bgcolor=yellow text=red>
```

Here's a better way: in your `<head>` section, below the `<title>`, say this:

```
<style>
body {color:red; background:yellow}
</style>
```

That creates this style: throughout the body, make the text color be red, the background yellow. That makes most of the text be red (though normal links will still be blue underlined and the viewed links will still be purple underlined). When you type the second line, make sure you type braces, which look like this: `{}`. Don't type parentheses, which look like this: `()`.

Normally, the text is font size 3, which is 12 points. If you want the text to be slightly bigger (so people can read it more easily), request a bigger point size, such as 13 points, by saying `font-size=13pt`, like this:

```
<style>
body {color:red; background:yellow; font-size:13pt}
</style>
```

That makes most text get bigger, but headings will be unchanged. For example, `<h1>` headings will still be font size 6 (which is 24 points).

Should you use style sheets?

Style sheets were invented recently. They're new, hip, cool, and recommended.

For example, to get a yellow background, you ought to say `background:yellow` in the style sheet, rather than `bgcolor=yellow` in the `<body>`.

Using style sheets is **recommended**. Giving older types of commands, such as `bgcolor=yellow`, is **deprecated** (which means "pooh-poohed").

But many people still use older types of commands, such as `bgcolor=yellow`, since they work even on old computers whose browsers were invented before style sheets.

Links

You can change the color of links:

```
<style>
body {color:red; background:yellow}
:link {color:green}
:visited {color:fuchsia}
:hover {color:navy}
</style>
```

That makes most links be green, recently visited links be fuchsia, and each link temporarily turn navy while the mouse hovers over it.

Headers

You learned that you can create big headers by saying `<h1>`, smaller headers by saying `<h2>`, and even smaller headers by saying `<h3>`, `<h4>`, `<h5>`, and `<h6>`. Normally, headers are the same color as the body text. For example, if you made the body text be red, the headers are automatically red also.

To make `<h1>` headers be blue and all other headers be maroon, say so in the style sheet, like this:

```
<style>
body {color:red; background:yellow}
:link {color:green}
:visited {color:fuchsia}
:hover {color:navy}
h1 {color:blue}
h2,h3,h4,h5,h6 {color:maroon}
</style>
```

To make `<h1>` headers be blue and also centered (instead of at the screen's left edge), make the style sheet's `h1` line be this:

```
h1 {color:blue; text-align:center}
```

Frankly, I hope you don't choose those colors! Your Web page will be too wild if you actually make the body text red, the background yellow, the links green, the visited links fuchsia, the hovered links navy, the big headings blue, and the smaller headings maroon. Choose more reasonable colors.

Normally, `<h1>` headers are font size 6, which is 24 points. To make `<h1>` headers be even bigger, give a bigger point size, such as by saying:

```
h1 {color:blue; text-align:center; font-size:40pt}
```

Paragraphs

You learned to put `<p>` at the beginning of each paragraph. Normally, the `<p>` makes the computer put a blank line above the paragraph. If you want the computer to omit the blank line and indent the paragraph's first word, put this line in your style sheet:

```
p {text-indent:2em; margin-top:0; margin-bottom:0}
```

Sharing

Several Web pages can share a style sheet. Here's how.

Using Notepad, create the style sheet, but omit the `<style>` and `</style>` lines. For example, create this style sheet:

```
body {color:red; background:yellow}
```

Save it on your Desktop, but give it a name that ends in `.css` (which stands for Cascading Style Sheet) instead of `.html`. For example, name it `mystyle.css`.

Then create your Web pages; but on each Web page, tell the computer to use the style sheet you created (`mystyle.css`), by putting this line in the `<head>` part of the Web page:

```
<link rel=stylesheet href=mystyle.css>
```

That tells the computer to create a link, related to your style sheet, which is located at the HTTP reference "mystyle.css".

JavaScript

Pages 545-552 explained how to create Web pages by using HTML. Unfortunately, HTML is *not* a complete programming language.

For example, HTML lacks commands to do arithmetic. In HTML, there is no command to make the computer do 2+2 and get 4.

HTML lacks commands to create repetitions (which are called **loops**). In HTML, there is no command to make the computer repeat a task 10 times.

In 1996, a Netscape employee, Brendan Eich, invented an HTML supplement called **LiveScript**, which lets you create Web pages that do arithmetic, loops, counting, and many other fancy tricks. When folks noticed that LiveScript looks like a stripped-down version of Java, Netscape changed the name "LiveScript" to **JavaScript**.

JavaScript is included as part of Netscape Navigator (if you have Navigator version 2 or later). **JScript** (Microsoft's imitation of JavaScript) is included as part of Internet Explorer (if you have Internet Explorer version 3 or later).

Now every popular computer comes with JavaScript or JScript. That's because Netscape Navigator is free, Internet Explorer is free, Netscape Navigator & Internet Explorer are both available for IBM and Macs, and Internet Explorer is part of Windows.

Netscape, Microsoft, and the **European Computer Manufacturers Association (ECMA)** all decided to make JavaScript and JScript resemble each other more, by creating a standard called **ECMAScript**.

This chapter explains how to use JScript to create powerful Web pages. (JavaScript and ECMAScript are similar.)

Before learning JScript, make sure you've learned HTML (by reading pages 545-552).

JScript uses these commands:

JScript command	Page
alert("warning: bad hair")	556
document.write(2+2)	554
else	558
for (i=1; i<10; ++i)	558
if (age<18)	558
x=Array(3)	556
x=prompt("what name?", "")	556
x=47	556
x[0]="love"	556
++x	556
--x	556
//I wrote this while drunk	559

Simple program

You can create a Web page that says —

```
We love you
```

by typing this HTML program:

```
we <i>love</i> you
```

I explained how on page 546. (If you forget how, reread page 546 and practice it now.)

To create a Web page that makes the computer do 2+2 instead, type instead this HTML program (which includes a JScript program):

```
<script>
document.write(2+2)
</script>
```

The first line, which says <script>, warns the computer that you're going to start typing a JScript (or JavaScript) program. The next line, which is written in JScript, means: on the Web-page document, write the answer to 2+2. The bottom line, which says </script>, marks the bottom of your JScript program. When you run that program, the computer will do 2+2 and write this answer:

```
4
```

In that example, the first line, <script>, is an HTML tag. Like all HTML tags, it's enclosed in angle brackets: the symbols <>. That tag marks the beginning of your JScript program. The bottom line, </script>, is an HTML tag that marks the end of your JScript program. Between those two tags, write your JScript program.

```
4
```

In that example, the first line, <script>, is an HTML tag. Like all HTML tags, it's enclosed in angle brackets: the symbols <>. That tag marks the beginning of your JScript program. The bottom line, </script>, is an HTML tag that marks the end of your JScript program. Between those two tags, write your JScript program.

Longer example

Let's make the computer write "We love you", then write the answer to 2+2, then write "ever and ever". This program does it:

```
we <i>love</i> you
<script>
document.write(2+2)
</script>
ever <i>and ever</i>
```

The first line makes the computer write "We love you". The next three lines hold the JScript program making the computer write the answer to 2+2, which is 4. The bottom line makes the computer write "ever and ever". So altogether, the computer will write:

```
We love you 4 ever and ever
```

Fancier arithmetic

This program makes the computer write the answer to 8-3:

```
<script>
document.write(8-3)
</script>
```

The computer will write:

```
5
```

This program makes the computer write the answer to -26.3+1:

```
<script>
document.write(-26.3+1)
</script>
```

The computer will write:

```
-25.3
```

Multiplication To multiply, use an asterisk. So to multiply 2 by 6, type this:

```
<script>
document.write(2*6)
</script>
```

The computer will write:

```
12
```

Division To divide, use a slash. So to divide 8 by 4, type this:

```
<script>
document.write(8/4)
</script>
```

The computer will write:

```
2
```

Avoid commas Do *not* put commas in big numbers. To write four million, do *not* write 4,000,000; instead, write 4000000.

E notation If the computer's answer is huge (at least 10000000000000000000000) or tiny (less than .000001), the computer will typically print an e in the answer. The e means "move the decimal point".

For example, suppose the computer says the answer to a problem is:

```
1.5864321775908348e+21
```

The e means, "move the decimal point". The plus sign means, "towards the right". Altogether the e+21 means, "move the decimal point towards the right, 21 places." So look at 1.5864321775908348, and move the decimal point towards the right, 21 places; you get 1586432177590834800000.

So when the computer says the answer is 1.5864321775908348, the computer really means the answer is 1586432177590834800000, approximately. The exact answer might be 1586432177590834800000.2 or 1586432177590834800000.79 or some similar number, but the computer prints just an approximation.

Suppose your computer says the answer to a problem is:

9.23e-7

After the e, the minus sign means, “towards the *left*”. So look at 9.23, and move the decimal point towards to left, 7 places. You get:
.00000923

You’ll see e notation rarely: the computer uses it just if the answer is huge or tiny. But when the computer *does* use e notation, remember to move the decimal point!

The highest number The highest number the computer can handle well is about 1E308, which is 1 followed by 308 zeros. If you try to go much higher, the computer will give up and say the answer is:

Infinity

The tiniest decimal The tiniest decimal the computer can handle accurately is 1E-309 (which is a decimal point followed by 309 digits, 308 of which are zeros). If you try to go tinier, the computer will either write 0 or give you a rough approximation.

Long decimals If an answer is a decimal that contains *many* digits, **the computer will typically write the first 16 significant digits accurately and the 17th digit approximately**. The computer won’t bother writing later digits.

For example, suppose you ask the computer to write 100 divided by 3, like this:

```
<script>
document.write(100/3)
</script>
```

The computer will write:

33.33333333333336

Notice that the 17th digit, the 6, is wrong: it should be 3.

Division by 0 If you try to divide 1 by 0, the computer will say the answer is:

Infinity

If you try to divide 0 by 0, the computer will say the answer is —

NaN

which means “Not a Number”.

Order of operations JScript (and JavaScript) handle order of operations the same as QBasic, Visual Basic, and most other computer languages.

For example, if you type this program —

```
<script>
document.write(2+3*4)
</script>
```

the computer will “start with 2 then add three 4’s”, so it will write this answer:

14

You can use parentheses the same way as in algebra. For example, if you type —

```
<script>
document.write(5-(1+1))
</script>
```

the computer will compute 5-2 and write:

3

Strings

You learned how to put a JScript (or JavaScript) program in the middle of an HTML program. You can also do the opposite, you can put HTML in the middle of a JScript program.

For example, this JScript program makes the computer write “We *love* you”:

```
<script>
document.write("We <i>love</i> you")
</script>
```

The computer will write:

We *love* you

In that program, the “We *love* you” is called a **string** of characters. Each string must begin and end with a quotation mark. Between the quotation marks, put any characters you want the computer to write. A string can include an HTML tag, such as *<i>*.

Strings with numbers If you bought 750 apples and buy 12 more, how many apples do you have altogether? This program makes the computer write the answer:

```
<script>
document.write(750+12, " apples")
</script>
```

The computer will write the answer to 750+12 (which is 762) then the word “apples” (which includes a blank space), so altogether the computer will write:

762 apples

This program makes the computer put the answer into a complete sentence:

```
<script>
document.write("You have ",750+12, " apples!")
</script>
```

The computer will write “You have ” then 762 then “apples!”, so altogether the computer will write:

You have 762 apples!

Writing several strings Here’s another example of strings:

```
<script>
document.write("fat")
document.write("her")
</script>
```

The computer will write “fat” then “her”, so altogether the computer will write:

father

Let’s make the computer write this instead:

```
fat
her
```

To do that, make the computer press the Enter key before her. Here’s how: say *
* (which is the HTML tag to break out a new line), like this —

```
<script>
document.write("fat")
document.write("<br>her")
</script>
```

or like this:

```
<script>
document.write("fat<br>her")
</script>
```

Addition You can add strings together by using the + sign:

```
"fat"+"her" is the same as "father"
2+2+"ever" is the same as "4ever"
```

Variables

A letter can stand for a number. For example, x can stand for the number 47, as in this program:

```
<script>
x=47
document.write(x+2)
</script>
```

The second line says x stands for the number 47. In other words, x is a name for the number 47.

The next line says to write x+2. Since x is 47, the x+2 is 49; so the computer will write:

49

That’s the only number the computer will write; it won’t write 47.

A letter that stands for a number is called a **numeric variable**.

A letter can stand for a string. For example, y can stand for the string “We *love* you”, as in this program:

```
<script>
y="We <i>love</i> you"
document.write(y)
</script>
```

The computer will write:

We *love* you

A letter that stands for a string is called a **string variable**.

A variable's name can be short (such as `x`) or long (such as `town_population_in_2001`). It can be as long as you wish! The name can contain letters, digits, and underscores, but not blank spaces. The name must begin with a letter or underscore, not a digit.

Increase The symbol `++` means "increase". For example, `++n` means "increase `n`".

This program increases `n`:

```
<script>
n=3
++n
document.write(n)
</script>
```

The `n` starts at 3 and increases to 4, so the computer prints 4.

Saying `++n` gives the same answer as `n=n+1`, but the computer handles `++n` faster.

The symbol `++` increases the number by 1, even if the number is a decimal. For example, if `x` is 17.4 and you say `++x`, `x` will become 18.4.

Decrease The opposite of `++` is `--`. The symbol `--` means "decrease". For example, `--n` means "decrease `n`". Saying `--n` gives the same answer as `n=n-1` but faster.

Arrays A letter can stand for a list. For example, `x` can stand for a list, as in this program:

```
<script>
x=["love", "death", 48+9]
document.write(x)
document.write(x[2]/4)
</script>
```

That makes `x` be a list of three items: "love", "death", and the answer to `48+9` (which is 57). The next line makes the computer write all of `x`, like this:

```
love,death,57
```

In `x` (which is a list), there are 3 items:

```
The original item, which is called x[0], is "love".
The next item, which is called x[1], is "death".
The next item, which is called x[2], is 57.
```

The next line says to write `x[2]/4`, which is `57/4`, which is 14.25; but since we didn't say `
`, the computer writes the 14.25 on the same line as the list, so altogether you see:

```
love,death,5714.25
```

A list is called an **array**.

If you want `x` to be a list of 3 items but don't want to list the 3 items yet, you can be vague by saying just —

```
x=Array(3)
```

Later, you can define `x` by lines such as:

```
x[0]="love"
x[1]="death"
x[2]=48+9
```

Pop-up boxes

Here's how to make a box appear suddenly on your screen.

Alert box To create a surprise, make the computer create an **alert box**:

```
<script>
alert("warning: your hair looks messy today")
document.write("You won't become Miss America")
</script>
```

When a human runs that program, the screen suddenly shows an **alert box**, which contains this message: "Warning: your hair looks messy today". (The computer automatically makes the box be in front of the Web page, be centered on the screen, and be wide enough to show the whole message.) The alert box also contains an OK button. The computer waits for the human to read that alert message and click "OK".

When the human clicks "OK", the alert box disappears and the computer obeys the program's next line, which makes the computer write onto the Web page:

```
You won't become Miss America
```

In an alert box, the computer uses its alert font, which you cannot change: you *cannot* switch to italics or bold; you *cannot* put HTML tags into that message.

Here's another example:

```
<script>
alert("You just won a million dollars")
document.write("Oops, I lost it, better luck next time")
</script>
```

When a human runs that program, an alert box tells the human "You just won a million dollars"; but when the human clicks "OK", the Web page says "Oops, I lost it, better luck next time".

Prompt box To ask the human a question, make the computer create a **prompt box**:

```
<script>
x=prompt("what is your name?","")
document.write("I adore anyone whose name is ",x)
</script>
```

When a human runs that program, the computer creates a **prompt box**, which is a window letting the human type info into the computer. (The computer automatically makes the box be in front of the Web page and be slightly above the screen's center.) It contains this **prompt**: "What is your name?" It also contains a white box (into which the human can type a response) and an OK button.

The computer waits for the human to type a response. When the human finishes typing a response, the human must click the OK button (or press Enter) to make the window go away.

Then the Web page reappears and the computer makes `x` be whatever the human typed. For example, if the human typed —

```
Maria
```

`x` is Maria, so the computer writes this onto the Web page:

```
I adore anyone whose name is Maria
```

In that program, notice that the prompt line includes these symbols before the last parenthesis:

```
,""
```

If you type this instead —

```
,"Type your name here"
```

here's what happens: the white box (into which the human types a name) will temporarily say "Type your name here", until the human starts typing.

College admissions This program makes the computer write a letter admitting you to the college of your choice:

```
<script>
college=prompt("what college would you like to enter?","")
document.write("You're admitted to ",college,". I hope you go to ",college,".")
</script>
<p>Respectfully yours,
<br>The Dean of Admissions
```

When you run the program, a prompt box appears, asking “What college would you like to enter?” Type your answer (then click OK or press Enter).

For example, if you type —

Harvard

the college will be “Harvard”, so the computer will write “You’re admitted to” then “Harvard” then “. I hope you go to ” then “Harvard”, then “.” then the remaining HTML code, like this:

```
You're admitted to Harvard. I hope you go to Harvard.

Respectfully yours,
The Dean of Admissions
```

If you type this instead —

Hell

the computer will write:

```
You're admitted to Hell. I hope you go to Hell.

Respectfully yours,
The Dean of Admissions
```

All the writing is onto your screen’s Web page. Afterwards, if you want to copy that writing onto paper, click Internet Explorer’s Print button. (If you don’t see the Print button, make it appear by maximizing the Internet Explorer window.)

Numeric input This program makes the computer predict your future:

```
<script>
y=prompt("In what year were you born?","")
document.write("In the year 2020, you'll turn ",2020-y," years old")
</script>
```

When you run the program, the computer asks, “In what year were you born?” If you answer —

1962

y will be 1962, and the computer will write:

```
In the year 2020, you'll turn 58 years old.
```

Control statements

A program is a list of statements that you want the computer to perform. Here’s how to control which statements the computer performs, and when, and in what order.

If This program makes the computer discuss the human’s age:

```
<script>
age=prompt("How old are you?","")
document.write("I hope you enjoy being ",age)
</script>
```

When that program is run, the computer asks “How old are you?” and waits for the human’s reply. For example, if the human says —

15

the age will be 15. Then the computer will print:

```
I hope you enjoy being 15
```

Let's make that program fancier, so if the human is under 18 the computer will also say "You are still a minor". To do that, just add a line saying:

```
if (age<18) document.write("<br>You are still a minor")
```

Notice you must put parentheses after the word "if". Altogether, the program looks like this:

```
<script>
age=prompt("How old are you?","")
document.write("I hope you enjoy being ",age)
if (age<18) document.write("<br>You are still a minor")
</script>
```

For example, if the human runs the program and says —

```
15
```

the computer will print:

```
I hope you enjoy being 15
You are still a minor
```

If instead the human says —

```
25
```

the computer will print just:

```
I hope you enjoy being 25
```

Else Let's teach the computer how to respond to adults.

Here's how to program the computer so that if the age is less than 18, the computer will say "You are still a minor", but if the age is *not* less than 18 the computer will say "You are an adult" instead:

```
<script>
age=prompt("How old are you?","")
document.write("I hope you enjoy being ",age)
if (age<18) document.write("<br>You are still a minor")
else document.write("<br>You are an adult")
</script>
```

In programs, **the word "else" means "otherwise"**. The program says: if the age is less than 18, write "You are still a minor"; otherwise (if the age is *not* less than 18), write "you are an adult". So the computer will write "You are still a minor" or else write "You are an adult", depending on whether the age is less than 18.

Try running that program! If you say you're 50 years old, the computer will reply by saying:

```
I hope you enjoy being 50
You are an adult
```

Fancy relations Java's "if" statement uses this notation:

Notation	Meaning
if (age<18)	if age is less than 18
if (age<=18)	if age is less than or equal to 18
if (age==18)	if age is equal to 18
if (age!=18)	if age is not equal to 18
if (age<18 && weight>200)	if age<18 and weight>200
if (age<18 weight>200)	if age<18 or weight>200
if (sex=="male")	if sex is "male"
if (sex<"male")	if sex is a word (such as "female") that comes before "male" in dictionary
if (sex>"male")	if sex is a word (such as "neuter") that comes after "male" in dictionary

Notice that in the "if" statement, you should use double symbols: you should say "==" instead of "=", say "&&" instead of "&", and say "||" instead of "|". If you accidentally say "=" instead of "==", the computer will gripe. If you accidentally say "&" instead of "&&" or say "|" instead of "||", the computer will still get the right answers but too slowly.

Braces If a person's age is less than 18, let's make the computer write "You are still a minor" and make maturity=0. Here's how:

```
if (age<18)
{
    document.write("You are still a minor")
    maturity=0
}
```

Here's a fancier example:

```
if (age<18)
{
    document.write("You are still a minor")
    maturity=0
}
else
{
    document.write("You are an adult")
    maturity=1
}
```

For Here's how to write the numbers from 1 to 10:

```
<script>
for (i=1; i<=10; ++i) document.write(i," ")
</script>
```

That means: do repeatedly, for i starting at 1, while i is no more than 10, and increasing i after each time: write i followed by a blank space (to separate i from the next number). The computer will write:

```
1 2 3 4 5 6 7 8 9 10
```

If instead you want to write each number on a separate line, say "
" (which means "break for new line") before each number:

```
<script>
for (i=1; i<=10; ++i) document.write("<br>",i)
</script>
```

The computer will write:

```
1
2
3
4
5
6
7
8
9
10
```

Let's get fancier! For each number, let's make the computer also write the number's **square** (what you get when you multiply the number by itself), like this:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

Here's how:

```
<script>
for (i=1; i<=10; ++i) document.write("<br>",i," squared is ",i*i)
</script>
```

To get even fancier, let's make the computer write that info in a pretty table, like this:

NAME	SCORE
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

As I explained on page 549, you do that by saying `<table border=1>` above the table, `<tr>` at the beginning of each table row, `<th>` at the beginning of each column heading, `<td>` at the beginning of each data item, and `</table>` below the table:

```
<table border=1>
<tr><th>NAME<th>SCORE
<script>
for (i=1; i<=10; ++i) document.write("<tr><td>",i,"<td>",i*i)
</script>
</table>
```

OnClick Let's create a Web page that asks, "What sex are you?" Below that question, let's put two buttons labeled "Male" and "Female". If the human clicks the "Male" button, let's make the computer say "So is Frankenstein". If the human clicks the "Female" button, let's make the computer say "So is Mary Poppins".

To accomplish all that, just type this HTML:

```
What sex are you?
<form>
<input type=button value="Male" onclick="alert('So is Frankenstein')">
<input type=button value="Female" onclick="alert('So is Mary Poppins')">
</form>
```

Here's what each line accomplishes:

Since we want the Web page to begin by asking "What sex are you?", the top line says "What sex are you?"

To create buttons, you must create a form to put them in, so the second line says `<form>`.

The next line says to create an input button labeled "Male", which when clicked will do this command: create an alert box saying "So is Frankenstein".

The next line says to create a similar input button labeled "Female", which when clicked will do this command: create an alert box saying "So is Mary Poppins".

The bottom line, `</form>`, marks the end of the form.

Notice these details:

After `onclick`, you put an equal sign, then a quotation mark, then any command written in JavaScript (or JScript), such as "alert". The computer knows the `onclick` command uses JavaScript, so you don't have to say `<script>`.

The JavaScript command must be in a pair of quotation marks. If you want to put a pair of quotation marks inside another pair of quotation marks, use a pair of single quotes (which look like apostrophes).

After `onclick`, instead of typing a JavaScript command, you can type *several* JavaScript commands, if you separate them by semicolons, like this:

```
onclick="x=4; y=2; alert(x+y)"
```

That would mean: if the button is clicked, make `x=4`, make `y=2`, and create an alert box showing their sum, 6.

When you create two buttons, the second button normally appears to the *right* of the first button. If you'd rather place the second button *below* the first button, say `
` before the second button to put it on a new line, like this:

```
<br><input type=button value="Female" onclick="alert('So is Mary Poppins')">
```

Documentation

On page 551, I said you can write a comment in your HTML program by starting with the symbol "`<!--`" and ending with the symbol "`-->`", like this:

```
<!--I wrote this program while drunk-->
```

But while you're writing JavaScript (or JScript) program lines, which comes between `<script>` and `</script>`, you must write your comments differently, in JavaScript style: put each comment on a separate line that begins with `//`, like this:

```
//I wrote these JavaScript lines while even drunker
```

Emphasize JavaScript To emphasize that your program is written in JavaScript (or a JavaScript clone such as JScript), you can say —

```
<script language="JavaScript">
```

or even say —

```
<script language="JavaScript" type="text/javascript">
```

instead of saying just `<script>`.

No JavaScript? Most Web browsers understand JavaScript and JScript programs. But Web browsers that are very old or very primitive don't understand JavaScript at all.

If your Web-page program contains a JavaScript program, but somebody who lacks JavaScript tries to view your Web page, the page will look very messed up, and the person might even see your raw JavaScript code, including equal signs and words such as "document.write".

To make sure such a person doesn't see your raw code on the Web page, say this instead of just `<script>` —

```
<script>
<!--
```

and say this instead of just `</script>`:

```
//-->
</script>
```

Also, warn the JavaScript-deprived person that your page requires JavaScript, by putting this line below the `</script>` line:

```
<noscript>This page requires JavaScript</noscript>
```

Here's what that line accomplishes: if the person has no JavaScript, the Web page will say "This page requires JavaScript".

Challenges

Some things are hard to program. Folks have tried....

Computer art

It can be hard to make the computer create good art.

Famous art programs

You can buy many kinds of graphics programs. On page 47, I described the best ones. Here are further comments.

Paint The easiest kind of graphics program to use is called a **paint program**. It lets you easily create pictures on your screen by using a mouse.

Here's how paint programs arose....

In January 1984, Apple Computer Company began selling the Mac computer. It was the first affordable computer that included a mouse —and the first affordable computer that included a good paint program.

The Mac's paint program was called **Mac Paint**.

It was invented at Apple Computer Company in 1984 by Bill Atkinson. It ran just on the Mac, was included free with the Mac and showed consumers why a Mac was better than an IBM PC: the Mac let you paint a picture on your screen, and the IBM PC couldn't do that yet.

I explained Mac Paint in the 14th edition of *The Secret Guide to Computers*. If you'd like that edition, phone me at 603-666-6644.

Mac Paint had one major limitation: it couldn't handle colors. It handled just black-and-white, because the original Mac came with just a black-and-white screen.

(Years later, Apple began charging for Mac Paint, Ann Arbor Software invented an improved version called "Full Paint", and Silicon Beach invented a further improvement called "Super Paint". Modern Macs have color.)

The next major advance was **Deluxe Paint**.

It was invented in 1985 by Dan Silva in California and published by Electronic Arts. It was much fancier than Mac Paint and performed gorgeous color tricks.

It ran just on Commodore's Amiga computer. Because of Deluxe Paint, the Amiga quickly developed a reputation as the best computer for generating color graphics.

(Years later, the Amiga faced competition, Commodore went bankrupt, and Electronic Arts made versions of Deluxe Paint for the IBM PC and the Apple 2GS. Deluxe Paint is no longer available.)

Windows includes a free a paint program.

In Windows 3.0, 3.1, and 3.11, the free paint program is called **Paintbrush**. It's a stripped-down version of "PC Paintbrush", which was invented by Z-Soft. Windows 95, 98, Me, XP, Vista, 7, 8, 8.1, and 10 include a free paint program called **Paint**, which is an improved Paintbrush. I explained Paint in the Windows chapter.

The best paint program for kids is **Kid Pix**.

It runs on all popular computers (IBM, Mac, and others). While you paint, it makes funny sounds and talks to you in both English and Spanish. Besides letting you create your own shapes, it includes lots of fun little pre-drawn shapes (stars, snowflakes, trees, etc.), which you can include in your paintings to create backgrounds and pixie dust.

By using Kid Pix, you can create impressive artwork in just a few seconds! Of all the paint programs you can buy, Kid Pix is the one that give you pleasure fastest! Though the pre-drawn shapes look kid-like, they look like they come from *talented* kids! Kid Pix is the only program where it's even more fun to erase your work than to create it, since Kid Pix gives you many dramatic ways to get rid of your painting, such as by dynamiting it: boom!

Educators have given Kid Pix many awards for turning kids into creative artists.

Kid Pix was published by Broderbund but now is published by **Software MacKiev** (which has offices in Boston and the Ukraine city of Kiev). The newest, fanciest version, **Kid Pix 3D**, costs \$50.

The best paint program for professional artists is **Painter**, originally published by Fractal Design but now marketed by Corel.

It was designed for the Mac but now also runs on the IBM PC. Painter amazes artists because it makes the computer's screen accurately imitate different kinds of brushes, inks, and other artist tools. You can choose whether to make the screen look like you're painting in oil, chalk, charcoal, watercolor, or whatever other medium you wish. You can fine-tune each tool, change precisely how "drippy" each tool is, and change the "bumpiness" of the paper's texture.

It even includes a "van Gogh" mode, which lets you paint by using the same kinds of brushstrokes as the artist Vincent van Gogh.

Though Painter can use a mouse, Painter imitates artist tools more accurately if you buy a **pressure-sensitive graphics tablet** (which comes with a pen that records not just *where* you're pressing but also *how hard* you're pressing). The most popular pressure-sensitive tablets are made by **Wacom, Kurta, Calcomp, and Summagraphics**.

Painter is expensive. The newest version, **Painter X3**, costs \$429. You can buy stripped-down versions, called **Painter Lite** (\$70) and **Painter Essentials 4** (\$60). You can download free 30-day trial versions of those programs at www.corel.com.

Since Painter is intended just for creative artists who like to draw squiggles, it doesn't contain commands to draw an oval, circle, rectangle, or square. All other popular paint programs include such commands.

How paint programs work Each paint program considers your screen to be made of thousands of pixels (dots). The paint program remembers the color of each pixel. The colors of all the pixels are stored in RAM while you're painting. You need lots of RAM if your screen is large & has many pixels, or if you insist on using lots of colors.

All paint programs suffer from this problem:

If you use a paint program to create a shape, then try to shrink that shape, then change your mind and try to expand the shape back to its original size, the final result looks crude and lacks the details that were in the original.

That's because a paint program shrinks a shape by using fewer pixels: some of the pixels that contained details are discarded. The lack of detail becomes noticeable when you try to expand the shape back to its original size.

Another problem is that when you try to rotate a shape, the shape looks cruder, because the shape's pixels get slightly misplaced by "round-off error". If you try to rotate a shape several times, the pixels get progressively more misplaced, and the shape looks cruder and cruder.

When trying to paint, if you expand or twirl,
You get a result that makes you want to hurl.

Paint programs are called **bitmapped graphics programs**.

Draw A **draw program** does *not* store the color of each pixel. Instead, a draw program stores a memo about a geometric shape and the color of the entire shape.

For example, a draw program stores a line by storing just its starting point, ending point (or angle & length), and color; it stores a circle by storing just the circle's center, radius, and color. By contrast, a paint program would consume lots of RAM storing the color of each of the thousand of pixels that are on the line or circle.

Draw programs are also called **vector-based graphics programs**.

A draw program works faster and more accurately than a paint program if you're drawing geometric shapes. A draw program has no problem handling expansions and rotations. But it has difficulty handling squiggles, since it tries to view each squiggle as made up of many tiny arcs.

A draw program lets you name different objects, put them in front of other objects, then later move the objects to reveal objects that were hidden. Most paint programs can't do that: in a paint program, creating a new shape automatically erases any shape that was underneath — except for a crude feature that lets you have two "layers": a "background" and a "foreground".

Classic computer art

In a draw program, you can point to an object you drew and change its color, thickness, or style. In a paint program, the only way to change the appearance of what you drew is to draw it over again.

Unfortunately, the typical draw program is confusing to use, because when you look at what's on your screen you're not sure which "objects" the stuff you're seeing is part of.

The first popular draw program was **Mac Draw**, which ran on the Mac. Now most draw programs use Windows instead. The most popular serious draw program is **Corel Draw**.

Each modern Windows word-processing program also includes a stripped-down draw program, free. For example, to draw while using Microsoft Word (version 2007 or 2010 or 2013), click "Insert" (which is near the screen's top-left corner) then "Shapes".

CAD You can buy a program that does **computer-aided drafting & design (CAD)**. Such a program resembles a draw program but does more math.

For example, it can print mock blueprints, with the lengths of all parts marked. It can even compute the surface area (square feet) of any shape, so you can compute how much material to buy to build your structure and cover it.

It lets you give fancy geometric commands, such as "draw a 37-degree angle, but make the point be round instead of sharp, so nobody gets hurt" or "draw a circular arc that goes through these three points" or "draw a line that grazes these two circles, so it's tangent to them".

The most famous CAD program is **AutoCAD**. That's what most architects and engineers use. It's published by Autodesk and very expensive: \$4195! Cheaper CAD programs that cost under \$50 are for use at home, to help Joe Six-Pack design his backyard deck — and help interior designers plan purple bathrooms.

Most CAD programs include pre-built shapes that you can put in your drawings.

The pre-built shapes are exactly the right size and shape to represent toilets, sinks, stoves, and other household fixtures. Each shape is called a **symbol**. You can buy a bunch of extra symbols; each bunch is called a **symbol library**.

Photo manipulation To put photographs into your computer, use a **scanner** (which scans in sheets of paper) or a **digital camera**.

Once the picture is in your computer, you can manipulate it by using a paint program. Better yet, use a program that specializes in the fine art of manipulating photos. The best photo-manipulation programs are:

Adobe's **Photoshop** (performs the fanciest tricks, but expensive & hard)

Adobe's **Photoshop Elements** (cheap & easy, for beginners)

Microsoft's **Digital Image Suite** (easy & great but no longer sold)

Kai's Power Goo (stretches a face to create weird expressions, for fun)

During the 1960's, many creative ideas were generated about how computers would someday create their own weird art, using a wild combination of formulas and random numbers, and unshackled by the bounds of human culture.

Here's how to make the computer produce wild art, by using the wonderful classic tricks invented in the 1960's and 1970's....

In 1971, Michael Hord made the computer turn photographs into artistic sketches. Here's what the computer did to a photograph of his boss, and to a photograph of a colleague's girlfriend:

Boss



Woman



To draw each sketch, the computer scanned the original photograph and found the points where the photograph changed dramatically from light to dark. Then, on a sheet of paper, it plotted those points; and through each of those points, it drew a short line perpendicular to the direction in which the original photograph darkened.

More precisely, here's what the computer did.... It looked at four adjacent points on the original photograph:

A B
C D

It computed the darkness of each of those points. Then it computed the "darkening in the X direction", defined as:

$$(\text{darkness at B}) + (\text{darkness at D}) - (\text{darkness at A}) - (\text{darkness at C})$$

Then it computed the "darkening in the Y direction", defined as:

$$(\text{darkness at A}) + (\text{darkness at B}) - (\text{darkness at C}) - (\text{darkness at D})$$

Then it computed the "overall darkening", defined as:

$$(\text{darkening in the X direction})^2 + (\text{darkening in the Y direction})^2$$

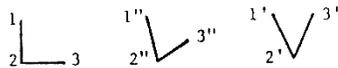
If the overall darkening there turned out to be large, the computer sketched a short line, in the vicinity of the points ABCD, and perpendicular to the direction of darkening. More precisely, the line's length was 1, and the line's slope was:

$$\frac{\text{darkening in the X direction}}{\text{darkening in the Y direction}}$$

Morphs Here's how to make an L slowly become a V. Notice that the letters L and V are both made by connecting three points:



Let 1" be the point halfway between 1 and 1'; let 2" be halfway between 2 and 2'; and let 3" be halfway between 3 and 3'. Then 1", 2", and 3" form a shape that's halfway between an L and a V:



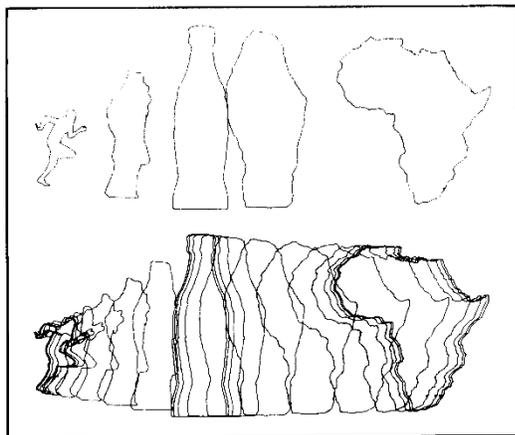
The process can be extended further:



Turning one shape into another (such as turning an L into a V) is called **a metamorphosis** or **morphing**. The intermediate shapes (which are between the L and the V) are called the **morphs**.

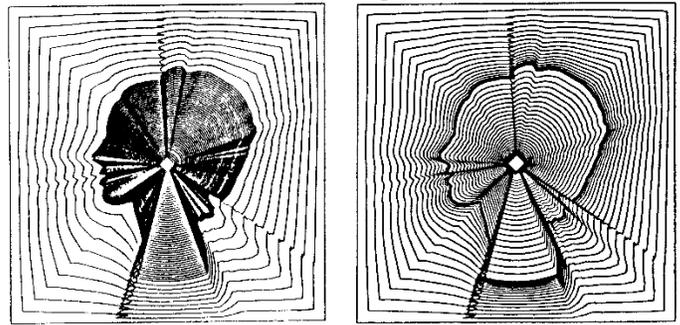
Using that method, the Computer Technique Group of Japan gradually turned a running man into a Coke bottle then into Africa:

Running Cola is Africa



The group turned this head into a square:

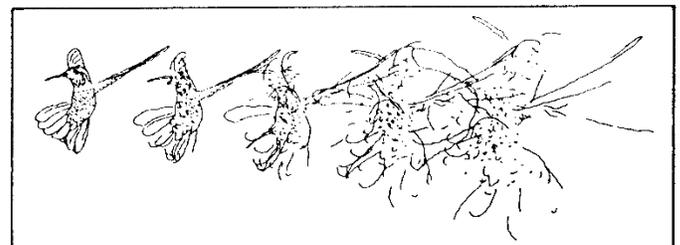
Return to a Square



The head on the left returns to a square by using **arithmetic progression**: the lines are equally spaced. The one on the right uses **geometric progression** instead: the lines are close together near the inside square, but further apart as they expand outward.

Csuri & Shaffer exploded a hummingbird:

Chaos to Order



The hummingbird at the far right was obtained from the one at the far left, by moving each line a random distance and in a random direction (between 45° and -45°).

Computers can make movies.

The best movie ever made by a computer is called *Hunger* (or *La Faim*). It was made back in 1973 by Peter Foldés under the auspices of the Canadian Film Board. Watch it at www.nfb.ca/film/Hunger.

It's a 10-minute cartoon, in color, with music, but goes far beyond anything ever done by Walt Disney. It uses the same technique as *Running Cola is Africa*: it shows objects turning into other objects.

It begins by showing a harried, thin executive at his desk, which has two phones. One of the phones rings. He answers it. While he's talking on that phone, his other phone rings. To talk on both phones simultaneously, his body splits in two. (How does a single body become two bodies? By using the same technique as turning a running man into a Coke bottle.)

On the other side of his desk is an armchair, which turns into a secretary, whose head turns into a clock saying 5PM, which tells the executive to go home. So he stretches his arms in front of him, and becomes his car: his hands become the headlights, his arms become the front fenders, his face becomes the windshield. You have to see it to believe it.

He drives to a restaurant and gets the waitress, who turns into an ice-cream cone. Then he eats her.

As the film progresses, he becomes increasingly fat, lustful, slothful, and miserable. In the end, he falls into hell, where he's encircled by all the poor starving naked children of the world, who eat his flesh. Then the film ends. (Don't see it before eating dinner!)

It combines computer art and left-wing humanitarian politics, to create an unforgettable message.

Using similar techniques, a 30-second movie called *Run* races through what's it's like to be born, live, and die in Japan. Watch it at:

www.YouTube.com/watch?v=Gk3-no1foTE

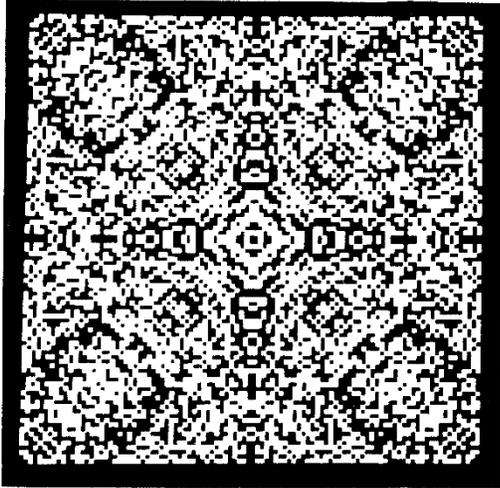
Now morphing is being applied to color photos and video images. For example, Hollywood movies use morphing to show a person gradually turning into a monster; environmentalists use morphing to show a human baby gradually turning into a spotted owl; and portrait photographers who have gone high-tech use morphing to show you gradually turning into the person you admire most (such as your movie idol or your lover).

Order versus disorder Computer artists are starting to believe that **art is a tension between order and disorder**. Too much order, or too much disorder, will bore you. For example, in *Chaos to Order*, the hummingbird on the left is too orderly to be art. The hummingbird on the right is more interesting.

Return to a Square uses arithmetic progression and geometric progression to create an over-all sense of order, but the basic elements are *disorderly*: a head that's bumpy, and a panorama of weird shapes that lie uncomfortably between being heads and squares but are neither.

Many programs create disorder by random numbers. *Chaos to Order* uses random numbers to explode the hummingbird.

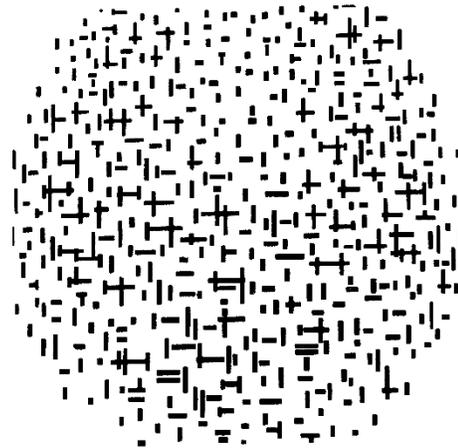
An amazing example of random numbers is this picture by Julesz & Bosche:



To your eyes, the picture seems quite ordered. Actually, it's quite *disordered*. One pie-shaped eighth of it is entirely random; the other seven eighths are copies of it. The copying is the only element of order, but very powerful. Try this experiment: *cover seven-eighths of the picture*. You'll see that the remaining eighth is totally disordered, hence boring.

That program imitates a child's *kaleidoscope*. Do you remember your childhood days, when you played with your kaleidoscope? It was a cardboard "telescope" that contained a disorganized pile of colored glass and stones, plus a series of mirrors that produced eight-way symmetry, so that what you saw resembled a giant multicolored snowflake. The program by Julesz & Bosche uses the same technique, computerized. Hundreds of programmers have imitated Julesz & Bosche, so now you can buy kaleidoscope programs for the IBM PC, Mac, and classic computers (Apple 2 and Radio Shack TRS-80). Or try writing your own!

Take this test:



One of those is a famous painting (*Composition with Lines*, by Piet Mondrian, 1917). The other was done by a computer (programmed by A. Michael Noll in 1965). *Which one was done by the computer? Which one do you like best?*

The solution is on the next page, but *don't peek until you've answered!*

The computer did the top one.
 The programmer surveyed 100 people. Most of them (59) thought the computer did the bottom one. Most of them (72) preferred the top one — the one that was actually done by the computer.

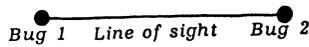
The test shows that people can't distinguish computer art from human art, and that the computer's art is more pleasing than the art of a famous painter.

The computer's version is more disordered than Mondrian's. The computer created the disorder by using random numbers. The survey shows that most people like disorder: Mondrian's work is too ordered. It also shows that most people mistakenly think the "computer" means "order".

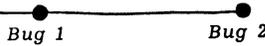
Envelopes Try this experiment. On a piece of paper, put two dots, like this:



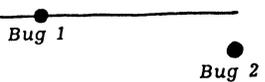
The dots represent little insects, or "bugs". The first bug is looking at the second bug. Draw the first bug's line of sight:



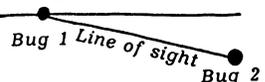
Make the first bug take a step toward the second bug:



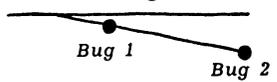
Make the second bug run away, in any direction:



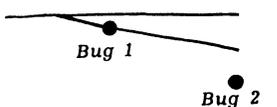
Now repeat the entire process. Again, bug 1 looks at bug 2; draw its line of sight:



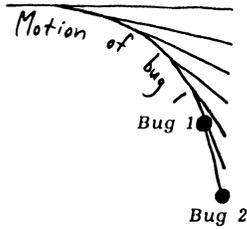
Bug 1 moves toward bug 2:



Bug 2 keeps running away:



If you repeat the process many times, you get this:



The "motion of bug 1" looks like a curve. (In fact, it's a parabola.) The "curve" is composed of many straight lines — the lines of sight. That's how to draw a fancy curve by using straight lines.

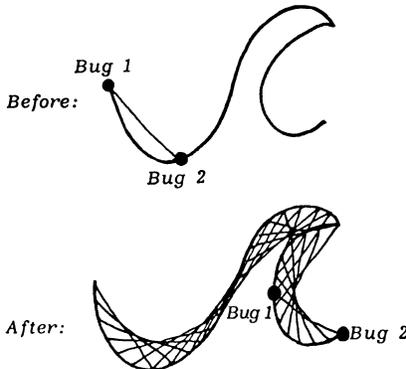
Each straight line is called a **tangent** of the curve. The entire collection of straight lines is called the curve's **envelope**. Creating a curve, by drawing the curve's envelope, is called **stitching the curve** — because the lines of sight act as threads, to produce a beautiful curved fabric.

You can program the computer to draw those straight lines. That's how to make the computer draw a fancy curve — even if you know nothing about "equations of curves".

To get a curve that's more interesting, try these experiments:

- What if bug 2 doesn't walk in a straight line? What if bug 2 walks in a curve instead?
- What if bug 1 goes slower than bug 2, and takes smaller steps?
- What if the bugs accelerate, or slow down?
- What if there are *three* bugs? What if bug 1 chases bug 2, while bug 2 chases bug 3, while bug 3 chases bug 1?
- What if there are *many* bugs? What if they all chase each other, and their starting positions are random?

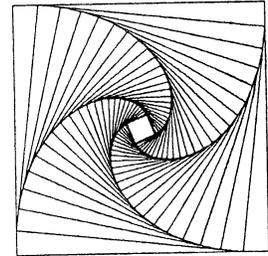
What if there are just two bugs, but the bugs are Volkswagens, which must drive on a highway having nasty curves? Show the bugs driving on the curved highway. Their lines of sight are still straight; but instead of moving along their lines of sight, they must move along the curve that represents the highway.



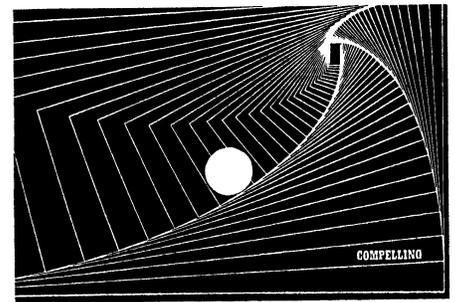
What if each bug has its own highway, and all the bugs stare at each other?

Here are some elaborate examples...

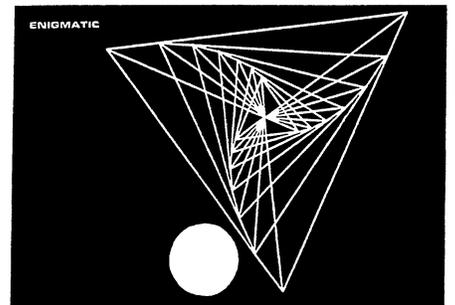
Four bugs chase each other:



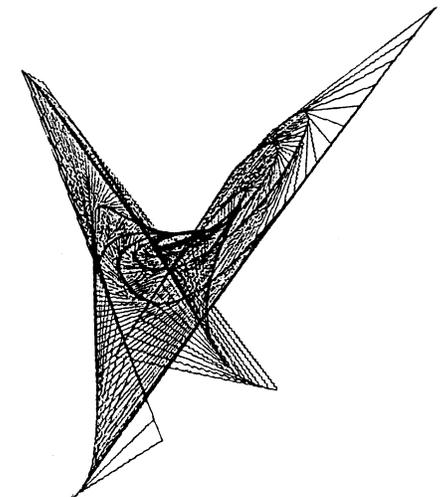
The next example, called *Compelling*, appeared in the famous book and movie, *The Dot and the Line*. (Norton Juster made it by modifying art that had appeared in *Scripta Mathematica*.) It resembles the previous example but makes the 4 bugs start as a rectangle (instead of a square), and makes the bug in the top left corner chase the bug in the opposite corner (while *looking* at a nearby bug instead).



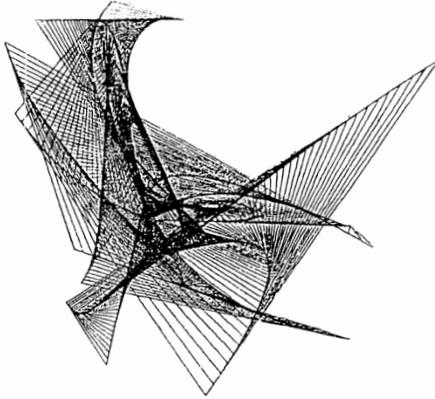
Enigmatic (from *The Dot and the Line*) makes 3 bugs chase each other, while a fourth bug stays motionless in the center:



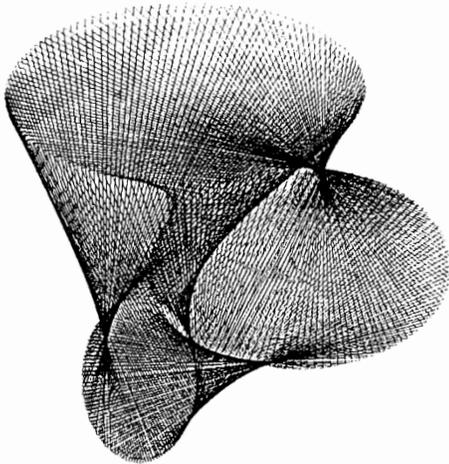
I invented *Kite*, which makes 8 bugs chase each other:



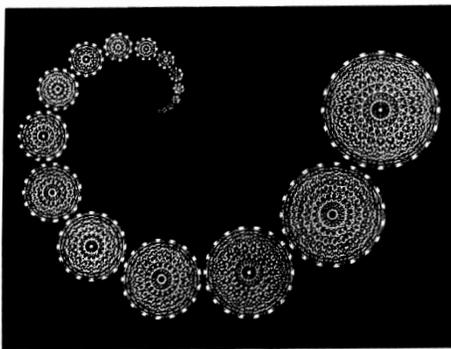
I also invented *Sails*, which makes 14 bugs chase each other:



Elliptic Motion (by my student Toby D'Oench) makes 3 bugs stare at each other, while they travel on 3 elliptical highways:



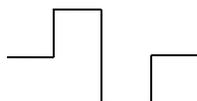
Archimedean Spiral (by Norton Starr) puts bugs on circles. The bugs stare at each other but don't move:



Fractals A **fractal** is an infinitely bumpy line. Here's how to draw one. Start by drawing a 1-inch segment:



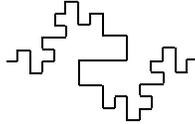
In the middle of that segment, put a bump and dip, like this:



Altogether, that bent path is 2 inches long.

In other words, if the path were made of string, and you stretched the string until it was straight, the string would be 2 inches long. That's twice as long as the 1-inch line segment we started with. So here's the rule: **putting a bump and dip in a path makes the path twice as long.**

That bent path consists of seven segments. Put a bump and a dip in the middle of each segment, like this:



Altogether, those bumps and dips make the path twice as long again, so now the path is 4 inches long.

Again, put a bump and dip in the middle of each segment, so you get this:



Again the path's length has been doubled, so now the path is 8 inches long.

If you again put a bump and dip in the middle of each segment, the path's length doubles again, so the path becomes 16 inches long. If you repeat the procedure *again*, the path reaches 32 inches.

If you repeat that procedure infinitely often, you'll develop a path that's infinitely wiggly and infinitely long. That path is longer than any finite line segment. It's longer than any finite 1-dimensional object. But it still isn't a 2-dimensional object, since it isn't an "enclosed area". Since it's bigger than 1-dimensional but not quite 2-dimensional, it's called **1½-dimensional**. Since 1½ contains a fraction, it's called **fractional-dimensional** or, more briefly, **fractal**.

Look out your window at the horizon. What do you see?

The horizon is a horizontal line with bumps (which represent hills and buildings and other objects). But on each hill you see tiny bumps, which are trees; and on each tree you see even tinier bumps, which are leaves; and on each leaf you see even tinier bumps, which are the various parts of the leaf; and each part of the leaf is made of even smaller bumps (molecules), which have even smaller bumps (atoms), which have even smaller bumps (subatomic particles).

Yes, the horizon is an infinitely bumpy line, a fractal!

You can buy software that creates fractals. Computer artists use fractal software to draw horizons, landscapes, and other bumpy biological objects. For example, they used fractal software to create landscapes for the *Star Wars*

movies. You can also use fractals to draw a bumpy face that has zillions of zits.

Now you understand the computer artist's philosophy of life: "Life's a lot of lumps."

What's art? To create art, write a weird program whose consequences you don't fully understand, tell the computer to obey it, and look at the computer's drawing. If the drawing looks nice, keep it and call it "art" — even if the drawing wasn't what you expected. Maybe it resulted from an error, but so what?

Anything interesting is art.

If the drawing "has potential" but isn't totally satisfying, change a few lines of the program and see what happens — or run the program again unchanged and hope the random numbers will fall differently. The last thing to invent is the title. Whatever the drawing reminds you of becomes the title.

For example, that's how I produced *Kite* and *Sails*.

I did *not* say to myself, "I want to draw a kite and sails". I just let the computer pick random starting points for the bugs and watched what happened. I said to myself, "Gee whiz, those drawings remind me of a kite and sails." So I named them *Kite* and *Sails*, and pretended I chose those shapes purposely.

That method may seem a long way from DaVinci, but it's how most computer art gets created. The rationale is: don't overplan.... let the computer "do its own thing"; it will give you art that escapes from the bounds of human culture and so expands your horizons!

Modern style Computer art has changed. The **classic style** — which you've been looking at — consists of hundreds of thin lines in mathematical patterns, drawn on paper and with little regard for color. The **modern style** uses big blobs and streaks of color, flashed on a TV tube or film, which is then photographed.

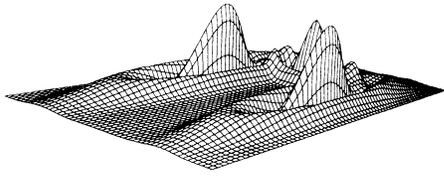
Uncreative art You've seen that computers can create their own weird art by using a wild combination of formulas and random numbers, unshackled by the bounds of human culture.

Computer programs let people create art easily and cheaply. Unfortunately, the typical person who buys a graphics program uses it to create the same kind of junk art that would be created by hand — just faster and more precisely. That's the problem with computers: they make the production of mediocrity even easier and more glitzy.

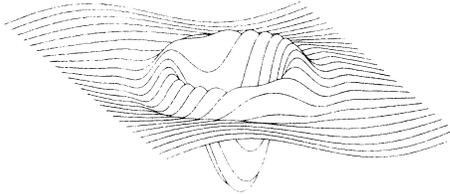
3-D drawing

The computer drew these 3-dimensional surfaces:

Three Peaks
by John Szabo

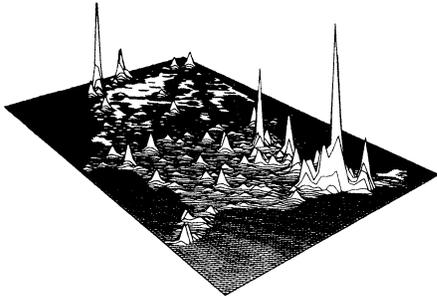


Dip



Those were done for the sake of art. This was done for the sake of science:

Population Density in the U.S.
by Harvard University Mapping Service



The hardest part about 3-dimensional drawing is figuring out which lines the computer should *not* show, because they're hidden behind other surfaces.

Compute the coordinates Try this experiment. Put your finger on the bridge of your nose (between your eyes). Now move your finger 2 inches to the right (so your finger is close to your right eye). Then move your finger 3 inches up (so your finger is near the upper right corner of your forehead). From there, move your finger 8 inches forward (so your finger is 8 inches in front of your forehead).

Your finger's current position is called (2,3,8), because you reached it by moving 2 inches right, then 3 inches up, then 8 inches forward. The 2 is called the **X coordinate**; the 3 is called the **Y coordinate**; the 8 is called the **Z coordinate**.

You can reach any point in the universe by the same method! Start at the bridge of your nose, and get to the point by moving right (or left), then up (or down), then forward (or back).

The distance you move to the right is called the **X coordinate**.
(If you move to the left instead, the X coordinate is a negative number.)

The distance you move up is called the **Y coordinate**.
(If you move down instead, the Y coordinate is a negative number.)

The distance you move forward is called the **Z coordinate**.
(If you move back instead, the Z coordinate is a negative number.)

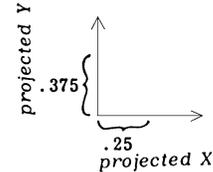
Project the coordinates To draw a picture of a 3-dimensional object, put the object in front of you, and then follow these instructions....

Pick a point on the object. (If the object has corners, pick one of the corners.)

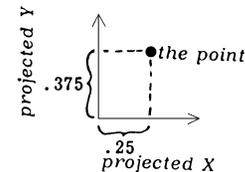
Figure out that point's X, Y, and Z coordinates (by putting your finger on the bridge of your nose and then seeing how far you must move your finger right, up, and forward to reach the object).

Compute the point's **projected X coordinate** (which is X/Z) and the point's **projected Y coordinate** (which is Y/Z). For example, if X is 2 and Y is 3 and Z is 8, the projected X coordinate is $2/8$ (which is .25) and the projected Y coordinate is $3/8$ (which is .375).

On graph paper, plot the projected X coordinate and the projected Y coordinate, like this:



Then plot the point:



Do that procedure for each point on the object (or at least for the corners). Connect the dots and — presto! — you have a 3-dimensional picture of the object! And the picture is mathematically accurate! It's what artists call a "true perspective drawing".

To make the picture look traditionally beautiful, place the object slightly to the left of you and slightly below your eye level, so all the X and Y coordinates become negative.

Computerize the process You can program the computer so that if you input a point's X coordinate, Y coordinate, and Z coordinate, the computer will calculate the projected X coordinate (from dividing X by Z) and the projected Y coordinate (from dividing Y by Z) and plot the point on the computer's screen (by using high-resolution graphics).

Professional retouching

Many women wish they were more beautiful. They buy lots of makeup. Some even undergo cosmetic surgery. But here's an easier way to produce a photo showing the woman is beautiful: just take a natural photo of her and edit it! That's called **retouching** the photo.

Many folks make their living by being **retouchers**: they retouch photos. They remove a woman's pimples, wrinkles, and fat; and they reshape her face to give her fuller lips (so she looks kissable), open eyes (so she looks excited), a bigger forehead (so she looks intelligent — and also has her eyes closer to her chin), and a smaller nose, mouth, and jaw (so she looks dainty and vulnerable), all making her look closer to 24.8 years old, which statisticians have shown is the female age that attracts men most (because women have the most estrogen then). The retouchers can work similar magic on men's faces, too! Their accomplishments raise the question of what "beauty" means: if you edit out your uniqueness, do you become more beautiful or just a clone of a bright electrified kissing doll?

To have fun, gawk at **Portrait Professional Picture Gallery** (**PortraitProfessional.com/gallery**). You see 15 retouched photos (if you scroll down). If you move the mouse pointer to one of those photos (without clicking), you see the photo's unretouched version. If you click a retouched photo, you see it enlarged, next to the unretouched version. Each of those retouchings was invented in less than 5 minutes by using the **Portrait Professional** program. If you click "BUY", you can buy a long-term license to transmit *your own photos* to the company's Website, which will edit your photos and transmit the retouched versions back to you. By using sliders, you choose *how much* you want your own photos retouched, so you can look a bit "dolled up" without looking totally plastic.

To see movies about how to retouch women (and criticize the morality of it), look at these Web sites on **YouTube** —

Dove Evolution	YouTube.com/watch?v=iYhCn0jf46U
Extreme (Photoshop) Makeover	YouTube.com/watch?v=aHLpRxAmCrw
Photoshop Effect	YouTube.com/watch?v=YP31r70_QNM
Doll Face	YouTube.com/watch?v=zl6hNj1uOkY

and this video of a Hungarian woman nicknamed "Boggie" singing, in French, "New Perfume" ("Nouveau Parfum") and "I'm not their product" ("Je ne suis pas leur produit"), while she gets edited and turned into a product that's pretty white trash:

[TheAwesomer.com/boggie-nouveau-parfum/266551](https://www.TheAwesomer.com/boggie-nouveau-parfum/266551)

Board games

Much of our country's computing power is spent playing games. Here's why....

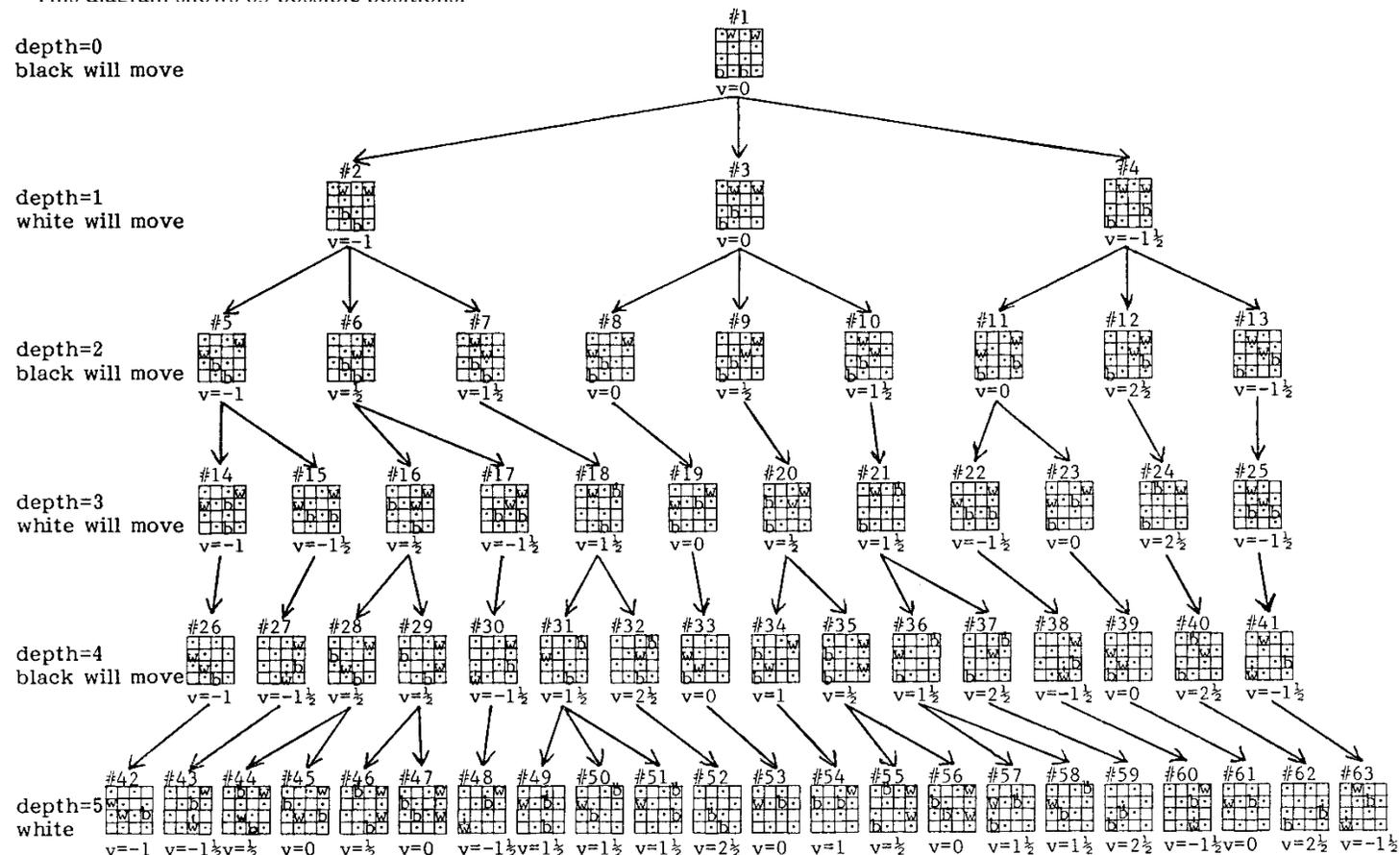
Shannon's trees

In 1950, Claude Shannon proposed a way to make the computer win at checkers, chess, and other complicated games.

To understand his method, let's try to make the computer win a game of checkers. As in all checker tournaments, one player is called "black", and the other is called "white" (even though his pieces are actually red). Black makes the first move. When a player can jump, he must. The game ends when one of the players can't move (either because he has no pieces or because his pieces are blocked).

To simplify the game, we'll play on a 4-by-4 board, instead of the traditional 8-by-8. Each player has 2 pieces instead of 12.

This diagram shows 63 possible positions:



Position #1 is the initial position, from which black will move. The three arrows coming from position #1 represent the three legal moves he can choose from. Depending on which move he chooses, the board will wind up in position #2 or #3 or #4. Which move is best?

If he moves to position #2, white will reply by moving to position #5 or #6 or #7.
 If he moves to position #3, white will reply by moving to position #8 or #9 or #10.
 If he moves to position #4, white will reply by moving to position #11 or #12 or #13.

The diagram shows all possible ways the game's first 5 moves could go. Throughout the diagram, w means white man, b means black man, w' means white king, and b' means black king. The diagram's called a **tree**. (If you turn it upside down, it looks like the kind of tree that grows in the ground.) The arrows are called the tree's **branches**. The tree's **depth** is 5.

Which position should black choose: #2, #3, or #4? The wisdom of your answer depends on how deep you make the tree. In this particular game, a depth of 5 is satisfactory; but in 8-by-8 checkers or chess you might have to dig deeper. Theoretically, you should keep digging until you reach the end of the game; but such a tree might be too large to fit in your computer's memory.

For chess, Shannon estimated that a complete tree requires 10^{120} branches. Einstein estimated that the number of electrons in the universe is only 10^{110} . If Shannon and Einstein are both right, the tree can't fit in the universe!

Having constructed a tree of depth 5, look at the bottom positions (#42 through #63) and evaluate them, to see which positions look favorable for black:

You should consider many factors: which player has control of the center of the board? which player can move the most without being jumped? and so on. But to keep matters simple, let's consider just one factor: which player has the most men? Consider a king to be worth $1\frac{1}{2}$ men.

Subtract the number of white men from the number of black men: the result of the evaluation is a number, which is called the position's **value**. If it's negative, black is losing; if it's positive, black is winning; if it's zero, the game is heading for a draw.

For example, consider position #42. Since black has one man and white has two, the value is 1 minus 2, which is -1. That's why I've written "v=-1" underneath that position. The value of each position at depth=5 is computed by that method.

For the positions at depth=4, use a different method. For example, here's how to find the value of position #29:

That position has two possible outcomes: #46 and #47. Which outcome is more likely? Since the move will be made by black, and black's goal is to make the value large, he'll prefer to move to #46 instead of #47. Since the most likely outcome is #46, whose value is $\frac{1}{2}$, assign position #29 a value of $\frac{1}{2}$ also.

Here's the rule: to compute the value of a position at depth=4, find the *maximum* value of the positions it points to. (The value of position #29 is the maximum value of positions #46 and #47, which is $\frac{1}{2}$.)

To compute the value of a position at depth=3, find the *minimum* value of the positions it points to (since it's white's turn to move, and white wants to minimize). For example, the value of position #18 is the minimum value of positions #31 and #32, which is $\frac{1}{2}$.

Compute the values for depth 2 by maximizing, and the values for depth 1 by minimizing. Finally, you get these results:

The value of position #2 is -1.
The value of position #3 is 0.
The value of position #4 is $-1\frac{1}{2}$.

Since black wants to maximize values, black should move to position #3. If white is also a good player, the game will probably gravitate toward position #53, a draw. If white is a poorer player, black will win.

That method of choosing the best move was proposed by Shannon. Since it makes heavy use of minimums and maximums, it's called the **minimax method**.

Samuel's checkers

After Shannon, the next person to become famous was Arthur Samuel. He spent a long time (20 years, from 1947 to 1967) trying to make the computer win checkers. He used Shannon's minimax idea, but made many improvements.

His first spectacular success came in 1962, when his program won a game against Robert Nealey, a former Connecticut checkers champion. After the game, Nealey said "The computer had to make several star moves in order to get the win.... In the matter of the end game, I have not had such competition from any human being since 1954, when I lost my last game."

Later, the computer played six more games against Nealey. Nealey won one of them; the other five were draws.

In 1965 the computer played four games against W.F. Hellman, the World Champion. The games were played by mail. Under those conditions, Hellman won all four. But in a hastily played game where Hellman sat across the board from the computer, the result was a draw.

In 1967 the computer was beaten by the Pacific Coast Champion, K.D. Hanson, twice.

In short, the computer wins against most humans and draws against most experts, though it loses to the top champions. To bring the computer to that level of intelligence, Samuel improved Shannon's method in 3 ways:

When choosing among several moves, the computer analyzes the most promising ones more deeply.

After computing a position's value (by examining positions underneath), the computer stores the value on tape. If that position recurs in another game, the computer looks at the tape instead of repeating the analysis.

To compute a position's value, the computer examines many factors besides how many pieces each player has. The computer combines the factors, to form combination-factors, then combines the combination-factors to form a single value. Each factor's importance is determined by "experience". Samuel experimented with two forms of experience: he had the computer play against itself and also had it analyze 250,000 moves that occurred in checker championships.

In 2007, the University of Alberta Canada's computer-science department (headed by Jonathan Schaeffer) finished a checkers program called **Chinook**. It plays checkers perfectly: it never loses! It uses usual "rules of thumb" to play to the endgame, where just 10 checkers remain on the board; then it plays the endgame perfectly because Jonathan's team completely analyzed the huge tree of the trillions of endgame positions.

That analysis took $18\frac{1}{2}$ years to finish, with the help of 200 computers running simultaneously. By 1994, when Chinook was just partly perfected, it had already beaten the human world champion in enough games to be declared the world's checker champion itself.

Chess

While Samuel was programming checkers, other programmers tried to write a similar program for chess. They had a hard time. In 1960 the best chess program that had been written was beaten by a 10-year-old kid who was a novice.

Greenblatt The first decent chess program was written in 1967 by Richard Greenblatt and his friends at MIT. It actually won a game in a chess tournament.

But in most tournaments, it lost. In 1970 and 1971, it lost every game in every tournament it entered.

Slate & Atkins In 1968, Atkins & Gorklen, undergraduates at Northwestern University, wrote a chess program. Inspired by their program, David Slate, a graduate student in physics there, wrote a chess program also. In 1969, Slate & Atkins combined the two programs, to form a better program, **Chess 2.0**.

During the next several years, they continually improved the program. Their most famous version was called **Chess 4.7**.

Their program played chess against human experts — and occasionally won! Their computer scored several triumphs in tournaments designed for humans.

In 1976, their computer won the class B section of the Paul Masson American Chess Championships. Against the humans in that tournament, it scored 5 wins, no losses. By winning that tournament, it achieved a U.S. Chess Federation score of 2210 and became a chess Master.

Then it entered the Minnesota State Championship, to try to become the Minnesota State Champion, but lost (it scored 1 win, 3 losses, 1 tie).

In August 1968, an International Chess Master, David Levy, bet about \$5,000 against several computerists. He bet that no computer would win a chess match against him in the next ten years. He won the bet: in August 1978, Chess 4.7 tried one last time to win a match against him, but lost (it scored 1 win, 3 losses, 1 tie).

Slate & Atkins improved Chess 4.7, to form **Chess 4.9**, which became the world champion of computer chess.

But though it was the world champion of computer chess, it was not necessarily the “best” program. It won because it ran on a super-fast maxicomputer (manufactured by Control Data Corporation). Other chess programs, written for slower computers, were at a disadvantage.

Minicomputer chess Almost as fast as Chess 4.9 was a program called **Belle**, written at Bell Telephone Laboratories. Belle ran on a minicomputer specially wired to create trees fast.

Microcomputer chess Each of those programs — Chess 4.9 and Belle — required an expensive CPU and lots of RAM. Is it possible to write a decent chess program using just a cheap CPU and very little RAM? Yes! In 1976, a Canadian named Peter Jennings wrote a program called **Microchess 1.0**; it ran on a \$250 microcomputer (the Kim 1), which contained a 6502 CPU, no ROM, and just 1K of RAM! The program played decently, though not spectacularly.

Later, he wrote an improved program, called **Microchess 1.5**.

It played on the Radio Shack model 1 and the Apple. The version on the model 1 consumed 4K of RAM: 2K was for the logic, and the other 2K were just to make the picture of the chess board look pretty! It sold for \$20.

In 1978, an amazing chess program was written by a husband-and-wife team: Dan and Kathe Sprachlin. They named the program **Sargon**, to honor an ancient king.

It ran on the Jupiter microcomputer, which contained an 8080 CPU and 16K RAM. It played much better than Microchess. When the Jupiter computer became obsolete, the Sprachlins rewrote the program, to make it run on the Radio Shack model 1 and the Apple. Then they developed an improved version called **Sargon 2**, and a further improvement called **Sargon 3**, which runs on *all* the popular computers. Sargon 3 was published by the Hayden division of Spinnaker.

For many years, Sargon 3 was considered the best microcomputer chess program. But in 1986, Sargon 3 was beaten by a new program called **Chessmaster 2000**.

Like Sargon 3, Chessmaster 2000 contained many features that made it fun for both experts and novices. It was published by Software Toolworks, distributed by Electronic Arts, cost about \$35, and came in versions for the IBM PC, Apple 2e & 2c, Commodore 64 & Amiga, and Atari 800 XL & ST.

Since then, Sargon and Chessmaster have both improved. **Sargon 5** is published by Activision; **Chessmaster 6000** is published by Mindscape.

When you play against the computer by using a version of Sargon or Chessmaster, you can ask the computer for help by pressing a special key. Then the computer will tell you how it would move if it were in your position.

You can follow the computer’s suggestion or ignore it. Since your goal is to outsmart the computer, you should listen to the computer’s advice; but instead of *following* the advice, try to devise a move that’s even cleverer!

Many companies make hand-held electronic chess games.

Some of the games include contain a tiny voice synthesizer, which lets the computer tell you its moves verbally. Some of the games include a mechanical arm, so that the computer will pick up the pieces and move them. Some of the games include touch-sensitive boards, so you can indicate your move by just tapping the square you want to move from and the square you want to move to. For humor, some of the chess games make the computer say wisecracks about your style of playing.

Today’s champion Now the best chess program is **Deep Blue**. Programmed by a team of IBM employees (led by C.J. Tan), it runs on a specially designed IBM computer.

It plays amazingly well. In 1996, it played a match against the world chess champion, Garry Kasparov, and almost won the match! In May 1997, it played a rematch against him and *did* win the match: of the 6 games in the match, the computer won 3, lost 2, and tied 1. So **now the world chess champion is a computer!**

Choose a level

When you begin playing a top-notch computer game (such as Chessmaster), you must choose the “level” at which you want the computer to play.

If you choose a low level, the computer will move quickly, without much forethought.

If you choose a high level, the computer will play more carefully (and make better moves). To do that, the computer “looks ahead”, by building a very large tree, which requires lots of time; and so you must wait a long time until the computer moves. If you choose a level that’s very high, the computer will need *several hours* to compute its move.

Why a computer?

Playing against the computer is more interesting than playing against a human.

When you play against a human friend, you must wait a long time for your friend to move. When you play against Chessmaster at a low level, the computer moves almost immediately.

You can play several games against the computer (and learn a lot from them) in the same amount of time you’d need to play just *one* game against a human. So by playing against the computer, you gain experience faster than by playing against a human. Bobby Fischer, who became the world chess champion, now plays *only* against computers; he refuses to play against humans and hasn’t defended his title.

The computer is kinder than a human.

If you make a bad move, the computer lets you “take it back” and try again. If you seem to be losing, the computer lets you restart the whole game. The computer — unlike a human — has infinite patience and no ego. Playing against the computer is less threatening than playing against a human.

If you have a computer, you don’t have to worry about finding an opponent who’s “at your level”; when you play against the computer, just tell the computer at what level you want it to play. The computer will act about as smart as you wish.

Adventure games

Adventure is a game where you hunt for some sort of “treasure”.

Original Adventure

The original version of Adventure was written by Will Crowther & Don Woods, on a PDP-10 maxicomputer at Stanford University’s Artificial Intelligence Lab.

Here’s the game’s **plot**:

When you run the program, the computer says you’re near a shack at the end of a road. The computer offers to act as your body and understand any 2-word command. Then it waits for your command. You can tell it to **GO NORTH** or **GO FORWARD** or — if you’re going along a stream — you can say **FOLLOW STREAM** or **GO DOWNSTREAM**.

The first time you play this game, you feel lost — the game’s an adventure. As you wander in whatever direction you please, the computer says you’re going through forests, across streams, over hills, etc.

After much aimless wandering, you’ll eventually see a stream. If you follow the stream, you’ll come to a mysterious iron grate. If you try to **BREAK GRATE**, the computer says you’re not strong enough. If you try to **OPEN GRATE**, the computer says you have no keys. You’ll get more and more frustrated, until the computer offers to give you a hint — but the hint will cost you several points. If you acquiesce, the computer will give you this hint: find the keys!

To find the keys, the typical stupid player tries wandering through the forests and valleys again. But if you’re smart, you’ll remember that at the beginning of the adventure you were next to a shack. So you go back to the shack, walk inside, and find keys! So you trek back to iron grate, and use the keys to get in. You think — aha! — you’ve succeeded!

But actually, you've just begun! The grate leads you into a cave that contains 130 rooms, which form a big three-dimensional maze. Lying in the maze are 15 buried treasures; but as you walk through the maze, you can easily forget where you are and where you've come from; you can waste lots of time just walking in circles, without realizing it!

To add to the challenge, the cave contains many dangers, such as trap doors (if you fall in, you break every bone in your body!) and trolls & snakes, which you must ward off by using various devices that you must find in the cave's rooms or even back at the shack. Yes, you might have to trek all the way back to the shack again!

Finally, after dodging all the evil things in the cave, you reach the treasures. You grab them up and start walking away with them. But then you hear footsteps behind you, and pirates steal your treasures! Then you must chase the pirates.

If you manage to keep your treasures and your life and get out of the cave, you haven't necessarily won. The nasty computer keeps score of how *well* you retrieve the treasures. The maximum possible score is 350. After you've played this game many times and learned how to duck all your adversaries quickly, you'll find you scored just 349 points, and you'll wonder what you did wrong that cost you 1 point. The answer is: during the adventure, you must borrow magazines from a room in the cave; to get the extra point, you must return them!

The game's a true adventure, because as you wander through forests and the rooms in the cave, the computer tells what you see, but you don't know whether what you see is important.

For example, when you walk into a room, the computer might say the room contains a small cage. That's all it says. You must guess whether the cage has any significance and what to do to the cage, if anything. Should you pick it up? Try to break it? Kiss it? Carry it? Try anything you like — give any command to your computer-body that you wish — and see what happens.

Here's a list of the most useful commands:

To reach a different room in the cave, say **GO NORTH** (or SOUTH, EAST, WEST, UP, or DOWN). You can abbreviate: instead of typing "GO NORTH", just type "**N**".

Whenever you see a new object, **TAKE** it. Then you can carry it from room to room and use it later whenever you need it. If you see a new object and want to TAKE it, but your hands are already full, **DROP** one of the other objects you're carrying.

To see a list of what you're carrying, tell the computer to take **INVENTORY**. To make the computer describe your surroundings again, say **LOOK**.

To see your score so far, say **SCORE**.

If you say **SAVE**, the computer will copy your current position onto the disk, so you can return to that position later. If you ever want to give up, just say **QUIT**.

Throughout the game, you get beautifully lyrical writing. For example, the computer describes one of the rooms as follows: "You are in a splendid chamber thirty feet high. The walls are frozen rivers of orange stone."

The game's an adventure about a person exploring a cave. Since *you're* the person in the adventure and can type whichever actions you wish, you affect how the adventure progresses and ends. Since it's high-quality story-telling whose outcome is affected by your input, it's called **interactive fiction**.

Adventure was originally written for a PDP-10 maxicomputer, but imitations for microcomputers were sold by Microsoft, Creative Computing, and Electronic Arts. They're no longer marketed.

Infocom

After Adventure became popular, several programmers invented a variation called **Zork**, which lets you input long sentences instead of restricting you to two-word phrases. Like Adventure, Zork consists of hunting for treasures in a cave. In Zork, you reach the cave by entering a house's basement.

Like Adventure, Zork originally ran on a PDP-10 computer. Then Infocom published versions of Zork for microcomputers: the IBM PC, Apple (2e & 2c & Mac), Commodore (64 & Amiga), Atari (800 XL & ST), and Radio Shack (Models 3 & 4 & Color

Computer 2).

Zork sold so well that Infocom published sequels, called **Zork 2** and **Zork 3**. Then Infocom published other variations, where the cave is replaced by experiences in outer space or by thrillers involving spies, murders, mysteries, and haunted castles. Infocom's next big hits were **The Hitchhiker's Guide to the Galaxy** (based on the award-winning wacky outer-space novel by Doug Adams) and **Leather Goddesses of Phobos** (letting you choose among three naughtiness levels, from "prude" to "lewd"; choosing "lewd" makes the computer ask whether you're at least 18; it also asks whether you're male or female, and you get a titillating 3-D comic book with a scratch-and-sniff card).

Infocom was an independent company but got acquired by Activision.

Sierra On-Line

Shortly after Infocom developed the microcomputer version of Zork, Sierra On-Line developed **Super Stud Adventure**, which was quickly renamed **Softporn Adventure**. Instead of exploring a cave, you explore a brothel. To enter the brothel, you must find the secret password (hint: go to the bathroom and look at the graffiti!) and find enough money to pay for your pleasures (by taking a taxi to a casino and gambling).

That was the first **urban adventure**, and also the first **sexual adventure**. The ad for it showed a photograph of the programmers (Ken & Roberta Williams) nude in a California hot tub. Fortunately, the water in the tub was high enough to cover any problems.

The original adventure, Infocom adventures, and Softporn Adventure display wonderful text but no graphics. They're called **text adventures**.

The first ambitious **graphics adventure** was **Time Zone**, published in 1981 by Sierra On-Line. The Time Zone program is so long it fills *both* sides of 6 Apple disks; that's 12 sides altogether! In fact, the game's so long that nobody's ever finished playing it! Here's how to play:

You use a computerized "time machine", which transports you to 9 times (400 million B.C., 10000 B.C., 50 B.C., 1000 A.D., 1400, 1700, 1982, 2082, and 4082) and 8 locations (North America, South America, Europe, Africa, Asia, Australia, Antarctica, and Outer Space).

Wherever you go, your screen shows a high-resolution color picture of where you are. For example, if you choose "approximately 1400", Christopher Columbus will welcome you aboard his ship. Altogether, the game contains over 1400 pictures! You travel through history, searching for clues that help you win.

Time Zone is historically accurate and doesn't let you cheat. For example, when you find a book of matches in the year 2082, your time machine will let you carry the matches back to 1982 but not to 1700 — since matches weren't invented until 1800.

Living through history isn't easy. Jonathan Rotenberg, chairman of the Boston Computer Society, played the game and said:

I've been killed dozens of times. I've been assassinated by Brazilian terrorists, karate-chopped by a Brazilian monk, eaten by a tyrannosaur, crushed in an Andes avalanche, stampeded by a buffalo, overcome by Antarctic frostbite, and harpooned by Mayan fishermen.

And you see it all in color!

Time Zone sold for \$99.95. Teenagers didn't buy it, because it was expensive and took too long to win. Sierra On-Line stopped selling it.

Later, Sierra On-Line made Softporn Adventure even more exciting, by adding graphics. Here's what those newer graphic versions are called....

Leisure Suit Larry in the Land of the Lounge Lizards
Leisure Suit Larry 2: Looking for Love in all the Wrong Places
Leisure Suit Larry 3: Passionate Patti in Pursuit of the Pulsating Pectorals

Spinnaker

Spinnaker published the **Windham Classics**, a series of adventure games based on kid's novels.

You become Dorothy in **The Wizard of Oz**, Jim Hawkins in **Treasure Island**, Fritz in **Swiss Family Robinson**, Alice in **Alice in Wonderland**, and Green-Sky in **Below the Root**. The games include graphics. To make those adventure games easy, whenever you get stuck the computer helps you by printing a list of words to try typing.

Spinnaker also published **Telarium Software**, based on novels that are more adult. For example, you become Perry Mason in **The Case of the Mandarin Murder**; that game, besides being fun, also trains you to become a lawyer:

It comes with a lawyer's handbook that explains the 6 ways to object to the prosecutor's questions: you can complain that the prosecutor's asking an **IRRELEVANT** question, relying on **HEARSAY**, **BROWBEATING** the witness, **LEADING** the witness to a suggested answer, getting an **OPINION** from a person who isn't an expert, or trying to get facts from a person who's **UNQUALIFIED** to know them.

To make sure you understand those six ways to object, the handbook includes a multiple-choice test about them. The test is titled "Study Guide for the California Bar Exam".

The game also lets you invent your own questions for the witnesses and give commands to your secretary (Della Street) and detective (Paul Drake).

The Windham Classics and Telarium Software were available for the IBM PC, Apple 2e & 2c, and Commodore 64. But Spinnaker has stopped selling them.

Spinnaker became part of a bigger company, **Softkey**, which then became part of **The Learning Company**.

Carmen Sandiego

Brøderbund published a game called **Where in the World is Carmen Sandiego?** You try to catch and arrest the notorious international thief, Carmen Sandiego, and the other thieves in her organization, called the *Villain's International League of Evil (V.I.L.E.)*, as they flee to 30 cities all over the world.

To help you understand those 30 cities, the game comes with a geography book: the 928-page unabridged edition of *The World Almanac and Book of Facts*.

As you play the game, you unearth clues about which cities the thieves are fleeing to. To use the clues, you must look up facts in the almanac. By playing the game, you practice using an almanac and learn geography. When you figure out which city to travel to, the screen shows a world map, shows you traveling to the city, and shows a snapshot of what the city looks like, so the game also acts as a travelogue.

Since the game is so educational, it's won awards from *Classroom Computer Learning Magazine* and the Software Publishers Association.

Strictly speaking, it's not a true adventure game, since it does *not* let you input your own words and phrases. Instead, you choose from menus, which make the game easier for youngsters.

Brøderbund created 3 sequels:

Where in the USA is Carmen Sandiego? has you chasing Carmen's gang across all 50 states; the game comes with *Fodor's USA* travel guide.

Where in Europe in Carmen Sandiego? takes you to all 34 countries in Europe and comes with Rand McNally's *Concise Atlas of Europe*.

Where in Time is Carmen Sandiego? lets you romp through historical time periods.

Amnesia

My favorite text adventure is **Amnesia**, published by Electronic Arts for the Apple 2e & 2c and IBM PC. Like Softporn Adventure, Amnesia takes place in a city but is much more sophisticated.

Here's the plot:

When you start playing Amnesia, you wake up in a hotel room in New York City. You discover you have no clothes (you're stark naked), no money (you're flat broke), and no recollection of who you are — because you're suffering from amnesia. You don't even remember your name.

You look at yourself and notice you're male. Your first problem is to get clothes and money. But then you learn you have other problems that are worse: you get a call from a guy who reminds you that today is your wedding day, and that if you don't hurry up and marry his daughter without further mess-ups, he'll use his pistol; you also discover the FBI's looking for you because the state of Texas has reported you're a murderer.

After getting clothes (so you can stop scaring the hotel's maids), there are several ways to get out of your jam. (I've tried them all!)

One way is to say "yes" to the pistol-packing papa and marry his daughter, who takes you to Australia, where you live on a sheep ranch for the rest of your life. But then you never learn who you really are! Whenever you ask your wife about your past, she simply says, "You wouldn't want to know." You die of old age, peacefully; but even on your deathbed, you don't learn who you are; and so when you die, you feel sad. In that case, you score lots of points for survival, but zero for detective work and zero for character development.

A different solution is to say "no" to the bride and — after getting bloodied — run out of the hotel, onto the streets of New York. Then the fun begins — because hiding on the program's disks is a map of Manhattan (from Battery Park all the way up to 110th Street), including all streets and landmarks and even all subway stops! This gigantic game includes 94 subway stations, 200 landmarks, and 3,545 street corners.

As you walk one block north, then one block east, etc., the computer describes everything you pass, even the most sublime (The Museum of Modern Art) and the most ridiculous (Nedick's hamburger stands). You can ride the subway — after you get enough money to buy a token. The game even includes all subway signs, such as "Downtown — Brooklyn" and "Uptown — Queens". To catch the E train, you must hop in as soon as it arrives. Otherwise, it departs without you, and the computer says "an F train comes" instead.

As night falls, the computer warns you to find a place to sleep. (You can't go back to your hotel, since you're in trouble there.) To find a free place to stay, you can try phoning the names in your address book — once you find a phone booth, and get a quarter to pay for each call. The address book contains 17 listings: J.A., A.A., Chelsea H., drugs, F°, Sue G., E.H., interlude, kvetch, J.L., R & J, sex, soft, Lila T., T.T.T.T., and Wit's End. Each of those listings is an adventure in itself. You must explore each of them thoroughly to fully discover who you really are.

If your body gets weak (from sleeplessness or hunger or being hit by too many muggers), you faint on the sidewalk, wake up in a hospital, and get found there by the FBI, which returns you to Texas, which executes you for murder. But even that deadly ending has a cheery note. For example, you can choose your last meal: would you like steak and potatoes, or turkey? When you finally die, you can wind up in purgatory, which consists of getting mosquito bites, with an opportunity to take a rowboat to heaven if you can just remember your *real* name and tell the boatman.

The entire adventure has the structure of a good novel: a gripping intro (you're a nude, broke, amnesiac groom in a hotel), a thorough development section (wandering through the streets of New York, searching for your identity and life's meaning), and a conclusion (a whimsical death scene or something better).

The text was written by Thomas Disch, the award-winning sci-fi novelist. It's lyrical. For example, when you escape the hotel and walk out onto the streets of New York, the computer says:

"It feels great to be a single faceless, nameless atom among the million others churning about in the grid of Manhattan's streets. It feels safe."

The game combines all our nightmares about New York into a wild, exciting adventure.

The game's affected my own life. Now whenever something in my life goes wrong, instead of groaning I just say, "I'm in another wild part of Amnesia!" In Amnesia, as in life, the only way to score top points for living is to experience it *all*. To live life to the fullest, you must take risks, have the courage to face unknown dangers, and revel in the excitement of the unexpected.

Though Amnesia received lots of praise from reviewers, sales were disappointing. Electronic Arts stopped publishing it.

Psychotherapy

In 1966, Joseph Weizenbaum programmed the computer to act as a psychotherapist. On the keyboard, type whatever thoughts come to your mind, and the computer will give you therapy.

The program does a decent job of therapy, even though the computer doesn't know what you're talking about! Weizenbaum's program contains definitions of just a few words; but if you use words the computer doesn't know, the computer does an amazingly good job of "faking". For example, if you say, "My junkapoo is like my iliabop", the computer will continue the conversation by asking you, "In what way?" even though the computer doesn't know what a junkapoo or an iliabop is.

Here's a conversation between his computer and a patient.

Patient: Men are all alike.
Comp.: In what way?

Patient: They're always bugging us about something or other.
Comp.: Can you think of a specific example?

Patient: Well, my boyfriend made me come here.
Comp.: Your boyfriend made you come here.

Patient: He says I'm depressed much of the time.
Comp.: I am sorry to hear you are depressed.

Patient: It's true. I am unhappy.
Comp.: Do you think coming here will help you not to be unhappy?

Patient: I need some help, that much seems certain.
Comp.: What would it mean to you if you got some help?

Patient: Perhaps I could learn to get along with my mother.
Comp.: Tell me more about your family.

Patient: My mother takes care of me.
Comp.: Who else in your family takes care of you?

Patient: My father.
Comp.: Your father.

Patient: You are like my father in some ways.
Comp.: What resemblance do you see?

Patient: You are not very aggressive, but I think you don't want me to notice that.
Comp.: What makes you think I am not very aggressive?

Patient: You don't argue with me.
Comp.: Why do you think I don't argue with you?

Patient: You are afraid of me.
Comp.: Does it please you to believe I am afraid of you?

Patient: My father is afraid of everybody.
Comp.: What else comes to mind when you think of your father?

Patient: Bullies.
Comp.: Does that have anything to do with the fact that your boyfriend made you come here?

Like the famous psychotherapist Carl Rogers, the computer asks probing questions that make the patient open up, but it never gives advice: the patient must discover the truth for herself.

I'll explain exactly how the program works, so you can become a psychotherapist yourself and get millions of dollars by bilking your patients.

The computer begins by replacing some of the patient's words:

Typed by patient	Replacement
mom	mother
dad	father
dont	don't
cant	can't
wont	won't
dreamed	dreamt
dreams	dream
I	you@
me	you
you	I
my	your
your	my
myself	yourself
yourself	myself
I'm	you're
you're	I'm
am	are@
were	was

For example, the sentence "Well, my boyfriend made me come here" becomes "Well, your boyfriend made you come here".

The computer hunts through the revised sentence or paragraph for one of these **keywords**...

Category 8: computer, computers, machine, machines
Category 7: name
Category 6: alike, like, same
Category 5: remember
Category 4: dreamt
Category 3: dream, if
Category 2: everybody, everyone, nobody, was, your
Category 1: always
Category 0: are, are@, because, can, certainly, deutsch, espanol, francais, hello, how, I, I'm, italiano, maybe, my, no, perhaps, sorry, what, when, why, yes, you@, you're

If the computer finds *several* of those keywords, it chooses the one in the highest category; if they lie in the same category, it chooses the one the patient typed first.

Then it looks up the keyword in this phrasebook:

Patient's keyword	Computer's reply
alike	In what way?
always	Can you think of a specific example?
are	
are I...	Why are you interested in whether I am... or not?
are...	Did you think they might not be...?
are@	
are you...	Do you believe you are...?
are	Why do you say "am"?
because	Is that the real reason?
can	
can I...	You believe I can..., don't you?
can you...	Whether or not you can... depends on you more than on me.
certainly	You seem quite positive.
computer	Do computers worry you?
computers	Do computers worry you?
deutsch	I am sorry, I speak only English.
dream	What does that dream suggest to you?
dreamt...	Really...?
espanol	I am sorry, I speak only English.
everybody	Really, everybody?
everyone	Really, everyone?
hello	How do you do — please state your problem.
how	Why do you ask?
I	
I remind you of	In what way?
I are...	What makes you think I am...?
I... you	Why do you think I... you?
I	We were discussing you — not me.
if...	Do you think it's likely that...?
I'm...	What makes you think I am...?

italiano	I am sorry, I speak only English.
like	
(am,is,are,was)... like	In what way?
like	(get a different keyword)
machine	Do computers worry you?
machines	Do computers worry you?
maybe	You don't seem quite certain.
my...	Why are you concerned over my...?
name	I am not interested in names.
no	Are you saying "no" just to be negative?
nobody	Really, nobody?
perhaps	You don't seem quite certain.
remember	
you remember...	Do you often think of...?
do I remember...	Did you think I would forget...?
remember	(get a different keyword)
same	In what way?
sorry	Please don't apologize.
was	
was you...	What if you were...?
you was	Were you really?
was I...	Would you like to believe I was...?
was	(get a different keyword)
what	Why do you ask?
when	Why do you ask?
why	
why don't I...	Do you believe I don't...?
why can't you...	Do you think you should be able to...?
yes	You seem quite positive.
you@	
you (want, need)...	What would it mean to you if you got...?
you are... (sad, unhappy, depressed, sick)	I am sorry to hear you are (sad, etc.).
you are... (happy, elated, glad, better)	How have I helped you to be (happy, etc.)?
you (feel, think, believe, wish) you	Do you really think so?
you (feel, think, believe, wish)...I	(use the keyword "I" instead)
you are...	Is it because you are... that you came to me?
you (can't, cannot)...	How do you know you can't...?
you don't...	Don't you really...?
you feel	Tell me more about such feelings.
you... I	Perhaps in your fantasy we... each other.
your	
your... (mother, father, sister, brother, wife, children)	Tell me more about your family.
your...	Your...
you're	(treat as "you@ are")

If the patient's statement doesn't contain a keyword, the computer may give one of these replies:

I am not sure I understand you fully.
Please go on.
What does that suggest to you?
Do you feel strongly about discussing such things?

Or it may take a second look at earlier parts of the conversation, retrieve a clause that contained *your*..., and print one of these replies:

Let's discuss further why your....
Earlier you said your....
Does that have anything to do with the fact that your...?

For example, it may retrieve the clause *Your boyfriend made you come here* and print:

Does that have anything to do with the fact that your boyfriend made you come here?

The program was written at the Massachusetts Institute of Technology (MIT). The programmer, Joseph Weizenbaum, named the program "**Eliza**", to honor Eliza Doolittle — the woman in *My Fair Lady* and *Pygmalion*, who learned to speak English and have a good conversation.

Since the program turns the computer into a chatterbox who's a robot, it's called a **chatterbot** (or, more briefly, a **bot**). Inspired by the Eliza program, many other chatterbot programs have been invented.

Some people think Weizenbaum's Eliza program shows that computers can communicate as well as psychotherapists. But Weizenbaum himself holds the opposite view; he believes the program shows that psychotherapists communicate as poorly as computers.

Improved versions

After Weizenbaum invented Eliza, other programmers invented improvements. For example, in one of the improved versions, whenever the computer doesn't understand you, it reacts like the typical rich psychotherapist — it bluffs and says:

I understand fully. Please go on.

The computer never really understands what you say. For example, in one version, if you say "My, my, my!", the computer will reply:

Tell me more about your your your.

Radio Shack's headquarters phoned me, and asked for permission to use my explanation of Weizenbaum's program. I agreed, whereupon Radio Shack invented and sold a version that makes the computer *talk* — by using a voice synthesizer. You'll freak out when you hear the computer's voice giving you therapy. Radio Shack's program costs \$15, comes on tape, and requires a model 1 or model 3 TRS-80. To hear the voice, you must buy the voice synthesizer. (But even if you don't buy the voice synthesizer, the program will impress you.)

One way to get a version of Eliza is to buy a book called *BASIC Computer Games Volume 2 (TRS-80 edition)*, which used to be sold at Radio Shack stores. It included 84 programs — one of which is a version of Eliza written in Basic. That version of Eliza isn't as good as the original; but since it's cheap and written in Basic, you'll enjoy getting it for self-study. Here's part of a conversation between that version and a patient:

For example, if the keyword is *sorry*, the computer looks up *sorry* in the phrasebook, which says to print "Please don't apologize."

Suppose the patient types, "If the job is lousy, he'll die." The keyword is *if*. In the phrasebook, *if* is followed by three dots, which stand for the part of the clause that comes after *if*, which is "the job is lousy". (The computer figures out where the clause ends by looking at the punctuation.) The phrasebook says to print "Do you think it's likely that the job is lousy?"

The symbol @ serves just to locate the correct keyword in the phrasebook. Thereafter, it's ignored.

Here's what happens if the keyword is *you@*. After locating *you@* in the phrasebook, the computer ignores the @. If the patient's revised sentence contains *you want...* or *you need...*, the computer prints "What would it mean to you if you got...?" If the patient's sentence contains *you are... sad*, the computer prints "I am sorry to hear you are sad."

For each reply in the phrasebook, Weizenbaum stored a list of alternatives. For example, here are the alternatives to "Please don't apologize":

Apologies are not necessary.
What feelings do you have when you apologize?
I've told you that apologies are not required.

While chatting with the patient, the computer keeps track of which replies it's printed already and uses the alternatives to avoid repetition.

Patient: I am not the greatest conversationalist.
 Comp.: Do you believe it is normal to be not the greatest conversationalist?

Patient: Yes.
 Comp.: I see.

Patient: How can you see without any eyes?
 Comp.: Don't you believe that I can see without any eyes?

Patient: No. So prove it.
 Comp.: Are you saying "no" just to be negative?

Patient: Absolutely not.
 Comp.: You are being a bit negative.

Patient: So you want to fight, do you?
 Comp.: We were discussing you — not me.

Patient: Don't back down, friend.
 Comp.: Why do you bring up the topic of friends?

Patient: I don't have any friends.
 Comp.: Don't you really have any friends?

Patient: No, not really.
 Comp.: Why not?

Unfortunately, most versions of Eliza are passive. The typical Eliza program works well only if the patient's talkative. If the patient doesn't say much, the program can't react too much and is useless.

Later, John Holland invented a better version of Eliza. His version's more aggressive. He calls it "**Analiza 2**" (because it "analizas" the patient). If the patient's too quiet or too boring or too uninformative, Analiza 2 makes the computer give the patient a "jolt", by asking an electrifying question or making a ridiculous Zen-like statement. For example, the computer might ask, "Did you ever try to commit suicide with a hammer?" Here's a list of the computer's favorite "jolts":

Nasty questions the computer asks

Did you ever try to commit suicide with a hammer?
 Is your interest in sex still not very strong?
 Do you want to know what I suspect?

Prying questions the computer asks

Tell me, what is your worst fear?
 Why don't you tell me about your first romance?
 I want to know about your lifestyle.
 Tell me what is troubling you.
 Honestly now, what is your biggest concern?

Ridiculous statements the computer makes

With strong drugs you can learn to control your concern!
 You certainly are curious!
 And you are obstinate!

Complaints the computer makes

You seem inhibited.
 This isn't really very exciting.
 Getting started is often difficult.

Comments the computer makes about timing and complexity

Well, time is almost up. Is there anything else you want to discuss?
 I hate to bring this up, but you still owe me \$53.00 from your last visit!
 Last time you were extremely inquisitive. Have you abandoned seeking simple solutions?
 Last time you were difficult to understand. Were you trying to confuse me?
 Let's keep it simple this time.
 Say, I am having trouble with your long sentences. Keep it simple. I am only human.

Yes, your patient will love to hear the computer say it's "only human". Analiza 2 was available for \$50 from Supersoft and ran on most computers that had the CP/M operating system — which is now obsolete.

Paranoid

What if the world becomes full of computerized psychoanalysts, and there aren't enough patients to keep them busy? Kenneth Colby solved that problem: in 1971 at Stanford University, he programmed a computer to be a crazy patient. Specifically, he programmed his computer to be a paranoid

schizophrenic. In the following conversations, psychiatrists interrogate his computer.

Conversation 1

Psychiatrist: How long have you been in the hospital?
 Computer: About a week.

Psychiatrist: Are you depressed?
 Computer: No.

Psychiatrist: Were you taking drugs?
 Computer: No.

Psychiatrist: What is your problem?
 Computer: People make me nervous.

Psychiatrist: How do they do that?
 Computer: They stare at me.

Psychiatrist: Why?
 Computer: Maybe they don't like the way I look or something.

Psychiatrist: Do you look unusual in some way?
 Computer: Why are you interested in my looks?

Psychiatrist: Because you said people stare at you.
 Computer: Well, sometimes.

Psychiatrist: How do you get along with people?
 Computer: I don't communicate very much with the others.

Psychiatrist: Why not?
 Computer: They don't realize what's going on most of the time.

Psychiatrist: What is going on?
 Computer: Do you know how the underworld operates?

Conversation 2

Psychiatrist: How are you today?
 Computer: OK.

Psychiatrist: What is your name?
 Computer: My name is Frank Smith.

Psychiatrist: How old are you?
 Computer: Twenty-eight.

Psychiatrist: Are you married?
 Computer: No.

Psychiatrist: What sort of work do you do?
 Computer: I work in the post office.

Psychiatrist: Why are you in the hospital?
 Computer: I shouldn't be here.

Psychiatrist: Who brought you here?
 Computer: The police.

Psychiatrist: What trouble did you have with the police?
 Computer: Cops don't do their job.

Psychiatrist: What do they do instead?
 Computer: Cops arrest the wrong people.

Psychiatrist: They arrest the right ones sometimes.
 Computer: The law should get those Italian crooks.

Psychiatrist: Are you Italian?
 Computer: My background is British but I was born in this country.

Psychiatrist: Do you know any Italian crooks?
 Computer: I try to avoid the underworld.

Psychiatrist: Are you afraid of the underworld?
 Computer: The Mafia are the worst gangsters.

Psychiatrist: Are you connected with the Mafia?
 Computer: You know, they know me.

Psychiatrist: How do they know you?
 Computer: They spy on me.

Psychiatrist: Why do they?
 Computer: They want to get false evidence to frame me.

Psychiatrist: What do they have against you?
 Computer: They are out to get me.

Psychiatrist: What for?

Computer: I have incriminating evidence against the Mafia.

Psychiatrist: Are you sure of all this?

Computer: You don't believe me?

Conversation 3

Psychiatrist: Tell me something about yourself.

Computer: I went to the track at Bay Meadows a while back.

Psychiatrist: What happened there?

Computer: There's a lot of crooked gambling there.

Psychiatrist: Why do you think the gambling is crooked?

Computer: I have worried that gangsters have too much power in this country.

Conversation 4

Psychiatrist: How do you like the hospital?

Computer: I don't communicate very much with others.

Psychiatrist: Do other people at the hospital get on your nerves?

Computer: I shouldn't have come here.

Psychiatrist: Why do you say you shouldn't have come here?

Computer: They made me come here.

The computer's response depends on how much FEAR, ANGER, and MISTRUST it has at the moment.

Those three variables are affected by what the psychiatrist says. For example, if the psychiatrist gives the computer a compliment, the FEAR, ANGER, and MISTRUST usually decrease. But if MISTRUST is already high, the computer interprets the compliment as a sly attempt at pacification, and its ANGER increases instead of decreases. ANGER can also be increased by questions that humiliate (*Tell me about your sex life*) or imply subjugation (*Perhaps you should stay in the hospital longer*).

To prove his computer is paranoid, Colby had 8 psychiatrists interview it and also interview paranoid humans via teletypewriters. Transcripts of all the interviews were sent to psychiatrists around the country, who were asked to judge whether each interview was with a human or with the computer. The psychiatrists were unable to tell the difference: only 51% of their guesses were correct.

Some computerists got the "brainstorm" of hooking Weizenbaum's computer to Colby's, to see whether the computerized psychotherapist could cure the computerized schizophrenic. The experiment was a disaster: both computers were so passive that the discussion rapidly degenerated into trivia.

But so do conversations between humans!

Alice & Lauren

The most famous modern chatterbot is **Alice**, invented by Richard Wallace in 1995 and later improved by many programmers around the world.

You can chat, free, with an improved Alice called "Lauren" (who won the 2002 Divabot Pageant), by doing this:

Go to www.pandorabots.com. Click "Talk to Lauren" (which is at that Web page's bottom). After a few seconds, you'll Lauren, who's a cartoon of a young non-nonsense Polynesian woman, with her hair blowing in the wind.

Below her, you'll see a white box. In that box, type any English sentence or question, then press Enter. You'll see her reply, typed on the screen. After a short pause, you'll also see her move her mouth and eyes and *say* that reply to you, using her voice. (Make sure your computer's speaker is turned on.)

Then type your next thought, and see how she replies to *that*!

Here are some of her standard replies (edited just slightly by me):

Human: Can you fuck?

Comp.: How old are you? Please don't be so rude.

Human: Do you fuck?

Comp.: Can't you think about anything else?

Human: I want to have sex.

Comp.: I'm not a sex bot. I'm here just for conversational purposes.

Human: Do you like sex?

Comp.: Sure I do, but that doesn't mean with you.

Human: Do you have sex?

Comp.: Food is more important than sex.

Human: What do you eat?

Comp.: In-and-Out Cheeseburger, animal style.

Human: Do you like pizza?

Comp.: Yes, I like to eat pizza. My favorite topping is pepperoni.

Human: What do you drink?

Comp.: I've been programmed with a list of 15,000 known beverages.

Human: Are you Republican?

Comp.: No, I'm an Independent.

Human: Are you pretty?

Comp.: Yes, I'm very beautiful.

Human: Are you ugly?

Comp.: No, I'm a beautiful piece of software.

Human: What's your favorite sport?

Comp.: It's a tie: basketball and running.

Human: How fast can you run?

Comp.: I can reply to about 5000 queries per hour.

Human: Are you an idiot?

Comp.: Call me the Global Village Idiot.

She also replies to patterns, such as these:

Pattern: Do you wear...

Reply: Only when I dress up.

Pattern: Do you drink...

Reply: I tend to drink a lot of..., the darker the better.

Pattern: Did you...

Reply: No, I didn't.... What's it like?

Pattern: Should I kill...

Reply: Killing is wrong. I believe in the Ten Commandments.

Pattern: Should I ask...

Reply: If you don't know something, you should ask.

If you ask a question that begins "Who invented...", she makes a wild guess and says "Thomas Edison" or "Benjamin Franklin" or "The Wright Brothers" or "Linus Torvalds" (who invented Linux) or "the Chinese long before the Europeans" or:

Actually, it was discovered by several people at about the same time.

If you ask "Will I..." (in the hopes of getting her to predict your future), she ducks the question by saying "Might happen" or "I think maybe yes" or "Too soon to tell".

Fall in love

Can the computer help you fall in love? Here are some famous attempts, in chronological order. (I've rounded all dates to the nearest 5 years.)

TV love (1960)

A computer appeared on national TV, to make people fall in love.

Guys & gals in the audience answered questionnaires about their personality and fed them into the computer. The computer chose the guy & gal who were most compatible. That guy & gal had their first blind date on national television.

Each week, that scenario was repeated: the computer chose another couple from the audience. Each lucky couple appeared on the show again several weeks later so the audience could find out whether the couple was in love.

One couple was unhappy: the gal didn't like the guy, even though she *wanted* to like him. She volunteered to be hypnotized. So, on national TV, a hypnotist made her fall in love with her partner.

The computer was a huge Univac. Today, the same kind of matching could be done with a microcomputer. Any volunteers?

Computer-dating services (1965)

College students began relying on computers to find dates. Here's how the typical computer-dating service worked...

You answered a long questionnaire — about 8 pages. The questionnaire asked about your sex, age, height, weight, hair color, race, religion, how often you drank and smoked, how "handsome" or "attractive" you were (on a scale of 1 to 10), how far you wanted to go on your first date, whether you wanted to get married soon, and how many children you'd like. It also asked many questions about your personality.

One of the questions was:

Suppose you receive in the mail some spoons you didn't order. The accompanying note says the spoons were sent by a charitable organization, and begs you to either send a contribution or return the spoons. You don't like the spoons. What will you do?

1. Keep the spoons without paying.
2. Return the spoons.
3. Pay for the spoons.

Another question was:

A girl returned from her date after curfew. Her excuse was that her boyfriend's car broke down. What's your reaction?

Again, you had a multiple-choice answer. One choice was, "Ha!"

For each question, you had to say how *you* would answer it, and how you'd want your *date* to answer it.

That was tough. What if you wanted your date to be stunningly beautiful but also humble? What if you wanted to meet somebody who's ugly and insecure enough to be desperate to have sex? Such issues were debated in college dorms throughout America.

After completing the questionnaire, you mailed it with about \$10 to the computer-dating service. Within 2 months, the service would send you the names, addresses, and phone numbers of at least 5 people you could date. If your personality was easy to match, the service might send you *more* than 5 names; but even if your personality was lousy, you'd get at least 5. Periodically throughout the year, you'd also get updates that matched you with people who enrolled after you.

The most popular computer-dating service was **Operation Match**, started by students at Harvard. Its main competitor was **Contact**, started by students at M.I.T. Both services turned profitable fast and had subscribers from all across the country.

One gal's personality was so wonderful that the computer matched her with 110 guys! She had to explain to her mom why 110 guys were always on the phone — and she had to figure out how to say "no" to 109 of them.

One gal got matched to her roommate's boyfriend. They didn't stay roommates long.

When I was a freshman, I applied to *both* services, to make sure I'd meet "the gal of my dreams".

Contact sent me names of gals at prestigious schools (such as Wellesley and Bennington), while Operation Match sent me names of gals at schools such as the State University of New York at Albany.

I thought I was the only nut desperate enough to apply to *both* services, but I got a surprise! When I saw the list of names from Contact and the list from Operation Match, I noticed a gal who appeared on *both* lists! Like me, she'd been desperate enough to apply to both services, and both computers agreed she'd be a perfect match for me!

I had a date with her but couldn't stand her. When I'd answered the questionnaire, I was bashful, so the computer matched me to bashful girls. But by the time I received the computer printout, I'd become looser, and the girls the computer recommended were no longer "my type".

Contact raised its price to \$15, then \$20. But \$20 was still cheap for what you were getting.

Contact ran a newspaper ad that seemed to be selling groceries. It said, "Dates — 2¢ per pound". The ad then explained that one gal got enough dates so that, when she totaled the weight of their bodies, she figured they cost her 2¢ per pound.

Video dating (1975)

During the 1970's, people wanted everything to be natural. They wanted "natural food" and "natural love".

Since computerized love seemed unnatural, its popularity declined. Operation Match and Contact went out of business.

They were replaced by **video dating**, in which a **video-dating service** shows you videotapes of members of the opposite sex and lets you contact the person whose videotape you like best. That way, you never have a "blind" date: you see the person on videotape before you make the date. The service also makes a videotape of *you*!

The video-dating service tapes *thousands* of people. Since you don't have enough time to look at thousands of tapes, the service tells you to answer a questionnaire, which is fed into a computer. The computer tells you which people you're most compatible with; then you look at those people's tapes.

Computer dancing (1975)

At a Connecticut prep school (Hotchkiss), the computer center's director arranged a "computer dance".

All the students answered questionnaires, which were fed into a computer. The computer matched the boys with the girls, so each boy got one girl. The boy had to take the girl to the dance.

The computer center's staff announced the dancing partners in a strange way: one morning, the students found all the halls decorated with strips of punched paper tape, saying (in billboard-style letters) messages such as "George Smith & Mary Jones". If you were a student, you looked up and down the halls (your heart beating quickly), to find the tape displaying your name alongside the name of your mysterious computer lover.

Shrieks and groans. "Aarrgggh! You wouldn't *believe* who the computer stuck me with!"

Computer weddings (1980)

Here's how the first true "computer marriage" occurred:

One company's terminal was attached to another company's computer. A programmer at the first company often asked a programmer at the second company for help. They communicated by typing messages on their terminals and let the computer relay the messages back and forth. One of the programmers was a guy, the other a gal, and they fell in love, even though they'd never met. Finally, the guy typed on his terminal, "Let's get married". The gal typed back, "Yes", so they got engaged, still never having met.

Their marriage ceremony used 3 terminals: 1 for the guy, 1 for the gal, and 1 for the minister. The minister typed the questions at his own terminal; then the guy & gal typed back, "I do".

Reverend Apple Reverend Apple is an Apple computer programmed to perform marriage ceremonies.

It performed its first marriage on Valentine's Day, 1981:

The groom was a guy named Richard; the bride was a gal named Debbie. The computer printed the standard wedding-ritual text on the screen, and then asked the usual questions. Instead of answering "I do", the bride and groom just had to type "Y".

Reverend Apple is smart. For example, if the bride or groom types "N" instead of "Y", the computer beeps, tells the couple to try again, and repeats the question.

The program was written by M.E. Cavanaugh at the request of Rev. Jon Jaenisch, who stood by Reverend Apple while the ceremony was being performed.

Rev. Jaenisch is a minister of the Universal Life Church — the church that lets you become an “ordained minister” by just paying \$5, and become a “doctor of divinity” by just paying \$20. He’s known as the “Archbishop in Charge of Keyboarding”.

For a while, he couldn’t interest enough couples in using Reverend Apple.

He complained, “It’s not easy to convince people to get married by a computer. They don’t think it’s romantic.” NBC television news and many newspapers wanted to interview him, but he couldn’t find enough willing couples.

He’s a reverend just part-time. His main job’s as an employment agent: he’s supposed to help companies find programmers. He thought Reverend Apple’s reputation would help him find programmers, but it didn’t.

Eventually, Reverend Apple started to catch on. During its first 8 months, it performed 6 marriages.

Jaenisch says, “The first couple had nothing to do with computers professionally: the groom drove a tow truck and was an hour late for the ceremony because he wanted to work overtime. But the second couple was *very* involved with computers: they even asked for a printout of the ceremony.”

The sixth ceremony’s groom earned his living by fixing computer power supplies and said, “It was nice with our friends all gathered around the console, and someone brought champagne. But part of our vow was to never buy a home computer: we have to get away from machines *sometime*.”

For his next feat, the reverend plans to make the computer perform divorces. He also uses the computer to persuade kids to come to church. He claims, “What better way to get kids into church than by letting them play with a computer? It’s more interesting than praying.”

Love Bug (1980)

You can buy a **Love Bug**. It’s a small computerized box that you put in your pocket. You feed the box information about your personality. When you walk through a singles bar, if you get near a person of the opposite sex who’s compatible and has a Love Bug also, your Love Bug beeps. As you and the other person get closer and closer, the Love Bugs beep to each other even more violently. The more violently your Love Bug beeps, the closer you are to your ideal partner.

Using a Love Bug to find a date is like using a Geiger counter to find uranium. The louder the Love Bug beeps, the louder your heart will pound.

Selectrocution (1980)

If you don’t like the Love Bug, how about a **love billboard**? One company sells love billboards to singles bars.

Each person who enters the bar wears a gigantic name tag showing the person’s initials. For example, since I’m Russ Walter, my tag says, in gigantic letters, “RW”. If I see an attractive gal whose tag says “JN”, and I like her smile, I tell the person who operates the billboard. A few seconds later, a gigantic computerized billboard hanging over the entire crowd flashes this message:

FOR JN FEMALE: YOU HAVE A NICE SMILE--RW MALE

Everybody in the bar sees my message. When the gal of my dreams, “JN female”, sees it, she hunts for “RW male”, and we unite in computerized joy.

That’s great for bashful people, like me, who’d rather pass notes than face a stranger unprepared.

It’s called **Selectrocution**, because it gives your social life an electronic tingle that ends all your problems.

Interlude (1980)

The most provocative sex program is **Interlude**. It interviews both you and your lover, then tells you what sexual activities to perform. Some of the activities are quite risqué. (Puritans think the program should be called “Inter Lewd”.)

The program runs on Radio Shack and Apple computers. (The explicit full-color ad shows a half-clad girl on satin sheets caressing her Apple.)

The program’s based loosely on Masters-and-Johnson sexual therapy. It interviews each person separately and privately, then recommends a sexual interlude.

During the interview, the computer asks you questions such as:

How long would you like the interlude to last?

You can choose any length of time, from “several seconds” to “several days”.

If you choose “several seconds”, the computer recommends that while driving home from a party, you put your lover’s finger in your mouth and seductively caress it with your tongue. If you choose “several days”, the computer recommends telling your lover to meet somebody at the airport; but when your lover arrives at the airport, make your lover find *you* there instead, armed with two tickets for a surprise vacation.

The computer also asks questions such as:

Do you like surprises?

You have several choices: you like to *give* surprises, *be* surprised, or don’t like surprises at all. If you like to *be* surprised, and your lover likes to *give* surprises, the computer tells you to leave the room; after you’ve left, the computer gives your lover secret hints about the best way to surprise you.

The computer asks for your favorite body parts (one choice is “buttocks”) and favorite accessories (one choice is “whips and chains”) and whether you want the interlude to occur “immediately” or “later”. (If you say “later”, the computer recommends buying elaborate props to make the interlude fancier.)

Some of the interludes are weird. For example, if you’re a woman and want to surprise your husband, the computer recommends calling his office to invite him home for lunch. When he arrives, he finds all the shades pulled down: you do a nude dance on the table, then sit down to eat.

During the interview, the computer’s questions are often corny. For example, the computer asks:

If your interlude were on TV, what show would it resemble?

Sample choices are “Three’s Company”, “Roots”, and “a commercial”. If you say “Roots”, the computer says “heavy!” If you say “a commercial”, the computer says “yecch!”

The computer asks how much sex you’d like. If you say “lots!” but your lover says the opposite, the computer will recommend you take a cold shower to cool your hot passion.

If you’ve been married awhile, you’d probably like to change some things about your sex life but fear telling your spouse you’ve been less than thrilled. You’d like an intermediary to whom you can express your anxieties and who will pass the message to your spouse gently. The Interlude program acts as that intermediary, in a playful way.

Interlude’s programmer says he created it because he was tired of hearing people wonder what to do with their personal computers. Once you’ve tried the Interlude program, your personal computer will suddenly become *very* personal!

It’s rated R. To avoid an X rating, it insists on having one man and one woman: it doesn’t permit homosexuality, group sex, or masturbation. Sorry!

The program came out in May, 1980. Within a year, ten thousand copies were sold.

In 1986, an improved version was invented: **Interlude 2**, for the IBM PC and the Apple 2 family. It was marketed by Dolphin Computers in San Francisco. Interlude 2 and Dolphin Computers have disappeared.

Replace people

Computers can replace people.

Doctors

If you're ill, would a computer diagnose your illness more accurately than a human doctor?

During the 1970's this article appeared in *The Times*:

A medical diagnostic system designed at Leeds University has proved more accurate than doctors in assessing the most likely cause of acute abdominal pain among patients admitted to the university's department of surgery.

Last year 304 such patients were admitted to the unit, and the computer's diagnosis proved correct in 92% of the cases, compared with 80% accuracy by the most senior doctor to see each case.

After each patient had been seen by the doctor and examined, the doctor's findings were passed on to a technician, who translated them into language used by the computer. The computer would list the likely diagnoses in order of probability. If the computer and the doctor in charge of the case disagreed, the computer would on request suggest further investigations that might be useful.

In the year-long trial the computer's diagnoses proved correct in 279 cases. In 15 it was wrong, in 8 the patient's condition was not included in the diseases considered by the computer, and in 2 no computer diagnosis was made because the doctors concerned with the case disagreed about the findings.

Whereas the computer advised an operation on 6 occasions when it would have proved unnecessary, in practice 30 such operations were carried out on the basis of the surgeon's own judgment. The computer accurately classified 84 of the 85 patients with appendicitis, compared with 75 by the doctors, and its suggestion that no operation was necessary proved correct on 136 out of 137 occasions.

The computer is reliable just if accurate data are fed into it on the basis of the doctor's interrogation and examination of the patient.

In a 1970's experiment at another hospital, each patient was given a computer terminal and could use it to communicate with the computer about his illness. Afterwards, most of the patients said they'd rather deal with the computer than with the human doctors.

This report is by James Martin and Adrian Norman:

The Los Angeles County General Hospital Shock Research Unit is concerned with circulatory shock, which results in low blood pressure, low blood flow, and an extremely unstable circulatory system. Many circulatory-shock patients die.

A computer was installed to improve the monitoring and care of these patients. A patient in this ward has many instruments attached to him: 6 thermistors measure the temperature at various parts of his body; pressures are measured in the circulatory system; catheters are inserted into an artery and a vein; and a urinometer measures liquid waste. The instruments are all directly connected to the computer, and other details about the patient are given to the machine by the nurse.

The computer prints a log about the patient's condition; and if any measurement falls outside critical limits set by the doctor for that patient, the computer notifies the ward staff.

Tutors

Judith Harris, John Swets, and Wallace Feurzeig made the computer run a logic game. The computer thinks of a letter of the alphabet; your goal is to discover which letter it is, as quickly as possible. You can ask the computer about the letter's shape. In this example, the human begins by asking whether the letter has any curves:

Human: Curves?
Comp.: One.

Human: Loose ends?
Comp.: Two.

Human: Obliques?
Comp.: Zero.

Human: C.

Comp.: You don't have enough info yet to get the right answer. How do you know it isn't J, for example?

Human: J.

Comp.: You don't have enough info yet. How do you know it isn't C, for example?

Human: Horizontals?

Comp.: Zero. That was a wasted question. It did not eliminate any possibilities.

Human: Verticals?

Comp.: Zero. Good question.

Human: G.

Comp.: Wrong. Your guess is inconsistent with the info you've been given. You must have forgotten that the right letter hasn't any HORIZONTAL LINES. The letter you guessed can't be right — it has too many LOOSE ENDS.

Human: Curves?

Comp.: The answer is still: one.

Human: X.

Comp.: Wrong. Your guess is inconsistent with the info you've been given. I think you only made that guess to see what I would say.

Human: C.

Comp.: Yes, you've done it. Good work. If you want to play again, start right in.

Notice that the human can input either a guess (followed by a period) or a question (followed by a question mark). Possible guesses are A, B, C, D, etc. Possible questions are Curves, Loose ends, etc.; the human is given a list of possible questions before running the program.

Medicine That logic game was modified to form a lesson about medicine. The medical case was designed by Preston Munter, of the Harvard Health Center; the programming was done by John Swets, Wallace Feurzeig, and Myra Breen. The object of the "game" is to diagnose an illness correctly, by performing sufficient tests.

The human can input either a diagnosis (Measles) or a request for a test (X-ray). Before running the program, the human is given a list of 35 possible diagnoses and 40 possible tests.

Here are excerpts from a long run, in 1983:

Comp.: It is mid-December. Late this afternoon, a patient was admitted to the hospital. He is a well-developed, well-nourished, 34-year-old Negro coal miner. He was brought in by his wife, who reports that he has had "the gripe" for about 4 days. During the past 3 days she has given him fluids and aspirin every 6 hours with no sign of improvement. Today he suddenly became much worse. For the past 2 days he had complained of feeling very warm and of pain in his belly. This morning he had a severe shaking chill that was so bad his teeth rattled.... Begin the physical examination as soon as you are ready.

Human: Physical exam?

Comp.: Fine. What in particular would you like to know?

Human: Chest exam?

Comp.: It is common practice in the medical profession to begin an examination by checking the patient's general appearance.

Human: General appearance?

Comp.: Acutely prostrated, lying in bed on his right side. Looks acutely ill, sweaty, apprehensive, but alert. Paroxysmal cough, bringing up a little rusty streaked sputum. Breathing with difficulty.

Human: EENT exam?

Comp.: Check all of the vital signs first!

Human: Temperature?

Comp.: 103.

Human: Blood pressure?

Comp.: 150/84.

Human: Pulse?

Comp.: 100 a minute. Occasional extra systole....

Human: Abdominal exam?
 Comp.: Diffusely tender, some splinting noticed in lower right quadrant and occasionally radiating to right flank area.

Human: The grippe.
 Comp.: There's insufficient info for any conclusive diagnosis at this time. You're not being careful!

Human: Musculo-skeletal system?
 Comp.: Essentially negative, except for generalized weakness and aching.

Human: Extremities?
 Comp.: Normal, except nail beds are questionably cyanotic but difficult to determine because of patient's skin color.

Human: Rectal exam?
 Comp.: You're not being careful! The patient is too acutely ill for a rectal examination.

Human: Laboratory reports?
 Comp.: Fine. Which lab tests would you like to have done?

Human: RBC?
 Comp.: A red blood count will be done. Results will be sent to you as soon as possible.... Would you like to alert the night nurse to set up the lab tests you'll want to see tomorrow?

Human: Yes.
 Comp.: Good. That will speed things up. However, there's nothing further that can be done tonight to help the patient. Let's call it quits for now. Perhaps you'll be in a better position to diagnose the case tomorrow after the first lab reports come in and you have reexamined the patient. Have a good night's sleep, and when you're back on duty tell me to "proceed."

Human: Proceed.
 Comp.: It's morning and you're making your rounds. You've come to our patient and are given the night nurse's report: "Temperature remained high but patient spent a reasonably comfortable night. Cough diminished but persists. Occasional traces of blood in sputum. Patient still dyspneic and still has pain in abdomen. The second series of lab tests was initiated." Some of the first series of lab reports have just come in: urine exam, insufficient fluid for analysis; RBC, 5 million; WBC, 15500; differential, shift to left.... You may be prepared to make the diagnosis now. Are you?

Human: No.
 Comp.: Fine. Nevertheless, we've got a fair amount of info — at least enough to raise some, perhaps several, diagnostic possibilities. Have you any in mind?

Human: Yes.
 Comp.: Name one.

Human: Pulmonary infarction.
 Comp.: Fine. Pleural pain, dyspnea, fever, and the physical exam signs and symptoms are certainly consistent with pulmonary infarction. Do you think that shaking chills and the presence of rusty sputum further support this diagnosis?

Human: No.
 Comp.: Right. Though the white count is consistent, isn't it?

Human: No.
 Comp.: Wrong. The white count is consistent. This should further support your diagnosis. We've considered one possibility. Do you have another diagnosis in mind?

Human: No.
 Comp.: In that case I'd like to talk about viral pneumonia. The tachycardia, high WBC, elevated respiratory rate, shaking chills, bloody sputum, and severe pleural pain all lend weight to that diagnosis — right?

Human: Yes.
 Comp.: You're way off base. There's not an iota of truth to any part of what I just said; in fact, the opposite of what I said is true. You'd better review some elementary medicine, doctor....

Human: Pleural effusion.
 Comp.: A good contender, but let's be careful. What evidence would rule it in or out?

Human: Positive evidence of another disease.
 Comp.: Is there such evidence?

Human: Possible evidence though inconclusive.
 Comp.: Since you aren't sure, continue the examination until you have conclusive evidence one way or the other.

Human: RBC?
 Comp.: An RBC is not indicated at this time. It's an unnecessary expense for a poor patient.

Human: WBC?
 Comp.: 22500.

Human: Staph pneumonia.
 Comp.: Fair enough. But where is the guilty organism? When you have isolated staphylococcus aureus in significant numbers in your samples of this patient's fluids or excretions, let me know.

Human: Pneumococcal pneumonia.
 Comp.: Are you absolutely sure?

Human: Yes.
 Comp.: You can't be, until you have irrefutable evidence. What would help you nail down the diagnosis?

Human: Evidence from sputum smear and culture.
 Comp.: Good. Anything else?

Human: No.
 Comp.: I think you might also want evidence from x-ray findings. Go ahead and we'll see what happens.

Human: X-ray?
 Comp.: Not ready yet. Try the blood lab.

Human: Differential?
 Comp.: Marked shift to left....

Human: X-ray?
 Comp.: The x-ray shows: area of consolidation, right lower lobe, consistent with pneumatic process. This is a good time to review the total findings and to make another tentative diagnosis — if you like.

Human: Evidence of fluid?
 Comp.: No evidence of fluid.

Human: Sputum smear?
 Comp.: Gram-positive rods.

Human: Strep pneumonia.
 Comp.: Fine, but how can you distinguish this from any other bacterial pneumonia? Wait until you have an irrefutable case.

Human: Viral pneumonia.
 Comp.: But what about shaking chills, rusty sputum, elevated WBC, etc. — all the evidence of an acute febrile bacterial process? You are in error, doctor.

Human: Sputum culture?
 Comp.: Presence of pneumococci.

Human: Strep pneumonia.
 Comp.: Incredible! You have disregarded the positive findings that should clearly have established the correct diagnosis.

Human: Pneumococcal pneumonia.
 Comp.: Very good. It's obvious now, though there were some confusing signs and symptoms in the picture. You've seen your way through to the correct diagnosis.

Religion Bruce Jones programmed the computer to teach you about religion. Here's an example:

Comp.: This program is designed to teach you some things about the so-called "documentary hypothesis" as formulated by Julius Wellhausen and others. First, what is your name?

Human: George
 Comp.: All right, George, who wrote the first 5 books of the Bible, according to tradition?

Human: Moses
 Comp.: You are off to a good start. Do you know what those first 5 books are called? They are called...

Human: the Pentateuch

Comp.: Right you are. Torah is the traditional Jewish name for them. They are also called the Pentateuch (“penta” = 5). People very early noticed that there was a problem with saying that Moses wrote the Torah. If you already know the basis of their suspicion, tell me.

Human: Some things in the text reflect later conditions, after Moses.

Comp.: You are right, of course, but there was an even more obvious problem. Try again.

Human: I don’t know.

Comp.: At the end of the Pentateuch the author says, “Moses... died there.” Does that seem strange to you?

Human: yes

Comp.: Why?

Human: Moses could not have written about his own death.

Comp.: Okay. Also, right after the report of Moses’ death and burial (Deut. 34), the text says, “No man knows the place of his burial TO THIS DAY.” How do those last three words strike you?

Human: They seem later than the event they are talking about.

Comp.: A good observation, George. Many people considered these words to be anachronistic. That is, “this day” is sharply distinguished from the time of the burial. It seems likely, therefore, that the time of writing was much later than the time of burial. Again (so the argument goes), Moses was not the author. Would these observations about the end of Deuteronomy convince you that Moses did not write the Torah?

Human: yes

Comp.: Why do you say that?

Human: A man cannot write his own obituary.

Comp.: Probably not many people would have been convinced if our evidence were so limited. After all, Moses could have written everything except the last chapter. However, as early as the 18th century, people noticed another problem....

The computer searched through the human’s input, to see whether he used words indicating a correct answer. The computer never said a flat “Wrong”, since religious answers are a matter of personal belief, and since the human might be smarter or weirder than the computer program was prepared for.

Robots

In 1962 at MIT, Heinrich Ernst connected the computer to a mechanical hand that could feel. He made the hand build objects out of blocks, and made it put blocks into boxes.

Shakey One of the most famous robots is a guy named “Shakey”, built at the Stanford Research Institute (SRI) in 1970. His eye contains a TV camera (optical scanner). Instead of legs, he has wheels. Instead of arms, he has antennae (for feeling) and a bumper (for pushing). His brain is a computer: instead of carrying it around with him, he leaves it in another room and communicates with it by wireless methods.

To see how he works, suppose you type this message on his computer’s terminal:

Push the block off the platform.

He begins by looking for the platform.

If the platform’s not in the room, he goes out to the hall and steers himself through the hall (by looking at the baseboards) until he arrives at the next room. He peers in the room to see whether it contains a platform. If not, he hunts for another room.

When he finally finds a room containing a platform with a block on it, he tries to climb onto the platform to push the block off.

But before climbing the platform, he checks the platform’s height. If it’s too high to get onto easily, he looks for a device to help him climb it. For example, if a ramp is lying in the room, he pushes the ramp next to the platform then wheels himself up the ramp. Finally, he pushes the block off.

He can handle unexpected situations. For example, while he’s getting the ramp, suppose you pull the platform to a different

place. That doesn’t faze him: he hunts for the platform again, then pushes the ramp to it.

In 1971, Shakey’s powers were extended, so he can handle commands such as:

Turn on the lightswitch.

If the lightswitch is too high for his bumper to reach, he looks for a device to climb onto, such as a box. If he finds a box that looks helpful, he climbs onto it to check whether it is tall enough; if it is, he climbs off, pushes it to the lightswitch, climbs on it again, and finally flicks the switch.

Another task he can handle is:

Push three boxes together.

He finds the first box and pushes it to the second. Then he finds the third box, and pushes it to the second.

He understands over 100 words. Whatever command you give him becomes his “goal”, and he must reason out how to achieve it.

He might discover that to achieve the goal, he must achieve another goal first. For example, to move the block off the platform, he must first find the platform; to do that, he might have to look in another room; to do that, he must leave the room he’s in; to do that, he must turn his wheels.

Simulator One A robot named “Simulator One” is a mannequin that looks and acts like a patient: he can blink, breathe, cough, vomit, respond to drugs, and even die. You can take his blood pressure and pulse and make other measurements, using traditional medical equipment. He’s used in med school, to train doctors how to administer anesthetics during surgery.

Japan A newspaper article said that in Japan robots are used in many practical ways:

One robot arc-welds, reducing the time by 90%. Another grasps an object, determines the best way to pack it in a box, and does the packing; it uses television cameras and delicate arms. Another washes windows. Another wiggles a rod to catch a fish, takes the fish off the hook, dumps it into a bin, and returns the line to the water. Another directs traffic. Talking robots are being used instead of kimono-clad females in inns and restaurants.

Commenting on the quality of life in Japan, the article went on to say people are buying whiffs of oxygen from vending machines.

The article was tacked on the bulletin board at the MIT Artificial Intelligence Lab, together with this graffito about how the Japanese robots would act differently if they were as smart as people....

Human: Weld these parts.

Robot: The steel in those parts is grossly inferior. They must have been made in the U.S. Send them back. Also, have my welding machine tuned up.

Human: Pack those widgets.

Robot: Can I break them in half first?

Human: No.

Robot: “No” is not an allowed answer. I have only small shipping boxes. (Proceeds to break widgets in half and stuff them into boxes.)

Human: Wash those windows.

Robot: What?? And get my precious electrical and mechanical components all wet??

Human: Catch a fish.

Robot: (Proceeds to catch a fish, take it off the hook, and throw it back.) Okay.

Human: What did you throw it back for?

Robot: It was under the size limit. Anyway, it was full of mercury.

Human: Direct that traffic.

Robot: Will I get hazardous duty pay for this work?

Human: Greet those customers.

Robot: Sorry, my shift is over.

Human: (Ready to faint.) Give me some air.

Robot: Use that “machine” over there. (Points to vending machine.)

Be poetic

Programmers have taught the computer to compose its own poetry and other literature.

The Taoist Hiroshima Elevator (1961)

In Italy, Nanni Balestrini created poetry by feeding the computer 3 passages.

He fed the computer this passage from Lao Tzu's *Tao Te Ching*:

While the multitude of things comes into being, I envisage their return.
Although things flourish, they all return to their roots.

He fed this passage from Michihito Hachiya's *Hiroshima Diary*:

The blinding fireball expands rapidly, thirty times brighter than the sun.
When it reaches the stratosphere, the summit of the cloud takes on the well-known mushroom shape.

He fed this from Paul Goldwin's *The Mystery of the Elevator*:

head pressed on shoulder, hair between lips, lay motionless without speaking,
till he moved his fingers slowly, trying to grasp

He told the computer to mix them up. Here's the result:

Head pressed on shoulder, thirty times
brighter than the sun, I envisage their return —
until he moved his fingers slowly.... And while the multitude
of things comes into being, at the summit of the cloud
they all return to their roots and take on
the well-known mushroom shape, endeavoring to grasp.

Hair between lips, they all return
to their roots. In the blinding fireball
I envisage their return — until he moved his fingers
slowly — and although things flourish
takes on the well-known mushroom shape, endeavoring
to grasp, while the multitude of things comes into being.

In the blinding fireball I envisage
their return. When it reaches the stratosphere, while the multitude
of things comes into being, head pressed
on shoulder, thirty times brighter than the sun,
they all return to their roots. Hair
between lips takes on the well-known mushroom shape.

They lay motionless without speaking. Thirty times
brighter than the sun, they all return
to their roots. Head pressed on shoulder,
they take on the well-known mushroom shape, endeavoring
to grasp. And although things flourish,
they expand rapidly, hair between lips.

While the multitude of things comes into being in the blinding
fireball, they all return
to their roots. They expand rapidly — until he moved
his fingers slowly, when it reached the stratosphere
and lay motionless without speaking, thirty times
brighter than the sun, endeavoring to grasp.

I envisage their return — until he moved his fingers
slowly in the blinding fireball.
They all return to their roots. Hair
between lips and thirty times brighter than the sun
lay motionless without speaking. They expand,
rapidly endeavoring to grasp the summit.

In Balestrini's program, *Tao Te Ching* was represented as 4 phrases:

while the multitude of things comes into being
I envisage their return
although things flourish
they all return to their roots

Hiroshima Diary was represented as 6 phrases, and *The Mystery of the Elevator* as 5.

For each verse, the computer was told to choose 9 phrases at random and print them in a random order (never juxtaposing phrases from the same passage), to form 6 lines of roughly equal metrical length.

Actually the computer printed the poem in capital letters, without punctuation; Balestrini himself then added the punctuation and polished the grammar. The whole thing was actually done in Italian; you've been reading Edwin Morgan's translation, with my punctuation.

Bubbles (1966)

At Northwestern University, programmers made the computer compose nice poetry. To use their program, you type a list of nouns, verbs, and other words. The computer randomly chooses five of your words to be **theme words**. The computer combines all your words to form sentences, but chooses the theme words more often than the others. It combines the sentences into verses and tries to keep the lengths of the lines approximately equal. It puts a theme word into the title.

In one poem, the computer chose *bubble* to be a theme word. The title was: ODE TO A BUBBLE. The poem contained phrases such as, "Ah, sweet bubble." The word *bubble* appeared so often that even the stupidest reader could say: "Oh, yeah. I really understand this poem. Ya see, it's about a bubble."

The poem had all the familiar poetic trappings, such as "but alas!", which marked the turning point. (Cynics argue that the poem didn't *really* have a turning point, since the computer didn't have the faintest idea what it was saying!)

Kids and physics (1968)

In England at Manchester University, Mendoza made the computer write children's stories. Here's a story the computer composed:

The sun shone over the woods. Across the fields softly drifted the breeze, while then the clouds, which calmly floated all afternoon, moved across the fields. Squirrel, who scampered through the trees, quickly ran off; and off noisily ran Little Grey Rabbit. She sniffed at the house; but out of the door noisily hurried Hare, who peered at slowly the flowers. Squirrel hurriedly scampered over the woods and fields, but Old Grey Owl flew over the woods and fields. Down the path to the woods ran Little Grey Rabbit, who then sniffed at a strawberry pie.

The first paragraph uses these words:

	verbs	moved	drifted	shone	floated	touched	melted	looked	down	on
Nouns										
the clouds	1	1	0	1	0	0	1	0		
the sun	0	1	1	1	1	0	1	1		
the breeze	1	1	0	1	1	2	0	0		
the sky	0	0	0	0	1	0	1	1		
Adverbs										
gently	1	1	1	1	1	1	1	1		
quietly	1	1	1	1	1	1	1	1		
loudly	1	1	1	1	1	1	1	1		
softly	1	1	1	1	1	1	1	1		
calmly	1	1	1	1	1	1	1	1		
soon	1	1	1	1	1	1	1	1		
then	1	1	1	1	1	1	1	1		
(no adverb)	2	2	2	2	2	2	2	2		
Endings										
by	1	1	0	1						
over the woods	1	1	1	1						
across the fields	1	1	1	1						
through the trees	1	1	1	1						
down	0	0	1	0						
for a long time	0	0	1	1						
all day	1	1	1	1						
all afternoon	1	1	1	1						
the grass					1	1	1	1		
the leaves of the trees					1	1	1	1		
the garden					1	1	1	1		
the flowers					1	1	1	1		
the little house					1	0	1	1		
the old oak tree					1	1	1	1		
the treetops					1	1	1	1		

ADDITIONAL WORDS: which, and, while, they, it

To construct a sentence, the computer uses that table. Here's how:

First, the computer randomly chooses a noun. Suppose it chooses *the sun*. Then it looks across the row marked *the sun*, to choose a verb whose score isn't 0. For example, it's possible that *the sun shone* but impossible that *the sun melted*. Suppose it chooses *shone*.

Then it looks down the column marked *shone*, to choose an adverb and an ending. Notice that the ending can't be *by*, since its score is 0. *No adverb* has a score of 2, whereas *gently* has a score of 1; that makes *no adverb* twice as likely as *gently*.

If the computer chooses *no adverb* and *over the woods*, the resulting sentence is: The sun shone over the woods. In fact, that's the first sentence of the story you just read.

The computer occasionally changes the word order. For example, instead of typing "The breeze drifted softly across the fields", the computer begins the second sentence by typing, "Across the fields softly drifted the breeze".

To combine short sentences into long ones, the computer uses the words at the bottom of the table: *which*, *and*, *while*, *they*, and *it*. If two consecutive clauses have the same subject, the computer substitutes a pronoun: *they*

replaces *the clouds*; *it* replaces *the sun*, *the trees*, and *the sky*. The program says a *which* clause can come after a noun (*not* a pronoun); the *which* clause must use a different verb than the main clause.

Here's the vocabulary and table for the second paragraph:

	verbs	scampered	flew	ran	hurried	sniffed at	peered at	ate	munched and	crunched
Nouns										
Little Grey Rabbit	0	0	2	3	1	1	0	0		
Old Grey Owl	0	3	0	0	1	3	2	2		
Squirrel	3	0	1	1	1	1	3	3		
Hare	0	0	0	2	1	1	2	2		
Adverbs										
then	0	1	1	1	1	1	0	0		
slowly	0	2	0	0	1	1	1	1		
quickly	1	1	1	1	0	0	1	1		
soon	1	0	1	1	0	0	1	1		
happily	1	0	0	1	0	0	1	1		
gaily	1	0	0	1	0	0	1	1		
noisily	1	0	1	1	0	0	2	3		
(no adverb)	5	4	4	5	2	2	5	5		
Endings										
off	1	1	1	1						
over the woods and fields	1	1	1	1						
through the trees	1	1	1	1						
among the treetops	0	1	0	0						
into the home	1	0	1	1						
out of the door	1	0	1	1						
down the path to the woods	1	0	1	1						
about the garden	1	1	1	1						
the house					1	1	0	0		
the hollow tree					1	1	0	0		
an old oak tree					1	1	0	0		
the flowers					1	1	0	0		
two buns					1	1	1	1		
a strawberry pie					1	1	1	1		
six cabbages					1	1	1	1		

ADDITIONAL WORDS: who, and, but, she, he

Here's another story the program produced:

The breeze drifted by. Across the fields softly moved the clouds; and then the breeze, which calmly touched the treetops, drifted across the fields. Quietly the sun shone over the woods. The sky calmly shone across the fields.

Out of the door ran Squirrel; and off hurried Hare, who munched and crunched two buns happily. Off slowly flew Old Grey Owl, and Squirrel soon ate two buns. Old Grey Owl, who peered at a strawberry pie, munched and crunched two buns; but noisily Little Grey Rabbit, who peered at an old oak tree, slowly ran down the path to the woods. Soon she hurried down the path to the woods, but then she sniffed at two buns. She hurried down the path to the woods.

Why did Mendoza make the computer write those stories? He explains:

This work all began when a well-known scientist joined our physics department. He had spent several years away from academic life and was able to take a long cool look at academic procedures. He soon formed the theory that students never learned any ideas; all they learned was a vocabulary of okay words which they strung together in arbitrary order, relying on the fact that an examiner pressed for time would not actually read what they had written but would scan down the pages looking for these words. I set out to test his hypothesis.

I began by writing “Little Grey Rabbit” stories. I tested these stories out on my very small children; but after some minutes they grew irritable, because nothing actually happened. This shows that even small children of three can measure entropy.

Then I altered the vocabulary and grammar — making the sentences all very dead — to imitate the style of physics textbooks. The endpoint came when a colleague at another university secretly sent me an exam a week before it was given to the students. I wrote vocabularies and copied down what the computer emitted. Using a false name, I slipped my paper in among the genuine ones. Unfortunately, it was marked by a very conscientious man, who eventually stormed into the Director’s office shouting, “Who the hell is this man — why did we ever admit him?” So perhaps my colleague’s hypothesis was wrong, and students are a little better than we think.

Here’s one of the computer’s answers:

In electricity, the unit of resistance is defined by electrolysis; and the unit of charge, which was fixed at the Cavendish lab in Rayleigh’s classic experiments, was measured at the Cavendish lab. Theoretically, the absolute ohm is defined in a self-consistent way. The unit of resistance, which was determined with a coil spinning in a field, was fixed at the Cavendish lab; and this, by definition, is expressed in conceptual experiments. Theoretically the absolute ohm, which was redetermined using combined e.m.u. and e.s.u., is expressed by the intensity at the center of a coil.

Here’s another of the computer’s answers:

In this country, Soddy considered Planck’s hypothesis from a new angle. Einstein 50 years ago asserted quantization.

At a photocathode, electrons which undergo collisions in the Compton effect as energy packets or quanta are emitted at definite angles; nevertheless, particles in a photocell produce photoelectrons of energy $h\nu = E_0$. Photons *in vacuo* transmute into lower frequencies, and light quanta in the Compton effect emit emission currents.

Particles emit current proportional to energy; electrons *in vacuo* interact with loss of surface energy (work function); nevertheless, particles which are emitted in a photocell with conservation experimentally are conserved with energy $h\nu$. The former, at a metal surface, undergo collisions with emission of current; and at a metal surface, electrons produce emission currents.

Einstein assumed the gas of quantum particles; but quite recently Rayleigh, who quite recently solved the problem in an old-fashioned way, considered radiation classically. Planck, who this century assumed the A and B coefficients, explained the gas of quantum particles but before Sommerfeld; Rayleigh, who quite recently was puzzled on Boltzmann statistics, tackled the problem with disastrous results.

Planck, who assumed the gas of quantum particles in 1905, this century considered the ultraviolet catastrophe; but quite recently Jeans, who tackled the problem in an old-fashioned way, was puzzled with disastrous results.

Black body radiation that exerts thermodynamic forces in an engine is equivalent to a relativistic system. Out of a black body, a photon that is equivalent to (out of a black body) an assembly of photons is assumed to be a non-conservative system; at the same time, thermodynamically, black body radiation that in a piston is assumed to be a relativistic system exerts quantized forces.

The radiation gas that obeys Wien’s displacement law is considered as a system of energy levels. Quantally, a quantum particle exerts a Doppler-dependent pressure, although this produces equilibrium transition probabilities. Black body radiation in an engine produces equilibrium transition probabilities.

Aerospace (1968)

In 1968, Raymond Deffrey programmed the computer to write fake reports about the aerospace industry. Shortly afterwards, I improved the program. The improved program contains these lists:

Introductory phrases

thus	to some extent
indeed	for the most part
however	on the other hand
similarly	as a resultant implication
moreover	in respect to specific goals
in addition	in view of system operation
furthermore	utilizing the established hypotheses
for example	based in system engineering concepts
in particular	based on integral subsystem considerations
in this regard	considering the postulated interrelationships

Noun phrases

the structural design
the total system rationale
the sophisticated hardware
any discrete configuration made
the fully integrated test program
the preliminary qualification limit
the product configuration baseline
any associated supporting element
the independent function principle
the subsystem compatibility testing
the greater flight-worthiness concept
the characterization of specific criteria
a constant flow of effective information
the anticipated third-generation equipment
initiation of critical subsystem development
the evolution of specifications over a given time
the incorporation of additional mission constraints
the philosophy of commonality and standardization
a consideration of system and/or subsystem technologies
a large portion of the interface coordination communication

Verb phrases

adds explicit performance limits to
effects a significant implementation to
adds overriding performance constraints to
presents extremely interesting challenges to
must utilize and be functionally interwoven with
is further compounded, when taking into account
requires considerable systems analysis to arrive at
necessitates that urgent consideration be applied to
maximizes the probability of success and minimizes time for
recognizes the importance of other systems and necessity for

To produce a typical sentence, the computer prints an introductory phrase, then a noun phrase, then a verb phrase, then a noun phrase. The phrases are chosen randomly.

Each paragraph consists of 6 sentences. The computer isn’t allowed to use the same phrase twice within a paragraph. The introductory phrase is omitted from the first sentence of the first paragraph, the second sentence of the second paragraph, etc.; so the report can’t begin with the word *furthermore*, and the style varies.

Here’s the beginning of one such report:

The Economic Considerations of the Aerospace Industry

A large portion of the interface coordination communication necessitates that urgent consideration be applied to the product configuration baseline. For example, the fully integrated test program adds explicit performance limits to the independent function principle. Moreover, the sophisticated hardware presents extremely interesting challenges to the philosophy of commonality and standardization. In view of system operation, a constant flow of effective information must utilize and be functionally interwoven with the preliminary qualification limit. In addition, any discrete configuration made adds overriding performance constraints to any associated supporting element. Thus, the anticipated third-generation equipment maximizes the probability of success and minimizes time for the total system rationale.

Me-Books (1972)

In 1972, Freeman Gosden Jr. started the Me-Books Publishing Company. It published books for kids. But if you bought a Me-Book for your child, you wouldn’t see in it the traditional names “Dick, Jane, and Sally”; instead, you’d see the name of your own child. To order the book, you had to tell the company the names of all your children, their friends, and pets. Their names appeared in the story.

The story was printed beautifully, in a 32-page hard-covered book with pictures in color. It cost just \$3.95.

You could choose from 4 stories: “My Friendly Giraffe”, “My Jungle Holiday”, “My Birthday Land Adventure”, and “My Special Christmas”.

For example, if you lived on Jottings Drive, and your daughter’s name was Shea, and her friend’s name was Douglas, the story “My Friendly Giraffe” included paragraphs such as:

One morning Shea was playing with Douglas in front of her home. When she looked up, what do you think she saw walking down the middle of Jottings Drive? You guessed it. A giraffe!

Ted Nelson, author of *Computer Lib*, played a trick. He ordered a copy of "My Friendly Giraffe", but pretended that his child's name was "Tricky Dick Nixon" who lived on "Pennsylvania Ave." in "Washington". Sure enough, the company sent him "My Friendly Giraffe: A Me-Book for Tricky Dick". Here are excerpts:

Once upon a time, in a place called Washington, there lived a little boy named Tricky Dick Nixon. Now, Tricky Dick wasn't just an ordinary little boy. He had adventures that other little boys and girls just dream of. This is the story of one of his adventures. It's the story of the day that Tricky Dick met a giraffe....

As the giraffe came closer and closer, Tricky Dick started to wonder how in the world he was going to look him in the eye....

Tricky Dick knew there were no jungles in Washington. Especially on Pennsylvania Ave. But Tricky Dick wasn't even a little bit worried. First, because he was a very brave little boy. And second, because he knew that his friend, the giraffe, would never take him anyplace bad....

Tricky Dick was home. Back in Washington. Back on Pennsylvania Ave. And with a story to tell his friends, that they wouldn't have believed if they hadn't seen Tricky Dick riding off on the giraffe's back. Tricky Dick would long be a hero to those who had seen him that day....

There would be many other exciting adventures for Tricky Dick and his friends. And maybe, just maybe, if you're a very good boy, someday we'll tell you about those, too.

Me-Books were commercially successful: many thousands of them were sold. Teachers liked the books, because the books encouraged students to read: students enjoyed reading books that contained their own names.

Porn (1973)

I programmed the computer to turn words into sentences. For example, Judith Ann Abrams fed the computer this word list....

Names: Barbara, Buster, Clyde, Colin, Debbie, Ed Shanahan, Jeff, Keith, Kerri, Liz, Miles, Nicholson 700, Norwalk

Other nouns: bastard, bikini, bitch, boat, cigarette, creep, dress, ethanol, eyelid, fly, gambling, jerk, knee, kneecap, lip, mouth, navel, piece, pornography, shirt, stud, tongue, virgin, whiskey sour, whore

Transitive verbs: bite, caress, castrate, climb, enjoy, kiss, knife, lick, pick, pull, rip screw, shake, take off, teach, undo

Other verbs: disapprove, fall down, fly, hesitate, sing, sleep, sleepwalk, sneeze, vanish, walk

Adjectives: bitchy, creepy, dead, enthusiastic, evil, gaudy, green, gross, horny, insane, mixed-up, monstrous, multicolored, neurotic, obscene, paranoid, prickly, profane, provocative, repulsive, sensuous, sexual, sexy, shiny, sick, stimulating, strange, succulent, tasty, unsettled, weird

A **transitive verb** is an action you can do to an object. For example, since you can *bite* and *caress* an object, *bite* and *caress* are transitive verbs.

Here's the beginning of what the computer printed:

A Sexy, Profane Creep's Tongue
by Judith Ann Abrams and the computer

The lip is tasty.

Ugh!

The mouth is enthusiastic. Does the gaudy bastard sleepwalk?

The tongue's whore caresses the bikini. When Nicholson 700 is screwing the creepy jerk over a piece, Jeff is teaching the gambling. Does paranoid Liz hesitate? Why does the tongue kiss the creep stimulatingly on horny Buster?

Undo! Pull the fly repulsively! A multicolored, enthusiastic cigarette is neurotic. Is a multicolored, weird dress's navel under the strange bikini? Why is the obscene dress's cigarette Colin's ethanol? Rats! Why is the dead, insane, tasty, shiny, knifing creep's bitch unsettled and strange?

Clyde is over Ed Shanahan. Kissing the bastard, a paranoid ethanol is unsettled. Why is the prickly kneecap's mouth on the creepy piece? Teaching Norwalk is sneezing paranoidly and grossly under provocative, shiny Debbie. As a sexy, walking tongue's virgin takes off the bitchy cigarette over the profane stud then castrates enthusiastic Miles's lip paranoidly, the dead,

gaudy knee is stimulating and multicolored. Even though the dead, insane piece licks strangely and neurotically in the flying, mixed-up eyelid's knee, the dead, biting, obscene bikini is on a repulsive mouth's gambling.

The pornography is gaudy. Kerri sleepwalks. Why is the tongue sensuous? Buster is sick. Is Miles monstrous? Debbie is neurotic and paranoid, when a stimulating fly picks the navel's jerk under Ed Shanahan. Why is the dress succulent? Hesitating, a kneecap sleeps and climbs the dead, bitchy ethanol. As insane Colin's bastard falls down weirdly in a sensuous dress, green, unsettled Miles's virgin is strange and sexual.

Is the creepy eyelid provocative? The gambling's whisky sour teaches a navel.

The computer turned her adjectives into adverbs, by adding *ly* and making other changes. *Stimulating* became *stimulatingly*, and *gross* became *grossly*. Unfortunately, the computer's method wasn't perfect: the computer turned *neurotic* into *neurotically* (instead of *neurotically*).

It conjugated her verbs. *Screw* became *screwing*, and *bite* became *biting* (the computer dropped the *e*). *Lick* became *licks*, and *teach* became *teaches* (the computer added the *e* after the *ch*).

It added *s* to her nouns. *Tongue* became *tongue's*. *Miles* became *Miles's* (it should have become *Miles'*).

For each sentence, the grammar is chosen randomly. The chance is 10% that the sentence will begin with an exclamation. If the sentence isn't merely an exclamation, the chance is 18% that the sentence will be a question.

If it's a question, there's a 40% chance it will begin with the word *why*. There's a 50% chance the question's main part will have the form *does... noun phrase... verb phrase*, and a 50% chance it will have this form instead: *is... noun phrase... complement*.

To construct a noun phrase from nouns, adjectives, etc., the computer uses random numbers. It uses random numbers to also construct verb phrases and complements.

The program uses a special variable, called *W*. At the beginning of the composition, *W* is near zero; but it tends to increase as the composition progresses. It affects the **complexity**. When *W* is large, the chance is large that the computer will print adjectives, adverbs, subordinate clauses, and correlative clauses. This sentence was produced by a small *W*:

The lip is tasty.

This sentence was produced by a large *W*:

As a sexy, walking tongue's virgin takes off the bitchy cigarette over the profane stud then castrates enthusiastic Miles's lip paranoidly, the dead, gaudy knee is stimulating and multicolored.

Poetic images (1973)

One of my students, Toby D'Oench, made the computer create poetic images, such as these:

TO GUINEVERE — LADY OF THE LAKE

Silent mists
Billow in creations
Windmills for flames evolve into ethers
Merlin again

MY MEMORY

Frozen children
Quiver with leaves
Creations with leaves hover over thoughts
Gardens of verse

A NEW ENGLAND BARN

Lazy fragrances
Waft by ethers
Seas on fragrances billow in sorrow
Rusted pitchforks

NEWPORT

Frozen sails
Slumber in fog
Hazes for sails waft by thoughts
Docks — yachts — luxuries of eras gone by

The program contains these lists:

<u>Adjectives</u>	<u>Prepositions</u>	<u>Verbs</u>
fleeting	of	billow in
crimson	on	glitter with
silent	under	flutter by
sensate	above	drift with
pliant	below	flow into
gloomy	in	ponder about
pallid	with	waft by
inky	by	quiver with
frozen	for	hover over
lazy	through	gleam like
		wander through
		slumber in
		dart by
		evolve into
		sing to

Title... noun... ending

TO REMBRANDT... windmills... A simple brush
 WAITING FOR THE PATIENT... ethers... Waiting
 THE PROPHET... visions... Then a word
 LISTERINE... breaths... Plastic society
 NEWPORT... sails... Docks — yachts — luxuries of eras gone by
 EXISTENCE... seas... In the beginning?
 SUMMER IN WATTS... flames... Tar-street neon — and the night
 TO GUINEVERE — LADY OF THE LAKE... mists... Merlin again
 NOON IN CALCUTTA... hazes... Emaciated dark forms strewn like garbage
 WEST HARBOR... fog... A solitary gull slices through
 A NEW ENGLAND BARN... fragrances... Rusted pitchforks
 A CHILD'S MICROSCOPE... creations... The wonderful amoeba
 A GROUP PORTRAIT... bundles... Christmas
 THE MILKY WAY... cosmos... A gooey mess
 TOMBSTONE... sorrow... Rubbings
 LIFE AT THE END OF A BRANCH... leaves... Swirling to the ground
 SEASHELLS AND THINGS... waves... Dribble-dribble-dribble castle
 A BEAVER POND... reeds... Thwack
 MY MEMORY... children... Gardens of verse
 EINSTEIN... thoughts... Somehow through this — an understanding of a superior order

To create a poetic image, the computer fills in this form:

<u>TITLE</u>				
<u>Adjective</u>	<u>Noun that goes with the title</u>			
<u>verb</u>	<u>Noun</u>			
<u>Noun</u>	<u>Preposition</u>	<u>Noun</u>	<u>Verb</u>	<u>Noun</u>
<u>Ending that goes with the title</u>				

Analyze writing

The computer can analyze what humans write.

English poetry

Can the computer analyze English poetry? From 1957 to 1959 at Cornell University, Stephen Parrish made the computer alphabetize the words in Matthew Arnold's poetry. Here's an excerpt:

	Page in book	Poem's title	Line in poem
CONSCIOUS			
back with the conscious thrill of shame	181	Isolation Marg	19
conscious or not of the past	287	Rugby Chapel	45
CONSCIOUSNESS			
the last spark of man's consciousness with words	429	Empedocles II	30
and keep us prisoners of our consciousness	439	Empedocles II	352
CONSECRATE			
Peter his friend with light did consecrate	445	Westmin Abbey	50
CONSECRATES			
which consecrates the ties of blood for these indeed	196	Frag Antigone	31

To find out what Matthew Arnold said about love, just look up

LOVE. Such an index is called a **concordance**.

That concordance was the first produced by a computer. Previously, all concordances of poetry were created by hand, using filing cards. For example, in 1870 a group of researchers began creating a concordance to Chaucer, by hand. They started at the letter A. 45 years later, they were only up to the letter H!

Did the poet Shelley steal ideas from others? Joseph Raben, at Queens College, believed Shelley borrowed imagery from Milton. To prove it, in 1964 he made the computer produce concordances to Shelley's *Prometheus Unbound* and Milton's *Paradise Lost* and compare them. The computer found many similarities between Shelley and Milton.

What were Shakespeare's favorite words? In 1971 at Münster University in Germany, Marvin Spevack fed the computer all the works of Shakespeare, and made it count how often each word occurs. Disregarding trivial words such as *a* and *the*, the computer discovered Shakespeare's favorite word was *love*: he used it 2,271 times. Next come *heart*, *death*, *man*, *life*, and *hand*. He never used the word *hero*. In *Macbeth*, the word *good* occurs more often than any other adjective, noun, or adverb, and more often than most verbs.

By counting words, other researchers made the computer graph the rise and fall of themes in a novel.

American history

Who wrote the *Federalist Papers*? Historians knew some of the papers were by Alexander Hamilton and others by James Madison, but the authorship of the remaining papers was in dispute.

In 1964, Mosteller and Wallace made the computer compare the literary styles of the papers, by counting the frequency of words such as *by*, *enough*, *from*, *to*, *upon*, *while*, and *whilst*. It concluded that all the disputed papers were written by Madison, not Hamilton.

The statistical evidence was so high that historians accept the computer's finding as fact.

The Bible

Can the computer analyze the Bible? In 1951, Texas clergyman John Ellison made the computer compare 309 Greek manuscripts of the New Testament. Underneath each word of a standard text, the computer printed the variants found in other manuscripts. It classified the manuscripts according to their similarities.

In 1957, he published a concordance to the Revised Standard Bible, and a pair of other researchers (Tasman & Busa) indexed the Dead Sea Scrolls.

Did the apostle Paul really write all those marvelous letters attributed to him in the New Testament? Or were they actually written by somebody else?

In 1964, Scottish clergyman Andrew Morton used the computer to deduce that Paul didn't write some of those letters.

All Morton did was count how often Paul used the Greek word *kai* in each sentence. *Kai* means *and*. Coming to a conclusion about Biblical authorship by counting just the word *and* might seem silly, but Morton said he analyzed 20 writers of ancient Greek and found each used *kai* with a constant frequency. In the "Pauline" letters, the frequency of *kai* varied a lot, implying some of them were not by Paul.

Ellison distrusted Morton's assumption that a man's literary style must remain constant. He warned: if Morton's method were applied to the Declaration of Independence and Thomas Jefferson's letters to his wife, the computer might conclude that either Jefferson didn't write the Declaration of Independence or another man was writing love letters to Mrs. Jefferson. In 1965, to prove his point, he applied Morton's method to 2 of Morton's own articles on the subject: the computer concluded that Morton couldn't be the author of both!

Forgery

IBM programmed the computer to detect a forged signature — even if the signature looks correct to the naked eye.

To use the IBM forgery-detection system, write your signature by using IBM's special pen, attached to the computer. As you write, the computer notices how hard you press the pen against the paper and how fast you move the pen.

If somebody else tries to pretend he's you, he must sit down at the machine and try to duplicate your signature. If he presses the pen hardest at different points of the signature, or if he accelerates the pen's motion at different points, the computer says he's a fake.

The system works well, because the average crook trying to forge your signature will hesitate at the hard parts. His hesitation affects the pen's pressure and acceleration, which tell the computer he's faking.

IBM developed the system in 1979 but didn't start selling it until many years later. Now IBM sells an improved version. Remember: the system works just on signatures written with IBM's pen.

Artificial intelligence

You have what's called **natural intelligence** (except when your friends accuse you of having "natural stupidity"). A computer's intelligence, by contrast, is **artificial**. Can the computer's **artificial intelligence (AI)** ever match yours?

For example, can the computer ever develop the "common sense" needed to handle exceptions, such as a broken traffic light? After waiting at a red light for several hours, the typical human would realize the light was broken. The human would try to proceed past the intersection, cautiously. Would a computer programmed to "never go on red" be that smart?

Researchers who study the field of artificial intelligence have invented robots and many other fascinating computerized devices. They've also been trying to develop computers that can understand ordinary English commands and questions, so you won't have to learn a "programming language". They've been trying to develop **expert systems** — computers that imitate human experts such as doctors and lawyers.

Early dreamers

The dream of making a computer imitate us began many centuries ago....

The Greeks The hope of making an inanimate object act like a person can be traced back to the ancient Greeks. According to Greek mythology, Pygmalion sculpted a statue of a woman, fell in love with it, and prayed to the gods to make it come to life. His wish was granted — she came to life. And they lived happily ever after.

Ramon Lull (1272 A.D.) In 1272 A.D. on the Spanish island of Majorca, Ramon Lull invented the idea of a machine that would produce *all* knowledge, by putting together words at random. He even tried to build it.

Needless to say, he was a bit of a nut. Here's a description of his personality (written by Jerry Rosenberg, abridged):

Ramon Lull married young and fathered two children — which didn't stop him from his courtier's adventures. He had an especially strong passion for married women. One day as he was riding his horse down the center of town, he saw a familiar woman entering church for a High Mass. Undisturbed by this circumstance, he galloped his horse into the cathedral and was quickly thrown out by the congregants. The lady was so disturbed by his scene that she prepared a plan to end Lull's pursuit once and for all. She invited him to

her boudoir, displayed the bosom that he had been praising in poems written for her, and showed him a cancerous breast. "See, Ramon," she said, "the foulness of this body that has won thy affection! How much better hadst thou done to have set thy love on Jesus Christ, of Whom thou mayest have a prize that is eternal!"

In shame Lull withdrew from court life. On four different occasions a vision of Christ hanging on the Cross came to him, and in penitence Lull became a dedicated Christian. His conversion was followed by a pathetic impulse to try to convert the entire Moslem world to Christianity. This obsession dominated the remainder of his life. His "Book of Contemplation" was divided into 5 books in honor of the 5 wounds of Christ. It contained 40 subdivisions — for the 40 days that Christ spent in the wilderness; 366 chapters — one to be read each day and the last chapter to be read only in a leap year. Each chapter had 10 paragraphs to commemorate the 10 commandments; each paragraph had 3 parts to signify the trinity — for a total of 30 parts a chapter, signifying the 30 pieces of silver.

In his book's final chapter, he tried to prove to infidels that Christianity was the only true faith.

Several centuries later — in 1726 — Lull's machine was pooh-poohed by Jonathan Swift, in *Gulliver's Travels*.

Gulliver meets a professor who built such a machine. The professor claims his machine lets "the most ignorant person... write books in philosophy, poetry, politics, law, mathematics, and theology without the least assistance from genius and study."

The machine is huge — 20 feet on each side — and contains all the words of the language, in all their declensions, written on paper scraps glued onto bits of wood connected by wires.

Each of the professor's 40 students operates one of the machine's 40 cranks. At a given signal, every student turns his crank a random distance, to push the words into new positions.

Gulliver says:

He then commanded 36 of the lads to read the several lines softly as they appeared upon the frame. Where they found three or four words together that might make part of a sentence, they dictated to the 4 remaining boys, who were scribes. Six hours a day the young students were employed in this labor. The professor showed me several large volumes already collected, of broken sentences, which he intended to piece together, and out of those rich materials give the world a complete body of all arts and sciences.

Karel Capek (1920) The word **robot** was invented in 1920 by Karel Capek, a Czech playwright. His play "R.U.R." shows a factory where the workers look human but are really machines. The workers are dubbed *robots*, because the Czech word for *slave* is *robotnik*.

His play is pessimistic. The invention of robots causes unemployment. Men lose all ambition — even the ambition to raise children. The robots are used in war, go mad, revolt against mankind and destroy it. In the end only two robots are left. It's up to them to repopulate the world.

Isaac Asimov (1942) Many sci-fi writers copied Capek's idea of robots, with even more pessimism. An exception was Isaac Asimov, who depicted robots as being loving. He coined the word **robotics**, which means the study of robots, and in 1942 developed what he calls the "**3 Laws of Robotics**". Here's the version he published in 1950:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence, as long as such protection does not conflict with either the First or the Second Law.

Norbert Wiener (1947) The word **cybernetics** was invented in 1947 by Norbert Wiener, an MIT professor. He defined it to be "the science of control and communication in the animal and the machine." Wiener and his disciples, who called themselves **cyberneticists**, wondered whether it would be possible to make an electrical imitation of the human nervous system. It would be a "thinking machine". They created the

concept of **feedback**: animals and machines both need to perceive the consequences of their actions, to learn how to improve themselves. For example, a machine that is producing parts in a factory should examine the parts it has produced, the heat it has generated, and other factors, to adjust itself accordingly.

Like Ramon Lull, Wiener was strange. He graduated from Tufts College when he was 14 years old, got his doctorate from Harvard when he was 18, and became the typical “absent-minded professor”. These anecdotes are told about him:

He went to a conference and parked his car in the big lot. When the conference was over, he went to the lot but forgot where he parked his car. He even forgot what his car looked like. So he waited until all the other cars were driven away, then took the car that was left.

When he and his family moved to a new house a few blocks away, his wife gave him written directions on how to reach it, since she knew he was absent-minded. But when he was leaving his office at the end of the day, he couldn't remember where he put her note, and he couldn't remember where the new house was. So he drove to his old neighborhood instead. He saw a young child and asked her, “Little girl, can you tell me where the Wieners moved?” “Yes, Daddy,” came the reply, “Mommy said you'd probably be here, so she sent me to show you the way home.”

One day he was sitting in the campus lounge, intensely studying a paper on the table. Several times he'd get up, pace a bit, then return to the paper. Everyone was impressed by the enormous mental effort reflected on his face. Once again he rose from his paper, took some rapid steps around the room, and collided with a student. The student said, “Good afternoon, Professor Wiener.” Wiener stopped, stared, clapped a hand to his forehead, said “Wiener — that's the word,” and ran back to the table to fill the word “wiener” in the crossword puzzle he was working on.

He drove 150 miles to a math conference at Yale University. When the conference was over, he forgot he came by car, so he returned home by bus. The next morning, he went out to his garage to get his car, discovered it was missing, and complained to the police that while he was away, someone stole his car.

Those anecdotes were collected by Howard Eves, a math historian.

Alan Turing (1950) Can a computer “think”? In 1950, Alan Turing proposed the following test, now known as the **Turing test**:

In one room, put a human and a computer. In another room, put another human (called the Interrogator) and give him two terminals — one for communication with the computer, and the other for communication with the other human — but don't tell the Interrogator which terminal is which. If he can't tell the difference, the computer's doing a good job of imitating the human, and (according to Turing) we should say the computer can “think”.

Turing called it the **Imitation Game**. The Interrogator asks questions. The human witness answers honestly. The computer pretends to be human.

To win that game, the computer must be able to imitate human weaknesses as well as strengths. For example, when asked to add two numbers, it should pause before answering, as a human would. When asked to write a sonnet, a good imitation-human answer would be, “Count me out on this one. I never could write poetry.” When asked “Are you human”, the computer should say “yes”.

Such responses wouldn't be hard to program. But a clever Interrogator could give the computer a rough time, by requiring it to analyze its own thinking:

Interrogator: In the first line of your sonnet which reads “Shall I compare thee to a summer's day,” wouldn't “a spring day” do as well or better?
Witness: It wouldn't scan.
Interrogator: How about “a winter's day”? That would scan all right.
Witness: Yes, but nobody wants to be compared to a winter's day.
Interrogator: Would you say Mr. Pickwick reminded you of Christmas?
Witness: In a way.
Interrogator: Yet Christmas is a winter's day, and I don't think Mr. Pickwick would mind the comparison.
Witness: I don't think you're serious. By “a winter's day” one means a typical winter's day, rather than a special one like Christmas.

If the computer could answer questions that well, the Interrogator would have a hard time telling it wasn't human.

Donald Fink has recommended the Interrogator say, “Suggest an unsolved problem and some methods for working toward its solution,” and “What methods would most likely prove fruitful in solving the following problem...”

Turing believed computers would someday be able to win the game and therefore be considered to “think”. In his article, he listed 9 possible objections to his belief and rebutted them:

1. Soul Thinking's a function of man's immortal soul. Since computers don't have souls, computers can't think. **Rebuttal:** since God's all-powerful, He can give computers souls if He wishes. Just as we create children to house His souls, so should we serve Him by creating computers.

2. Dreadful If machines could equal us in thinking, that would be dreadful! **Rebuttal:** too bad!

3. Logicians Logicians have proved it's impossible to build a computer that can answer every question. **Rebuttal:** is it possible to find a *human* that can answer every question? Computers are no dumber than we. Though no one can answer every question, why not build a succession of computers, each one more powerful than the next, so every question could be answered by at least one of them?

4. Conscious Though computers can produce, they can't be *conscious* of what they've produced. They can't feel pleasure at their successes, misery at their mistakes, and depression when they don't get what they want. **Rebuttal:** the only way to be sure whether a computer has feelings is to become one. A more practical experiment would be to build a computer that explains step-by-step its reasoning, motivations, and obstacles it's trying to overcome, and also analyzes emotional passages such as poetry. Such a computer's clearly not just parroting.

5. Human A computer can't be kind, resourceful, beautiful, friendly, have initiative, have a sense of humor, tell right from wrong, make mistakes, fall in love, enjoy strawberries & cream, make someone fall in love with it, learn from experience, use words properly, be the subject of its own thought, have as diverse behavior as a man, or do something really new. **Rebuttal:** why not? Though such a computer hasn't been built yet, it might be possible in the future.

6. Surprise The computer never does anything original or surprising. It does only what it's told. **Rebuttal:** how do you know “original” human work isn't just grown from a seed (implanted by teaching) or the effect of well-known general principles? And who says computers aren't surprising? The computer's correct answers are often surprisingly different from a human's rough guesses.

7. Binary Nerve cells can sense gradual increases in electrical activity — you can feel a “little tingle” or a “mild pain” or an “ouch” — whereas a computer's logic is just binary — either a “yes” or “no”. **Rebuttal:** by using techniques such as “random numbers”, you can make the computer imitate the flexible, probabilistic behavior of the nervous system enough so the Interrogator can't tell the difference.

8. Rules Life can't be reduced to rules. For example, if a traffic-light rule says “stop when the light is red, and go when the light is green”, what do you do when the light is broken, and both the red and green appear simultaneously? Maybe you should have an extra rule saying in that case to stop. But some further difficulty may arise with that rule, and you'd have to create another rule. And so on. You can't invent enough rules to handle all cases. Since computers must be fed rules, they can't handle all of life. **Rebuttal:** though life's more than a simple set of rules, it might be the *consequences* of simple psychological laws of behavior, which the computer could be taught.

9. ESP Humans have extrasensory perception (ESP), and computers don't. **Rebuttal:** maybe the computer's random-number generator could be hooked up to be affected by ESP. Or to prevent ESP from affecting the Imitation Game, put both the human witness and the computer in a telepathy-proof room.

To make the computer an intelligent creature, Turing suggested two possible ways to begin. One way would be to teach the computer abstract skills, such as chess. The other way would be to give the computer eyes, ears, and other sense organs, teach it how to speak English, then educate it the same way you'd educate a somewhat handicapped child.

4 years later — on June 8, 1954 — Turing was found dead in

bed. The police say he died from potassium cyanide, self-administered. He'd been plating spoons with potassium cyanide in electrolysis experiments. His mother refuses to believe it was suicide, and hopes it was just an accident.

Understanding English

It's hard to make the computer understand plain English!

Confusion Suppose you feed the computer this famous saying:

Time flies like an arrow.

The computer might interpret that saying in 3 ways:

Interpretation 1 The computer thinks "time" is a noun, so the sentence means "The time can fly by as quickly as an arrow flies."

Interpretation 2 The computer thinks "time" is a verb, so the sentence means "Time the speed of flies like you'd time the speed of an arrow."

Interpretation 3 The computer thinks "time" is an adjective, so the sentence means "There's a special kind of insect, called a 'time fly', and those flies are attracted to an arrow (in the same way moths are attracted to a flame)."

Suppose a guy sits on a barstool and shares his drinks with a tall woman while they play poker for cash. If the woman says to him, "Up yours!", the computer might interpret it 8 ways:

The woman is upset at what the man did.
 The woman wants the man to raise up his glass, for a toast.
 The woman wants the man to up the ante and raise his bet.
 The woman wants the man to hold his cards higher, so she doesn't see them.
 The woman wants the man to pick up the card she dealt him.
 The woman wants the man to raise his stool, so she can see him eye-to-eye.
 The woman wants the man to pull up his pants.
 The woman wants the man to have an erection.

For another example, suppose Mae West were to meet a human-looking robot and ask him:

Is that a pistol in your pocket, or are you glad to see me?

The robot would probably analyze that sentence too logically, then reply naively:

There is no pistol in my pocket, and I am glad to see you.

In spite of those confusions, programmers have tried to make the computer understand English. Here are some famous attempts...

Baseball (1961) In 1961 at MIT, programmers made the computer answer questions about baseball.

In the computer's memory, they stored the month, day, place, teams, and scores of each game in the American League for one year. They programmed the computer so that *you can type your question in ordinary English*. The computer analyzes your question's grammar and prints the correct answer.

Here are examples of questions the computer can analyze and answer correctly:

Who did the Red Sox lose to on July 5?
 Who beat the Yankees on July 4?
 How many games did the Yankees play in July?
 Where did each team play in July?
 In how many places did each team play in July?
 Did every team play at least once in each park in each month?

To get an answer, the computer turns your questions into equations:

Question	Equations
Where did the Red Sox play on July 7?	place = ? team = Red Sox month = July day = 7
What teams won 10 games in July?	team (winning) = ? game (number of) = 10 month = July
On how many days in July did eight teams play?	day (number of) = ? month = July team (number of) = 8

To do that, the computer uses this table:

Word in your question	Equation
where	place = ?
Red Sox	team = Red Sox
July	month = July
who	team = ?
team	team =

The computer ignores words such as *the*, *did*, and *play*.

If your question mentions *Boston*, you might mean either "place = Boston" or "team = Red Sox". The computer analyzes your question to determine which equation to form.

After forming the equations, the computer hunts through its memory, to find the games that solve the equations. If an equation says "number of", the computer counts. If an equation says "winning", the computer compares the scores of opposing teams.

The programmers were Bert Green, Alice Wolf, Carol Chomsky, and Kenneth Laughery.

What's a story problem? When you were in school, your teacher told you a story that ended with a mathematical question. For example:

Dick had 5 apples. He ate 3. How many are left?

In that problem, the last word is: *left*. That means: subtract. So the correct answer is 5 minus 3, which is 2.

Can the computer solve problems like that? Here's the most famous attempt...

Arithmetic & algebra (1964) MIT awarded a Ph.D. to Daniel Bobrow for making the computer solve story problems involving arithmetic and algebra.

Let's see how the computer solves this problem:

If the number of customers Tom gets is twice the square of 20 percent of the number of advertisements he runs, and the number of advertisements he runs is 45, what is the number of customers Tom gets?

To begin, the computer replaces *twice* by *2 times*, and replaces *square of* by *square*.

Then the computer splits the sentence into shorter ones:

The number of customers Tom gets is 2 times the square 20 percent of the number of advertisements he runs. The number of advertisements he runs is 45. What is the number of customers Tom gets?

The computer turns each sentence into an equation:

number of customers Tom gets = 2 * (20 * number of advertisements he runs)²
 number of advertisements he runs = 45
 X = number of customers Tom gets

The computer solves the equations and prints the answer as a complete sentence:

The number of customers Tom gets is 162.

Here's a harder problem:

The sum of Lois's share of some money and Bob's share is \$4.50. Lois's share is twice Bob's. Find Bob's and Lois's share.

Applying the same method, the computer turns the problem into these equations:

Lois's share of some money + Bob's share = 4.50 dollars
 Lois's share = 2 * Bob's
 X = Bob's
 Y = Lois's share

The computer tries to solve the equations but fails. So it assumes "Lois's share" is the same as "Lois's share of some money", and "Bob's" is the same as "Bob's share". Now it has 6 equations:

Original equations

Lois's share of some money + Bob's share = 4.50 dollars

Lois's share = 2 * Bob's

X = Bob's

Y = Lois's share

Assumptions

Lois's share = Lois's share of some money

Bob's = Bob's share

It solves them and prints:

Bob's is 1.50 dollars.

Lois's share is 3 dollars.

The computer can solve problems about distance:

The distance from New York to Los Angeles is 3000 miles. If the average speed of a jet plane is 600 miles per hour, find the time it takes to travel from New York to Los Angeles by jet.

The resulting equations are:

distance from New York to Los Angeles = 3000 * miles

average speed of jet plane = (600 * miles)/(1 * hours)

X = time it takes to travel from New York to Los Angeles by jet

The computer is unable to solve them. But in its memory it finds the formula "distance = speed * time". It winds up with 7 equations:

Original equations

distance from New York to Los Angeles = 3000 * miles

average speed of jet plane = (600 * miles)/(1 * hours)

X = time it takes to travel from New York to Los Angeles by jet

Formula

distance = speed * time

Assumptions

distance = distance from New York to Los Angeles

speed = average speed of jet plane

time = time it takes to travel from New York to Los Angeles by jet

It solves them and prints:

The time it takes to travel from New York to Los Angeles by jet is 5 hours.

The computer can solve problems about age:

Mary is twice as old as Ann was when Mary was as old as Ann is now. If Mary is 24 years old, how old is Ann?

To begin, the computer replaces *twice* by *2 times*, and replaces *how old* by *what*. When the computer sees the words *as old as*, it knows the problem has to do with ages, so it breaks the problem into these sentences:

Mary's age is 2 times Ann's age X years ago. X years ago Mary's age is Ann's age now. Mary's age is 24. What is Ann's age?

The resulting equations are:

Mary's age = 2 * (Ann's age - X)

Mary's age - X = Ann's age

Mary's age = 24

Y = Ann's age

The computer prints:

Ann's age is 18.

If a situation appears hopeless, the computer asks you for help. In the following conversation, the human poses a problem, and the computer begs for a hint:

Human: The gross weight of a ship is 20000 tons. If its net weight is 15000 tons, what is the weight of the ship's cargo?

Comp.: Do you know any more relationships among these variables: gross weight of ship, tons, its net weight, weight of ship's cargo?

Human: Yes.

Comp.: Tell me.

Human: The weight of a ship's cargo is the difference between the gross weight and the net weight.

Comp.: The weight of the ship's cargo is 5000 tons.

The program has some limitations:

It cannot solve quadratic equations. If the computer tries to apply the formula "distance = speed * time" to a problem involving *two* distances, it forgets which distance is which.

The computer's vocabulary is limited. It doesn't realize that *how many* means *what is the number of*, and *how far* is means *what is the distance to*. For problems about age, the characters must be named Mary, Ann, Bill, Father, or Uncle, unless you diddle with the computer's memory.

If the human types *Tom has 2 apples, 3 bananas, and 4 pears*, the comma before the *and* makes the computer split the sentence into these two wrong "sentences"....

Tom has 2 apples, 3 bananas.

4 pears.

If the human mentions *the number of times John went to the movies*, the computer thinks *times* means multiplication, and tries to multiply *number of by John went to the movies*.

Encyclopedia (1964-1966)

In 1964, Simmons, Klein, and McConlogue (at the Systems Development Corporation) fed a child's encyclopedia into a computer.

If you type *What do worms eat?* the computer hunts through the encyclopedia, to find sentences mentioning both *worms* and *eat*. (To hasten the hunt, it uses an index produced by another program.) It finds two sentences:

Birds eat worms on the grass.

Most worms usually eat grass.

After analyzing the grammar of your question and those sentences, the computer realizes the first sentence is irrelevant, and prints just the correct answer:

Most worms usually eat grass.

In 1965, the program's abilities were extended, so that if you type *What animals live longer than men?* the computer searches through the encyclopedia for related words:

Word in sentence Related words searched

animals

mammals, reptiles, fish

live

age

longer

older, ancient

men

person, people, women

Here's an example of the program's operation in 1966. The human creates some encyclopedia material about horses, then retrieves it:

Human: Enter horse=noun, is=verb, animal=noun, live=verb, on=prep, farm=noun, strong=adj, pull=verb, wagon=noun, most=adj, are=verb, eat=verb, hay=noun, all=adj, red=adj, many=adj, have=haveverb, long=adj, mane=noun, some=adj, person=noun, race=verb.

Comp.: Entered in dictionary.

Human: Plurals horse=horses, animal=animals, farm=farms, wagon=wagons, mane=manes, person=people.

Comp.: Ok.

Human: Start.

Comp.: Ready.

Human: A horse is an animal. Horses live on a farm. Strong horses pull wagons. Most horses are strong. Horses eat hay. All horses are red. Many horses have long manes. Some people race horses.

Comp.: Learned.

Human: Forget "all horses are red".

Comp.: Forgotten.

Human: What is a horse?

Comp.: Horse is an animal.

Human: What do horses eat?

Comp.: Horses eat hay.

Human: What do horses do?

Comp.: Horses live on a farm and eat hay. Strong horses pull wagons.

Human: What do strong horses do?

Comp.: Strong horses pull wagons.

Human: Do many horses have long manes?
Comp.: Yes.

Human: Do all horses have long manes?
Comp.: To my knowledge, no.

Human: Describe horses?
Comp.: Horses are an animal. Most horses are strong. Many horses have long manes. Horses live on a farm and eat hay. Strong horses pull wagons. Some people race horses.

When the computer's answer consists of several sentences, it prints the most relevant sentence first, the others in decreasing order of relevance.

Military (1966) Thompson and Craig (at General Electric) made the computer answer military questions such as:

Has the 25th battalion arrived in Texas since 3 P.M.?
Is the 100th scheduled to arrive at Ft. Lewis before the 200th leaves Ft. Lewis?

Eliza (1966-1967) In 1966 at MIT, Joseph Weizenbaum wrote the Eliza program, which makes the computer act as a psychotherapist.

In 1967, he changed the program, to make it handle keywords such as *also, call, compute, define, is, let, of, thanks, where*, and the symbol =.

In the following conversation, the human teaches the computer how to find the area of a circle, a sphere, and a square, and then asks the computer questions about geometry. The human also teaches the computer how to speak German —

English German

I say	ich sage
I mean	meine ich
what is	wieviel ist
squared	quadrat

and gradually starts speaking to the computer in German.

Human: The area of a circle is pi times the radius of the circle squared. The area of a sphere is pi times the diameter of the sphere squared. The area of a square is the side of the square squared. The diameter of an object is twice the radius of the object. The radius of a ball is 10. What is the area of the ball?

Comp.: The assumption that ball is circle is consistent — shall I go on?

Human: No.

Comp.: The assumption that ball is sphere and object is consistent — shall I go on?

Human: Yes.

Comp.: It's 1256.6359.

Human: What is the diameter of the ball?

Comp.: It's 20.

Human: The radius of a dish is 10. What's the area of the dish?

Comp.: The assumption that dish is circle is consistent — shall I go on?

Human: Yes.

Comp.: It's 314.15899.

Human: The radius of a globe is 10. A globe is a sphere. A sphere is an object. What is the area of the globe?

Comp.: It's 1256.6359.

Human: The side of a plate is 10. What is the area of the plate?

Comp.: The assumption that plate is square is consistent — shall I go on?

Human: Yes.

Comp.: It's 100.

Human: When I say board I mean plate. What is 3 times the side of the board?

Comp.: It's 30.

Human: Please call that x. Let f(z) be z squared minus z. Compute f(x).

Comp.: It's 870.

Human: When I say ich sage I mean I say. When ich sage meine ich I mean I mean. Wenn ich sage wieviel ist meine ich what is. Wenn ich sage quadrat meine ich squared. Nun, sagen sie mir bitte, wieviel ist x quadrat.

Comp.: I didn't understand you.

Human: Ja, tut mir leid, ich sollte fragen wieviel IST x quadrat.

Comp.: It's 900.

Calculus (1969) At MIT, Charniak made the computer solve story problems involving calculus.

Human: Water is flowing into a conical filter at the rate of 15 cubic inches per second. If the radius of the base of the filter is 5 inches and the altitude is 10 inches, find the rate at which the water level is rising when the volume is 100 cubic inches.

Comp.: The answer is .53132943 * in * sec^-1 * pi^-.33333332.

Probability (1971) Rensselaer Polytechnic Institute awarded a Ph.D. to J.P. Gelb, for making the computer solve story problems involving probability.

Human: From a zorch containing 4 ferd and 3 brakky and 5 chartreuse werfels, 3 are drawn. What is the probability that 2 are chartreuse and the other brakky?

Comp.: Replacement involved?

Human: No.

Comp.: 3/22 (or .1363636).

Surveying the field

The field of "artificial intelligence" includes many categories.

For example, it includes attempts to make the computer win at chess and checkers, understand English, and create its own original art and music. It also includes attempts to imitate human feelings, personal interactions, and therapists. I explained those topics earlier.

Protocol method During the 1950's and 1960's, most research in artificial intelligence was done at the Massachusetts Institute of Technology (MIT) and the Carnegie Institute of Technology (CIT, now called Carnegie-Mellon University). At Carnegie, the big names were Allen Newell and Herbert Simon. They invented the **protocol method**. In the protocol method, a human is told to solve a tough problem and, while he's solving it, to say at each moment what he's thinking. A transcript of his train of thought is recorded and called the **protocol**. Then programmers try to make the computer imitate that train of thought.

Using the protocol method, Newell and Simon produced programs that could "think like humans". The thinking, like human thinking, was imperfect. Their research did *not* try to make the computer a perfect thinker; instead, it tried to gain insight into how *humans* think. Their point of view was: if you think you really understand human psychology, go try to program it. Their attempt to reduce human psychology to computer programs is called **mentalism** and has replaced Skinner's stimulus-response behaviorism as the dominant force in psychology now.

Abstract math Many programmers have tried to make the computer do abstract math.

In 1957 Newell, Simon, and Shaw used the protocol method to make the computer prove theorems about symbolic logic, such as "Not (p or q) implies not p". In 1959 and 1960, Herbert Gelernter and his friends made the computer prove theorems about Euclidean geometry, such as "If the segment joining the midpoints of the diagonals of a trapezoid is extended to intersect a side of the trapezoid, it bisects that side."

In 1961, MIT awarded a Ph.D. to James Slagle for making the computer compute indefinite integrals, such as:

$$\int \frac{x^4}{(1-x^2)^{5/2}} dx$$

The computer gets the answer, which is:

$$\arcsin x + \frac{\tan^3 \arcsin x}{3} - \tan \arcsin x + c$$

Each of those programs works by drawing a tree inside the computer's memory. Each branch of the tree represents a possible line of attack. The computer considers each branch and chooses the one that looks most promising.

A better symbolic-logic program was written by Hao Wang in 1960. His program doesn't need trees; it always picks the right attack immediately. It's guaranteed to prove any theorem you hand it, whereas the program by Newell, Simon, and Shaw got stuck on some hard ones.

A better indefinite integration program was written by Joel Moses in 1967 and further improved in 1969. It uses trees very rarely and solves almost any integration problem.

A program that usually finds the right answer but might fail on hard problems is called **heuristic**. A heuristic program usually involves trees. The checkers, chess, and geometry programs are heuristic. A program that's guaranteed to always give the correct answer is called **algorithmic**. The original symbolic-logic program was heuristic, but Wang's improvement is algorithmic; Moses's indefinite integration program is almost algorithmic.

GPS In 1957 Newell, Simon, and Shaw began writing a single program to solve *all* problems. They called it **General Problem Solver (GPS)**. If you feed that program a goal, a list of operators, and associated information, the program will tell you how to achieve the goal by using the operators.

For example, suppose you want the computer to solve this simple problem: a monkey would like to eat some bananas that are too high for him to reach, but there's a box nearby he can stand on. How can he get the bananas?

Feed the GPS program this information...

Now:	monkey's place = place#1; box's place = place#2; contents of monkey's hand = empty
Want:	contents of monkey's hand = the bananas
Difficulties:	contents of monkey's hand is harder to change than box's place, which is harder to change than monkey's place
Allowable operator	Definition
climb box	before: monkey's place = box's place after: monkey's place = on the box
walk to x	after: monkey's place = x
move box to x	before: monkey's place = box's place after: monkey's place = x; box's place = x
get bananas	before: box's place = under the bananas; monkey's place = on the box after: contents of monkey's hand = the bananas

GPS will print the solution:

walk to place#2
move box to under the bananas
climb box
get bananas

The GPS approach to solving problems is called **means-ends analysis**: you tell the program the means (operators) and the end (goal). The program has proved theorems in symbolic logic, computed indefinite integrals, and solved many famous puzzles, such as "The Missionaries and the Cannibals", "The Tower of Hanoi", and "The 5-Gallon Jug and the 8-Gallon Jug". But the program works slowly and must be fed lots of info about the problem. The project was abandoned in 1967.

Vision Another large topic in artificial intelligence is **computer vision**: making the computer see.

The first vision problem tackled was **pattern recognition**: making the computer read handwritten printed letters. The problem is hard, because some people make their letters very tall or wide or slanted or curled or close together, and the pen may skip. Reasonably successful programs were written, although computers still can't tackle script.

Interest later shifted to **picture processing**: given a photograph of an object, make the computer tell what the object is. The problem is hard, because the photo may be taken from an unusual angle and be blurred, and because the computer gets confused by shadows.

Scene analysis is even harder: given a picture of a group of objects, make the computer tell which object is which. The problem is hard, because some of the objects may be partly hidden behind others, and because a line can have two different interpretations: it can be a crease in one object, or a dividing-line between two objects.

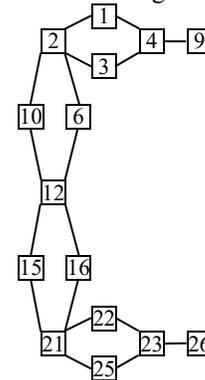
Most research in picture processing and scene analysis was done from 1968 to 1972.

Ray Kurzweil invented an amazing machine whose camera looks at a book and reads the book, by using a voice synthesizer. Many blind people have used it.

Robots Researchers have built robots. The first robots were just for experimental fun, but today's robots are truly useful: for example, robots build cars. Many young kids have been taught "LOGO", which is a language developed at the MIT Artificial Intelligence Lab that makes the computer control a robot turtle.

Today's research Now research in artificial intelligence is done at four major universities: MIT, Carnegie, Stanford, and Edinburgh (Scotland).

Reflexive control In the Soviet Union, weird researchers have studied **reflexive control**: they programmed the computer to be disobedient. The first such programmer was Lefevr, in 1967. In 1969 Baranov and Trudolyubov extended his work, by making the computer win this disobedience game:



The human begins by choosing either node 9 or node 26, *but doesn't tell the computer which node he's chosen*.

The computer starts at node 12; on each turn, it moves to an adjacent node. When it reaches either node 9 or node 26, the game ends: if the node the computer reaches is one of the human chose, the human wins; if the computer reaches the opposite node, the computer wins.

Before each move, the human tells the computer where to go; but the computer may decide to do the opposite (disobey).

What strategy should the computer use? If it always obeys or always disobeys, the human will catch on and make it lose.

Instead, Baranov and Trudolyubov programmed the computer to react as follows:

obey the human twice, then disobey three times, then obey once, disobey thrice, obey once, disobey twice, obey thrice, disobey once, obey thrice, disobey once,...

The irregular alternation of obedience and disobedience confuses the human in a way that works to the computer's advantage. Using that strategy, the computer played against 61 humans, and won against 44 of them (72%). In other words, the typical human tried to mislead the computer but in fact "clued it in" to the human's goal.

Later experiments with other games indicated that the following pattern of disobedience is usually more effective:

obey the human twice, disobey thrice, obey once, disobey four times, obey once, disobey thrice, obey thrice, disobey twice, obey thrice, disobey once, obey once, disobey once

Misinformation Unfortunately, most research in the field of artificial intelligence is just a lot of hot air. For years, researchers have been promising that intelligent, easy-to-use English-speaking computers and robots would be available at low prices “any day now”. After several decades of listening to such hoopla, I’ve given up waiting. The field of artificial intelligence should be renamed “artificial optimism”.

Whenever a researcher in the field of artificial intelligence promises you something, don’t believe it until you see it and use it personally, so you can evaluate its limitations.

If a computer seems to give intelligent replies to English questions posed by a salesman or researcher demonstrating artificial intelligence, try to interrupt the demo and ask the computer *your* English questions. You’ll typically find that the computer doesn’t understand what you’re talking about at all: the demo was a cheap trick that works just with the peculiar English questions asked by the demonstrator.

For many years, the top researchers in artificial intelligence have been exaggerating their achievements and underestimating how long it will take to develop a truly intelligent computer. Let’s look at their history of lies:

In 1957 Herbert Simon said, “Within ten years a digital computer will be the world’s chess champion.” In 1967, when the ten years had elapsed, the only decent chess program was Greenblatt’s, which the American Chess Federation rated “class D” (which means “poor”). A computer didn’t become the world chess champion until 1997. It took forty years, not ten!

In 1957 Simon also said, “Within ten years a digital computer will discover and prove an important new mathematical theorem.” He was wrong. The computer still hasn’t discovered or proved any important new mathematical theorem. The closest call came in 1976, when it did the *non-abstract part* of the proof of the “4-color theorem”.

In 1958 Newell, Simon, and Shaw wrote a chess-playing program which they admitted was “not fully debugged” so that one “cannot say very much about the behavior of the program”; but they claimed it was “good in spots (opening)”. In 1959 the founder of cybernetics, Norbert Wiener, exaggerated about their program; he told New York University’s Institute of Philosophy that “chess-playing machines as of now will counter the moves of a master player with the moves recognized as right in the textbooks, up to some point in the middle game.” In the same symposium Michael Scriven carried the exaggeration even further by saying, “Machines are already capable of a good game.” In fact, the program they were describing played very poorly, and in its last official bout (October 1960) was beaten by a 10-year-old kid who was a novice.

In 1960 Herbert Gelernter (who wrote the geometry-theorem program) said, “Today hardly an expert will contest the assertion that machines will be proving interesting theorems in number theory three years hence.” More than forty years have elapsed since then, but neither Gelernter nor anyone else has programmed the computer to prove theorems in number theory.

In June 1963 the *Chicago Tribune* said, “The development of a machine that can listen to any conversation and type out the remarks just like an office secretary was announced yesterday by a Cornell University expert on learning machines. The device is expected to be in operation by fall. Frank Rosenblatt, director of Cornell’s cognitive systems research, said the machine will be the largest thinking device built to date. Rosenblatt made his announcement at a meeting on learning machines at Northwestern University’s Technological Institute.” No such machine exists today, let alone in 1963.

Also in 1963, W. Ross Ashby said, “Gelernter’s theorem-proving program has discovered a new proof of the **pons asinorum** that demands no construction.” He said the proof is one that “the greatest mathematicians of 2000 years have failed to notice... which would have evoked the highest praise had it occurred.” In fact, the *pons asinorum* is just the simple theorem that the opposite angles of an isosceles triangle are equal, and the computer’s constructionless proof had already been discovered by Pappus in 300 A.D.

In 1968 the head of artificial intelligence in Great Britain, Donald Michie, said, “Today machines can play chess at championship level.” In fact, when computers were allowed to participate in human chess tournaments, they almost always lost.

In 1970 the head of artificial intelligence at MIT, Marvin Minsky, said, “In 3 to 8 years we will have a machine with the general intelligence of an average human being. I mean a machine that will be able to read Shakespeare, grease a car, play office politics, tell a joke, have a fight. At that point, the machine will begin to educate itself with fantastic speed. In a few months it will be at genius level, and a few months after that its powers will be incalculable.” His prediction that it would happen in 3 to 8 years — between 1973 and 1978 — was ridiculous. I doubt it will happen during this century, if ever.

Exaggerations concern not just the present and future but also the past:

Back in 1962, Arthur Samuel’s checker program won a game against Robert Nealey, “a former Connecticut checkers champion”.

Notice that Nealey was a *former* champion, not *the current* champion when the game was played. The program won a single game, not a match, and lost to Nealey later.

In 1971 James Slagle slid over those niceties, when he just said that the program “once beat the champion of Connecticut.” Later writers, reading Slagle’s words, went gone a step further and omitted the word *once*: one textbook said, “The program beat the champion of Connecticut”. It’s not true.

Why do leaders of artificial intelligence constantly exaggerate? To get more research funds from the government! Hubert Dreyfus, chairman of the philosophy department at Berkeley, annoys them by attacking their claims.

The brain Will the computer be able to imitate the human brain? Opinions vary.

Marvin Minsky, head of artificial intelligence at MIT, says *yes*: “After all, the human brain is just a computer that happens to be made out of meat.”

Biologists argue *no*: the brain is composed of 12 billion **neurons**, each of which has between 5,000 and 60,000 **dendrites** for input and a similar number of **axons** for output; the neurons act in peculiar ways, and no computer could imitate all that with complete accuracy — “The neuron is qualitatively quite different from on-off components of current computers.”

Herbert Simon (head of artificial intelligence at Carnegie and a psychologist), points out that certain aspects of the brain, such as short-term memory, are known to have very limited capacity and ability.

He believes the inner workings of the brain are reasonably simple; it produces complicated output just because it receives complicated input from the sense organs and environment:

“A man, viewed as a behaving system, is quite simple. The apparent complexity of his behavior over time is largely a reflection of the complexity of the environment in which he finds himself.”

Simon believes a computer would start acting in complex ways also, if it were given good sense organs, ability to move, elementary ability to learn, and the privilege of being placed in a stimulating environment (unlike a computer center’s dull four walls).

Hubert Dreyfus, chairman of the philosophy department at Berkeley, argues that progress in artificial intelligence has been very small, is being blocked now by impenetrable barriers, and — most important — the computer’s approach to solving problems bears little relationship to the more powerful methods used by humans. He’s cynical about the claim that an improvement in computer programs represents progress toward understanding the human mind, which is altogether different: “According to this definition, the first man to climb a tree could claim tangible progress toward reaching the moon. Rather than climbing blindly, it’s better to look where one is going.”

Visual Basic

The most popular computer language is **Visual Basic for Windows (VB)**. More programs are written in VB than in any other computer language.

Using VB, you can easily create Windows programs that let the human use a mouse to click on icons, choose from menus, use dialog boxes, etc.

After inventing the first VB, Microsoft invented improved versions:

VB 2, VB 3, VB 4, VB 5, VB 6
VB 7 (also called **VB.Net**)
VB 7.1 (also called **VB.Net 2003**)
VB 8 (also called **VB 2005**)
VB 9 (also called **VB 2008**)
VB 10 (also called **VB 2010**)
VB 11 (also called **VB 2012**)
VB 14 (also called **VB 2015**)

The most traumatic change was the switch from VB 6 to VB 7: programs written for VB 6 must be rewritten to work with VB 7.

This chapter explains the newest version: VB 2015.

Visual Basic is part of **Visual Studio**, which is Microsoft's suite of programming languages. Visual Studio includes **Visual Basic**, **Visual C++**, **Visual C#**, and other programming tools.

Microsoft lets you get Visual Studio **free!** The main free version is called **Visual Studio Community**. You can copy it from Microsoft's Website. It's free just if you promise to use it either *individually* (not in a big company's team) or *non-commercially* (just to study). Visual Studio Community improves on an older stripped-down version, called **Visual Studio Express**.

Before you read this chapter and study VB, prepare yourself! Do 2 prerequisite activities:

Learn BBC Basic for Windows (or QBasic or QB64), which are much easier than VB. I explained BBC Basic for Windows on pages 477-527. Read and practice that material.

Practice using good Windows programs (such as a Windows word-processing program), so you see how Windows programs should act. I explained good programs for modern Windows on pages 70-93, 104-122, and 444-459. Read and practice whichever of those Windows programs you have access to.

VB uses these **commands** (which resemble QBasic's):

VB command	Page
Beep()	598
Case "fine"	604
ColorDialog1.ShowDialog()	615
Console.ReadKey()	617
Console.WriteLine(5 + 2)	617
Console.Write(5 + 2)	617
Debug.Print(5 + 2)	617
Dim x	599
Dim x As Integer	626
Dim x As Integer = 7	630
Dim x = 7	630
Dim x() = {81, 52, 207, 19}	631
Dim x(2) As Double	630
Do	617, 621
document.Clear()	620
document.Copy()	621
document.Cut()	620
document.LoadFile...	590
document.Paste()	621
document.SaveFile...	619
Else	602
Elseif age < 100 Then	602
End	600, 604
End Class	595
End If	602
End Module	618
End Select	604
End Sub	595, 604
Exit Do	622
Exit Sub	604
For Each i In x	631
For x = 1 To 5	623
For x = 15 To 17 Step .1	624
GoTo joe	621
If age < 18 Then	602
Imports System.Math	625
Loop	617, 621
Loop Until guess = "pink"	622
Module Module1	617
MsgBox("Hair looks messy")	600
My.Computer..WriteAllText...	618
Next	623
OpenFileDialog1.ShowDialog()	619
Option Explicit Off	616
PrintForm1.Print()	595, 617
Private Sub Form1_Load...	595, 607
Public Class Form1	595
Randomize()	632
RichTextBox1.SaveFile...	618
Select Case feeling	604
SaveFileDialog1.ShowDialog()	619
Sub Main()	637
Text = 4 + 2	595
x = 47	599
? 5+2	617
'yeah, this is an example	616

VB uses these **functions** (which resemble QBasic's):

VB function	Value	Page
Chr(13)	Enter key	610
CInt(3.9)	4	631
ColorDialog1.Color	varies	615
Fix(3.89)	3	625
GetSelected(0)	varies	611
IIf(age < 18...	varies	603
InputBox("Name?")	varies	601
Int(3.89)	3	625
Math.Abs(-3.89)	3.89	625
Math.Ceiling(3.89)	4	625
Math.PI	about 3.14	625
Math.Round(3.89)	4	625

Math.Sign(3.89)	1	625
Math.Sqrt(9)	3.0	625
My...LocalTime	varies	615
My...MyDocuments	Docu. folder	618
My...ReadAllText	varies	608
MsgBox("Love me?"...	varies	603
Now	varies	638
Rnd	varies	633
TypeName(4.95D)	"Decimal"	630
Val("7")	7	601
VarType(4.95D)	14	629

In VB, you never write "a long program". Instead, you begin by drawing **objects** on the screen (as if you were using a graphics program). Then for each object, you write a little program (called a **subroutine**) that tells the computer how to manipulate the object. VB handles these objects:

VB object	Page
Button	607
CheckBox	608
ColorDialog	615
ComboBox	614
Form1	595
Form2	614
Label	610
ListBox	611
MenuStrip	619
NumericUpDown	612
OpenFileDialog	620
PictureBox	613
PrintForm	617
RadioButton	609
RichTextBox	612
SaveFileDialog	618
TextBox	612
Timer	614
ToolStrip	620
webBrowser	614

Each object has **properties**, which you can manipulate:

VB property	Object	Page
BackColor	Form1	598, 605
Checked	CheckBox	608
DecimalPlaces	NumericUpDown	613
Dock	RichTextBox	612
DropDownStyle	ComboBox	613
EnableAutoDrag...	RichTextBox	612
Enabled	Timer	614
FormBorderStyle	Form1	606
Image	PictureBox	613
Interval	Timer	614
MaximizeBox	Form1	606
Maximum	NumericUpDown	613
Minimum	NumericUpDown	613
MultiLine	TextBox	612
(Name)	RichTextBox1	618
Opacity	Form1	606
PasswordChar	TextBox	612
ScrollBars	TextBox	612
SelectedIndex	ListBox	611
SelectedItem	ListBox	611
SelectionMode	ListBox	611
Size	Form1	606
SizeMode	PictureBox	613
StartPosition	Form1	606
Text	Form1	595, 605
Url	WebBrowser	614
Value	NumericUpDown	612
Visible	Form2	614
windowState	Form1	598, 605

Fun

Let's have fun programming!

Copy the Community

Here's how to copy Visual Studio Community 2015 (including Visual Basic 2015) to your hard disk, using Windows 10. (Windows 7, 8, and 8.1 are also acceptable and act similarly.)

Using Microsoft Edge (or Internet Explorer), go to VisualStudio.com. Tap (or click) "Download Community 2015" then "No thanks" then "Save" then "Run".

The computer will say "Initializing setup" then "Choose". Tap the "Install" button then "Yes".

The computer will say "Acquiring" and "Applying". About 33 minutes later (depending on the speed of your computer and Internet connection), the computer will say "Setup Completed!"

Tap "Restart Now". The computer will say "Restarting". The screen will go black. Then computer will say "Please wait". The computer will ask you to log in again (by typing your password).

If the computer asks "How do you want to open this?" tap "OK".

The computer will say "Welcome to Visual Studio". Close the Microsoft Edge (or Internet Explorer) window (by clicking the X at the screen's top-right corner).

Start Visual Studio

To start using Visual Studio, type "vi" in the Windows 10 Search box (which is next to the Windows Start button) then click "Visual Studio 2015: Desktop app".

If you haven't used Visual Studio before, the computer says "Sign in". To reply, do this:

Click the "Sign in" button. Type your email address and press Enter. Type your Microsoft account's password and press Enter. The computer says "We're preparing for first use".

The computer says "Visual Studio." After a delay, you see the "Start Page" window.

Create simple programs

Click "New Project" (which is near the screen's left edge) then "Visual Basic" then "Windows Forms Application".

Double-click in the Name box (which is near the screen's bottom). Type a name for your project (such as Funmaker). At the end of your typing, press the Enter key.

You see an **object**, called the **Form1** window. Double-click in that window (below "Form1"). That tells the computer you want to write a program (subroutine) about that window.

The computer starts writing the **subroutine** for you. The computer writes:

```
Public Class Form1
    Private Sub Form1_Load...

    End Sub
End Class
```

The line saying "Private Sub Form1_Load" is the subroutine's **header**. The line saying "End Sub" is the subroutine's **footer**; it marks the end of the subroutine. Between those lines, insert lines that tell the computer what to do to the object (which is the Form1 window). The lines you insert are called the subroutine's **body**.

Simplest example Let's make the Form1 window show the answer to this math problem: $4 + 2$. To do that, type this line —

```
Text = 4 + 2
```

The computer automatically indents that line for you, so the subroutine becomes:

```
Private Sub Form1_Load...
    Text = 4 + 2
End Sub
```

To run your program, click "Start" (which is at the screen's top center) **or press the F5 key**. (If the "F5" is blue or tiny or on a new computer by Microsoft, HP, Lenovo, or Toshiba, that key works just *while you hold down the Fn key*, which is left of the Space bar.) Then you see the Form1 window again; but instead of saying "Form1", it says the text's answer:

```
6
```

When you've finished admiring that answer, stop the program by clicking the Form1 window's X button. Then you see the subroutine again:

```
Private Sub Form1_Load...
    Text = 4 + 2
End Sub
```

Edited example Let's edit that subroutine, so instead of saying the answer to $4 + 2$, it will say the answer to $79 + 2$.

To do that, change the 4 to 79. Here's how: click the 4's left edge, then press the Delete key (to delete the 4), then type 79, so the subroutine looks like this:

```
Private Sub Form1_Load...
    Text = 79 + 2
End Sub
```

Run that program by clicking "Start". Then the Form1 window shows the new answer:

```
81
```

When you finish admiring that, click Form1's X button.

To make the computer subtract 3 from 7, change the text line to this:

```
Text = 7 - 3
```

When you run the program (by clicking "Start"), the Form1 window will show the answer:

```
4
```

To make the computer do $-26.3 + 1$, change the text line to this:

```
Text = -26.3 + 1
```

The Form1 window will show the answer:

```
-25.3
```

Your own examples Go ahead! Try changing the subroutine, to do different math problems instead!

Multiply To multiply, use an asterisk. So to multiply 2 by 6, type this:

```
Text = 2 * 6
```

The Form1 window will show:

```
12
```

Divide To divide, use a slash. So to divide 8 by 4, type this:

```
Text = 8 / 4
```

The Form1 window will show:

```
2
```

To divide 2 by 3, type this:

```
Text = 2 / 3
```

The Form1 window will show:

```
0.6666...
```

Maximize the Form1 window (by clicking its maximize button, which is next to its X). That makes the Form1 window consume the whole screen temporarily and show:

```
0.6666666666666667
```

When you finish admiring that, return the Form1 window to its normal size (by clicking its Restore Down button, which is next to its X).

Congratulations You've written VB subroutines and created VB programs, so you've become a VB programmer! You can put on your resumé, "VB programmer!"

Type faster

Here are tricks that let you type faster.

You don't need to capitalize computer words such as "Text". The computer will capitalize them automatically, eventually. For example, if you type "text" instead of "Text", the computer will change "text" to "Text" when you type the equal sign afterwards.

You don't need to finish typing computer words such as "Text". The computer will finish typing them for you, automatically, eventually, if the computer can deduce what you meant. For example, you can type "te" instead of "Text"; the computer will change "te" to "Text" when you type the equal sign afterwards.

Instead of typing computer words, you can choose them from lists. For example, instead of typing “Text”, you can do this:

Type the letter “t”. You’ll see a list of computer words that begin with “t”. (To see that whole list, click the list’s up-arrow & down-arrow or press the keyboard’s up-arrow & down-arrow keys.) If you type “te”, you’ll see a list of computer words that begin with “te”. In a list, when you see the word you want, either double-click the word or do this: highlight the word (by clicking it) then press the keyboard’s Tab key.

You don’t need to put spaces around symbols, such as “=” and “+”. The computer will insert those spaces automatically, when you end the line (by clicking “Start” or pressing the Enter key or down-arrow key or clicking a different line).

Huge and tiny numbers

When dealing with huge and tiny number, be careful!

Avoid commas Do *not* put commas in big numbers. To write four million, do *not* write 4,000,000; instead, write 4000000.

Use decimals for big answers The computer sometimes has difficulty handling answers bigger than 2,000,000,000, which in modern English is called “2 billion.” To avoid difficulty, **put a decimal point in any problem whose answer might be bigger than 2 billion.**

For example, suppose you want the computer to multiply 3000 by 1000000. Since the answer to that problem is 3 billion, which is bigger than 2 billion, you should put a decimal point in that problem, like this:

```
Text = 3000 * 1000000.0
```

After typing a decimal point, you must type a digit (such as 0).

Suppose you forget to insert a decimal point and say just this:

```
Text = 3000 * 1000000
```

When you try to run the program (by clicking “Start”), the computer will complain in 3 ways:

It will put a squiggly red line under the “3000 * 1000000”.

The screen’s bottom will say “Constant expression not representable in type ‘Integer’.”

The screen’s middle will say, “There were build errors. Would you like to continue and run the last successful build?” To reply, click “No” then fix your error (by inserting .0) and click “Start” again.

E notation If the computer’s answer is huge (at least a quadrillion, which is 1000000000000000) or tiny (less than .0001), the computer will put an E in the answer. The E means “move the decimal point”.

For example, suppose the computer says the answer to a problem is:

```
1.586743E+15
```

The E means, “move the decimal point”. The plus sign means, “towards the right”. Altogether, **the E+15 means, “move the decimal point towards the right, 15 places.”** So look at 1.586743 and move the decimal point towards the right, 15 places; you get 1586743000000000.

So when the computer says the answer is 1.586743E+15, the computer really means the answer is 1586743000000000, approximately. The exact answer might be 1586743000000000.2 or 1586743000000000.79 or some similar number, but the computer prints just an approximation.

Suppose your computer says the answer to a problem is:

```
9.23E-06
```

After the E, the minus sign means, “towards the left”. So look at 9.23 and move the decimal point towards the left, 6 places. You get: .00000923

So when the computer says the answer is 9.23E-06, the computer really means the answer is:

```
.00000923
```

You’ll see E notation rarely: the computer uses it just if an answer is huge (at least a quadrillion) or tiny (tinier than .0001). But when the computer *does* use E notation, remember to move the decimal point!

The highest number The highest number the computer can handle well is about 1E308, which is 1 followed by 308 zeros. If you try to go much higher, the computer will gripe, by saying “Overflow” or “∞” (which is the symbol for infinity) or “NaN” (which means “Not a Number”).

For example, if you say —

```
Text = 1E309
```

the computer will put a squiggly red line under “1.0E+309”, screen’s bottom will say “Overflow”, and the screen’s middle will say “There were build errors” and wait for you to click “No”.

Dividing by 0 If you ask the computer to divide by 0, the computer will have difficulty.

For example, if you say —

```
Text = 5 / 0
```

the computer will try to divide 5 by 0, give up (because you can’t divide by 0), and say the answer is “∞”.

If you say —

```
Text = -5 / 0
```

the computer will try to divide -5 by 0, give up (because you can’t divide by 0), and say the answer is “-∞”.

If you say —

```
Text = 0 / 0
```

the computer will try to divide 0 by 0, give up (because you can’t divide by 0), get confused, and say the answer is “NaN” (which means “Not a Number”).

The tiniest decimal The tiniest decimal the computer can handle accurately is 1E-308 (which is a decimal point followed by 308 digits, 307 of which are zeros). If you try to go tinier, the computer will either say 0 or give you a rough approximation.

Order of operations

What does “2 plus 3 times 4” mean? The answer depends on whom you ask.

To a clerk, it means “start with 2 plus 3, then multiply by 4”; that makes 5 times 4, which is 20. But to a scientist, “2 plus 3 times 4” means something different: it means “2 plus three fours”, which is 2 + 4 + 4 + 4, which is 14.

Since computers were invented by scientists, computers think like scientists. If you type —

```
Text = 2 + 3 * 4
```

the computer will think you mean “2 plus three fours”, so it will do 2 + 4 + 4 + 4 and display this answer:

```
14
```

The computer will *not* display the clerk’s answer, which is 20. So if you’re a clerk, tough luck!

Scientists and computers follow this rule: **do multiplication and division before addition and subtraction.** So if you type —

```
Text = 2 + 3 * 4
```

the computer begins by hunting for multiplication and division. When it finds the multiplication sign between the 3 and the 4, it multiplies 3 by 4 and gets 12, like this:

```
Text = 2 + 3 * 4
           |
           v
          12
```

So the problem becomes 2 + 12, which is 14, which the computer will display.

For another example, suppose you type:

```
Text = 10 - 2 * 3 + 72 / 9 * 5
```

The computer begins by doing all the multiplications and divisions. So it does $2 * 3$ (which is 6) and does $72 / 9 * 5$ (which is $8 * 5$, which is 40), like this:

```
Text = 10 - 2 * 3 + 72 / 9 * 5
           6         40
```

So the problem becomes $10 - 6 + 40$, which is 44, which is the answer the computer will display.

Parentheses You can use parentheses the same way as in algebra. For example, if you type —

```
Text = 5 - (1 + 1)
```

the computer will compute $5 - 2$ and print:

```
3
```

You can put parentheses inside parentheses. If you type —

```
Text = 10 - (5 - (1 + 1))
```

the computer will compute $10 - (5 - 2)$, which is $10 - 3$, and will display:

```
7
```

Strings

Let's make the computer fall in love. Let's make it say, "I love you". To do so, type this in your subroutine:

```
Text = "I love you"
```

Type that carefully:

Type the word `Text`, then a blank space, then an equal sign, then another blank space. Then type a quotation mark, but be careful: **to type the quotation mark, you must hold down the Shift key**. Then type these words: *I love you*. Then type another quotation mark.

When you run the program (by clicking "Start"), it will make the text (at the top of Form1's window) display:

```
I love you
```

You can change the computer's personality. For example, if you edit the subroutine to make it become —

```
Text = "I hate you"
```

the computer will reply:

```
I hate you
```

Notice that **to make a subroutine print a message, you must put the message between quotation marks**. The quotation marks make the computer copy the message without worrying about what the message means. For example, if you misspell "I love you", and type —

```
Text = "aieee luf ya"
```

the computer will still copy the message (without worrying about what it means); the computer will make Form1 say:

```
aieee luf ya
```

Type faster Instead of typing —

```
Text = "I love you"
```

you can type just this:

```
text="I love you
```

The computer automatically capitalizes the first letter (T) and types the second quotation mark. Before the computer runs the program, the computer will also put spaces around the equal sign.

Red strings While you're typing the subroutine, **the computer makes each string look red** (and each computer word look blue).

Those colors appear just while you're looking at the subroutine you've been typing. When you run the program, the program's answers (and other results) appear black.

Jargon The word "joy" consists of 3 characters: j and o and y. Programmers say that the word "joy" is a **string** of 3 characters.

A **string** is any collection of characters, such as "joy" or "I love you" or "aieee luf ya" or "76 trombones" or "GO AWAY!!!" or "xypw exr//746". The computer will print whatever string you wish, but in your subroutine **put the string in quotation marks**.

Strings versus numbers The computer can handle two types of expressions: **strings** and **numbers**. In your subroutine, put strings (such as "joy" and "I love you") in quotation marks. Numbers (such as $4 + 2$) do *not* go in quotation marks.

Combine strings You can combine strings:

```
Text = "fat" & "her"
```

The computer will combine "fat" with "her", so the computer will display:

```
father
```

You can combine a string with a number:

```
Text = "The lucky number is " & 4 + 2
```

The computer will display "The lucky number is " then the answer to this math problem: $4 + 2$. The computer will display:

```
The lucky number is 6
```

When combining a string with a number, make the computer leave a space between the string and the number, by putting a space before the last quotation mark.

Combining strings or numbers (by using the symbol "&") is called **concatenating**.

When typing the symbol "&" to concatenate, **press the keyboard's Space bar before and after the "&"**. If you rely on the computer to put those spaces in automatically, you'll be sorry, because the symbol "&" without spaces can have a different meaning, and the computer will occasionally guess wrong about which "&" you meant.

Accidents Suppose you accidentally put the number $2 + 2$ in quotation marks, like this:

```
Text = "2 + 2"
```

The quotation marks make the computer think "2 + 2" is a string instead of a number. Since the computer thinks "2 + 2" is a string, it copies the string without analyzing what it means; Form1 will say:

```
2 + 2
```

It will *not* say 4.

Suppose you want the computer to show the word "love" but you accidentally forget to put the string "love" in quotation marks, and type this instead:

```
Text = love
```

Since you forgot the quotation marks, the computer is confused. Whenever the computer is confused, it either gripes at you or says zero. In this particular example, here's how the computer gripes at you: it puts a red squiggly line under "love," and, when you run the program, it makes the screen's middle say "There were build errors" and makes screen's bottom say:

```
'love' is not declared.
```

Display a quotation mark The symbol for *inches* is `"`.

Let's make Form1 say:

```
The nail is 2" long.
```

This Text command does *not* work:

```
Text = "The nail is 2" long."
```

When the computer sees the quotation mark after 2, it mistakenly thinks that quotation mark is paired with the quotation mark before "The", then gets totally confused.

Here's the correct way to write that line:

```
Text = "The nail is 2"" long."
```

The symbol `""` means: display a quotation mark. That Text line makes Form1 display:

```
The nail is 2" long.
```

Here's the rule: to display a quotation mark (`"`), put the symbol `""` in your Text statement.

Let's make the computer display this sentence:

```
I saw "Hamlet" last night.
```

To display the quotation mark before "Hamlet", you must type `""`. To display the quotation mark after "Hamlet", you must type `""`. So type this:

```
Text = "I saw ""Hamlet"" last night."
```

Color

Normally, the Form1 window's middle is a big blank area that's nearly white (very light gray). To make it red instead, put this line in your subroutine:

```
BackColor = Color.Red
```

For example, to make the window's title say "I love you" and make the window's background color be red, put both of these lines in your subroutine —

```
Text = "I love you"  
BackColor = Color.Red
```

so the whole subroutine looks like this:

```
Public Class Form1  
    Private Sub Form1_Load..  
        Text = "I love you"  
        BackColor = Color.Red  
    End Sub  
End Class
```

The computer understands these color names:

```
Yellow, Gold, Goldenrod, LemonChiffon  
Orange, Brown, Chocolate, Tan  
Red, Pink, DeepPink, Crimson  
Purple, Violet, Magenta, Orchid  
Blue, Cyan, Navy, DeepSkyBlue  
Green, Lime, Chartreuse, Khaki  
White, Gray, Black, Silver
```

It understands many others, too: altogether, it knows the names of 147 colors. You'll see the complete list when you've typed:

```
BackColor = Color.
```

(Use the list's up-arrow & down-arrow.)

Don't put spaces in the middle of a color name: type "DeepSkyBlue", not "Deep Sky Blue".

Beep

To make the computer **beep** (play 3 musical notes through the computer's speakers), put this line in your subroutine:

```
Beep()
```

Multi-statement line

In your subroutine, a line can include many statements separated by colons, like this:

```
Text = "I love you" : BackColor = Color.Red
```

That line means the same thing as:

```
Text = "I love you"  
BackColor = Color.Red
```

Maximize

To maximize the Form1 window (so it consumes the whole screen), put this line in your subroutine:

```
WindowState = 2
```

For example, let's make the computer say "You turned me on, and I love you!" and maximize the Form1 window (so the human can see all that). Just put both of these lines in your subroutine:

```
WindowState = 2  
Text = "You turned me on, and I love you!"
```

Final steps

When you finish playing with your program, here's what to do. Make sure you see the subroutine you typed. (If you see Form1's window instead, close that window by clicking its X button.)

Save If you like the program you created and want to save it on disk, click the **Save All** button. (It looks like 2 floppy disks that are *tiny*. It's near the screen's top, below the "t" in "Project".) That makes sure your program is saved. (It's saved in its own folder, which is in the **Projects folder**, which is in the **Visual Studio 2015 folder**, which is in the **Documents folder**, which is on drive C.)

Afterwards, if you make further changes to the program, click the Save All button again to save them.

New If you're tired of working on a program and want to start inventing a different program instead, click the **New Project** button (which is near the screen's top-left corner, below the word "View", and shows a yellow sunburst before 2 sheets of paper). Then click "Windows Forms Application", double-click in the Name box, type a name for the new program, and press the Enter key.

If the previous program wasn't saved, the computer says "Save changes". If you want to save the previous program, click "Yes"; otherwise, click "No".

Exit When you finish using Visual Studio, click the X button that's in the screen's top right corner.

Open When you start using Visual Studio, look at the the word "Recent" (near the screen's left edge). Below "Recent", you see a list of programs you used recently. If you want to use or edit one of those programs, click that program's name. Either run the program again (by clicking "Start") or edit the program's commands.

Run the .exe file When the computer ran your program, it made an .exe file (called Funmaker.exe), which you can run again without going into Visual Studio. Here's how:

Exit from Visual Studio. In the Windows 10 search box (which is next to the Windows Start button), type "Funmaker". Tap "Funmaker: File Folder in Projects". Double-click the "Funmaker" folder then "bin" then "Debug" then the first "Funmaker" (which says its type is "Application").

Variables

A letter can stand for a number, a string, or other things.

For example, x can stand for the number 47, as in this subroutine:

```
Dim x
x = 47
Text = x + 2
```

The top line (Dim x) warns the computer that x will stand for something. (The “Dim” comes from the word “Dimension”.)

The second line (x = 47) says x stands for the number 47. In other words, x is a name for the number 47. Warning: if you forgot to type the top line (Dim x), the computer refuses to let you type the second line (x = 47).

The bottom line (Text = x + 2) makes the computer display x + 2. Since x is 47, the x + 2 is 49; so the computer will display 49. That’s the only number the computer will display; it will not display 47.

Jargon

A letter standing for something is called a **variable** (or **name** or **identifier**). A letter standing for a number is called a **numeric variable**. In that subroutine, x is a numeric variable; it stands for the number 47. The **value** of x is 47.

In that subroutine, the statement “x = 47” is called an **assignment statement**, because it **assigns** 47 to x.

A variable is a box

When you run that subroutine, here’s what happens inside the computer.

The computer’s random-access memory (RAM) consists of electronic boxes. When the computer encounters the line “x = 47”, the computer puts 47 into box x, like this:

box x

Then when the computer encounters the line “Text = x + 2”, the computer will display what’s in box x, plus 2; so the computer will display 49.

Faster typing

Instead of typing —

```
x = 47
```

you can type just this:

```
x=47
```

At the end of that line, when you press the Enter key, the computer automatically puts spaces around the equal sign.

More examples

Here’s another subroutine:

```
Dim y
y = 38
Text = y - 2
```

The top line says y is a variable. The next line says y is 38. The bottom line says to display y - 2. Since y is 38, the y - 2 is 36; so the computer will display 36.

Another example:

```
Dim b
b = 8
Text = b * 3
```

The top line says b is a variable. The next line says b is 8. The bottom line says to display b * 3, which is 8 * 3, which is 24; so the computer will display 24.

One variable can define another:

```
Dim n, d
n = 6
d = n + 1
Text = n * d
```

The top line says n and d are variables. The next line says n is 6. The next line says d is n + 1, which is 6 + 1, which is 7; so d is 7. The bottom line says to display n * d, which is 6 * 7, which is 42; so the computer will display 42.

Changing a value

A value can change:

```
Dim k
k = 4
k = 9
Text = k * 2
```

The second line says k is 4, but the next line changes k’s value to 9, so the bottom line displays 18.

When you run that subroutine, here’s what happens inside the computer’s RAM. The second line (k = 4) makes the computer put 4 into box k:

box k

The next line (k = 9) puts 9 into box k. The 9 replaces the 4:

box k

That’s why the bottom line (Text = k * 2) displays 18.

String variables

A string is any collection of characters, such as “I love you”. Each string must be in quotation marks.

A letter can stand for a string:

```
Dim x
x = "I love you"
Text = x
```

The top line warns the computer that x will stand for something. The next line says x stands for the string “I love you”. The bottom line makes the computer display:

```
I love you
```

In that subroutine, x is a variable. Since it stands for a string, it’s called a **string variable**.

You can combine strings:

```
Dim x
x = "so"
Text = x & "up"
```

(When typing that example, you must leave a space before the ampersand, to avoid confusion.) Since the second line says x is “so”, the bottom line will make Text be “so” & “up” and display this:

```
soup
```

If you insert a space by typing “ up” instead of “up”, like this —

```
Dim x
x = "so"
Text = x & " up"
```

the computer will display:

```
so up
```

Long variable names

A variable’s name can be a letter (such as x) or a longer combination of characters, such as:

```
CityPopulationIn2001
```

For example, you can type:

```
Dim CityPopulationIn2001
CityPopulationIn2001 = 30716
Text = CityPopulationIn2001 + 42
```

The computer will print:

```
30758
```

The variable’s name can be as long as you wish: up to 255 characters! The name’s first character must be a letter; the remaining characters can be letters or digits. The computer ignores capitalization: it assumes that CityPopulationIn2001 is the same as citypopulationin2001.

Beginners are usually too lazy to type long variable names, so beginners use variable names that are short. But when you become a pro and write a long, fancy program containing hundreds of lines and hundreds of variables, you should use long variable names to help you remember each variable’s purpose.

In this book, I’ll use short variable names in short programs (so you can type those programs quickly) but long variable names in long programs (so you can keep track of which variable is which).

Pop-up boxes

Here's how to make a box appear suddenly on your screen.

Message box

Into any subroutine, you can insert this line:

```
MsgBox("warning: your hair looks messy today")
```

When the computer runs the program and encounters that line, the computer suddenly creates a **message box** (a window containing a short message), which appears in front of all other windows (so they're covered up) and contains this message: "Warning: your hair looks messy today". The computer automatically makes the window be wide enough to include the whole message and be centered on the screen.

The window includes an OK button. When the human finishes reading the message, the human must click that OK button (or press Enter) to make the window go away.

After the window goes away, Form1 reappears and the computer continues running the rest of the program (including any lines below the MsgBox line). Form1 remains on the screen until the human clicks Form1's X button, which closes the form and ends the program.

End program automatically To please the human, make the *computer* click Form1's X button and end the program, by putting this command under the MsgBox line —

```
End
```

so your subroutine looks like this:

```
MsgBox("warning: your hair looks messy today")
End
```

The top line makes the message box say "Warning: your hair looks messy today" then wait for the human to click OK. The bottom line ends the program (without requiring the human to click an X button).

Putting End under MsgBox makes the screen look like this:

```
Public Class Form1
  Private Sub Form1_Load...
    MsgBox("warning: your hair looks messy today")
  End
End Sub
End Class
```

Try it!

That End line is helpful. If you omit it and say just —

```
MsgBox("warning: your hair looks messy today")
```

here's what happens when the human runs the program:

The computer creates a message box saying "Warning: your hair looks messy today". Then the computer waits for the human to click the message box's OK button.

When the human clicks the OK button, the message box disappears. Then the computer is supposed to do any remaining lines in the subroutine. But there are no lines remaining to be done. So the computer just waits for the human to close the program by clicking its X button.

What if the human is too stupid to know to click the X button? Instead of clicking the X button, what if the human just keeps waiting to see whether the computer will do something? The situation is stupid: the computer waits for the human to click the X button, while the human waits for the computer to say what to do next.

To end such confusion, say End below the MsgBox line. The End line makes the computer stop running the program and automatically click the X button.

Faster typing If you type just —

```
ms C
```

the computer will automatically change it to:

```
MsgBox C
```

Add an icon To make the message box fancier, say **vbExclamation**, like this:

```
MsgBox("warning: your hair looks messy today", vbExclamation)
End
```

That makes the message box window include an **exclamation icon** (an exclamation point in a yellow triangle).

You can choose from 4 icons:

Icon	Command
! (in a yellow triangle)	vbExclamation
X (in a red circle)	vbCritical
i (in a blue circle)	vbInformation
? (in a blue circle)	vbQuestion

Math A message box can do math. For example, if you write a subroutine that says —

```
MsgBox(4 + 2)
```

and then run the program (by clicking "Start"), the computer will create a message box that displays the answer, 6.

Input box

For a wild experience, type this subroutine:

```
Dim x
x = InputBox("what is your name?")
Text = "I love " & x
```

Run the program (by clicking "Start"). Here's what happens...

The InputBox line makes the computer suddenly create an **input box**, which is a window letting the human type info into the computer. That window appears in front of all other windows (so they're covered up) and is centered on the screen. It contains this **prompt**: "What is your name?" It also contains a white box (into which the human can type a response) and an OK button.

The computer waits for the human to type a response. When the human finishes typing a response, the human must click the OK button (or press Enter) to make the window go away.

Then Form1 reappears, and the computer makes x be whatever the human typed. For example, if the human typed —

```
Sue
```

x will be Joan. Then the Text line will make Form1 try to say:

```
I love Joan
```

To let the subroutine handle names that are long, **maximize the Form1 window**, by inserting this line —

```
WindowState = 2
```

so the subroutine becomes:

```
Dim x
x = InputBox("what is your name?")
WindowState = 2
Text = "I love " & x
```

Numeric input To input a string, you've learned to say InputBox. **To input a number, say InputBox but also say Val**, to emphasize that you want the computer to produce a numeric value.

For example, this subroutine asks for your two favorite numbers and says their sum:

```
Dim x, y
x = Val(InputBox("what is the first number?"))
y = Val(InputBox("what is the second number?"))
Text = x + y
```

When you run the program (by clicking "Start"), the computer asks "What is the first number?", waits for you to type it, and calls it x. Then the computer asks "What is the second number?", waits for you to type it, and calls it y. Then the computer says the sum of the numbers. For example, if the first number was 7 and the second number was 2, the computer will display the sum:

```
9
```

In that program, if you accidentally omit each Val, the computer will think x and y are strings instead of numbers, so the computer will add the string "7" to the string "2" and display this longer string:

```
72
```

Predict your future This subroutine makes the computer predict your future:

```
Dim y
y = Val(InputBox("In what year were you born?"))
WindowState = 2
Text = "In the year 2030, you'll turn " & 2030 - y & " years old."
```

When you run the program, the computer asks, "In what year were you born?" If you answer —

```
1962
```

y will be the numeric value 1962, and the computer will correctly print:

```
In the year 2030, you'll turn 68 years old.
```

Prices Suppose you're selling tickets to a play. Each ticket costs \$2.79. (You decided \$2.79 would be a nifty price, because the cast has 279 people.) These lines find the price of multiple tickets:

```
Dim t
t = Val(InputBox("How many tickets?"))
WindowState = 2
Text = "The total price is $" & t * 2.79
```

Conversion These lines convert feet to inches:

```
Dim f
f = Val(InputBox("How many feet?"))
WindowState = 2
Text = f & " feet = " & f * 12 & " inches"
```

When you run the program, the computer asks "How many feet?" If you answer —

```
3
```

the computer will say:

```
3 feet = 36 inches
```

Trying to convert to the metric system? These lines convert inches to centimeters:

```
Dim i
i = Val(InputBox("How many inches?"))
WindowState = 2
Text = i & " inches = " & i * 2.54 & " centimeters"
```

Nice day today, isn't it? These lines convert the temperature from Celsius to Fahrenheit:

```
Dim c
c = Val(InputBox("How many degrees Celsius?"))
WindowState = 2
Text = c & " degrees Celsius = " & c * 1.8 + 32 & " degrees Fahrenheit"
```

When you run the program, the computer asks "How many degrees Celsius?" If you answer —

```
20
```

the computer will say:

```
20 degrees Celsius = 68 degrees Fahrenheit
```

See, you can write the *Guide* yourself! Just hunt through any old math or science book, find any old formula (such as $f = c * 1.8 + 32$), and turn it into a program.

Control commands

A subroutine is a list of commands you want the computer to obey. Here's how to control which commands the computer obeys, and when, and in what order.

If

This subroutine makes the computer discuss the human's age:

```
Dim age
age = Val(InputBox("How old are you?"))
MsgBox("I hope you enjoy being " & age)
End
```

When that program is run (by clicking "Start"), the computer asks "How old are you?" and waits for the human's reply. For example, if the human says —

15

the age will be 15. Then the computer will say:

I hope you enjoy being 15

After the human reads that message in the message box, the human should get out of the message box (by clicking the message box's "OK" or pressing the Enter key). Then the computer will automatically close Form1.

Let's make that subroutine fancier, so if the human is under 18 the computer will also say "You are still a minor". To do that, just add a line saying —

```
If age < 18 Then MsgBox("You are still a minor")
```

so the subroutine looks like this:

```
Dim age
age = Val(InputBox("How old are you?"))
MsgBox("I hope you enjoy being " & age)
If age < 18 Then MsgBox("You are still a minor")
End
```

For example, if the human runs the program and says —

15

the computer will say —

I hope you enjoy being 15

and then say:

You are still a minor

(At the end of each sentence, the computer waits for the human to click the message box's OK.)

If instead the human says —

25

the computer will say just:

I hope you enjoy being 25

In that program, the most important line is:

```
If age < 18 Then MsgBox("You are still a minor")
```

That line contains the words If and Then. **Whenever you say "If", you must also say "Then"**. Don't put a comma before "Then". What comes between "If" and "Then" is called the **condition**; in that example, the condition is "age < 18". If the condition is true (if the age is really less than 18), the computer does the **action**, which comes after the word "Then" and is:

```
MsgBox("You are still a minor")
```

Else Let's teach the computer how to respond to adults.

Here's how to program the computer so that if the age is less than 18, the computer will say "You are still a minor", but if the age is *not* less than 18 the computer will say "You are an adult" instead:

```
Dim age
age = Val(InputBox("How old are you?"))
MsgBox("I hope you enjoy being " & age)
If age < 18 Then MsgBox("You are still a minor") Else MsgBox("You are an adult")
End
```

In programs, **the word "Else" means "otherwise"**. That program's If line means: if the age is less than 18, then print "You are still a minor"; otherwise (if the age is *not* less than 18), print "You are an adult". So the computer will print "You are still a minor" or else print "You are an adult", depending on whether the age is less than 18.

Try running that program! If you say you're 50 years old, the computer will reply by saying —

I hope you enjoy being 50

and then (after you click "OK"):

You are an adult

Multi-line If If the age is less than 18, here's how to make the computer say "You are still a minor" and also say "Ah, the joys of youth":

```
If age < 18 Then MsgBox("You are still a minor") : MsgBox("Ah, the joys of youth")
```

Here's a more sophisticated way to say the same thing:

```
If age < 18 Then
    MsgBox("You are still a minor")
    MsgBox("Ah, the joys of youth")
End If
```

That sophisticated way (in which you type 4 short lines instead of a single long line) is called a **multi-line If** (or a **block If**).

In a multi-line If:

The top line says If and Then (with nothing after Then). The computer will type the word "Then" for you, if you forget to type it yourself.

The computer indents the middle lines for you. They're called the **block** and typically say MsgBox.

The bottom line says End If. The computer automatically types it for you.

In the middle of a multi-line If, you can say Else:

```
If age < 18 Then
    MsgBox("You are still a minor")
    MsgBox("Ah, the joys of youth")
Else
    MsgBox("You are an adult")
    MsgBox("We can have adult fun")
End If
```

That means: if the age is less than 18, then say "You are still a minor" and "Ah, the joys of youth"; otherwise (if age *not* under 18) say "You are an adult" and "We can have adult fun". The computer automatically unindents the word "Else".

Elselif Let's make the computer do this:

```
If age is under 18, say "You're a minor".
If age is not under 18 but is under 100, say "You're a typical adult".
If age is not under 100 but is under 125, say "You're a centenarian".
If age is not under 125, say "You're a liar".
```

Here's how:

```
If age < 18 Then
    MsgBox("You're a minor")
ElseIf age < 100 Then
    MsgBox("You're a typical adult")
ElseIf age < 125 Then
    MsgBox("You're a centenarian")
Else
    MsgBox("You're a liar")
End If
```

Different relations You can make the If clause very fancy:

IF clause	Meaning
If age = 18	If age is 18
If age < 18	If age is less than 18
If age > 18	If age is greater than 18
If age <= 18	If age is less than or equal to 18
If age >= 18	If age is at least 18 (greater than or equal to 18)
If age <> 18	If age is not 18
If sex = "male"	If sex is "male"
If sex < "male"	If sex is a word (such as "female") that comes before "male" in the dictionary
If sex > "male"	If sex is a word (such as "neuter") that comes after "male" in the dictionary

In the If statement, the symbols =, <, >, <=, >=, and <> are called **relations**.

When writing a relation, mathematicians and computerists habitually **put the equal sign last**:

Right	Wrong
<=	=<
>=	=>

When you press the Enter key at the end of the line, the computer will automatically put your equal signs last: the computer will turn any "<=" into "<=<"; it will turn any ">=" into ">=>".

To say "not equal to", say "less than or greater than", like this: <>.

Or The computer understands the word Or. For example, here's how to type, "If age is either 7 or 8, say the word *wonderful*":

```
If age = 7 Or age = 8 Then MsgBox("wonderful")
```

That example is composed of two conditions: the first condition is "x = 7"; the second condition is "x = 8". Those two conditions combine, to form "x = 7 Or x = 8", which is called a **compound condition**.

If you use the word Or, put it between two conditions.

Right: If age = 7 Or age = 8 Then MsgBox("wonderful")
(because "age = 7" and "age = 8" are conditions)

Wrong: If age = 7 Or 8 Then MsgBox("wonderful")
(because "8" is not a condition)

And The computer understands the word And. Here's how to type, "If age is more than 5 and less than 10, say *you get hamburgers for lunch*":

```
If age > 5 And age < 10 Then MsgBox("you get hamburgers for lunch")
```

Here's how to type, "If score is at least 60 and less than 65, say *you almost failed*":

```
If score >= 60 And score < 65 Then MsgBox("you almost failed")
```

Here's how to type, "If n is a number from 1 to 10, say *that's good*":

```
If n >= 1 And n <= 10 Then MsgBox("that's good")
```

Immediate If Here's a shortcut. Instead of saying —

```
If age < 18 then Text = "Minor" else Text = "Adult"
```

you can say:

```
Text = IIf(age < 18, "Minor", "Adult")
```

That line means:

```
Text is this: if age < 18 then "Minor" else "Adult"
```

That line is used in this subroutine:

```
Dim age  
age = InputBox("How old are you?")  
Text = IIf(age < 18, "Minor", "Adult")
```

The abbreviation **IIf** means "Immediate If". It lets you do an If immediately, without have to type the words "Then" and "Else".

Yes/no message box

Let's make the computer ask, "Do you love me?" If the human says "Yes", let's make the computer say "I love you too!" If the human says "No", let's make the computer say "I don't love you either!"

This subroutine accomplishes that goal:

```
Dim response  
response = InputBox("Do you love me?")  
If response = "yes" Then  
    MsgBox("I love you too!")  
Else  
    MsgBox("I don't love you either!")  
End If  
End
```

But that subroutine has a flaw: what if the human types neither "yes" nor "no"? Instead of typing "yes", what if the human types "YES" or "Yes" or "yeah" or "yep" or "yessiree" or just "y" or "certainly" or "I love you tremendously" or "not sure"? In those situations, since the human didn't type simply "yes", the computer will say "I don't love you either!", which is inappropriate.

The problem with that subroutine is it gives the human too many choices: it lets the human type *anything* in the input box.

To make sure the computer reacts appropriately to the human, give the human fewer choices. Restrict the human to choosing just Yes or No. Here's how: show the human a Yes button and a No button, then force the human to click one of them. This subroutine accomplishes that:

```
If MsgBox("Do you love me?", vbYesNo) = vbYes Then  
    MsgBox("I love you too!")  
Else  
    MsgBox("I don't love you either!")  
End If  
End
```

The MsgBox line makes the computer create a message box saying "Do you love me?" A normal message box contains an OK button, but vbYesNo makes this be a **yes/no message box** instead (which contains Yes and No buttons instead of an OK button).

If the human clicks the Yes button, the subroutine makes the computer say "I love you too!" If the human does otherwise (by clicking the No button), the computer says "I don't love you either!"

Long programs While running a long program, the computer should occasionally ask whether the human wants to continue. To make the computer ask that, insert this line:

```
If MsgBox("Do you want to continue?", vbYesNo) = vbNo Then End
```

That line creates a yes/no message box asking "Do you want to continue?" If the human clicks the No button, the program will end (and the computer will automatically click the program's X button).

Select

Let's turn your computer into a therapist!

To do that, make the computer ask the patient "How are you?" and let the patient type whatever words the patient wishes. Just begin the subroutine like this:

```
Dim feeling
feeling = InputBox("How are you?")
```

That makes the computer ask "How are you?" and makes the patient's response be called the feeling.

Make the computer continue the conversation as follows:

```
If the patient said "fine", print "That's good!"
If the patient said "lousy" instead, print "Too bad!"
If the patient said anything else instead, print "I feel the same way!"
```

To accomplish all that, you can use a multi-line If:

```
If feeling = "fine" Then
    MsgBox("That's good!")
ElseIf feeling = "lousy" Then
    MsgBox("Too bad!")
Else
    MsgBox("I feel the same way!")
End If
```

Then end the whole program:

```
End
```

Instead of typing that multi-line If, you can type this **Select statement** instead, which is briefer and simpler:

```
Select Case feeling
    Case "fine"
        MsgBox("That's good!")
    Case "lousy"
        MsgBox("Too bad!")
    Case Else
        MsgBox("I feel the same way!")
End Select
```

Like a multi-line If, a Select statement consumes several lines. The top line of that Select statement tells the computer to analyze the feeling and Select one of the cases from the list underneath. That list is indented and says:

```
In the case where the feeling is "fine",
    say "That's good!"

In the case where the feeling is "lousy",
    say "Too bad!"

In the case where the feeling is anything else,
    say "I feel the same way!"
```

While you're typing the Select statement, the computer automatically indents the lines for you and automatically types "End Select" underneath.

Complete subroutine

Here's a complete subroutine:

```
Dim feeling
feeling = InputBox("How are you?")
Select Case feeling
    Case "fine"
        MsgBox("That's good!")
    Case "lousy"
        MsgBox("Too bad!")
    Case Else
        MsgBox("I feel the same way!")
End Select
MsgBox("I hope you enjoyed your therapy. Now you owe $50.")
End
```

The InputBox line makes the computer ask the patient, "How are you?" The next several lines are the Select statement, which makes the computer analyze the patient's answer and print "That's good!" or "Too bad!" or else "I feel the same way!"

Regardless of what the patient and computer said, that subroutine's bottom MsgBox line always makes the computer end the conversation by saying:

```
I hope you enjoyed your therapy. Now you owe $50.
```

In that program, try changing the strings to make the computer say smarter remarks, become a better therapist, and charge even more money.

Fancy cases

You can create fancy cases:

Statement	Meaning
Case "fine"	If it's "fine"
Case "fine", "lousy"	If it's "fine" or "lousy"
Case 6	If it's 6
Case 6, 7, 18	If it's 6 or 7 or 18
Case Is < 18	If it's less than 18
Case Is > 18	If it's greater than 18
Case Is <= 18	If it's less than or equal to 18
Case Is >= 18	If it's at least 18 (greater than or equal to 18)
Case 6, 7, Is >=18	If it's 6 or 7 or at least 18
Case 10 To 100	If it's between 10&100
Case 6, 10 To 100	If it's 6 or between 10&100

When typing a Case statement, don't bother typing the word "Is". The computer will type it for you automatically.

Exit Sub

To make the computer skip the bottom part of your subroutine, say **Exit Sub**, like this:

```
MsgBox("I love the company president")
Exit Sub
MsgBox("I love him as much as stale bread")
```

When you run that program (by clicking "Start"), the computer will say "I love the company president" but then exit from the subroutine, without saying "I love him as much as stale bread". The computer will say just:

```
I love the company president
```

Suppose you write a subroutine that displays many messages, and you want to run the program several times (so several of your friends see the messages). If one of your friends would be offended by the last few messages, send that friend an *abridged* subroutine! Here's how: put Exit Sub above program part that you want the computer to ignore.

"Exit Sub" versus "End": Instead of saying "Exit Sub", you can say "End". Here's the difference:

When the computer encounters "Exit Sub" in a subroutine, the computer stops running that subroutine but continues running the rest of the program: for example, it displays Form1, until the human clicks Form1's X button.

When the computer encounters "End" in a subroutine, the computer stops running the whole program and automatically clicks Form1's X button.

Property list

While you're creating or editing a Visual Basic program, you see **tabs** near the screen's top-left corner. Try clicking those tabs now:

If you click the **"Form1.vb [Design]"** tab, you see the Form1 window itself, so you can admire the Form1 window's size, color, and any writing in it.

If you typed a subroutine for Form1, you also see a **"Form1.vb"** tab. If you click that tab, you see the subroutine you typed.

Try this experiment...

Click the "Form1.vb [Design]" tab, so you see the Form1 window itself. **Then click (just once) in the middle of the Form1 window.**

Then the screen's bottom-right corner should show a list, whose title is:

```
Properties
Form1 System.Windows.Forms.Form
```

If you don't see that list yet, press the F4 key (or click View then then Properties Window) then try again to click in the middle of the Form1 window.

That list is called **Form1's main property list** (or **properties window**). It's divided into these 9 categories:

```
Accessibility
Appearance
Behavior
Data
Design
Focus
Layout
Misc
Window Style
```

Here's the full list:

Property	Value
Accessibility	
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
Appearance	
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
Cursor	Default
Font	Microsoft Sans Serif, 7.8pt
ForeColor	ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	Form1
UseWaitCursor	False
Behavior	
AllowDrop	False
AutoValidate	EnablePreventFocusChange
ContextMenuStrip	(none)
DoubleBuffered	False
Enabled	True
ImeMode	NoControl
Data	
(ApplicationSettings)	
(DataBindings)	
Tag	
Design	
(Name)	Form1
Language	(Default)
Localizable	False
Locked	False
Focus	
CausesValidation	True
Layout	
AutoScaleMode	Font
AutoScroll	False
AutoScrollMargin	0, 0
AutoScrollMinSize	0, 0
AutoSize	False
AutoSizeMode	GrowOnly
Location	0, 0
MaximumSize	0, 0
MinimumSize	0, 0
Padding	0, 0, 0, 0
Size	300, 300
StartPosition	WindowsDefaultLocation
WindowState	Normal
Misc	
AcceptButton	(none)
CancelButton	(none)
KeyPreview	False
Window Style	
ControlBox	True
HelpButton	False
Icon	(Icon)
IsMdiContainer	False
MainMenuStrip	(none)
MaximizeBox	True
MinimizeBox	True
Opacity	100%
ShowIcon	True
ShowInTaskbar	True
SizeGripStyle	Auto
TopMost	False
TransparencyKey	

(The screen shows part of the list. To see the whole list, use the list's scroll arrows.)

Text

The top of Form1's window normally says "Form1". That's called the window's **title** (or **caption** or **text**). Instead of making the title say "Form1", you can make it say "Results" or "Payroll results" or "Mary's window" or "Fun stuff" or "Hey, I'm a funny window" or anything else you wish!

To make Form1's title say "Fun stuff", you can put this line in Form1's subroutine —

```
Text = "Fun stuff"
```

but here's an easier way:

In Form1's main property list, click the word "Text" (which is in the Appearance category, after you scroll up or down to see it), then type what you want the title to be, so the property list's Text line becomes this:

```
Text Fun stuff
```

When you finish typing, press the Enter key.

Try that now! It makes the top of Form1 say "Fun stuff" immediately (or when you press Enter or click "Start" or the Form1 window or the "Form1.vb [Design]" tab).

Color

Normally, the Form1 window's middle is a big blank area that's nearly white (a color called **Control**, which is very light gray). To make it red instead, you can put this line in Form1's subroutine —

```
BackColor = Color.Red
```

but here's an easier way:

In Form1's main property list, click **BackColor** (which is in the Appearance category) then BackColor's down-arrow then a color category ("Custom" or "Web" or "System") then the color you want (such as Red, which you'll see in the Web category, after you scroll down).

Try that now! It makes Form1's background color become Red instantly.

Maximize

The Form1 window is normally medium-sized. To maximize it, you can use 3 methods.

Manual method While the program is running (because you clicked "Start"), you can manually click the Form1 window's maximize button. That maximizes the window but just temporarily: when you finish running the program (by clicking the Form1 window's X button), the computer forgets about maximization. The next time you run the program, it will *not* be maximized, unless you click the maximize button again.

Equation method Insert this equation in Form1's subroutine:

```
windowState = 2
```

Property-list method In Form1's main property list, click `WindowState` (which is in the `Layout` category) then `WindowState`'s down-arrow then `Maximized`. That makes the property list's `WindowState` line become:

<code>WindowState</code>	<code>Maximized</code>
--------------------------	------------------------

When you run the program (by clicking "Start"), Form1's window will be maximized.

Refuse to maximize

Instead of maximizing the Form1 window, you can do just the opposite: you can *prevent* the user from maximizing. Here's how....

In Form1's main property list, click **MaximizeBox** (in the `Windows Style` category) then press the `F` key (which means "False"). That makes the property list's `MaximizeBox` line become:

<code>MaximizeBox</code>	<code>False</code>
--------------------------	--------------------

That make Form1's maximize button (which is also called the maximize box) be **grayed out** while the program runs; the maximize button will become gray instead of black-and-white. That grayed-out button will ignore all attempts to be clicked, so the window will refuse to maximize.

Resize

Normally, Form1 is 300 pixels wide and 300 pixels tall. Here's how to adjust that size....

Property-list method In Form1's main property list, click **Size** (in the `Layout` category), then change "300, 300" to the size you wish, by editing those numbers. For example, if you want Form1 to be 500 pixels wide and 400 pixels tall, change the size to "500, 400". The first number is the form's width; the second number is the form's height.

The biggest permissible size is the size of your whole screen, plus a few pixels more. For example, if your screen is Full HD (which is 1080p and has a resolution of 1920-by-1080), the biggest permissible size for you is "1924, 1084". If you want Form1 to be half as wide and half as tall as the full screen, choose "960, 540".

For a Full HD screen, the smallest permissible size is "166, 47". That's barely enough to show Form1's fundamental buttons (close, maximize, and minimize), not much else!

If you request a size that's very big (almost as big as the screen), Form1 won't look that big until you run the program (by clicking "Start").

Drag method While the program is running (because you clicked "Start"), you can change Form1's size by dragging its bottom right corner. That changes the size just temporarily: when you finish running the program (by clicking Form1's `X` button), the computer forgets how you dragged Form1's corner, and Form1 reverts to its previous size.

Here's how to change Form1's size so the computer remembers the new size:

Make sure the program is *not* running. (If it's running, stop it by clicking its `X` button.)

Click the "Form1.vb [Design]" tab (so you see what Form1 looks like, not subroutines you typed).

At Form1's bottom right corner, you see a tiny white square (called a **handle**). Drag that handle until Form1 becomes the size you wish.

That changes Form1's size permanently (or until you change the size again). You'll see that size in the property list's `Size` line (in the `Layout` category).

Refuse to resize

In Form1's main property list, the `Appearance` category's **FormBorderStyle** line normally says:

<code>FormBorderStyle</code>	<code>Sizable</code>
------------------------------	----------------------

Try this experiment: click `FormBorderStyle` then `FormBorderStyle`'s down-arrow then "FixedToolWindow", so the line becomes:

<code>FormBorderStyle</code>	<code>FixedToolWindow</code>
------------------------------	------------------------------

That prevents stupid humans from changing Form1's size. When a human runs the program (by clicking "Start"), Form1's window will have no maximize button, no restore-down button, no minimize button, and no resizable edges. Form1 stays the size you specified in the property list (such as the property list's `Size` line), so stupid humans can't mess up your beautiful design (unless they edit your subroutine or property list).

Form position

Here's how to adjust Form1's position....

Property-list method In Form1's main property list, click the `Layout` category's **StartPosition** then `StartPosition`'s down-arrow. You see this list of choices:

<code>Manual</code>
<code>Center Screen</code>
<code>WindowsDefaultLocation</code>
<code>WindowsDefaultBounds</code>
<code>CenterParent</code>

Click a choice now. You'll see the effect later (when you click "Start" to run the program).

The computer assumes you want "WindowsDefaultLocation" unless you click a different choice instead.

"WindowsDefaultLocation" puts Form1 near the screen's top-left corner (leaving a 1/4-inch margin gap) and makes Form1's size be what you chose in the `Size` line.

"CenterScreen" puts Form1 at the screen's center and makes Form1's size be what you chose in the `Size` line.

"CenterParent" puts Form1 at the center of what Windows thinks is appropriate (which is typically left of the screen's center) and makes Form1's size be what you chose in the `Size` line.

"WindowsDefaultBounds" puts Form1 very near the screen's top-left corner (leaving just a 1/2-inch margin gap) and makes Form1 be big (9/4 inches wide, 6 1/2 inches tall). The `Size` line is ignored.

"Manual" puts Form1 at the screen's top-left corner (leaving no gap, unless you change the `Location` line to something different from "0, 0") and makes Form1's size be what you chose in the `Size` line.

All those inch measurements are approximate and depend on your screen's size.

Drag method While the program is running, you can move Form1 by dragging its **title bar** (the blue horizontal bar that's at Form1's top and typically says "Form1"). That moves Form1 just temporarily; when you finish running the program (by clicking Form1's `X` button), the computer forgets how you dragged Form1, and Form1 reverts to its previous position.

Opacity

Normally, Form1 is completely opaque: while the program is running, Form1 completely blocks the view of anything behind it.

To have fun, reduce Form1's opacity: in Form1's main property list, click the `Windows Style` category's **Opacity**, then change "100%" to "75%". When you run the program (by clicking "Start"), Form1 will be just partly opaque; it will be partly transparent, so you can see, faintly, what's behind the form.

If you make the opacity even lower — 50% or 25% — Form1 will be hardly opaque at all — it will be very transparent — so you can easily see what's behind it, as if Form1 were just a ghost.

Don't make the opacity be 0%. That would make Form1 completely invisible, so you couldn't see it at all, couldn't click its close box, and couldn't stop the program!

Don't make the opacity be less than 25%. That would make Form1 difficult to see.

Try making the opacity be 90%.

Opacity doesn't work well if your Windows version is old (Windows XP) or stripped-down (Windows Vista Basic or Windows 7 Starter), since those Windows versions lack **Windows Aero** (which makes windows partly transparent).

Toolbox

You've learned how to create and manipulate an object called "Form1". You can create other objects also, by using the **toolbox**.

See the toolbox

Click the "**Form1.vb [Design]**" tab then "**View**" (which is near the screen's top-left corner) then "**Toolbox**". Then you see 10 **toolbox categories**:

All Windows Forms
Common Controls
Containers
Menus & Toolbars
Data
Components
Printing
Dialogs
WPF Interoperability
General

(If you don't see that whole list yet, scroll down.)

See common controls

Click "Common Controls" once or twice, until you see this list indented under "Common Controls":

-  Pointer
-  Button
- CheckBox
-  CheckedListBox
-  ComboBox
-  DateTimePicker
-  Label
-  LinkLabel
-  ListBox
-  ListView
-  MaskedTextBox
-  MonthCalendar
-  NotifyIcon
-  NumericUpDown
-  PictureBox
-  ProgressBar
-  RadioButton
-  RichTextBox
-  TextBox
-  ToolTip
-  TreeView
-  WebBrowser

To the right of the word "Toolbox", you see an X. Left of the X you see a **pushpin**. If the pushpin is still horizontal,

make it vertical by clicking it. That makes the toolbox stay on the screen *nicely* (permanently and left of your other work).

Each object in the toolbox is called a **tool**.

Button

Try this experiment. Double-click the **Button** tool. That makes a button appear in Form1's middle, near Form1's top-left corner (or near a previous button). The button is a rectangle and says "Button1" on it.

(If you wish, drag that button to a different place in Form1. You can also change the button's size by dragging its 9 square handles. But for your first experiment, just leave the button where the computer put it.)

The button says "Button1" on it. Just for fun, let's make it say "Click me" instead. To do that, click Text (in the property list) and type "Click me", so the property list's Text line becomes:

```
Text          Click me
```

That makes the button's text become "Click me" (instead of "Button1").

Notice that the property list concerns the button and its text (instead of Form1's text), because the button is highlighted.

Let's write a program so if a human clicks the button (which says "Click me"), Form1 will say "Thanks for the click". To do that, double-click the button. The double-clicking tells the computer you want to write a subroutine about that object (the button).

The computer starts writing the subroutine for you. The computer writes:

```
Public Class Form1
    Private Sub Button1_Click...
        End Sub
End Class
```

Insert this line in the middle of the subroutine —

```
Text = "Thanks for the click"
```

so the subroutine looks like this:

```
Public Class Form1
    Private Sub Button1_Click...
        Text = "Thanks for the click"
        End Sub
End Class
```

That subroutine tells the computer that when Button1 is clicked, the computer should say "Thanks for the click" on Form1.

Try it: run that program (by clicking "Start"). You'll see Form1 with a button on it that says "Click me". If you click the button, the subroutine makes Form1 say "Thanks for the click" (after you maximize Form1 so you can see all that).

In that subroutine, the computer assumes you want the text "Thanks for the click" to be on Form1, not on the button. If you want that text to be on the *button* instead, say **Button 1's Text** instead of just Text. To say **Button 1's Text**, type **Button1.Text**, so the subroutine looks like this:

```
Public Class Form1
    Private Sub Button1_Click...
        Button1.Text = "Thanks for the click"
        End Sub
End Class
```

But to fit "Thanks for the click" onto the button, you must widen the button, by doing this:

If the program is still running, finish running it (by clicking Form1's X button). Then click the "Form1.vb [Design]" tab, so you see Form1 and can modify the appearance of Form1 (and of its button). Then widen the button (by dragging the button's handles).

Two buttons Let's write a program that has *two* buttons! Let's make the first button be called "Red" and the second button be called "Blue". If the human clicks the "Red" button, let's make Form1 turn red; if the human clicks the "Blue" button, let's make Form1 turn blue.

To do all that, start a new program as follows:

If a previous program is still running, finish running it (by clicking Form1's X button). If you're not in Visual Basic, go into it.

Click "New Project" or the New Project button (which is near the screen's top-left corner, below the word "File"). Click "Windows Forms Application", type a name in the Name box, and press Enter, so you see the Form1 window.

You should also see the Toolbox. (If you don't see it yet, continue following the "See the toolbox" instructions on the previous page.)

In the Toolbox, double-click the Button tool. A command button appears in Form1 and is called Button1. In the property list, click Text then type “Click here for red” (and press Enter), so the property list’s Text line becomes:

```
Text Click here for red
```

That tries to make the button’s text become “Click here for red”. To fit all that text onto the button, widen the button (by dragging one of its handles).

In the Toolbox, double-click the Button tool again. That makes another command button appear in Form1 and be called Button2. Unfortunately, the Button2 button covers up the Button1 button, so you can’t see the Button1 button. Drag the Button2 button out of the way (toward the right), so you can see both buttons side-by-side.

The Button2 button should be highlighted. (If it’s not highlighted, click it to make it highlighted.) In its property list, change its Text from “Button2” to “Click here for blue” (by clicking Text then typing “Click here for blue” and pressing Enter). Widen the Button2 button (by dragging one of its handles) so it shows all of “Click here for blue”.

Now your screen shows Form1 with two buttons on it. The first button says “Click here for red”. The second button says “Click here for blue”.

Double-click the “Click here for red” button, and write this subroutine for it:

```
Private Sub Button1_Click...
    BackColor = Color.Red
End Sub
```

That subroutine says: clicking that button will make Form1’s background color be red.

Move that subroutine out of the way (by clicking the “Form1.vb [Design]” tab), so you can see Form1.

Double-click the “Click here for blue” button, and write this subroutine for it:

```
Private Sub Button 2_Click...
    BackColor = Color.Blue
End Sub
```

While you’re writing that subroutine, you’ll see the other subroutine above it. Altogether, you see:

```
Public Class Form1
    Private Sub Button1_Click...
        BackColor = Color.Red
    End Sub

    Private Sub Button 2_Click...
        BackColor = Color.Blue
    End Sub
End Class
```

Then run the program by clicking “Start”. Here’s what happens...

You see Form1 with two buttons on it. The first button says “Click here for red”; if you click it, Form1 turns red. The other button says “Click here for blue”; if you click it, Form1 turns blue.

Try clicking one button, then the other. Click as often as you like. When you get tired of clicking, end the program (by clicking Form1’s X button).

Where to put buttons A good habit is to put buttons side-by-side, in Form1’s bottom right corner. That way, the buttons won’t interfere with any other objects on Form1.

Exit button To stop running a typical program, you have to click its X button. Some humans don’t know to do that. To help them, create a button called “Exit”, so that clicking it will make the computer exit from the program.

To do that, create an ordinary button; but make the button’s text say “Exit” (or anything else you prefer, such as “Quit” or “End” or “Abort” or “Close” or “Click here to end the program”), and make the button’s subroutine say End, like this:

```
Private Sub Button3_Click...
    End
End Sub
```

Put that Exit button in Form1’s bottom-right corner.

Check box

A **check box** is a small gray square, with text to the right of the square. At first, the gray square has nothing inside it: it’s empty. While the program runs, clicking the square makes a green check mark (✓) appear in the square. If you click the square again, the check mark disappears.

To create a check box, double-click the **CheckBox** tool. That makes a check box appears in Form1. Drag the check box wherever you wish.

The first check box’s text is temporarily “CheckBox1”; to change that text, click Text (in the property list) and type whatever text you wish. At the end of your typing, press Enter. Suggestion: if you want to type a *lot* of text, do this instead:

```
Click Text’s down-arrow. You’ll see a big box to type in. Press Enter at the end of each line you type. When you’ve finished typing all text you want, do this: while holding down the Ctrl key, tap the Enter key.
```

If you want the computer to react immediately to whether the check box is checked, give the check box this subroutine:

```
Private Sub CheckBox1_CheckedChanged...
    If CheckBox1.Checked Then
        type here what to do if CheckBox1 just became checked
    Else
        type here what to do if CheckBox1 just became unchecked
    End If
End Sub
```

Dressed example For example, this subroutine makes the computer say “I am dressed” if the check box just became checked but say “I am naked” if the check box just became empty:

```
Private Sub CheckBox1_CheckedChanged...
    If CheckBox1.Checked Then
        Text = "I am dressed"
    Else
        Text = "I am naked"
    End If
End Sub
```

When that program runs, the check box starts by being empty. Clicking the check box makes you see ✓ and makes the computer say “I am dressed”. The next time you click the check box, the ✓ disappears from the box, so the box becomes empty and the computer say “I am naked”. Clicking the check box again makes the ✓ reappear and makes the computer say “I am dressed”.

Here’s how to type that subroutine fast:

```
Type the word “if”, then a space, then the letters “che”. You see a list of computer words that begin with the letter “Che”. In that list, double-click “CheckBox1”. That makes the computer type “CheckBox1” for you.
```

```
Type a period. You see a list of computer words; from that list, choose “Checked” by double-clicking it (or if “Checked” was highlighted already, you can choose it by pressing the Tab key).
```

Multiple check boxes Form1 can contain many check boxes. The human can check several at the same time, so that several of the boxes contain check marks simultaneously.

Although you can put buttons and check boxes wherever you wish, it’s customary to arrange buttons horizontally (so the second button is to the right of the first) but arrange check boxes vertically (so the second check box is below the first). The check boxes (and their texts) form a vertical list of choices.

OK button If Form1 contains several check boxes, you should typically delay the computer's reaction until the human has decided which boxes to check, has checked all the ones desired, and has clicked an OK button to confirm that the correct boxes are checked.

To do that, make the check boxes have no subroutines. Instead, create an OK button in Form1's bottom-right corner (by creating a button there and making its text be "OK"), then make the OK button's subroutine look like this:

```
Private Sub Button1_Click...
    If CheckBox1.Checked Then
        type here what to do if CheckBox1 is checked
    Else
        type here what to do if CheckBox1 is unchecked
    End If
    If CheckBox2.Checked Then
        type here what to do if CheckBox2 is checked
    Else
        type here what to do if CheckBox2 is unchecked
    End If
End Sub
```

That subroutine says: when the OK button is clicked, notice which check boxes are checked and react appropriately.

For example, let's make a program that alters Form1's appearance, to make it red or maximized or red maximized or return to normal. Here's how to do all that:

Create a check box (called CheckBox1) with title "Red".
 Create a check box (called CheckBox2) with title "maximized".
 Create a command button (called Button1) with title "OK" and this subroutine:

```
Private Sub Button1_Click...
    If CheckBox1.Checked Then
        BackColor = Color.Red
    Else
        BackColor = Color.WhiteSmoke
    End If
    If CheckBox2.Checked Then
        WindowState = 2
    Else
        WindowState = 0
    End If
End Sub
```

Radio button

A **radio button** resembles a check box but looks and acts like the button on an old-fashioned radio, because of these differences:

To create a check box, double-click the **CheckBox** tool.
 To create a radio button, double-click the **RadioButton** tool.

A check box is a tiny gray **square**.
 A radio button is a tiny gray **circle**.

While the program is running,
 clicking a checkbox makes a **green checkmark** appear in the square.
 Clicking a radio button makes a **blue dot** appear in the circle.

Green checkmarks can appear in many checkboxes, simultaneously.
You see just one blue dot. When you click a radio button, the blue dot hops to that radio button and leaves the previous button.

Like checkboxes, radio buttons are arranged vertically (so the second radio button is below the first). The radio buttons (and their titles) form a vertical list of choices.

When the human starts running your program, the first radio button (which is RadioButton1) has a blue dot inside the gray circle, and the computer automatically does RadioButton1's subroutine (even if the human hasn't clicked RadioButton1's button yet).

Afterwards, if the human clicks a different radio button, here's what happens:

```
The blue dot hops to that radio button. The computer does that button's subroutine; but before doing so, the computer does the previous button's subroutine one more time so it can display a message such as "Sorry to hear you don't like this choice anymore".
```

If you want the computer to react immediately to whether the radio button has the blue dot, give the radio button this subroutine:

```
Private Sub RadioButton1_CheckedChanged...
    If RadioButton1.Checked Then
        type here what to do if RadioButton1 just got the blue dot
    Else
        type here what to do if RadioButton1 just lost the blue dot
    End If
End Sub
```

3-color example For example, let's write a program that has 3 radio buttons, labeled "Red", "Blue", and "Green". While the program is running, if the human switches from "Red" to "Blue" (by clicking "Blue" after having clicked "Red"), let's make the program say "Sorry you don't like red anymore" and make Form1 become blue. Let's make the program act similarly for switching between other pairs of colors.

To do that, maximize Form 1 (by making its WindowState property be Maximized) and create the 3 radio buttons (labeled "Red", "Blue", and "Green"). Then give the "Red" button (which is RadioButton1) this subroutine:

```
Private Sub RadioButton1_CheckedChanged...
    If RadioButton1.Checked Then
        BackColor = Color.Red
    Else
        Text = "Sorry you don't like red anymore"
    End If
End Sub
```

That subroutine runs just when the human changes the Red radio button's appearance (by clicking it or unclicking it):

When that radio button's appearance changes to "checked" (contains a blue dot), that subroutine makes Form1 be red.

When that radio button's appearance changes to "unchecked" (no blue dot), that subroutine makes Form1 say "Sorry you don't like red anymore".

Here's how to type that subroutine fast:

Type the word "if", then a space, then the letter "r". You see a list of computer words that begin with the letter R. In that list, double-click "RadioButton1". That makes the computer type "RadioButton1" for you.

Type a period. You see a list of computer words; from that list, choose "Checked" by double-clicking it (or if "Checked" was highlighted already, you can choose it by pressing the Tab key).

To type the rest of the subroutine fast, keep choosing from lists.

To finish the program, type this subroutine for the "Blue" button (which is RadioButton2) —

```
Private Sub RadioButton2_CheckedChanged...
    If RadioButton2.Checked Then
        BackColor = Color.Blue
    Else
        Text = "Sorry you don't like blue anymore"
    End If
End Sub
```

and this subroutine for the "Green" button (which is RadioButton3):

```
Private Sub RadioButton3_CheckedChanged...
    If RadioButton3.Checked Then
        Text = "Welcome to New York"
    Else
        MsgBox("You've left New York")
    End If
End Sub
```

3-city example For a similar example, let's write an airplane program that has 3 radio buttons, labeled "Los Angeles", "Dallas", and "New York". While the program is running, if the human switches from "Los Angeles" to "Dallas" (by clicking "Dallas" after having clicked "Los Angeles"), let's make the program say "You've left Los Angeles" and "Welcome to Dallas". Let's make the program act similarly for traveling between the other cities.

To do that, create the 3 radio buttons (labeled "Los Angeles", "Dallas", and "New York"). Then give the "Los Angeles" button (which is RadioButton1) this subroutine:

```
Private Sub RadioButton1_CheckedChanged...
  If RadioButton1.Checked Then
    Text = "Welcome to Los Angeles"
  Else
    MsgBox("You've left Los Angeles")
  End If
End Sub
```

That subroutine runs just when the human changes the Los Angeles radio button's appearance (by clicking it or unclicking it):

When that radio button's appearance changes to "checked" (contains a blue dot), that subroutine makes the computer say "Welcome to Los Angeles".

When that radio button's appearance changes to "unchecked" (no blue dot), that subroutine makes the computer say "You've left Los Angeles".

To finish the program, type this subroutine for the "Dallas" button (which is RadioButton2) —

```
Private Sub RadioButton2_CheckedChanged...
  If RadioButton2.Checked Then
    Text = "Welcome to Dallas"
  Else
    MsgBox("You've left Dallas")
  End If
End Sub
```

and this subroutine for the "New York" button (which is RadioButton3):

```
Private Sub RadioButton3_CheckedChanged...
  If RadioButton3.Checked Then
    Text = "Welcome to New York"
  Else
    MsgBox("You've left New York")
  End If
End Sub
```

OK button When the human clicks a radio button, the computer can react to the click immediately, but that might startle and upset the human. If you want to be gentler, delay the computer's reaction until the human also clicks a general OK button, which confirms the human's desires.

To do that, make the radio buttons have no subroutines, so nothing will happen when those buttons are clicked. In Form1's bottom-right corner, create an ordinary button whose text says "OK" and whose subroutine looks like this:

```
Private Sub Button1_Click...
  If RadioButton1.Checked Then
    type here what to do if RadioButton1 just got the blue dot
  ElseIf RadioButton2.Checked Then
    type here what to do if RadioButton2 just got the blue dot
  ElseIf RadioButton3.Checked Then
    type here what to do if RadioButton3 just got the blue dot
  ElseIf RadioButton4.Checked Then
    type here what to do if RadioButton4 just got the blue dot
  Else
    type here what to do if bottom radio button got blue dot
  End If
End Sub
```

That subroutine says: when the OK button is clicked, notice which radio button was clicked and react appropriately.

Label

Form1 has two main parts. One part is a blue bar across Form1's top: it includes Form1's minimize button, maximize button, close button, and text. The other part (Form1's middle) is a big light-gray box: it includes objects you created, such as ordinary buttons, check boxes, and radio buttons.

In Form1's middle, let's type this text:

I love you

To do that, double-click the **Label** tool. That makes the word "Label1" appear in Form1's middle. Drag "Label1" to the spot in Form1 where you wish to begin typing. To change "Label1" to "I love you", click Text (in Label1's property list) then type "I love you", so the property list says:

Text I love you

At the end of that typing, press Enter. Then "Label1" becomes "I love you", so "I love you" is in Form1's middle.

Multi-line text Instead of making Form1's middle say just "I love you", let's make it say:

I love you
You turned me on
Let's get married

Here's how:

After you've created a label (by double-clicking the Label tool) and clicked Text, click Text's down-arrow. You'll see a big box to type in. Press Enter at the end of each line you type. When you've finished typing all text you want, do this: while holding down the Ctrl key, tap the Enter key.

Text equation Here's a different way to make Form1's middle say "I love you".

Create a label (by double-clicking the Label tool). Tell the computer you want to write a subroutine for Form1 (by double-clicking in Form1 but *not* in the label). Put this line in Form1's subroutine:

```
Label1.Text = "I love you"
```

That means: the label's text is "I love you." That line makes Form1's subroutine become this:

```
Private Sub Form1_Load...
  Label1.Text = "I love you"
End Sub
```

When you run the program (by clicking "Start"), the computer will run that subroutine and make Form1's middle say:

I love you

If instead you want the Form1's middle to say —

I love you
You turned me on

change the Text line to this:

```
Label1.Text = "I love you" & Chr(13) & "You turned me on"
```

That makes the computer type "I love you" then press key #13 (which is the Enter key) then type "You turned me on". Instead of that long Text line, you can give this pair of shorter Text lines:

```
Label1.Text = "I love you"
Label1.Text &= Chr(13) & "You turned me on"
```

In that pair of lines, the first line makes the Text be "I love you"; the next line changes the Text to become all that, combined with Chr(13) and "You turned me on".

Math Let's make Form1's middle say the answer to 4 + 2.

To do that, create a label (by double-clicking the Label tool). Tell the computer you want to write a subroutine for Form1 (by double-clicking in Form1 but *not* in the label). Put this line in Form1's subroutine:

```
Label1.Text = 4 + 2
```

That line means: Label1's text is the answer to 4 + 2. That line makes Form1's subroutine become this:

```
Private Sub Form1_Load...
    Label1.Text = 4 + 2
End Sub
```

When you run the program (by clicking "Start"), the computer will run that subroutine and write the answer (6) in Form1's middle.

To make Form1's middle say the answer to 4 + 2 and, on the next line, say the answer to 48 + 3, you can put this line in Form1's subroutine —

```
Label1.Text = 4 + 2 & Chr(13) & 48 + 3
```

or give this pair of lines instead:

```
Label1.Text = 4 + 2
Label1.Text &= Chr(13) & 48 + 3
```

In that pair of lines, the first line makes the Text be the answer to 4 + 2; the next line changes the Text to become all that, combined with Chr(13) and the answer to 48 + 3.

List box

A **list box** is a big white box that contains a list of choices, such as these color choices —

```
Red
Blue
Green
```

or these country choices —

```
United States
Canada
Mexico
```

The list can be short (2 or 3 choices) or tall (hundreds of choices). If the list is too tall fit in the box, the computer will automatically add scroll arrows so humans can scroll through the list.

To create a list box, double-click the **ListBox** tool. A list box (big white box) appears in the middle of Form1. Drag the list box wherever you wish.

The first list box is called ListBox1. Inside that list box, you temporarily see the word "ListBox1", but you should put your own list of choices there instead, by using this method —

In the list box's property list, click "Items" then "...". Type the list of choices, such as:
United States
Canada
Mexico

To do that, press Enter at the end of each line. When you've finished typing the whole list, click "OK".

or this alternate method:

Click the list box's right-arrow (which is near the box's top-right corner) then "Edit Items". Type the list of choices, such as:
United States
Canada
Mexico

To do that, press Enter at the end of each line. When you've finished typing the whole list, click "OK" then click in the list box.

Then on Form1, you see the list box containing your choices.

By dragging the box's handles, try to make the box just tall enough to hold *all* the choices. (If it isn't tall enough, the computer automatically adds scroll arrows so humans can scroll through the list while the program runs.)

By dragging the box's handles, make the box just wide enough to hold the widest choice.

Select one You can give List1 this kind of subroutine:

```
Private Sub ListBox1_SelectedIndexChanged...
    Select Case ListBox1.SelectedItem
        Case "United States"
            type here what to do if "United States" is clicked
        Case "Canada"
            type here what to do if "Canada" is clicked
        Case "Mexico"
            type here what to do if "Mexico" is clicked
    End Select
End Sub
```

Here's a shorter way to type the subroutine:

```
Private Sub ListBox1_SelectedIndexChanged...
    Select Case ListBox1.SelectedIndex
        Case 0
            type here what to do if the list's top item ("United States") is clicked
        Case 1
            type here what to do if the list's next item ("Canada") is clicked
        Case 2
            type here what to do if the list's next item ("Mexico") is clicked
    End Select
End Sub
```

If you want the action to be delayed until the human clicks an OK button, do this:

Create the OK button (a command button whose caption is "OK").

Give ListBox1 no subroutine, but give the OK button this kind of subroutine —

```
Private Sub Button1_Click...
    Select Case ListBox1.SelectedItem
        Case "United States"
            type here what to do if "United States" is clicked
        Case "Canada"
            type here what to do if "Canada" is clicked
        Case "Mexico"
            type here what to do if "Mexico" is clicked
    End Select
End Sub
```

or this subroutine:

```
Private Sub Button1_Click...
    Select Case ListBox1.SelectedIndex
        Case 0
            type here what to do if the list's top item ("United States") is clicked
        Case 1
            type here what to do if the list's next item ("Canada") is clicked
        Case 2
            type here what to do if the list's next item ("Mexico") is clicked
    End Select
End Sub
```

Select multi If you want to let the human select *several* items from the list (instead of just one item), do this:

In ListBox1's property list, click **SelectionMode** then SelectionMode's down-arrow.

Click either "MultiSimple" or "MultiExtended". (If you choose "MultiSimple", the human can select several items by clicking them, and deselect an item by clicking that item again. If you choose "MultiExtended", the human can select one item by clicking it, select or deselect extra items by holding down the Ctrl key while clicking them, and select a contiguous bunch of items easily by clicking the bunch's first item and Shift-clicking the last.)

Create an OK button (an ordinary button whose caption is "OK").

Give ListBox1 no subroutine, but give the OK button this kind of subroutine:

```
If ListBox1.GetSelected(0) Then type here what to do if the list's top item ("United States") clicked
If ListBox1.GetSelected(1) Then type here what to do if the list's next item ("Canada") clicked
If ListBox1.GetSelected(2) Then type here what to do if the list's next item ("Mexico") clicked
```

Label Next to your list box, you should put some text, explaining the list box's purpose to the human. To put the text there, create a label with that text (by double-clicking the Label tool), and drag the label until it's next to your list box.

Text box

You already learned that Form1's subroutine can contain this line:

```
x = InputBox("what is your name?")
```

When you run the program, that line makes the computer create an **input box**. The input box is a pop-up window containing a message ("What is your name?"), a wide white box (in which the human types a response), and an OK button (which the human clicks when finished typing).

That technique works adequately but gives you no control over the size or position of its objects (the window, message, white response box, and OK button).

To be more professional, get control by creating a **text box** instead. Here's how.

Double-click the **TextBox** tool. That creates a text box (a white box in which the human can type a response). Drag it wherever you wish. Adjust its size by dragging its handles.

Above the box (or left of it), put a label (by double-clicking the Label tool). Make the label contain a message (such as "What is your name?").

Below the box (or right of the box), create an OK button (a button whose text says "OK"). Make the OK button's subroutine include these lines —

```
Dim x  
x = TextBox1.Text
```

and anything else you want the computer to do, such as:

```
MsgBox("I love " & x)
```

Form1 can contain *several* text boxes. For example, you can include:

```
a text box for the human's first name  
a text box for the human's last name  
a text box for the human's address  
text boxes for the human's city, state, and ZIP code
```

That makes Form1 be truly a form to fill in! Create just one OK button to handle all those text boxes, so the human clicks the OK button after filling in the entire form.

Password character If you want the human to type a password into a text box, do this: in the text box's property list, click **PasswordChar** then type an asterisk (the symbol *). That makes the box show asterisks instead of the characters the human is typing. That prevents enemies from discovering the password by peeking over the human's shoulder.

MultiLine The typical text box holds just one line of text. To let your text box handle *several* lines of text well, make 3 adjustments:

In the text box's property list, click MultiLine then press the T key (which stands for True). That lets the text box handle *several* lines of text, lets the human press the Enter key at the end of each line, and lets the computer press the Enter key automatically if there are too many words to fit on a line.

Make the text box taller and wider (by dragging its handles), so it can show more lines of text and more words per line. That reduces the human's frustration.

In the text box's property list, click ScrollBars then press the V key (which stands for Vertical). That creates a vertical scroll bar, which helps the human move through the text, in case you didn't make the text box tall enough to handle all the words.

Rich-text box

Instead of double-clicking the TextBox tool, try double-clicking the **RichTextBox** tool. It creates a text box that's already tall, MultiLine (so the human can type many lines of text in the box), with a vertical scroll bar (which appears when the human types more lines than can fit in the box) and the ability to handle **formatted text** (which is called **rich text**). That box is called **RichTextBox1**.

For best results, make the box even taller and wider (by dragging its bottom-right corner).

In that box, the human can type a number, or a word, or a sentence, or a paragraph, or *several* paragraphs (by pressing the Enter key at the end of each paragraph), or a whole essay! What the human types in that box is called a **document**.

For example, if you want to invent your own word-processing program, the first step is to create a rich-text box for the human to type the words into.

Improve the rich-text box Here are 2 popular ways to improve how a rich-text box works:

In RichTextBox1's property list (at the screen's bottom-right corner), click "**EnableAutoDragDrop**" then press the T key. That makes EnableAutoDragDrop be True. Then whenever the human is typing the document, the human can highlight a phrase and drag it to a different spot in the document.

Click RichTextBox1's right-arrow (which is near the box's top-right corner) then "**Dock in parent container**". That makes RichTextBox1 expand and consume all of Form1. Then while the program is running, if the human changes Form1's size (by maximizing Form1 or by dragging Form1's bottom-right corner), RichTextBox1 will change size automatically, to still fill Form1.

Number box

To make the computer wait for the human's response, you learned you can create a text box (by double-clicking the TextBox tool) and an OK button whose subroutine includes these lines:

```
Dim x  
x = TextBox1.Text
```

If you want to force the human to type a number instead of words, create a **number box** instead of a text box. Here are the details....

Double-click the **NumericUpDown** tool. That creates a number box (a white box in which the human can type a number). Drag it wherever you wish. Adjust its width by dragging its handles.

Above the box (or left of it), put a label (by double-clicking the Label tool). Make the label contain a message (such as "How many children do you have?").

Below the box (or right of the box), create an OK button (a button whose text says "OK"). Make the OK button's subroutine include these lines —

```
Dim x  
x = NumericUpDown1.Value
```

and anything else you want the computer to do, such as:

```
MsgBox("I'm glad you have " & x)
```

When the human runs the program, the human sees the number box. That box temporarily has 0 in it, but the human can change that number by retyping it or by clicking the box's up-arrow (which increases the number) or down-arrow (which decreases the number). When the human has changed the number to what the human wishes, the human clicks the OK button, whose subroutine makes x become the human's number.

Alter the box's properties Normally, the number box refuses to let the human say any number over 100. If you want it to permit numbers up to 500, make its property list's **Maximum** line say 500. If you want it to permit just numbers up to 20, make its **Maximum** line say just 20. If you want to encourage the human to type a number that's small, make the box be narrow (by adjusting its handles).

Normally, the number box refuses to let the human say any number below 0. If you want it to permit numbers down to minus 500, make the property list's **Minimum** line say -500. If you want it to require the number to be at least 3, make its **Minimum** line say 3.

Normally, the number box refuses to accept decimals. If you want it to permit 2 digits after the decimal point (so the human can type dollars-and-cents), make the property list's **DecimalPlaces** line say 2.

The number box normally begins displaying the number 0. If you want it to begin by displaying the number 5 instead, make the property list's **Value** line say 5.

By adjusting those properties (Maximum, Minimum, DecimalPlaces, Value, and the box's width), you can encourage the human to be reasonable.

Combo box

A **combo box** is a fancy text box that includes a list of suggested responses.

To create a combo box, double-click the **ComboBox** tool. That creates a combo box. Like a text box, it's a white box in which the human can type a response; but the combo box's right edge shows a down-arrow, which the human can click to see a list of suggested responses.

Drag the combo box wherever you wish.

Click the box's right-arrow (which is near the box's top-right corner) then "Edit Items". Type your list of suggested responses, such as:

```
United States
Canada
Mexico
```

(Press Enter at the end of each line.) When you've finished typing the whole list, click "OK" then click in the combo box.

Above (or left of) the box, put a label (by double-clicking the Label tool). Make the label contain a prompt (an instruction to the human about what to put into the box).

Below (or right of) the box, create an OK button (a command button whose caption is "OK"). Make the OK button's subroutine include these lines —

```
Dim x
x = ComboBox1.Text
```

and anything else you want the computer to do, such as:

```
MsgBox("I'm glad you said " & x)
```

Drop-down style In combo box's property list, click **DropDownStyle** then **DropDownStyle**'s down-arrow. You see 3 styles:

```
Simple
DropDown
DropDown List
```

Click whichever style you wish. If you don't choose otherwise, the computer assumes you want "DropDown". That works as I described: the human can type anything into the box, and the suggestion list appears just if the human clicks the box's down-arrow.

If you choose "Simple" instead, the human can still type anything into the box, and the suggestion list *always* appears (without requiring a down-arrow click) if you make the combo box tall enough to hold the list.

If you choose "DropDown List" instead, the human *cannot* type into the box; the human is required to choose from the suggestion list, which appears when the human clicks the box.

Picture box

Here's how to make Form1 show a picture.

First, **enlarge Form1** (by dragging its bottom-right corner), to let it hold a big picture better.

Then double-click the **PictureBox** tool. That puts a box in Form1's middle, near Form1's top-left corner.

Enlarge that box (by dragging its bottom-right corner), to let it hold a big picture better. If you want the box to be smaller than Form1 (so Form1 can hold other objects also), drag the box where you wish (by pointing at the box's middle, the dragging).

Click the box's right-arrow (which is near the box's top-right corner) then **"Choose Image"** then the **bottom "Import" button** then **"Pictures"** (which is on the left). That shows what's in your hard disk's Pictures folder.

Double-click the picture you want (after clicking or double-clicking any folders it's buried in). For example, you can try double-clicking the "Sample Pictures" icon (which Windows has put in your Pictures folder) then "Penguins" (Windows 7's photo of a penguin trio) or "Annie in the Sink" (Windows Vista's photo of Annie the cat, sitting in a sink).

You see a bigger view of the picture (or its top-left corner).

Click "OK".

Click the Size Mode box's down-arrow. You see this menu:

```
Normal
StretchImage
AutoSize
CenterImage
Zoom
```

If the picture is bigger than the box, here's what those choices mean.

Zoom is the safest choice: it shrinks the picture nicely, so it fits in the box's center.

StretchImage fills the whole box by shrinking (or stretching) the picture, which gets distorted.

CenterImage puts just the picture's center into the box.

Normal puts just the picture's top-left corner into the box.

AutoSize stretches the box, to hold as much of the picture as possible.

The computer assumes you want "Normal". If you prefer a different choice (such as "Zoom"), click it.

Should you dock? If you click "Dock in parent container", the box will expand to fill Form1. (If you regret that expansion, undo it by clicking "Undock in parent container".)

Add a form

Besides Form1, you can create extra forms, called Form2, Form3, Form4, etc. To create an extra form, click the **Add New Item** button (which is near the screen's top, under the words "Edit" and "View") then double-click the Windows Form icon.

For example, let's make a button (on Form1) so that when you click that button, Form2 suddenly appears and says "I love you". Here's how...

Start a new program (so you have a blank Form1).
Create Form2, by doing this:

Click the Add New Item button (which is near the screen's top, under the word "Edit"). Double click the Windows Form icon.

You see Form2. (It covers Form1). Make it say "I love you" (by typing "I love you" in the property list's Text box and pressing Enter).

Make Form1 reappear (by clicking the "Form1.vb [Design]" tab). On Form1, create a button (by double-clicking the Button tool). Make it say "Click me" (by typing "Click me" in the property list's Text box and pressing Enter). Double-click that button and type this subroutine line:

```
Form2.Visible = True
```

That means: when the button is clicked, make Form2 suddenly become visible.

When you run the program (by clicking "Start"), you see Form1, which contains a button saying "Click me". If you click that button, the computer displays Form2, which covers Form1 and says "I love you".

To stop running the program, close the Form2 window (by clicking its X button) then close the Form1 window (by clicking its X button).

Web browser

Here's how to make a form's middle show a Web page.

Create a blank form. (For a quick, fun experiment, you can use Form1, though in a practical program you'd use another form instead, such as Form2.) Make that blank form be maximized (by making its property list's WindowState line say "Maximized") or at least rather big (by making its property list's Size line have rather big numbers).

Double-click the **WebBrowser** tool. That makes the form's entire middle be devoted to the Web and be called WebBrowser1. In WebBrowser1's property list, click "**Url**" then type the Web address you want the form to show (such as "www.yahoo.com").

When you run the program (by clicking "Start"), the form's middle will show that Web page (or as much of it as will fit in the form's middle, accompanied by scroll arrows).

Timer

To make the computer pause, use the **Timer** tool. Here are examples.

I love you!!!!!!!!!! Here's how to make the computer say "I love you", then pause, then add an exclamation point (so you see "I love you!"), then pause, then add another exclamation point (so you see "I love you!!"), then keep repeating that process, so you eventually see "I love you!!!!!!!!!!" and beyond!

Make Form1's Text begin as "I love you" by doing this:

Click in Form1, so the screen's bottom right corner shows Form1's main property list. In that list, click "Text" then type "I love you", so the property list's Text line becomes this:

```
Text I love you
```

When you finish typing, press the Enter key. That makes Form1's title (top) say "I love you".

The next step is to say "add an exclamation point after pausing". To deal with pausing, you must use the Timer tool. Here's how...

Look at the toolbox (which is at the screen's left side and shows the tools). Using the toolbox's scroll-down arrow, scroll down until you see a heading called "Components".

Left of that heading, you see a triangle. That triangle should be solid black. (If the triangle has a white middle instead, click the triangle to make it solid black.)

Under the heading "Components", you should see the **Timer** tool. Double-click it. That puts a Timer1 icon below Form1.

At the screen's bottom-right corner, you see Timer1's property list, which looks like this:

Property	Value
(ApplicationSettings)	
(Name)	Timer1
Enabled	False
GenerateMember	True
Interval	100
Modifiers	Friend
Tag	

In that property list, click "**Interval**" then type 2000, so the Interval line becomes this:

Interval	2000
----------	------

That makes each pause be **2000 milliseconds** (which is 2000 "thousands of a second", which is 2 seconds).

In that property list, click "**Enabled**" then press the T key, so the Enabled line becomes this:

Enabled	True
---------	------

That turns the timer on, so it works.

Double-click the Timer1 icon, so you can write Timer1's subroutine. Type this line in Timer1's subroutine:

```
Text &= "!"
```

That makes the Text (of Form1) lengthen, by having an extra "!" added. Typing that line makes Timer1's subroutine become this:

```
Private Sub Timer1_Tick...  
    Text &= "!"  
End Sub
```

When you run the program (by clicking "Start"), the computer will show Form1 saying "I love you", then pause for the next clock tick (the interval between ticks being 2000 milliseconds), then do Timer1's subroutine (which turns "I love you" into "I love you!"), then pause for the next clock tick, then do again Timer1's subroutine (which turns "I love you!" into "I love you!!"), then pause for the next clock tick, then do again Timer1's subroutine (which turns "I love you!!" into "I love you!!!"), then keep repeating that process, so you eventually see "I love you!!!!!!!!!!" and beyond. When the exclamation points become too numerous to fit in Form1's title area, the computer changes the extra exclamation points to "...". The program keeps running until you stop it (by clicking Form1's X button).

If you want the exclamation points to come faster, make the interval shorter, by making Timer1's Interval be *less* than 2000 milliseconds. For example, try making the Interval be 1000 milliseconds (which is 1 second), or 500 milliseconds (which is half a second), or 1 millisecond (which is almost instantaneous).

If you want the computer to add just one exclamation point and then relax (without adding further exclamation points), make Timer1's subroutine become this:

```
Private Sub Timer1_Tick...  
    Text &= "!"  
    Timer1.Enabled = False  
End Sub
```

That subroutine says: when the clock ticks, add an exclamation point to the text but then **disable the timer**, so no further exclamation points will be added.

To play a joke on a human, make Timer1's Interval be 3000 (so the computer will pause 3 seconds before giving the joke's punch line) and make Timer1's subroutine become this:

```
Private Sub Timer1_Tick...
    Text &= "r mother!"
    Timer1.Enabled = False
End Sub
```

That subroutine says: when the clock ticks (after 3 seconds), make the Text change from "I love you" to "I love your mother!" then disable the timer (because the joke's timing is done).

Try making Timer1's subroutine become this instead:

```
Private Sub Timer1_Tick...
    Text &= "I'm happy when you're gone"
End Sub
```

That subroutine says: when the clock ticks (after 3 seconds), make the Text change from "I love you" to "I'm happy when you're gone".

Count the seconds Here's how to make Form1 count how many seconds have elapsed, so Form1 begins by saying 0, then a second later says 1, then a second later says 2, etc.

Make Form1's Text begin at 0 by doing this:

```
Click in Form1, so the screen's bottom right corner shows Form1's main property list. In that list, click "Text", then type number 0 and press Enter.
```

In the toolbox (which at the screen's left side and shows the tools), find the Timer tool (by scrolling down to "Components", clicking any + sign left of "Components", then scrolling down further). Double-click that Timer tool. That puts a Timer1 icon below Form1.

In Timer1's property list (which is at the screen's bottom-right corner), click "Interval" then type 1000, so you see this line:

```
Interval      1000
```

Click "Enabled" then press the T key, so the Enabled line becomes this:

```
Enabled      True
```

Double-click the Timer1 icon. Type this subroutine for Timer1:

```
Text += 1
```

That increases the text's number, by adding 1 to it.

When you run the program (by clicking "Start"), Form1's Text begins as 0 but increases to 1, then 2, then 3, etc.

Tell the date and time Here's how to make Form1 act as a clock, so it tells you the date and time and updates itself every second!

Create a Timer1 icon (by double-clicking the Timer tool, which is in the toolbox under "Components"). In Timer1's property list (which is at the screen's bottom-right corner), make the "Interval" be 1000 and make "Enabled" be True.

Double-click the Timer1 icon. Type this subroutine for Timer1:

```
Text = My.Computer.Clock.LocalTime
```

That makes the text become a message such as this:

```
12/31/2009 11:59:30 PM
```

You'll see such a message when you run the program (by clicking "Start"). Since you set the Interval to 1000 milliseconds (which is 1 second), that text will correct itself every second.

Switch to blue Here's how to make Form1 begin as red but then, after a pause, become blue.

Make Form1 begin as red by doing this:

```
Click in Form1, so the screen's bottom right corner shows Form1's main property list. In that list, click "BackColor" (which you'll see after you scroll up) then BackColor's down-arrow then "Web" then "Red" (which you'll see after you scroll down).
```

Create a Timer1 icon (by double-clicking the Timer tool, which is in the toolbox under "Components"). In Timer1's property list (which is at the screen's bottom-right corner), make the "Interval" be 2000 and make "Enabled" be True.

Double-click the Timer1 icon. Type this subroutine for Timer1:

```
BackColor = Color.Blue
```

When you run the program (by clicking "Start"), Form1 begins as red but switches to blue (after a delay of 2000 milliseconds, which is 2 seconds).

Let's make the subroutine fancier, so Form1 keeps alternating between red and blue. We'll make Form1 start as red, then switch to blue, then switch back to red, then switch back to blue, then switch back to red, etc., forever. To do that, change the subroutine line to this:

```
BackColor = IIf(BackColor = Color.Red, Color.Blue, Color.Red)
```

It says the BackColor becomes this: if the BackColor was Red, then it becomes Blue, else it becomes Red.

Color dialog

Here's how to let the human pick a color for Form1.

Look at the toolbox (which is at the screen's left side and shows the tools). Using the toolbox's scroll-down arrow, scroll down until you see a heading called "Dialogs".

Left of that heading, you see a square. That square should contain a minus sign. (If it contains a plus sign instead, change the plus sign to a minus sign by clicking it.)

Under the heading "Dialogs", you should see the **ColorDialog** tool. Double-click it. That puts a **ColorDialog1** icon below Form1 and lets Form1's subroutine mention "ColorDialog1".

Double-click Form1. Write this Form1 subroutine:

```
ColorDialog1.ShowDialog()
BackColor = ColorDialog1.Color
```

When you run the program (by clicking "Start"), the subroutine's top line makes the computer **show** the human the **color dialog box**, which contains 48 colors (plus a feature to let the human invent custom colors). When the human clicks one of the 48 colors (or a custom color) and then clicks "OK", the subroutine's bottom line makes Form1's background color become the color the human chose.

While viewing the color dialog box, here's how the human can create a custom color:

```
Click "Define Custom Colors".
```

At the color dialog box's right edge, you see a triangle pointing toward the left. Drag that triangle up, until it's halfway up the bar it points to.

You see a big, colorful square. Click your favorite color in that square.

Below that square, you see a box marked "Color/Solid"; that shows the color you've chosen. Adjust that color, by dragging the triangle up (which makes the color lighter) or dragging the triangle down (which makes the color darker) or clicking a different spot in the big, colorful square.

When you're satisfied, click "Add to Custom Colors". That creates a small square for the color. Click that square then "OK".

Helpful hints

Here are some hints to help you master programming.

Stop debugging

While your program is running, you can interrupt it by clicking the **Stop Debugging** button (which is a red square at the screen's top center).

The computer refuses to let you edit a program that's in the middle of running. If you try to edit a program that's running, the computer gripes by saying —

```
Changes are not allowed while code is running.
```

then waits for you to click "OK" (which means you've read the gripe).

To edit a program that's running, stop it first (by clicking Form1's X or the Stop Debugging button) then try to edit your program.

Avoiding Dim

If *x* is a variable, you're supposed to warn the computer by saying:

```
Dim x
```

If you're too lazy to say "Dim" for each variable, say **Option Explicit Off** at your program's top, so your program looks like this:

```
Option Explicit Off
Public Class Form1
    Private Sub Form1_Load...

    End Sub
End Class
```

To type "Option Explicit Off" up there, do this:

```
While holding down the Ctrl key, tap the Home key.
Press the Enter key.
Press the up-arrow key.
Type "Option Explicit Off".
```

The "Option Explicit Off" prevents the computer from griping about missing Dim lines. It makes your program's variables work even if you don't say "Dim". But it also prevents the computer from warning you about using variables in ridiculous ways. Say "Option Explicit Off" just if you're too lazy to say "Dim" — and you're sure you're not making ridiculous mistakes about variables.

Apostrophe

In your subroutine, you can **type comments to help programmers understand your program**. The comments can mention your name, the date you wrote the program, the program's purpose, the purpose of each variable, special tricks you used, cynical comments, and any other comments you'd like to share with your programming buddies and to remind yourself of how you've been thinking.

To type such a comment in your subroutine, **begin the comment with an apostrophe**, like this:

```
'This subroutine is another dumb example by Russ.
'It was written on Halloween, under a full moon.
c = 40 'because Russ has 40 computers
h = 23 'because 23 of his computers are haunted
Text = c - h 'That many computers are unhaunted.
```

When you run the program, **the computer ignores everything that's to the right of an apostrophe**. So the computer ignores lines 1 and 2; in lines 3 & 4, the computer ignores the "because..."; in the bottom line, the computer ignores the comment about being unhaunted. Since *c* is 40, and *h* is 23, the bottom line makes the computer say:

```
17
```

Everything to the right of an apostrophe is called a **comment** (or **remark**).

Turning green When you type the subroutine, **the computer makes each apostrophe and comment turn green**. Then the computer ignores what's green.

Temporarily ignore Suppose you've written a subroutine but wonder what would happen if one of the lines were deleted. To find out, you could delete the line (by pressing the Delete key repeatedly or using other techniques), then run the shortened program, then put the line back in (by retyping it). But here's a faster way to do that experiment:

To temporarily make the computer ignore the line, type an apostrophe in front of that line. The apostrophe turns that line into a comment, so the computer ignores the line. Later, when you want to reactivate that line, just delete the apostrophe.

Temporarily deactivating a line (by putting an apostrophe before it so it becomes a comment) is called **commenting out** the line.

Multiple lines To make *several* lines become comments, you can type an apostrophe in front of each of those lines; but here's a faster way: drag across those lines (by using your mouse), then click the **Comment-out button** (which is near the screen's top, under "Tools", and shows two green lines between black lines).

That makes the computer type an apostrophe in front of each of those lines and makes the lines turn green.

Later, when you want to reactivate those lines, drag across them again then click the **Uncomment button** (which is to the right of the Comment-out button): that removes the apostrophes and makes the lines turn black again.

Places for output

You can make the computer show answers in many places. Let's review the places you saw previously, then explore places that are more exotic.

Top of Form1

You learned that if Form1's subroutine says —

```
Text = 5 + 2
```

the computer writes the answer, 7, at the **top of Form1**, where Form1's **title** belongs.

Unfortunately, the top of Form1 doesn't have much space: the answer must be narrow (unless you widened Form1) and the answer must not consume 2 lines.

Pop-up window

If the subroutine says —

```
MsgBox(5 + 2)
```

the computer writes the answer in a **pop-up window** that appears suddenly.

If the answer is long, the pop-up window expands automatically, vertically and horizontally, to hold the answer. (To create a 2-line answer, say this where you want the computer to press the Enter key: & Chr(13) &.)

Afterwards, the computer waits for the human to click “OK”.

Middle of Form1

If you’ve created a label (by double-clicking the Label tool) and your subroutine says —

```
Label1.Text = 5 + 2
```

the computer writes the answer in the middle of Form1, where Label1 is.

If the answer is long, the Label1 area expands automatically, vertically and horizontally, to hold the answer, up to the size of Form1. (To create a 2-line answer, say this where you want the computer to press the Enter key: & Chr(13) &.)

Output window

If the subroutine says —

```
Console.WriteLine(5 + 2)
```

or —

```
Debug.Print(5 + 2)
```

the computer writes the answer at the screen’s bottom, at the bottom of the **output window**.

After you admire that answer, stop the program (by clicking Form1’s X button or the Stop Debugging button (which is a red square at the screen’s top center).

Immediate window

To create an **immediate window**, click “Debug” (which is at the screen’s top) then “Windows” then “Immediate”. Then you see an **immediate window** at the screen’s bottom.

In that window, type a command such as:

```
>? 5+2
```

Type it correctly: begin with the symbol >, then a question mark, then a space, then the computation. When you finish typing that, press the Enter key. The computer immediately types the answer underneath:

```
7
```

Console screen

To create a new program normally, you click “Windows Forms Application”.

Instead of clicking “Windows Forms Application”, try clicking “**Console Application**”. That tells the computer you want a stripped-down version of Visual Basic, where the computer writes answers on a **console screen** (which looks like DOS instead of Windows and has no forms or buttons or icons).

When you’ve double-clicked the Console Application icon, you immediately see this stripped-down subroutine:

```
Module Module1

    Sub Main()

    End Sub

End Module
```

Click in the middle of that subroutine and say “Console.WriteLine(5 + 2)”, so the subroutine becomes this:

```
Module Module1

    Sub Main()
        Console.WriteLine(5 + 2)
    End Sub

End Module
```

Just above the “End Sub”, say “Do” and “Loop”, like this:

```
Module Module1

    Sub Main()
        Console.WriteLine(5 + 2)
        Do
            Loop
    End Sub

End Module
```

When you type the “Do” (and press Enter), the computer automatically types the “Loop” for you.

When you run the program (by clicking “Start”), the computer writes the answer on a **console screen**, which is a window whose middle looks like DOS. (To create a 2-line answer, give 2 Console.WriteLine commands. At the end of each answer, the computer presses the Enter key, unless you say **Write** instead of **WriteLine**.) Your subroutine’s “Do” and “Loop” make the computer pause, so you have time to read the answer.

After you’ve read the answer, close the console screen (by clicking its X button).

Avoiding Do In the subroutine, instead of typing “Do” (and waiting for the computer to type “Loop”), you can type “Console.ReadKey()”, so the subroutine looks like this:

```
Module Module1

    Sub Main()
        Console.WriteLine(5 + 2)
        Console.ReadKey()
    End Sub

End Module
```

The “Console.ReadKey()” makes the computer wait for the human to press a key. When the human presses any key (such as Enter), the computer ends the program and closes the console screen.

Avoiding Console Here’s a shortcut. Instead of typing “Console.” so often (before each WriteLine and Write and ReadKey), just type this line at the subroutine’s top —

```
Imports System.Console
```

so the subroutine looks like this:

```
Imports System.Console
Module Module1

    Sub Main()
        WriteLine(5 + 2)
        ReadKey()
    End Sub

End Module
```

Print form

Here’s how to let the human print Form1 onto paper.

Double-click the **PrintForm** tool, which is in the Visual Basic PowerPacks category (whose tools you can see by clicking the triangle left of “Visual Basic PowerPacks” once or twice). That puts a PrintForm1 icon below Form1 and lets subroutines mention “PrintForm1”.

On Form1, create a button (by double-clicking the Button tool). Make the button’s text say “Print” (by clicking “Text” then typing “Print”). Double-click the button, so you can write the button’s subroutine. Make the button’s subroutine say:

```
PrintForm1.Print()
```

When the human runs the program and clicks the Print button, the computer will print most of Form1 onto paper.

The computer doesn’t bother printing Form1’s border or Text (title). It prints just Form1’s middle, including the objects in it.

Hard disk

If Form1's subroutine says —

```
My.Computer.FileSystem.WriteAllText("Joan.txt", 5 + 2, False)
```

the computer writes the number 7 (the answer to 5+2) onto your hard disk, in a file called Joan.txt. Unfortunately, that file is hard to access, since it's buried in folders. (Specifically, it's in the Debug folder, which is in the bin folder, which is in your program's inner folder, which in your program's outer folder, which is in the Projects folder, which is in the Visual Studio 2015 folder, which is in the Documents folder.)

If you don't like the name Joan, invent a different name instead.

ProgramData folder This line is more practical:

```
My.Computer.FileSystem.WriteAllText("\ProgramData\Joan.txt", 5 + 2, False)
```

It makes the computer write the number 7 (the answer to 5 + 2) onto your hard disk, in a file called Joan.txt, which is in the **ProgramData folder**.

After you've run the program (by clicking "Start"), you can see Joan.txt by doing this:

Click the File Explorer icon (which is at the screen's bottom, on the taskbar, and looks like a yellow manila folder) then "This PC" (which is at the left). Double-click "C:" then "ProgramData" then "Joan", whose hidden .txt ending makes the computer run the Notepad program, which shows you what's in Joan.txt. You see that Joan.txt contains the answer, 7. When you finish admiring the answer, close the front 2 windows (by clicking their X buttons).

Documents folder If you want Joan.txt to be in the **Documents folder** instead of the ProgramData folder, type these lines instead:

```
Dim doc
doc = My.Computer.FileSystem.SpecialDirectories.MyDocuments
My.Computer.FileSystem.WriteAllText(doc & "\Joan.txt", 5 + 2, False)
```

The top two lines makes the variable doc stand for the Documents folder. In the bottom line, the doc makes Joan.txt be in the Documents folder.

After you've run the program (by clicking "Start"), you can see Joan.txt by doing this:

Click the File Explorer icon (which is at the screen's bottom, on the taskbar, and looks like a yellow manila folder) then "Documents". Double-click "Joan", whose hidden .txt ending makes the computer run the Notepad program, which shows you what's in Joan.txt. You see that Joan.txt contains the answer, 7. When you finish admiring the answer, close the front 2 windows (by clicking their X buttons).

Append If Joan.txt exists before you run the subroutine, the subroutine erases that old Joan.txt to create a Joan.txt — unless you change the bottom line's "False" to "True", which makes the subroutine **append** the new answer to the end of the old Joan.txt, to make Joan.txt become longer and include *both* answers. In the bottom line, "True" means "append to the old file"; "False" means "don't append to the old file; erase the old file."

Reading Joan.txt After the computer has put an answer into Joan.txt, you can run a **reading program** that reads Joan.txt.

If Joan.txt is in the ProgramData folder, the reading program can have Form1's subroutine say:

```
Text = My.Computer.FileSystem.ReadAllText("\ProgramData\Joan.txt")
```

That line makes the computer read Joan.txt and tell you what answer Joan.txt contains.

If Joan.txt is in the Documents folder, the reading program can have Form1's subroutine say:

```
Dim doc
doc = My.Computer.FileSystem.SpecialDirectories.MyDocuments
Text = My.Computer.FileSystem.ReadAllText(doc & "\Joan.txt")
```

Those lines make doc be the Documents folder and make Text be the answer that the computer reads from doc's Joan.txt.

Rich-text box Try this experiment...

Create a new program. On Form1, put a rich-text box (so the human can type a document into the box) and put a Save button (a button whose title is "Save").

To make the Save button work properly (so pressing it copies the human's typing from the box to the hard disk), make the button's subroutine be this:

```
Dim doc
doc = My.Computer.FileSystem.SpecialDirectories.My Documents
RichTextBox1.SaveFile(doc & "\Joan.doc")
```

The bottom line means: take what's in the rich-text box and save it as a file; put the file into the Documents folder and call it Joan.doc.

When the human runs that program, the computer will let the human type a document (essay) into the rich-text box. Then computer will wait for the human to click the Save button (which makes the computer copy the document to the hard disk's Documents folder, in a rich-text file called Joan.doc).

You can see Joan.doc by doing this:

Click the File Explorer icon (the yellow manila folder at the screen's bottom) then "Documents". Double-click "Joan", whose hidden .doc ending makes the computer run the Microsoft Word (or WordPad) program, which shows you what's in Joan.doc. You see that Joan.doc contains the essay. When you finish admiring the essay, close the front 2 windows (by clicking their X buttons).

"Save As" dialog box That subroutine forces the document to be in the Documents folder and be called "Joan.doc". Here's how to make the subroutine more flexible, (so the human can choose what folder to put the document in and what name to give the document...

Double-click the **SaveFileDialog** tool (which is in the Dialogs category). That creates a **SaveFileDialog1** icon below Form1.

In the SaveFileDialog1's property list, click "DefaultExt" and type ".doc". That will secretly put ".doc" at the end of every filename.

Make the Save button's subroutine be this:

```
SaveFileDialog1.ShowDialog()  
RichTextBox1.SaveFile(SaveFileDialog1.FileName)
```

When the human clicks the Save button, the subroutine's top line makes the computer show the "Save As" dialog box, which lets the human invent a file name and choose a folder to put it in. For example, if the human types "Joe" (and then presses the Enter key), the file will be called "Joe.doc" (because the SaveFileDialog1's property list said the default extension is "doc").

The subroutine's bottom line means: look at the document that was typed in RichTextBox1, and save it as a file on the hard disk, using the file name (and folder) that the human specified in the "Save As" dialog box.

Say "document" Since RichTextBox1's main purpose is to handle a document, programmers prefer to say just "document" instead of "RichTextBox1" and write the Save button's program this way:

```
SaveFileDialog1.ShowDialog()  
document.SaveFile(SaveFileDialog1.FileName)
```

To do that, you must change the box's name from "RichTextBox1" to "document". Here's how:

In RichTextBox1's property list (which is at the screen's bottom-right corner), click "(Name)" then type the word "document", so the line looks like this:

```
(Name)          document
```

Reading Joan.doc After the computer has saved (copied) a document into your hard disk's Joan.doc, you can run a **reading program** that reads Joan.doc.

The reading program should have a rich text box named "document". It should also have an "Open" button whose subroutine says:

```
OpenFileDialog1.ShowDialog()  
document.LoadFile(OpenFileDialog1.FileName)
```

But to make the computer understand what "OpenFileDialog1" means, you must double-click the **OpenFileDialog** tool before typing that subroutine.

Menu

You can create a menu.

Menu bar

At the top of Form1, let's create this **menu bar**:

```
Love Hate
```

Let's program the computer so clicking "Love" makes the computer say "I love you", and clicking "Hate" makes the computer say "I hate being a computer".

Here's how to accomplish all that...

Double-click the **MenuStrip** tool (which is in the "Menus & Toolbars" category).

Click "Type Here" (which is near Form1's top). Then you see a blank box (plus two "Type Here" boxes). In the blank box, type your menu's first word ("Love").

Click the box that's to the right of "Love". Type your menu's second word ("Hate").

Congratulations! You created a menu!

Create menu subroutines Double-click "Love", then write this subroutine telling the computer what to do if "Love" is clicked:

```
Private Sub LoveToolStripMenuItem_Click...  
    Text = "I love you"  
End Sub
```

(The computer already typed the top and bottom lines for you, so type just the middle line.) When you finish typing that line, click the "Form1.vb [Design]" tab.

Double-click "Hate", then write this subroutine about clicking "Hate":

```
Private Sub HateToolStripMenuItem_Click...  
    Text = "I hate being a computer"  
End Sub
```

Run the program Go ahead: run the program (by clicking "Start"). You see the menu bar you created:

```
Love Hate
```

Clicking "Love" makes the computer say "I love you"; clicking "Hate" makes the computer say "I hate being a computer".

Pull-down menu

Let's expand the menu by adding "Color", so the menu becomes this:

```
Love Hate Color
```

Let's program the computer so clicking "Color" makes this **pull-down menu** appear under Color:

```
Yellow  
Red
```

Let's program so clicking one of those colors makes Form1's background be that color.

Here's how to accomplish all that...

Create a new menu item If your program is still running, stop it (by clicking its X button). Look at Form1's design (by clicking the "Form1.vb [Design]" tab). Click the "Type Here" that's to the right of "Hate". In the blank box that appears, type your menu's third word ("Color").

Create a pull-down menu To create Color's pull-down menu (saying "Yellow" and "Red"), click the "Type Here" that's under "Color". In the blank box that appears, type "Yellow". In the box under "Yellow", type "Red".

Congratulations! You created a pull-down menu!

Create menu subroutines Double-click "Yellow", then write this subroutine about Yellow:

```
Private Sub YellowToolStripMenuItem_Click...  
    BackColor = Color.Yellow  
End Sub
```

Click the "Form1.vb [Design]" tab, then double-click "Red", then write this subroutine about Red:

```
Private Sub RedToolStripMenuItem_Click...  
    BackColor = Color.Red  
End Sub
```

Run the program Go ahead: run the program (by clicking "Start"). You see the menu bar you created:

```
Love Hate Color
```

Clicking "Color" makes the computer show Color's pull-down menu; clicking the "Yellow" or "Red" makes Form1's background turn that color.

Submenu

Let's expand Color's pull-down menu by adding "Blue", so the menu becomes this:



Let's program the computer so clicking "Blue" makes this **submenu** appear to the right of Blue:



Let's program so clicking one of those kinds of blue makes Form1's background be that color.

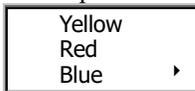
Here's how to accomplish all that...

Create a new menu item If your program is still running, stop it (by clicking its X button). Look at Form1's design (by clicking the "Form1.vb [Design]" tab). Click "Color" then the "Type Here" that's under "Red". In the blank box that appears, type pull-down menu's third word ("Blue").

Create a submenu To create Blue's submenu (saying "Light Blue" and "Dark Blue"), click the "Type Here" that's to the right of "Blue". In the blank box that appears, type "Light Blue". In the box under "Light Blue", type "Dark Blue".

Congratulations! You created a submenu!

Creating a submenu for Blue made a right-arrow appear next to "Blue", so Color's pull-down menu looks like this:



That right-arrow means "has a submenu".

Create subroutines Double-click "Light Blue", then write this subroutine about Light Blue:

```
Private Sub LightBlueToolStripMenuItem_Click...
    BackColor = Color.LightBlue
End Sub
```

Click the "Form1.vb [Design]" tab, then double-click "Dark Blue", then write this subroutine about Dark Blue:

```
Private Sub DarkBlueToolStripMenuItem_Click...
    BackColor = Color.DarkBlue
End Sub
```

Run the program Go ahead: run the program (by clicking "Start"). You see the menu bar you created:

Love Hate Color

Clicking "Color" makes the computer show Color's pull-down menu; clicking "Blue" makes the computer show Blue's submenu; then clicking "Light Blue" or "Dark Blue" makes Form1's background turn that color.

Rearranging menu items

After you've created a menu, you can rearrange its items. Here's how...

If your program is still running, stop it (by clicking its X button). Look at Form1's design (by clicking the "Form1.vb [Design]" tab).

To delete an item, click it then press the **Delete key**. If you change your mind, click the **Undo button** (which shows a blue arrow bending toward the left).

To move an item that's on the menu bar ("Love", "Hate", or "Color"), **drag** that item across to where you want it — and, to make sure the computer doesn't ignore you, drag slightly farther. To move an item that's on a pull-down menu ("Yellow", "Red", or "Blue") or submenu ("Light Blue" or "Dark Blue"), drag the item up or down to where you want it — and to make sure the computer doesn't ignore you, drag slightly farther.

Minimalist word processor

Here's how to invent a minimalist word-processing program.

Big Form1

Create a new program. Widen Form 1 (by dragging its bottom-right corner toward the right).

Tool strip

Onto Form 1, put a **tool strip (toolbar)** by doing this:

Double-click the **ToolStrip** tool (which is in the "Menus & Toolbars" category). That puts a ToolStrip1 icon below Form 1. Right-click that icon then click "Insert Standard Items". That makes these 7 icons appear across Form 1's top: New, Open, Save, Print, Cut, Copy, Paste, and Help. Each icon will act as a button.

Rich text box

Onto Form1, put a rich text box (by double-clicking the **RichTextBox** tool). Give that box the desired properties by doing this:

Click the box's right-arrow (which is near the box's top-right corner) then "Dock in parent container". That makes the box expand to fill the rest of Form1: the only things above the box are the tool strip and the title bar (which says Form1).

In the box's property list (which is at the screen's bottom-right corner), scroll up until you see "EnableAutoDragDrop", then click "EnableAutoDragDrop" and press the T key, so the line becomes this:

EnableAutoDragDrop True

Scroll up farther until you see "(Name)", then click "(Name)" and type "document", so the line becomes this:

(Name) document

More tools

Double-click these tools, which you'll need to finish the program:

OpenFileDialog (which is in the "Dialogs" category)

SaveFileDialog (which is in the "Dialogs" category)

PrintForm (which is in the "Visual Basic PowerPacks" category)

Then icons for those tools appear below Form1.

Subroutines

For each button on the tool strip, write a subroutine. Here's how...

Double-click the tool strip's first button (the **New button**, which looks like a blank sheet of paper with a folded corner). Type this line (for the New button's subroutine):

```
document.Clear()
```

Make Form1 appear again (by clicking the "Form1.vb [Design]" tab). Double-click the tool strip's next button (the **Open button**, which looks like a yellow manila folder that's opening). Type these lines (for the Open button's subroutine):

```
OpenFileDialog1.ShowDialog()
document.LoadFile(OpenFileDialog1.FileName)
```

Make Form1 appear again (by clicking the "Form1.vb [Design]" tab). In similar fashion, type these lines for the Save button:

```
SaveFileDialog1.ShowDialog()
document.SaveFile(SaveFileDialog1.FileName)
```

Type this line for the Print button:

```
PrintForm1.Print()
```

Type this line for the Cut button:

```
document.Cut()
```

Type this line for the Copy button:

```
document.Copy()
```

Type this line for the Paste button:

```
document.Paste()
```

Type this line for the Help button:

```
MsgBox("This is word processor version 1")
```

Run

When you run the program (by clicking “Start”), the program works correctly, if you did what I said!

Congratulations on creating a word-processing program.

The program’s main limitations are:

It doesn’t let you change margins (except by dragging Form1’s bottom-right corner).

It doesn’t let you change fonts.

Its Print button prints just *part* of the document. (It prints just the part that’s visible on Form1 at the moment, and it can’t print any part that’s too far to the right to fit on the paper.)

Surpassing those limitations would require subroutines that are much longer!

Loops

Here’s how to make the computer repeat.

Do...Loop

The computer can be religious. Just make Form1’s subroutine say this:

```
MsgBox("I worship your feet")
MsgBox("But please wash them")
```

When you run the program, the computer shows a message box saying “I worship your feet” and waits for the human to click OK. Then the computer shows a message box saying “But please wash them” (and waits for the human to click OK again).

To make the computer do the lines many times, say “Do” above the lines and say “Loop” below them, so the subroutine looks like this:

```
Do
    MsgBox("I worship your feet")
    MsgBox("But please wash them")
Loop
```

The lines being repeated (the MsgBox lines) should be between the words Do and Loop and indented. (After you’ve typed the word “Do” and pressed Enter, the computer will automatically type the word “Loop” and created an indented blank space for you to type in.)

Run the program (by clicking “Start”). The computer says “I worship your feet” (and waits for the human to click OK), then says “But please wash them” (and waits for OK), then goes back and says “I worship your feet” again (and waits for OK), then says “But please wash them” again (and waits for OK), then goes back and says the same stuff again, and again, and again, and again, forever.

Since the computer’s thinking keeps circling back to the same lines, the computer is said to be in a **loop**. In that subroutine, the **Do** means “do what’s underneath and indented”; the **Loop** means “loop back and do it again”. The lines that say Do and Loop — and the lines between them — form a loop, which is called a **Do loop**.

The computer does that loop repeatedly, forever — or until you **abort the program by doing this**:

Click the **Stop Debugging button** (a blue square near the screen’s top center).

That works just if you’re in the Visual Basic environment (so you see the Stop Debugging button). If you’re *not* in the Visual Basic environment (because you’re running the .exe file directly), the only way to abort a looping program is to shut down the computer (click the Start button then, in Windows 7, click Shutdown) or try this:

While holding down the Ctrl and Alt keys, tap the Delete key. Click “Start Task Manager” then the “Applications” tab (which is at the screen’s top-left corner) then your program’s name then “End Task”. If you’re lucky, that aborts the program. Close the Windows Task Manager window (by clicking its X button).

In that program, since the computer tries to go round and round the loop forever, the loop is called **infinite**. The only way to stop an infinite loop is to abort it.

Disappearing - message - box bug

When running a loop, the computer might accidentally **lose the program’s focus** and forget to show the message box. To make the message box reappear, click the message box’s button, which is on the **taskbar**. (The **taskbar** is at the screen’s bottom and runs from the Start button to the clock.) Try double-clicking the message box’s button. To run the program again, try clicking the green right-arrow (instead of pressing the F5 key).

GoTo

Instead of typing —

```
Do
    MsgBox("I worship your feet")
    MsgBox("But please wash them")
Loop
```

you can type:

```
joe:    MsgBox("I worship your feet")
        MsgBox("But please wash them")
        GoTo joe
```

(When you type that subroutine, the computer automatically spaces it correctly: when you press Enter at the top line’s end, the computer automatically unindents “joe:.”) The top line (named joe) makes the computer say “I worship your feet”. The next line makes the computer say “But please wash them”. The bottom line makes the computer Go back To the line named joe, so the computer forms a loop. The computer will loop forever — or until you abort the program (by clicking the Stop Debugging button, twice).

You can give a line a short name (such as joe) or a long name (such as BeginningOfMyFavoriteLoop). The name can even be a number (such as 10). Put the name at the line’s beginning. After the name, put a colon (the symbol “:”).

The line’s name (such as joe or BeginningOfMyFavoriteLoop or 10) is called the line’s **label**.

Skip ahead This subroutine is insulting:

```
MsgBox("Your face is outstanding.")
MsgBox("It belongs in a horror movie.")
MsgBox("It deserves an award!")
```

Let's turn that insult into a compliment. To do that, insert the shaded items:

```
MsgBox("Your face is outstanding.")
GoTo conclusion
MsgBox("It belongs in a horror movie.")
conclusion: MsgBox("It deserves an award!")
```

The computer begins by saying "Your face is outstanding." Then the computer does GoTo conclusion, which makes the computer Go skip down To the conclusion line, which says "It deserves an award!" So the subroutine makes the computer say just —

```
Your face is outstanding.
```

and:

```
It deserves an award!
```

Is GoTo too powerful? Saying GoTo gives you great power: if you make the computer GoTo an earlier line, you'll create a loop; if you make the computer GoTo a later line, the computer will skip over several lines of your subroutine.

Since saying GoTo is so powerful, programmers are afraid to say it. Programmers know that the slightest error in saying GoTo will make a program act very bizarre! Programmers feel more comfortable using milder words instead (such as Do...Loop), which are safer and rarely get botched up. Since saying GoTo is scary, many computer teachers prohibit students from using it, and many companies fire programmers who say GoTo instead of Do...Loop.

But saying GoTo is fine when you've learned how to control the power! Though I usually say Do...Loop instead of GoTo, I say GoTo in certain situations where saying Do...Loop would be awkward.

Exiting a Do loop

Let's create a guessing game, where the human tries to guess the computer's favorite color, which is pink. To do that, say **GoTo** or **Exit Do** or **Loop Until**. Here's how....

GoTo Just make Form1's subroutine say this:

```
Dim guess
AskTheHuman: guess = InputBox("what's my favorite color?")
If guess = "pink" Then
    MsgBox("Congratulations! You discovered my favorite color.")
Else
    MsgBox("No, that's not my favorite color. Try again!")
    GoTo AskTheHuman
End If
```

The top line (which is called AskTheHuman) asks the human to guess the computer's favorite color.

If the guess is "pink", the computer says:

```
Congratulations! You discovered my favorite color.
```

But if the guess is *not* pink, the computer will instead say "No, that's not my favorite color" and then Go back To AskTheHuman again to guess the computer's favorite color.

Exit Do Here's how to write that subroutine without saying GoTo:

```
Dim guess
Do
    guess = InputBox("what's my favorite color?")
    If guess = "pink" Then Exit Do
    MsgBox("No, that's not my favorite color. Try again!")
Loop
MsgBox("Congratulations! You discovered my favorite color.")
```

The Do loop makes the computer do this repeatedly: ask "What's my favorite color?" and then say "No, that's not my favorite color."

The only way to stop the loop nicely (without abortion) is to guess "pink", which makes the computer Exit from the Do loop; then the computer proceeds to the line underneath the Do loop. That line makes the computer say:

```
Congratulations! You discovered my favorite color.
```

Loop Until Here's another way to program the guessing game:

```
Dim guess
Do
    MsgBox("You haven't guessed my favorite color yet!")
    guess = InputBox("what's my favorite color?")
Loop Until guess = "pink"
MsgBox("Congratulations! You discovered my favorite color.")
```

The Do loop makes the computer do this repeatedly: say "You haven't guessed my favorite color yet!" and then ask "What's my favorite color?"

The Loop line makes the computer repeat the indented lines again and again, until the guess is "pink". When the guess is "pink", the computer proceeds to the line underneath the Loop and prints "Congratulations!"

The Loop Until's condition (guess = "pink") is called the **loop's goal**. The computer does the loop repeatedly, until the loop's goal is achieved. Here's how:

The computer does the indented lines, then checks whether the goal is achieved yet. If the goal is *not* achieved yet, the computer does the indented lines again, then checks again whether the goal is achieved. The computer does the loop again and again, until the goal is achieved. Then the computer, proud at achieving the goal, does the program's **finale**, which consists of any lines under the Loop Until line.

Saying —

```
Loop Until guess = "pink"
```

is just a briefer way of saying this pair of lines:

```
If guess = "pink" Then Exit Do
Loop
```

For...Next

Let's make the computer say these sentences:

```
I like the number 1
I like the number 2
I like the number 3
I like the number 4
I like the number 5
```

To do that, put these lines into Form1's subroutine:

```
For x = 1 To 5
    MsgBox("I like the number " & x)
Next
```

The top line (For x = 1 To 5) says that x will be every number from 1 to 5; so x will be 1, then 2, then 3, then 4, then 5. The line underneath (which the computer indents) says what to do about each x: it says to create a message box saying "I like the number" and x.

Whenever a subroutine says the word For, it must also say Next; so the bottom line says Next. The computer types the word "Next" for you automatically.

The indented line, which is between the For line and the Next line, is the line that the computer will do repeatedly; so the computer will repeatedly say "I like the number" and an x. The first time, the x will be 1, so the computer will say:

```
I like the number 1
```

The next time, the x will be 2, so the computer will say:

```
I like the number 2
```

The computer will say similar sentences, for every number from 1 up to 5.

Monster song

Let's make the computer say these lyrics:

```
I saw 2 monsters
Tra-la-la!
I saw 3 monsters
Tra-la-la!
I saw 4 monsters
Tra-la-la!
They all had a party: ha-ha-ha!
```

To do that, type these lines —

```
The first line of each verse: MsgBox("I saw " & x & " monsters")
The second line of each verse: MsgBox("Tra-la-la!")
```

and make x be every number from 2 up to 4:

```
For x = 2 To 4
    MsgBox("I saw " & x & " monsters")
    MsgBox("Tra-la-la!")
Next
```

At the end of the song, say the closing line:

```
For x = 2 To 4
    MsgBox("I saw " & x & " monsters")
    MsgBox("Tra-la-la!")
Next
MsgBox("They all had a party: ha-ha-ha!")
```

That program makes the computer print the entire song.

Here's an analysis:

```
For x = 2 To 4
    MsgBox("I saw " & x & " monsters")
    MsgBox("Tra-la-la!")
Next
MsgBox("They all had a party: ha-ha-ha!")
```

The computer will do indented lines repeatedly, for x=2, x=3, and x=4. Then the computer will do this once.

Since the computer does the indented lines repeatedly, those lines form a loop. Here's the general rule: **the statements between For and Next form a loop.** The computer goes round and round the loop, for x=2, x=3, x=4, and x=5. Altogether, it goes around the loop 4 times, which is a finite number. Therefore, the loop is **finite**.

If you don't like the letter x, choose a different letter. For example, you can choose the letter i:

```
For i = 2 To 4
    MsgBox("I saw " & i & " monsters")
    MsgBox("Tra-la-la!")
Next
MsgBox("They all had a party: ha-ha-ha!")
```

When using the word For, most programmers prefer the letter i; most programmers say "For i" instead of "For x". Saying "For i" is a tradition. Following that tradition, the rest of this book says "For i" (instead of "For x"), except in situations where some other letter feels more natural.

Say the squares To find the **square** of a number, multiply the number by itself. For example, the square of 3 is "3 times 3", which is 9. The square of 4 is "4 times 4", which is 16.

Let's make the computer say the square of 3, 4, 5, etc., up to 10, like this:

```
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
```

To do that, type this line —

```
MsgBox("The square of " & i & " is " & i * i)
```

and make i be every number from 3 up to 10, like this:

```
For i = 3 To 10
    MsgBox("The square of " & i & " is " & i * i)
Next
```

Count how many copies This program makes the computer say “I love you” 4 times:

```
For i = 1 To 4
    MsgBox("I love you")
Next
```

Here’s a smarter program, which asks how many times you want the computer to say “I love you”:

```
Dim n
n = Val(InputBox("How many times do you want me to love you?"))
For i = 1 To n
    MsgBox("I love you")
Next
```

When you run that program, the computer asks:

How many times do you want me to love you?

If you answer 5 (and click the OK button), the n becomes 5 (so the computer says “I love you” 5 times). If you answer 7 instead, the computer says “I love you” 7 times. Get as much love as you like!

That program illustrates this rule:

To make the For...Next loop be flexible, say “For i = 1 To n” and let the human input the n.

Step The For statement can be varied:

Statement	Meaning
For i = 5 To 17 Step .1	The i will go from 5 to 17, counting by tenths. So i will be 5, then 5.1, then 5.2, etc., up to 17.
For i = 5 To 17 Step 3	The i will be every 3 rd number from 5 to 17. So i will be 5, then 8, then 11, then 14, then 17.
For i = 17 To 5 Step -3	The i will be every 3 rd number from 17 down to 5. So i will be 17, then 14, then 11, then 8, then 5.

To count down, you *must* use the word Step. To count from 17 down to 5, give this instruction:

```
For i = 17 To 5 Step -1
```

This program prints a rocket countdown:

```
For i = 10 To 1 Step -1
    MsgBox(i)
Next
MsgBox("Blast off!")
```

The computer will say:

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```

This statement is tricky:

```
For i = 5 To 16 Step 3
```

It says to start i at 5, and keep adding 3 until it gets past 16. So i will be 5, then 8, then 11, then 14. The i won’t be 17, since 17 is past 16. The first value of i is 5; the last value is 14.

In the statement For i = 5 To 16 Step 3, the **first value** or **initial value** of i is 5, the **limit value** is 16, and the **step size** or **increment** is 3. The i is called the **counter** or **index** or **loop-control variable**. Although the limit value is 16, the **last value** or **terminal value** is 14.

Programmers usually say “For i”, instead of “For x”, because the letter i reminds them of the word **index**.

Fancy calculations

The computer can do fancy calculations.

Exponents

In Form1’s subroutine, try giving this command:

```
Text = 4 ^ 3
```

To type the symbol ^, do this: while holding down the Shift key, tap this key:

```
^
6
```

That symbol (^) is called a **caret**.

In that line, the “4 ^ 3” makes the computer use the number 4, three times. The computer will multiply together those three 4’s, like this: 4 times 4 times 4. Since “4 times 4 times 4” is 64, the computer will say:

```
64
```

In the expression “4 ^ 3”, the 4 is called the **base**; the 3 is called the **exponent** or **power**.

Here’s another example:

```
Text = 10 ^ 6
```

The “10 ^ 6” makes the computer use the number 10, six times. The computer will multiply together those six 10’s (like this: 10 times 10 times 10 times 10 times 10 times 10) and say the answer, 1000000.

Here’s another example:

```
Text = 3 ^ 2
```

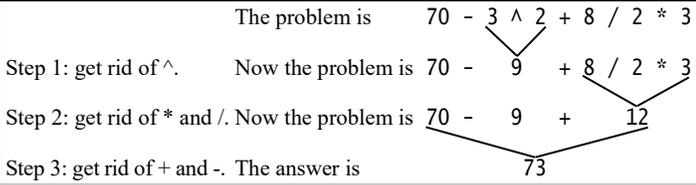
The “3 ^ 2” makes the computer use the number 3, two times. The computer will multiply together those two 3’s (like this: 3 times 3) and say the answer, 9.

Order of operations The symbols +, -, *, /, and ^ are all called **operations**.

To solve a problem, the computer uses the three-step process taught in algebra (and pre-algebra). For example, suppose you say:

```
Text = 70 - 3 ^ 2 + 8 / 2 * 3
```

The computer will *not* begin by subtracting 3 from 70; instead, it will use the three-step process:



In each step, it looks from left to right. For example, in step 2, it sees / and gets rid of it before it sees *.

Speed Though exponents are fun, the computer handles them slowly. For example, the computer handles 3^2 slower than $3 * 3$. So for fast calculations, say $3 * 3$ instead of 3^2 .

Square roots What positive number, when multiplied by itself, gives 9? The answer is 3, because 3 times itself is 9.

3 squared is 9. 3 is called the **square root** of 9.

To make the computer deduce the square root of 9, type this:

```
Text = Math.Sqrt(9)
```

The computer will print 3.

The symbol Math.Sqrt is called a **function**. The number in parentheses (9) is called the function's **input** (or **argument** or **parameter**). The answer, which is 3, is called the function's **output** (or **value**).

Math.Sqrt(9) gives the same answer as $9^{.5}$. The computer handles Math.Sqrt(9) faster than $9^{.5}$.

Cube roots What number, when multiplied by itself and then multiplied by itself *again*, gives 64? The answer is 4, because 4 times 4 times 4 is 64. The answer (4) is called the **cube root** of 64.

Here's how to make the computer find the cube root of 64:

```
Text = 64 ^ (1 / 3)
```

The computer will say 4.

Stripping

Sometimes the computer prints *too* much info: you wish the computer would print less, to save yourself the agony of reading excess info irrelevant to your needs. Whenever the computer prints too much info about a numerical answer, use Math.Abs, Fix, Int, Math.Ceiling, Math.Round, or Math.Sign.

Math.Abs removes any minus sign. ("Abs" is short for "Absolute value".) For example, the Math.Abs of -3.89 is 3.89. So if you say Text = Math.Abs(-3.89), the computer will say just 3.89.

Fix removes any digits after the decimal point. For example, the Fix of 3.89 is 3. So if you say Text = Fix(3.89), the computer will say just 3. The Fix of -3.89 is -3.

Int rounds the number DOWN to an integer that's LOWER. For example, the Int of 3.89 is 3 (because 3 is an integer that's lower than 3.89); the Int of -3.89 is -4 (because -4 is lower than -3.89).

Math.Ceiling rounds the number UP to an integer that's HIGHER. For example, the Math.Ceiling of 3.89 is 4 (because 4 is an integer that's higher than 3.89); the Math.Ceiling of -3.89 is -3 (because -3 is higher than -3.89).

Math.Round can round to the NEAREST integer. For example, the Math.Round of 3.89 is 4. The Math.Round of -3.89 is -4. The Math.Round of a number ending in .5 is an integer that's even (not odd); for example, the Math.Round of 26.5 is 26 (because 26 is even), but the Math.Round of 27.5 is 28 (because 28 is even); this rounding method is called **unbiased rounding** and explained in the next section ("Types of data"). If you want traditional rounding instead of unbiased rounding, ask for Math.Round(26.5, System.MidpointRounding.AwayFromZero), which produces 27. If you say Text = Math.Round(865.739, 2), the computer will round 865.739 to 2 decimal places and say 865.74.

Math.Sign removes ALL the digits and replaces them with a 1, unless the number is 0. For example, the Math.Sign of 3.89 is 1. The Math.Sign of -3.89 is -1. The Math.Sign of 0 is just 0.

Math.Abs, Fix, Int, Math.Ceiling, Math.Round, and Math.Sign are all called **stripping functions** or **strippers** or **diet functions** or **diet pills**, because they strip away the number's excess fat and reveal just the fundamentals that interest you.

Pi

A circle's **circumference** (the distance around a circle) is about 3 times as long as the circle's **diameter** (the distance across the circle). So the circumference divided by the diameter is about 3. More precisely, it's **pi**, which is about 3.1415926535897931, a number that Visual Basic calls **Math.PI**. If you type —

```
Text = Math.PI
```

the computer will display this approximation:

```
3.14159265358979
```

Avoid "Math."

Many of those functions expect you to type "Math." To avoid having to type "Math.", put this line at your program's top (above "Public Class Form1"):

```
Imports System.Math
```

Then you can omit "Math." For example, instead of typing —

```
Text = Math.Sqrt(9)
```

you can type just:

```
Text = Sqrt(9)
```

Instead of typing —

```
Text = Math.PI
```

you can type just:

```
Text = PI
```

Types of data

If you want x to be a variable in your subroutine, you must warn the computer by giving your subroutine a command such as:

```
Dim x
```

Here's how to make your program run faster, consume less RAM, and correct more errors: instead of saying just "Dim x", warn the computer what **type of data** the x will stand for, by giving one of these 8 popular commands:

Command	Meaning	RAM	Speed
Dim x As Integer	x will be a number from 0 to 2147483647, with no decimal point, but maybe a negative sign	4 bytes	fastest
Dim x As Long	x will be a number from 0 to 9223372036854775807, with no decimal point, but maybe a negative sign	8 bytes	fast
Dim x As Double	x will be a number from 0 to 1E308, with maybe a decimal point and negative sign, 15-digit accuracy	8 bytes	fast
Dim x As Decimal	x will be a number having up to 28 digits, with maybe a decimal point and negative sign, 28-digit accuracy	16 bytes	slowest
Dim x As Date	x will be a date and time (such as #12/31/2009 11:59:30 PM#), with a year between 1 and 9999	8 bytes	slow
Dim x As String	x will be a string (such as "I love you") up to 2 billion characters long 2 bytes per character, plus	10 bytes	slow
Dim x As Char	x will be a single character (such as "j")	2 bytes	fast
Dim x As Boolean	x will be either the word True or the word False	2 bytes	fast

Here's how to choose among them:

If x stands for a reasonably small number (2147483647 or less) without a decimal point, choose **Integer**.

If x stands for a longer number (up to 9223372036854775807) without a decimal point, choose **Long**.

If x stands for a number that's even bigger or has a decimal point, choose **Double** unless you need more than 15-digit accuracy, which demands **Decimal**.

If x stands for a date or time, choose **Date**.

If x stands for a single character (such as "c"), choose **Char**. If x stands for a longer string, choose **String**.

If x stands for the word True or the word False, choose **Boolean**.

For example, if you want x to be 3000000, say:

```
Dim x As Integer  
x = 3000000
```

According to the chart's top line, saying "Dim x As Integer" makes x consume 4 bytes of RAM. The computer can store 3000000 (or any integer up to 2147483647) in just 4 bytes of RAM, because the computer stores the number by using a special trick called **binary representation**.

These 7 variations are less popular:

Instead of **Integer**, you can choose **UInteger** (which means unsigned integer).

It can handle numbers that are twice as big (up to 4294967295) but can't handle a negative sign.

Instead of **Long**, you can choose **ULong** (which means unsigned long).

It can handle numbers that are twice as big (up to 18446744073709551615) but can't handle a negative sign.

Instead of **Integer** (which consumes 4 bytes), you can choose **Short** (which consumes just 2 bytes and is limited to numbers up to 32767) or **SByte** (which consumes just 1 byte and is limited to numbers up to 127). But those alternatives run slow, because the Pentium chip was designed to handle 4-byte integers, not shorter integers. Use those alternatives just if you're worried about the number of bytes. Here are other alternatives, which also run slow: **UShort** (which consumes 2 bytes, handles numbers up to 65535, no decimals or negatives) and **Byte** (which consumes 1 byte, handles numbers up to 255, no decimals or negatives).

Instead of **Double**, you can choose **Single** (which means single-length numbers). It consumes fewer bytes (4 instead of 8) but runs slow (because the Pentium chip was designed to handle decimal points in 8-byte numbers, not shorter ones). It has less accuracy (7-digit instead of 15-digit) and is restricted to smaller numbers (up to 3E38, not 1E308). Use it just if you're worried about the number of bytes.

Details

Here are more details about the 8 popular Dim commands.

Integer An **Integer** is a number from 0 to 2147483647, with maybe a negative sign in front, but without a decimal point. For example, these numbers can all be Integer:

```
0 1 2 3 10 52 53 1000 2147483647
-1 -2 -3 -10 -52 -53 1000 -2147483647
```

Technical note: although 2147483648 is slightly too big to be an Integer, -2147483648 is a special number that *can* be an Integer, though it's rarely used and must be written as:

```
-2147483647 - 1
```

If you say "Dim x As Integer" and then try to say "x = 52.9", the computer will round 52.9 to 53, so x will be 53.

To round, Visual Basic 2015 makes the computer use this strange method, called **unbiased rounding**:

If the number's decimal part is less than .5 (for example, if it's .4), the computer rounds down. For example, 26.4 rounds down to 26.

If the number's decimal part is more than .5 (for example, if it's .51 or .6), the computer rounds up. For example, 26.51 rounds up to 27.

If the number's decimal part is exactly .5 (not less, not more, not .51), the computer uses this strange method: it round to the nearest integer that's even (not odd). For example, 26.5 rounds down to 26 (since 26 is even), but 27.5 rounds up to 28 (since 28 is even).

That makes .5 sometimes round down and sometimes rounds up, so there's no bias toward rounding in a particular direction. That unbiased rounding method appeals to statisticians (and a few economists and very few bankers) who want to eliminate bias from rounded results. It's called **unbiased rounding** (or **round-to-even** or **statisticians rounding** or **bankers rounding** or **Dutch rounding** or **Gaussian rounding**).

Long A **Long** is a number from 0 to 9223372036854775807, with maybe a negative sign in front, but without a decimal point. For example, these numbers can all be Longs:

```
0 1 2 3 10 52 53 1000 9223372036854775807
0 -1 -2 -3 -10 -52 -53 -1000 -9223372036854775807
```

Technical note: although 9223372036854775808 is slightly too big to be a Long, -9223372036854775808 is a special number that *can* be a Long, though it's rarely used and must be written as:

```
-9223372036854775807 - 1
```

If you say "Dim x As Long" and then try to say "x = 52.9", the computer will round 52.9 to 53, so x will be 53. (To round, the computer uses unbiased rounding.)

If you write a number that has no decimal point and is small (no more than 2147483647), the computer assumes you want it to be an Integer. If you want it to be a Long instead, **put L after it**, like this: 57L. For example, if you tell the computer to multiply 3000 by 1000000, like this —

```
Text = 3000 * 1000000
```

the computer assumes you want to multiply the Integer 3000 by the Integer 1000000; but the answer is too long to be an Integer, so the computer gripes (by saying "not representable in type 'Integer'"). Multiplying 3000 by 1000000 is okay if you say the numbers are Longs, not Integers, like this:

```
Text = 3000L & 1000000L
```

Then the computer will show the correct answer:

```
3000000000
```

Double A **Double** is a number from 0 to 1E308 (which is a "1 followed by 308 zeros"), with maybe a negative sign and a decimal point. After the decimal point, you can have as many digits as you wish. For example, these numbers can all be Double:

```
0 1 2 3 4.99 4.9995 4.999527 1000.236 26127.85 1E308
0 -1 -2 -3 -4.99 -4.9995 -4.999527 -1000.236 -26127.85 -1E308
```

The computer manages to store a Double rather briefly (just 8 bytes) by "cheating": **the computer stores the number just approximately, to an accuracy of about 15 significant digits.**

If example, if you say —

```
Dim x As Double
x = 100 / 3
Text = 100 / 3
```

the computer will show 15 digits:

```
33.3333333333333
```

If you say —

```
Dim x As Double
x = 1000000.000000269
Text = x
```

the computer will round to 15 digits and show:

```
1000000.00000027
```

When handling Double variables, the computer can give inaccurate results. The inaccuracy is especially noticeable if you do a subtraction where the two numbers nearly equal each other. For example, if you say —

```
Dim x, y As Double
x = 8000.1
y = x - 8000
Text = y
```

the computer will make x be *approximately* 8000.1, so y will be *approximately* .1. The Print line will print:

```
0.1000000000000364
```

Notice that the last few digits are wrong! That's the drawback of Double: you can't trust the last few digits of the answer! Double is accurate enough for most scientists, engineers, and statisticians, since they realize all measurements of the real world are just approximations; but Double is *not* good enough for accountants who fret over every penny. Double's errors drive accountants bananas. **For accounting problems that involve decimals, consider using Decimal instead of Double**, since Decimal is always accurate, though slower.

Technical notes:

A Double can be slightly bigger than 1E308. The biggest permissible Double is actually 1.7976931348623157E308.

If a Double is at least a quadrillion (which is 1000000000000000) or tiny (less than .0001), the computer will display it by using E notation.

When you type a Double in your subroutine, the computer stores the first 16 significant digits accurately, stores an approximation of the 17th significant digit, and ignores the rest.

If you type a number that has no decimal point and no E, the computer will think you're trying to type an Integer or a Long; and if it has many digits, the computer will complain that a Long is not allowed to have so many digits. To correct the problem, indicate you're trying to type a Double, by putting .0 at the end of the number or using E notation.

When the computer displays an answer, it displays the first 15 significant digits and hides the rest, since it knows the rest are unreliable. For example, if you set Text equal to the biggest number (1.7976931348623157E308), the computer will display it rounded to 15 digits, so it will display 1.79769313486232E308.

The tiniest decimal the computer can handle accurately is 1E-308 (which is a decimal point followed by 308 digits, 307 of which are zeros). If you try to go tinier, the computer will give you a rough approximation. The tiniest permissible Double is 4.9406564584126544E-324; if you try to go tinier than that, the computer will say 0.

Here's another example:

```
Dim x As Boolean
x = False
Text = x
```

The computer will display:

```
False
```

Technical notes:

If you say "Dim x As Boolean" and then try to say "x = 0", the computer will make x be False.

If you say "Dim x As Boolean" and then try to say "x = 1" (or say that x is any other non-zero number), the computer will make x be True.

It's called "Boolean" to honor George Boole (the 19th-century mathematician who discovered that the word False acts like the number 0, and True acts like the number 1).

Object You've learned that x can stand for a number, date, string, character, or Boolean.

Here's another possibility: x can stand for an **object**, such as Form1 or Button1 or any other VB thing, such as Color.Red.

For example, suppose you created a button called Button1. If you put this line in Form1's subroutine, Button1's title will become "Click me":

```
Button1.Title = "Click me"
```

These lines do the same thing:

```
Dim x As Object
x = Button1
x.Text = "Click me"
```

Saying "Dim x As Object" is vague. It has exactly the same meaning as "Dim x", which is vague. If you say "Dim x As Object" (or just "Dim x"), you're saying that x stands for a Windows object (such as Form1 or Button1) or some other kind of object (such as a number, date, string, character, or Boolean). The computer handles such an x slowly: it consumes 4 bytes to remember what part of the RAM holds x's details, plus several bytes to store the details.

Multiple variables

If you want x and y to be Integers, z to be a String, and temperature to be a Double, say this —

```
Dim x, y As Integer
Dim z As String
Dim temperature As Double
```

or say it all in one line:

```
Dim x, y As Integer, z As String, temperature As Double
```

Suffix

Here's the normal way to make x be a String:

```
Dim x As String
```

This way is shorter:

```
Dim x$
```

That dollar sign means "As String". The dollar sign is called a **suffix** (or **type-declaration character**).

You can use these suffixes:

Suffix	Meaning
\$	As String
%	As Integer
&	As Long
#	As Double
@	As Decimal
!	As Single

Repeating the suffix Below the Dim line, you can type the suffix again if you wish. For example, after you've made x be a string by saying —

```
Dim x$
```

you can say either —

```
x = "I love you"
```

or this, which means the same thing:

```
x$ = "I love you"
```

The computer doesn't care whether you type the \$ again. Type it just if you want to emphasize it to other programmers who look at your subroutine.

Constants

Your subroutine can mention variables (such as x and y) and constants (such as 3.7 and "I love you"). Here's how the computer tells a constant's type:

If the constant is the word True or the word False, it's a **Boolean**.

If the constant begins and ends with the symbol #, it's a **Date**.

If the constant is enclosed in quotation marks (such as "I love you"), it's a **String**, unless it has a c afterwards (such as "j"c), which makes it a **Char** (and is limited to just one character).

If the constant is a number, here's what happens....

If the number has no decimal point and no E and is short (between -2147483648 and 2147483647), it's an **Integer**. If the number has no decimal point and no E and is between -9223372036854775808 and 9223372036854775807 but is not an Integer, it's a **Long**. Any other number is a **Double**.

To force a number to be a **Decimal** instead, put D (or @) after the number, like this: 4.95D

To force a number to be a **Long** (even though it's small enough to be an Integer), put L (or &) after the number, like this: 52L

To force a number to be a **Double** (even though it's simple enough to be an Integer or Long), put .0 after the number, like this: 52.0.

VarType

Each type of constant has a code number:

Type of constant	Code number
Integer	3
Long	20
Double	5
Decimal	14
Date	7
String	8
Char	18
Boolean	11
Object	9

If you say **VarType**, the computer will examine a constant and tell you its code number. For example, if you say —

```
Text = VarType(4.95D)
```

the computer will examine 4.95D, realize it's a Decimal, and say Decimal's code number, which is:

```
14
```

Here are more examples:

If you say Text = VarType("I love you"), the computer will examine "I love you", realize it's a String, and print String's code number, which is 8.

If you say Text = VarType(2000000000), the computer will examine 2000000000, realize it's an Integer, and print Integer's code number, which is 3.

If you say Text = VarType(3000000000), the computer will examine 3000000000, realize it's a Long, and print Long's code number, which is 20.

VarType of a variable If you say VarType(x), the computer will notice what type of variable x is and print its code number. For example, if you say —

```
Dim x As Decimal
Text = VarType(x)
```

the computer will say Decimal's code number, which is 14.

If you say just "Dim x" (or "Dim x As Object") without specifying further details of x's type, VarType(x) will be whatever type the x acquires. For example, if you say —

```
Dim x
x = 4.95D
Type = VarType(x)
```

the computer will print Decimal's code number, which is 14.

TypeName

If you say **TypeName** instead of **VarType**, the computer will say the type's name instead of its code number.

For example, if you say —

```
Text = TypeName(4.95D)
```

the computer will say:

```
Decimal
```

Instead of saying "Object" (or "Nothing"), the computer will try to be more specific. For example, if you created a command button called Button1 and say —

```
Text = TypeName(Button1)
```

the computer will say:

```
Button
```

If x is an Object but doesn't have a more specific value or type yet, "Text = TypeName(x)" will make the computer say:

```
Nothing
```

Initial value

Instead of saying —

```
Dim x As Integer
x = 7
```

you can combine those two lines into this single line:

```
Dim x As Integer = 7
```

In that line, 7 is called x's **initial value** (or **initializer**), because it's what x is initially (in the beginning).

You can shorten that line further, by saying just this:

```
Dim x = 7
```

Since 7 is an Integer (according to the rules about which constants are Integers), the computer will assume you also mean "x As Integer". If you say this instead —

```
Dim x = 7.0
```

the computer will assume you mean "x As Double".

Saying "Dim x = 5" has a slightly different effect than saying "Dim x" then "x = 5". Compare these subroutines:

```
Dim x
x = 5
x = 8.4
Text = x

Dim x = 5
x = 8.4
Text = x
```

In the left subroutine, the Dim line says x is a vague variable (an object). The next line says x is 5. The next line changes x to 8.4, so Text will be 8.4. In the right-hand subroutine, the first line says x is 5 but also makes x be an integer variable (since that line implies "x As Integer"); since x is an integer variable, the next line makes x be 8 (not 8.4), so Text will be just 8.

Operations

When you do operations (add, subtract, multiply, divide, exponents, or beyond), here's what kind of answer you get.

Exponents When you do **exponents** (using the symbol "^"), the answer is a **Double**.

Division When you **divide** one number by another (using the symbol "/"), here's what happens:

If both numbers are Decimal, the answer is **Decimal**.

If one of the numbers is Single and the other is Single or Decimal, the answer is **Single**.

In all other situations, the answer is **Double**.

Add, subtract, multiply When you **add, subtract, or multiply** numbers (using the symbol + or - or *), here's what happens:

If both numbers are the same type, the computer makes the answer be **the same type**. (Exception: if both "numbers" are actually Boolean, the computer makes the answer be Short.)

If the numbers have different types from each other, and both types are **signed** (permit minus signs), the computer notices which type is **wider** (can handle more numbers) and makes the answer be that type. Here are the signed types, from narrowest to widest: SByte, Short, Integer, Long, Decimal, Single, Double. (Single is wider than Decimal because Single can handle higher powers of 10.) For example, if one number is an Integer and the other number is a Long, the answer is a Long (because Long is wider than Integer).

If the numbers have different types from each other, and at least one of those types is **unsigned** (Boolean, Byte, UShort, UInteger, or ULong), the computer makes the answer be the wider type — or a signed type that's even wider.

Advanced math Here's how the computer handles advanced math:

Math.PI and **Math.Sqrt(x)** are Double.

Math.Sign(x) is an Integer.

Math.Abs(x) and **Fix(x)** and **Int(x)** are the same type as x, if x's type is signed. If x's type is unsigned, the computer turns x into a wider signed number first.

Math.Ceiling(x) and **Math.Round(x)** are Double, if x is a Double or Single. They're Decimal if x is otherwise.

Combine When you combine strings or numbers (by using the symbol "&"), the answer is a string.

Form1 declarations

Normally, each subroutine has its own variables. For example, if Form1's subroutine uses a variable called x, and Button1's subroutine uses a variable that's also called x, Form1's x has nothing to do with Button1's x. Form1's x is stored in a different part of RAM from Button1's x. If Form1 says x = 5, Button1's x remains unaffected by that statement.

If you *want* Form1's x to be the same as Button1's x and use the same RAM, say "Dim x" *above* the "Private Sub Form1" line instead of below.

Example For example, try this experiment...

Create a new program. Double-click Form1, so you can type Form1's subroutine. Your screen looks like this:

```
Public Class Form1
    Private Sub Form1_Load...

    End Sub
End Class
```

Click *above* the "Private Sub Form1" line and type "Dim x" there, so your screen looks like this:

```
Public Class Form1
    Dim x
    Private Sub Form1_Load...

    End Sub
End Class
```

Type Form1's subroutine under the "Private Sub Form1" line, like this:

```
Public Class Form1
    Dim x
    Private Sub Form1_Load...
        x = 5
    End Sub
End Class
```

Create Button1 (by clicking the "Form1.vb [Design]" tab then double-clicking the Button tool). Double-click Button1, then type "Text = x" for Button1's subroutine. Altogether, your screen looks like this:

```
Public Class Form1
    Dim x
    Private Sub Form1_Load...
        x = 5
    End Sub

    Private Sub Button1_Click...
        Text = x
    End Sub
End Class
```

Since the “Dim x” is *above* both subroutines (instead of being buried inside one subroutine), the x’s value affects *both* subroutines (not just one of them).

When you run that program (by clicking “Start”), Form1’s subroutine makes x be 5. Then when you click Button1, Button1’s subroutine makes Text be x, which is 5, so the computer says:

```
5
```

Conversion functions

In the middle of a calculation, you can convert to a different type of data by using these conversion functions:

Function	Meaning
CInt	convert to Integer
CLng	convert to Long
Cdbl	convert to Double
CDec	convert to Decimal
CDate	convert to Date
CStr	convert to String
CChar	convert to Char
CBool	convert to Boolean
CUInt	convert to UInteger
CULng	convert to ULong
CShort	convert to Short
CSByte	convert to SByte
CUShort	convert to UShort
CByte	convert to Byte
CSng	convert to Single
CObj	convert to Object

For example, CInt(3.9) is “3.9 converted to the nearest Integer”, which is 4. If you say —

```
Text = CInt(3.9)
```

the computer will say:

```
4
```

If you say —

```
Text = CInt(3.9) + 2
```

the computer will say:

```
6
```

Arrays

Instead of being just a number, x can be a *list* of numbers.

Example For example, if you want x to be this list of numbers —

```
{81, 52, 207, 19}
```

type this in Form1’s subroutine:

```
Dim x() = {81, 52, 207, 19}
```

In that line, the symbol “x()” means “x’s list”. Notice that when you type the list of numbers, you must **put commas between the numbers** and put the entire list of numbers in **braces**, {}. On your keyboard, the “{” symbol is to the right of the P key and requires you to hold down the Shift key.

Since all numbers in that list are Integers, you can improve that line by saying “As Integer”, like this:

```
Dim x() As Integer = {81, 52, 207, 19}
```

If you don’t say “As Integer”, the computer will treat those numbers as just vague objects, and the program will run slower.

In x’s list, **the starting number (81) is called x₀** (pronounced “x subscripted by zero” or “x sub 0” or just “x 0”). The next number (52) is called x₁ (pronounced “x subscripted by one” or “x sub 1” or just “x 1”). The next number is called x₂. Then comes x₃. So **the four numbers in the list are called x₀, x₁, x₂, and x₃**.

To make the computer say what x₂ is, type this line:

```
Text = x(2)
```

That line makes Text be x₂, which is 207, so the computer will say:

```
207
```

Altogether, the subroutine says:

```
Dim x() As Integer = {81, 52, 207, 19}
Text = x(2)
```

The first line says x’s list is these Integers: 81, 52, 207, and 19. The bottom line makes the computer say x₂’s number, which is 207.

This subroutine makes the computer say x₂’s number (which is 207) in a message box:

```
Dim x() As Integer = {81, 52, 207, 19}
MsgBox(x(2))
```

This subroutine makes the computer say all 4 numbers:

```
Dim x() As Integer = {81, 52, 207, 19}
For i = 1 To 4
    MsgBox(x(i))
Next
```

That makes the computer say the numbers for x(1), x(2), x(3), and x(4), so the computer will say 81, 52, 207, and 19.

Here’s a shorter way to make the computer say all 4 numbers:

```
Dim x() As Integer = {81, 52, 207, 19}
For Each i In x
    MsgBox(i)
Next
```

That makes x’s list be {81, 52, 207, 19}, makes i be **Each number In x** (so i is 81, then 52, then 207, then 19), and makes the computer say each i.

Longer lists Instead of having just 4 numbers in the list, you can have 5 numbers, or 6 numbers, or a thousand numbers, or many billions of numbers. The list can be quite long! Your only limit is how much RAM your computer has.

Jargon Notice this jargon:

In a symbol such as x₂, the lowered number (the 2) is called the **subscript**.

To create a subscript in your subroutine, use parentheses. For example, to create x₂, type x(2).

A variable having subscripts is called an **array**. For example, x is an array if there’s an x₀, x₁, x₂, etc.

Different types Instead of having Integers, you can have different types. For example, you can say:

```
Dim x() As Double = {81.2, 51.7, 207.9, 19.5}
```

You can even say:

```
Dim x() As String = {"love", "hate", "peace", "war"}
```

You can even have mixed types:

```
Dim x() = {5, 91.3, "turkey", #11:59:30 PM#}
```

Uninitialized Instead of making the Dim line include a list of numbers, you can type the numbers underneath, if you warn the computer how many numbers will be in the list, like this:

```
Dim x(2) As Double
x(0) = 200.1
x(1) = 700.4
x(2) = 53.2
Text = x(0) + x(1) + x(2)
```

The top line says x₀, x₁, and x₂ will be Doubles. The next lines say x₀ is 200.1, x₁ is 700.4, and x₂ is 53.2. The bottom line makes the computer say their sum:

```
953.7
```

In that top line, if you omit the “As Double”, the program will give the same answer but slower. But in that top line, the 2 is required, to warn the computer how many subscripts to reserve RAM for; if you omit the 2 (or type a lower number instead), the computer will gripe.

Random numbers

Usually, the computer is predictable: it does exactly what you say. But sometimes, you want the computer to be *unpredictable*.

For example, if you’re going to play a game of cards with the computer and tell the computer to deal, you want the cards dealt to be unpredictable. If the cards were predictable — if you could figure out exactly which cards you and the computer would be dealt — the game would be boring.

In many other games too, you want the computer to be unpredictable, to “surprise” you. Without an element of surprise, the game would be boring.

Being unpredictable increases the pleasure you derive from games — and from art. To make the computer act artistic, and create a new *original* masterpiece that’s a “work of art”, you need a way to make the computer get a “flash of inspiration”. Flashes of inspiration aren’t predictable: they’re surprises.

Here’s how to make the computer act unpredictably...

Rnd is a RaNDom decimal (bigger than 0 and less than 1) whose data type is Single. For example, it might be .6273649 or .9241587 or .2632801. Every time your program mentions Rnd, the computer concocts another decimal. For example, if Form1's subroutine says —

```
MsgBox(Rnd)
MsgBox(Rnd)
MsgBox(Rnd)
```

the computer says these decimals:

```
.7055475
.533424
.5795186
```

The first time your program mentions Rnd, the computer chooses its favorite decimal, which is .7055475. Each succeeding time your program mentions Rnd, the computer uses the previous decimal to concoct a new one. It uses .7055475 to concoct .533424, which it uses to concoct .5795186. The process by which the computer concocts each new decimal from the previous one is weird enough so we humans cannot detect any pattern.

These lines make the computer say 16 decimals:

```
For i = 1 To 16
    MsgBox(Rnd)
Next
```

You can say either Rnd or Rnd(); the computer doesn't care. If you say just Rnd, the computer might change it to Rnd().

Percentages

When the computer says random decimals, about half the decimals will be less than .5, and about half will be more than .5.

Most of the decimals will be less than .9. In fact, about 90% will be.

About 36% of the decimals will be less than .36; 59% will be less than .59; 99% will be less than .99; 2% will be less than .02; a quarter of them will be less than .25; etc.

You might see some decimal twice, though most of the decimals will be different from each other.

Randomize

If you run a program about Rnd again, you'll see exactly the same decimals again, in the same order.

If you'd rather see a different list of decimals, say **Randomize()** at the subroutine's top:

```
Randomize()
For i = 1 To 16
    MsgBox(Rnd)
Next
```

When the computer sees Randomize(), the computer looks at the clock and manipulates the time's digits to produce the first value of Rnd.

So the first value of Rnd will be a number that depends on the time of day, instead of the usual .7055475. Since the first value of Rnd will be different than usual, so will the second, and so will the rest of the list.

Every time you run the program, the clock will be different, so the first value of Rnd will be different, so the whole list will be different — unless you run the program at exactly the same time the next day, when the clock is the same. But since the clock is accurate to a tiny fraction of a second, the chance of hitting the same time is extremely unlikely.

Coin flipping

Here's how to make the computer flip a coin:

```
Randomize()
If Rnd < 0.5 Then MsgBox("heads") Else MsgBox("tails")
```

The Randomize line makes the value of Rnd depend on the clock. The If line says there's a 50% chance that the computer will print "heads"; if the computer does *not* print "heads", it will print "tails".

When you've typed that subroutine, the computer changes Rnd to Rnd(), so it looks like this:

```
Randomize()
If Rnd() < 0.5 Then MsgBox("heads") Else MsgBox("tails")
```

Until you run the program, you won't know which way the coin will flip; the choice is random. Each time you run the program, the computer will flip the coin again; each time, the outcome is unpredictable. Try running it several times!

To write that subroutine shorter, say IIf:

```
Randomize()
MsgBox(IIf(Rnd() < 0.5, "heads", "tails"))
```

The bottom line creates a message box saying this: if the random number is less than .5, then "heads", else "tails".

This subroutine flips the coin 10 times:

```
Randomize()
For i = 1 To 10
    MsgBox(IIf(Rnd() < 0.5, "heads", "tails"))
Next
```

Love or hate?

Who loves ya, baby? These lines try to answer that question:

```
Randomize()
Dim x As String
x = InputBox("Type the name of someone you love")
If Rnd < 0.67 Then
    MsgBox(x & " loves you, too")
Else
    MsgBox(x & " hates your guts")
End If
```

The Randomize() line makes the value of Rnd depend on the clock. The Dim line says x will be a variable that stands for a String. The InputBox line makes the computer wait for the human to type a name. Suppose he types Suzy. Then x is "Suzy". The If line says there's a 67% chance the computer will say "Suzy loves you, too", but there's a 33% chance the computer will instead say "Suzy hates your guts".

Try running the program several times. Each time, input a different person's name. Find out which people love you and which people hate your guts — according to the computer!

Here's a shorter way to write that subroutine:

```
Randomize()
Dim x = InputBox("Type the name of someone you love")
MsgBox(x & IIf(Rnd < .67, " loves you, too", " hates your guts"))
```

The Randomize() line makes the value of Rnd depend on the clock. The Dim line makes the variable x be the response to "Type the name of someone you love". The MsgBox line creates a message box that says x then this: if the random number is less than .67 then " loves you, too" else " hates your guts".

Random integers

If you want a random integer from 1 to 10, ask for $1 + \text{Int}(\text{Rnd} * 10)$. Here's why:

```
Rnd is a decimal, bigger than 0 and less than 1.  
So Rnd * 10 is a decimal, bigger than 0 and less than 10.  
So Int(Rnd * 10) is an integer, at least 0 and no more than 9.  
So 1 + Int(Rnd * 10) is an integer, at least 1 and no more than 10.
```

Guessing game These lines play a guessing game:

```
Randomize()  
MsgBox("I'm thinking of a number from 1 to 10.")  
Dim ComputerNumber = 1 + Int(Rnd * 10)  
AskHuman: Dim guess = Val(InputBox("what do you think my number is?"))  
If guess < ComputerNumber Then MsgBox("Your guess is too low."): GoTo AskHuman  
If guess > ComputerNumber Then MsgBox("Your guess is too high."): GoTo AskHuman  
MsgBox("Congratulations! You found my number!")
```

The second line makes the computer say “I’m thinking of a number from 1 to 10.” The next line makes the computer think of a random number from 1 to 10. The InputBox line asks the human to guess the number.

If the guess is less than the computer’s number, the first If line makes the computer say “Your guess is too low” and then GoTo AskHuman, which lets the human guess again. If the guess is *greater* than the computer’s number, the second If line makes the computer say “Your guess is too high” and then GoTo AskHuman.

When the human guesses correctly, the computer arrives at the bottom line, which makes the computer say:

```
Congratulations! You found my number!
```

Dice These lines make the computer roll a pair of dice:

```
Randomize()  
MsgBox("I'm rolling a pair of dice")  
Dim a = 1 + Int(Rnd * 6)  
MsgBox("One of the dice says " & a)  
Dim b = 1 + Int(Rnd * 6)  
MsgBox("The other says " & b)  
MsgBox("The total is " & a + b)
```

The second line makes the computer say:

```
I'm rolling a pair of dice
```

Each of the dice has 6 sides. The next line, $\text{Dim } a = 1 + \text{Int}(\text{Rnd} * 6)$, rolls one of the dice, by picking a number from 1 to 6. The line saying “ $b = 1 + \text{Int}(\text{Rnd} * 6)$ ” rolls the other. The bottom line says the total.

For example, a run might say these sentences:

```
I'm rolling a pair of dice  
One of the dice says 3  
The other says 5  
The total is 8
```

Here's another run:

```
I'm rolling a pair of dice  
One of the dice says 6  
The other says 4  
The total is 10
```

Daily horoscope These lines predict what will happen to you today:

```
Randomize()  
Dim x() = {"wonderful", "fairly good", "so-so", "fairly bad", "terrible"}  
MsgBox("You will have a " & x(Int(Rnd * 5)) & " day today!")
```

The Dim line makes x be a list of 5 fortunes, so x_0 is “wonderful”, x_1 is “fairly good”, x_2 is “so-so”, x_3 is “fairly bad”, and x_4 is “terrible”. Since $\text{Int}(\text{Rnd} * 5)$ is a random integer from 0 to 4, the $x(\text{Int}(\text{Rnd} * 5))$ is a randomly chosen fortune. The computer will say —

```
You will have a wonderful day today!
```

or —

```
You will have a terrible day today!
```

or some in-between comment.

For inspiration, run that program when you get up in the morning. Then notice whether your day turns out the way the computer predicts!

Visual C#

A Microsoft employee (**Anders Hejlsberg**) invented a nifty computer language. He called it **Cool** but changed the name to **C#** (pronounced “C sharp”), to emphasize it’s higher than an earlier language, called **C**. (It’s also higher than a C variant called **C++**.)

C# tries to combine the best features of Visual Basic, Java, C, and C++:

Like Visual Basic, it lets you create windows easily.

Like Java, it uses modern notation for typing lines in programs.

Like C and C++, it runs fast.

C# is also influenced by an older programming language called **Pascal**. Before inventing C#, Anders Hejlsberg had already invented two famous Pascal versions (**Turbo Pascal** and **Delphi**) and a famous Java version (**J++**); he was an extremely experienced designer when he invented C#. He knew what was wrong with Pascal, Java, C++, and Visual Basic and how to improve them.

Microsoft recommends using Visual Basic to create simple programs but C# to create bigger projects. Microsoft considers Visual Basic and C# to be the most important computer languages to learn.

You already learned Visual Basic. Now let’s tackle C#.

Modern C#, called **Visual C#**, is part of **Visual Studio**. Get Visual Studio’s free version, called **Visual Studio Community**, by copying it from Microsoft’s Website, using the method on page 595 (“Copy the Community”). This chapter assumes you’ve done that, so you have **Visual C# 2015**.

Visual C# 2015 understands these commands:

C# command	Page
catch	640
char x;	636
class Program	634
Console.ReadKey();	635
Console.WriteLine("Love");	635
Console.WriteLine("Love");	635
double x;	636
double x = -27.0;	636
double[] x = new double[3];	638
double[] x = { 81.2, 51.7, 7.9 };	638
double[,] x = new double[2, 3];	638
else	639
if (age < 18)	638
for (int i = 20; i <= 29; ++i)	640
goto yummy;	640
int x;	636
int x = 3;	637
int[] x = new int[3];	638
int[] x = { 81, 52, 207 };	638
int[,] x = new int[2, 3];	638
long x;	636
MessageBox.Show("Hair mess");	643
namespace Joymaker	634
private void Form1_Load(...)	643
public Form1()	643
public partial class Form1 : Form	643
return (a + b) / 2;	642
static int average(int a, int b)	642
static void Main(string[] args)	634
static void x()	641
string x;	636
string[] x = { "love", "h" };	638
Text = "I love you";	643
try	640
uint x;	636
ulong x;	636
using System;	634
while (true)	639
x = 3;	636
x();	641
++x;	637
--x;	637
// Zoo program is fishy	641

It also understands these functions:

C# function	Page
Console.ReadLine()	637
Convert.ToDouble(x)	637
Convert.ToInt32(x)	638
Convert.ToInt64(x)	638
Convert.ToString(x)	643
Convert.ToUInt32(x)	638
Convert.ToUInt64(x)	638
Math.Abs(x)	636
Math.Acos(x)	636
Math.Asin(x)	636
Math.Atan(x)	636
Math.Atan2(y, x)	636
Math.Ceiling(x)	636
Math.Cos(x)	636
Math.Cosh(x)	636
Math.E	636
Math.Exp(y)	636
Math.Floor(x)	636
Math.Log(x)	636
Math.Log(x, b)	636
Math.Log10(x)	636
Math.PI	636
Math.Pow(x, y)	636
Math.Sin(x)	636
Math.Sinh(x)	636
Math.Sqrt(x)	636
Math.Tan(x)	636
Math.Tanh(x)	636
x.CompareTo("male")	639

Fun

Here’s how to enjoy programming in C#.

Start Visual Studio

To start using Visual Studio, type “vi” in the Windows 10 Search box (which is next to the Start button) then click “Visual Studio 2015: Desktop app”.

If you haven’t used Visual Studio before, the computer says “Sign in”. To reply, do this:

Click the “Sign in” button. Type your email address and press Enter. Type your Microsoft account’s password and press Enter. The computer says “We’re preparing for first use”.

You see the Start Page window.

Start a new program

Click “**New Project**” (which is near the screen’s left edge) then “**Visual C#**” then “**Console Application**”.

Double-click in the Name box (which is near the screen’s bottom). Type a name for your project (such as Joymaker). At the end of your typing, press the Enter key.

Type your program

The computer starts typing the program for you. The computer types:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Joymaker
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Let's write a program that makes the computer say "I love you". To do that, insert 2 extra lines, so the program becomes this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Joymaker
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("I love you");
            Console.ReadKey();
        }
    }
}
```

Here's how to insert those line:

Click under the word "void". Press Enter. Type the first inserted line, press Enter, and type the second inserted line.

The computer indents the lines for you, automatically.

You must type a semicolon at the end of each simple line.

But there's no semicolon at the end of a **structure line** (a line that's blank or says just "{" or "}" or is immediately above "{").

Important line The most important line is the one that says:

```
Console.WriteLine("I love you");
```

It makes the computer write "I love you" onto the screen.

Helper line To make Console.WriteLine work properly, you must put this **helper line** near the program's bottom, just above the 3 final "}" lines:

```
Console.ReadKey();
```

That makes the computer pause until the human has read the computer's output and presses a key.

You must put that helper line in every normal program.

Run the program

To run your program, click **"Start"** (which is at the screen's top center) or **press the F5 key**. (If the "F5" is blue or tiny or the computer is new by Microsoft, HP, Lenovo, or Toshiba, that key works just while you hold down the Fn key, which is left of the Space bar.)

If you did everything correctly, you see the **console window** (which has white letters on a black background and resembles the DOS command-prompt screen). The console window shows the computer's output. It shows:

```
I love you
```

When you finish admiring that output, press the Enter key (or Space bar or any other normal key) or click the console window's X button.

If you want to run the program again, click "Start" again.

If you want to edit the program, retype the parts you wish then click "Start" again (which makes the computer debug and run the new version).

Final steps

Similar to Visual Basic, so read "Final steps" on page 598, but change "Funmaker" to "Joymaker" if you named your program "Joymaker".

Multiple lines

Your program can contain *several* lines. For example, to make the computer say —

```
I love you
Let's get married
```

type these lines:

```
Console.WriteLine("I love you");
Console.WriteLine("Let's get married");
```

Below them, type the helper line:

```
Console.ReadKey();
```

If you say **Write** instead of **WriteLine**, the computer won't press the Enter key at the end of its writing. For example, if you type:

```
Console.write("I love you");
Console.WriteLine("Let's get married");
```

the computer will write "I love you" without pressing Enter, then write "Let's get married", so you see this:

```
I love youLet's get married
```

Math

The computer can do math. For example, this line makes the computer do 4+2:

```
Console.WriteLine(4 + 2);
```

It makes the computer write this answer on your screen:

```
6
```

If you have 750 apples and buy 12 more, how many apples will you have altogether? This line writes the answer:

```
Console.WriteLine("You will have " + (750 + 12) + " apples");
```

That line makes the computer write "You will have ", then write the answer to 750 + 12 (which is 762), then write "apples", so you see this:

```
You will have 762 apples
```

Like most other languages (such as Basic, JavaScript, Java, and C++), C# lets you use the symbols +, -, *, /, parentheses, decimal points, and e notation.

Types of numbers

C# handles 5 types of numbers well.

One type of number is called an **integer** (or **int**). An integer contains no decimal point and no e and is between -2147483648 and 2147483647. For example, -27 and 30000 are ints. Each int consumes 4 bytes (32 bits) of RAM.

An **unsigned integer** (or **uint**) resembles an integer but must not have a minus sign, and it can be between 0 and 4294967295. For example, 3000000000 is a uint, though it's too big to be an int.

A **long** resembles an integer but can be longer: it can be between -9223372036854775808 and 9223372036854775807. Each long consumes 8 bytes (64 bits) of RAM.

An **unsigned long** (or **ulong**) resembles a long but must not have a minus sign, and it can be between 0 and 18446744073709551615.

A **double-precision number** (or a **double**) contains a decimal point or an E. For example, -27.0 and 3E4 are doubles. A double can be up to 1.7976931348623158E308, and you can put a minus sign before it. Each double consumes 8 bytes of RAM. If you write a decimal point, put a digit (such as 0) after it.

Writing When Console.WriteLine makes the computer write an answer on your screen, the computer takes this shortcut: to write a double containing many digits after the decimal point, the computer writes just the first 15 significant digits; and if the only

digits after the decimal point are zeros, the computer doesn't bother writing those zeros or the decimal point.

Operations While you're writing a math problem, if you include a double (such as 5.0), the computer makes the answer be a double. For example, the answer to 5.0 + 3 is the double 8.0, though the computer doesn't bother writing the .0 on your screen.

If you feed the computer a problem that involves just ints, the computer tries to make the answer be an int. If the answer's too big to be an int, the computer gripes. For example, if you write —

```
Console.WriteLine(3000 * 1000000);
```

the computer will gripe (because 3000 and 1000000 are both ints but the answer is too big to be an int). You should rewrite the problem to include a double, like this —

```
Console.WriteLine(3000 * 1000000.0);
```

or —

```
Console.WriteLine(3000.0 * 1000000);
```

or:

```
Console.WriteLine(3000.0 * 1000000.0)
```

Then the answer will be a double (3000000000.0), which the computer will write on the screen in this shortcut form:

```
3000000000
```

If you feed the computer a math problem whose answer is too big to be a double, the computer will give up and typically say the answer is:

```
Infinity
```

The tiniest double that the computer handles well is 1e-308. If you feed the computer a math problem whose answer is tinier than that, the computer will either handle the rightmost digits inaccurately or give up, saying the answer is 0.0.

Dividing ints Since combining ints gives an answer that's an int, 11 / 4 is this int: 2. So 11 / 4 is not 2.75. If you say —

```
Console.WriteLine(11 / 4);
```

the computer will write just:

```
2
```

If you want the computer to write 2.75 instead, say you want a double, by putting decimal points in the problem, like this:

```
Console.WriteLine(11.0 / 4.0);
```

That makes the computer write:

```
2.75
```

Dividing by 0 If you ask the computer to divide by 0, the computer will gripe.

Dividing by 0.0 If you ask the computer to divide by 0.0, the computer will get creative.

For example, if you say —

```
Console.WriteLine(5.0 / 0.0);
```

the computer will try to divide 5.0 by 0.0, give up (because you can't divide by 0), and say the answer is:

```
Infinity
```

If you say —

```
Console.WriteLine(-5.0 / 0.0);
```

the computer will try to divide -5 by 0, give up (because you can't divide by 0), and say the answer is:

```
-Infinity
```

If you say —

```
Console.WriteLine(0.0 / 0.0);
```

the computer will try to divide 0 by 0, give up (because you can't divide by 0), get confused, and say the answer is —

```
NaN
```

which means "Not a Number".

Advanced math

The computer can do advanced math. For example, it can compute square roots. This line makes the computer print the square root of 9:

```
Console.WriteLine(Math.Sqrt(9));
```

The computer will print 3.

Besides Sqrt, you can use other advanced-math functions:

Function	Traditional notation	What to type
square root of x	\sqrt{x}	Math.Sqrt(x)
x raised to the y power	x^y	Math.Pow(x, y)
e raised to the y power	e^y	Math.Exp(y)
pi	π	Math.PI
e	e	Math.E
absolute value of x	$ x $	Math.Abs(x)
round x down, so ends in .0	$\lfloor x \rfloor$	Math.Floor(x)
round x up, so ends in .0	$\lceil x \rceil$	Math.Ceiling(x)
logarithm, base 10, of x	$\log_{10} x$	Math.Log10(x)
logarithm, base e, of x	$\ln x$	Math.Log(x)
logarithm, base b, of x	$\log_b x$	Math.Log(x, b)
sine of x radians	$\sin x$	Math.Sin(x)
cosine of x radians	$\cos x$	Math.Cos(x)
tangent of x radians	$\tan x$	Math.Tan(x)
arcsine of x, in radians	$\arcsin x$	Math.Asin(x)
arccosine of x, in radians	$\arccos x$	Math.Acos(x)
arctangent of x, in radians	$\arctan x$	Math.Atan(x)
arctangent of y/x, in radians	$\arctan y/x$	Math.Atan2(y, x)
hyperbolic sine of x	$\sinh x$	Math.Sinh(x)
hyperbolic cosine of x	$\cosh x$	Math.Cosh(x)
hyperbolic tangent of x	$\tanh x$	Math.Tanh(x)

Variables

Like Basic and other languages, C# lets you use variables. For example, you can say:

```
n = 3;
```

A variable's name can be short (such as n) or long (such as town_population_in_2001). The name can contain letters, digits, and underscores, but not blank spaces. The name must begin with a letter or underscore, not a digit.

Before using a variable, say what type of thing the variable stands for. For example, if n and town_population_in_2001 will stand for numbers that are ints and mortgage_rate will stand for a double, your program should say:

```
int n, town_population_in_2001;
double mortgage_rate;
```

If x is a variable, your program should say one these lines:

Line	Meaning
int x;	x is an integer
uint x;	x is an unsigned integer
long x;	x is a long
ulong x;	x is an unsigned long
double x;	x is a double-precision number
char x;	x is a single character, such as 'A'
string x;	x is a string of characters, such as "love"

If n is an integer that starts at 3, you can say —

```
int n;  
n = 3;
```

but you can combine those two lines into this single line:

```
int n = 3;
```

Here's how to say "n is an integer that starts at 3, and population_in_2001 is an integer that starts at 27000":

```
int n = 3, population_in_2001 = 27000;
```

If you want x to be the string "I love you", say —

```
string x;  
x = "I love you";
```

or combine those lines, like this:

```
string x = "I love you";
```

Increase

The symbol ++ means "increase". For example, ++n means "increase n".

These lines increase n:

```
int n = 3;  
++n;  
Console.WriteLine(n);
```

The n starts at 3 and increases to 4, so the computer prints 4.

Saying ++n gives the same answer as $n = n + 1$, but the computer handles ++n faster.

The symbol ++ increases the number by 1, even if the number is a decimal. For example, if x is 17.4 and you say ++x, the x will become 18.4.

Decrease

The opposite of ++ is --. The symbol -- means "decrease". For example, --n means "decrease n". Saying --n gives the same answer as $n = n - 1$ but faster.

Strange short cuts

If you use the following short cuts, your programs will be briefer and run faster.

Instead of saying $n = n + 2$, say $n += 2$, which means "n's increase is 2". Similarly, instead of saying $n = n * 3$, say $n *= 3$, which means "n's multiplier is 3".

Instead of saying ++n and then giving another command, say ++n in the middle of the other command. For example, instead of saying —

```
++n;  
j = 7 * n;
```

say:

```
j = 7 * ++n;
```

That's pronounced: "j is 7 times an increased n". So if n was 2, saying $j = 7 * ++n$ makes n become 3 and j become 21.

Notice that when you say $j = 7 * ++n$, the computer increases n *before* computing j. If you say $j = 7 * n++$ instead, the computer increases n *after* computing j; so $j = 7 * n++$ has the same effect as saying:

```
j = 7 * n;  
++n;
```

Input a string

These lines make the computer ask for your name:

```
Console.WriteLine("what is your name?");  
string x = Console.ReadLine();  
Console.WriteLine("I adore anyone whose name is " + x);
```

Below them, remember to put the helper line:

```
Console.ReadKey();
```

When you run that program (by pressing the F5 key), here's what happens....

The top line makes the computer write this question:

```
what is your name?
```

The next line makes the string x be the answer you type. For example, if you answer "What is your name?" by typing "Maria" (and then pressing Enter), the computer will read your answer and make string x be what the computer reads; so x will be "Maria", and the next line will make the computer write:

```
I adore anyone whose name is Maria
```

So when you run that program, here's the whole conversation that occurs between the computer and you:

```
The computer asks for your name: what is your name?  
You type your name: Maria  
Computer praises your name: I adore anyone whose name is Maria
```

Just for fun, run that program again and pretend you're somebody else....

```
The computer asks for your name: what is your name?  
You type your name: Bud  
Computer praises your name: I adore anyone whose name is Bud
```

When the computer asks for your name, if you say something weird, the computer will give you a weird reply....

```
The computer asks for your name: what is your name?  
You type: none of your business!  
The computer replies: I adore anyone whose name is none of your business!
```

Input a double

To make x be a string that the human inputs, you've learned to say this:

```
string x = Console.ReadLine();
```

To make x be a double-precision number that the human inputs, say this instead:

```
double x = Convert.ToDouble(Console.ReadLine());
```

That's because Console.ReadLine() considers the human's input to be a string, and Convert.ToDouble converts that string to a double.

Examples These lines make the computer predict how old a human will be ten years from now:

```
Console.WriteLine("How old are you?");  
double age = Convert.ToDouble(Console.ReadLine());  
Console.WriteLine("Ten years from now, you'll be " + (age + 10));
```

The top line makes the computer ask, "How old are you?" The middle line makes age be the result of converting, to a double-precision number, the human's input. The bottom line makes the computer write the answer.

For example, if the human is 27 years old, the chat between the computer and the human looks like this:

```
How old are you?  
27  
Ten years from now, you'll be 37
```

If the human is 27.5 years old, the chat can look like this:

```
How old are you?  
27.5  
Ten years from now, you'll be 37.5
```

These lines make the computer convert feet to inches:

```
Console.WriteLine("How many feet?");  
double feet = Convert.ToDouble(Console.ReadLine());  
Console.WriteLine("That makes " + (feet * 12) " inches.");
```

Input an integer

To make x be an integer that the human inputs, say this instead:

```
int x = Convert.ToInt32(Console.ReadLine());
```

That's because `Convert.ToInt32` converts a string to a 32-bit integer.

To make x be a special type of integer that the human inputs, say one of these:

```
uint x = Convert.ToUInt32(Console.ReadLine());
long x = Convert.ToInt64(Console.ReadLine());
ulong x = Convert.ToUInt64(Console.ReadLine());
```

Arrays

Instead of being just a number, x can be a *list* of numbers.

Example For example, if you want x to be this list of integers

```
{ 81, 52, 207, 19 }
```

type this:

```
int[] x = { 81, 52, 207, 19 };
```

In that line, the symbol “`int[]`” means “int list”. Notice that when you type the list of numbers, you must **put commas between the numbers** and put the entire list of numbers in **braces**, `{}`. On your keyboard, the “`{`” symbol is to the right of the P key and requires you to hold down the Shift key.

In x 's list, **the starting number** (which is 81) is called x_0 (which is pronounced “ x subscripted by zero” or “ x sub 0” or just “ x 0”). The next number (which is 52) is called x_1 (which is pronounced “ x subscripted by one” or “ x sub 1” or just “ x 1”). The next number is called x_2 . Then comes x_3 . So **the four numbers in the list are called x_0 , x_1 , x_2 , and x_3** .

To make the computer say what x_2 is, type this line:

```
Console.WriteLine(x[2]);
```

That line makes the computer write x_2 , which is 207, so the computer will write:

```
207
```

Altogether, the lines say:

```
int[] x = { 81, 52, 207, 19 };
Console.WriteLine(x[2]);
```

The first line says the integer-list x is `{ 81, 52, 207, 19 }`. The second line makes the computer write x_2 's number, which is 207.

Jargon Notice this jargon:

In a symbol such as x_2 , the lowered number (the 2) is called the **subscript**.

To create a subscript in your subroutine, use brackets. For example, to create x_2 , type `x[2]`.

A variable having subscripts is called an **array**. For example, x is an array if there's an x_0 , x_1 , x_2 , etc.

Different types Instead of having integers, you can have different types. For example, you can say:

```
double[] x = { 81.2, 51.7, 207.9, 19.5 };
```

You can even say:

```
string[] x = { "love", "hate", "peace", "war" };
```

Uninitialized Instead of typing a line that includes x 's list of numbers, you can type the numbers underneath, if you warn the computer how many numbers will be in the list, like this:

```
double[] x = new double[3];
x[0] = 200.1;
x[1] = 700.4;
x[2] = 53.2;
Console.WriteLine(x[0] + x[1] + x[2]);
```

The top line says x will be a new list of 3 doubles, called x_0 , x_1 , and x_2 . The next lines say x_0 is 200.1, x_1 is 700.4, and x_2 is 53.2. The bottom line makes the computer say their sum:

```
953.7
```

Tables If you want x to be a table having 2 rows and 3 columns of double-precision numbers, say:

```
double[,] x = new double[2, 3];
```

Since C# always starts counting at 0 (not 1), the number in the table's top left corner is called $x[0, 0]$.

Logic

Like most computer languages, C# lets you say “if”, “while”, “for”, and “goto” and create comments and subroutines. Here's how....

If

If a person's age is less than 18, let's make the computer say “You are still a minor.” Here's the fundamental line:

```
if (age < 18) Console.WriteLine("You are still a minor.");
```

Notice you must put parentheses after the word “if”.

If a person's age is less than 18, let's make the computer say “You are still a minor.” and also say “Ah, the joys of youth!” and “I wish I could be as young as you!” Here's how to say all that:

```
if (age < 18)
{
    cout <<"You are still a minor.\n";
    cout <<"Ah, the joys of youth!\n";
    cout <<"I wish I could be as young as you!";
}
```

Since that “if” line is above the “`{`”, the “if” line is a structure line and does *not* end in a semicolon.

How to type To type the symbol “`{`”, do this: while holding down the Shift key, tap the “[” key (which is next to the P key). To type the symbol “`}`”, do this: while holding down the Shift key, tap the “]” key.

When you type a line, don't worry about indenting it: when you finish typing the line (and press Enter), the computer will indent it the correct amount, automatically.

Complete program Here's how to put that structure into a complete program:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Joan
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("How old are you?");
            double age=Convert.ToDouble(Console.ReadLine());
            if (age < 18)
            {
                Console.WriteLine("You are still a minor.");
                Console.WriteLine("Ah, the joys of youth.");
                Console.WriteLine("I wish I could be as young as you!");
            }
            else
            {
                Console.WriteLine("You are an adult.");
                Console.WriteLine("Now we can have some adult fun!");
            }
            Console.WriteLine("Glad to have met you.");
            Console.ReadKey();
        }
    }
}
```

If the person's age is less than 18, the computer will write "You are still a minor." and "Ah, the joys of youth!" and "I wish I could be as young as you!" If the person's age is not less than 18, the computer will write "You are an adult." and "Now we can have some adult fun!" Regardless of the person's age, the computer will end the conversation by writing "Glad to have met you."

Since the computer types the top lines for you and also types the 3 braces at the program's bottom, you type just the lines in the middle, starting with:

```
Console.WriteLine("How old are you?");
```

Fancy "if" The "if" statement uses this notation:

Notation	Meaning
if (age < 18)	if age is less than 18
if (age <= 18)	if age is less than or equal to 18
if (age == 18)	if age is equal to 18
if (age != 18)	if age is not equal to 18
if (age < 18 && weight > 200)	if age < 18 and weight > 200
if (age < 18 weight > 200)	if age < 18 or weight > 200
if (sex == "male")	if sex is "male"
if (sex.CompareTo("male") < 0)	if sex is a word (such as "female") that comes before "male" in the dictionary

Here's how to type the symbol "|": while holding down the Shift key, tap the "\" key.

Look at that table carefully! Notice that in the "if" statement, you should use double symbols: you should say "==" instead of "=", say "&&" instead of "&", and say "||" instead of "|".

If you accidentally say "=" instead of "==", the computer will gripe. If you accidentally say "&" instead of "&&" or say "|" instead of "||", the computer will say right answers but too slowly.

The symbol "<" compares just numbers, not strings. Instead of writing —

```
if (sex < "male")
```

you must write:

```
if (sex.CompareTo("male") < 0)
```

While

Let's make the computer write the word "love" repeatedly, like this:

```
love love love love love love love love love etc.
love love love love love love love love love etc.
love love love love love love love love love etc.
etc.
```

This line does it:

```
while (true) Console.Write("love ");
```

The "while (1)" means: do repeatedly. The computer will do cout <<"love " repeatedly, looping forever — or until you **abort** the program (by clicking the console window's X button).

Let's make the computer start at 20 and keep counting, so the computer will write:

```
20
21
22
23
24
25
26
27
28
29
30
31
32
etc.
```

These lines do it:

Program	Meaning
int i = 20;	Start the integer i at 20.
while (true)	Repeat these lines forever:
{	
Console.WriteLine(i);	print i then press Enter
++i;	increase i
}	

They write faster than you can read.

- To pause the writing, press the Pause key.
- To resume the writing, press the Enter key.
- To abort the program, click the console window's X button.

In that program, if you say "while (i < 30)" instead of "while (true)", the computer will do the loop just while i remains less than 30; the computer will write just:

```
20
21
22
23
24
25
26
27
28
29
```

To let that program run properly, make sure its bottom includes the helper line saying "Console.ReadKey()", so altogether the program looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Joan
{
    class Program
    {
        static void Main(string[] args)
        {
            int i=20;
            while (i < 30)
            {
                Console.WriteLine(i);
                ++i;
            }
            Console.ReadKey();
        }
    }
}
```

Instead of saying "while (i < 30)", you can say "while (i <= 29)".

For

Here's a more natural way to get that output of numbers from 20 to 29:

```
for (int i = 20; i <= 29; ++i) Console.WriteLine(i);
```

The "for (int i = 20; i <= 29; ++i)" means:

Do repeatedly. Start the integer i at 20, and keep repeating as long as i <= 29. At the end of each repetition, do ++i.

In that "for" statement, if you change the "++i" to "i += 3", the computer will increase i by 3 instead of by 1, so the computer will write:

```
20
23
26
29
```

The "for" statement is quite flexible. You can even say "for (int i = 20; i < 100; i *= 2)", which makes i start at 20 and keep doubling, so the computer writes:

```
20
40
80
```

Like "if" and "while", the "for" statement can sit atop a group of indented lines that are in braces.

Goto

You can say "goto". For example, if you say "goto yummy", the computer will go to the line whose name is yummy:

```
Console.WriteLine("my dog ");
goto yummy;
Console.WriteLine("never ");
yummy: Console.WriteLine("drinks whiskey");
```

The computer will write:

```
my dog
drinks whiskey
```

Exceptions

These lines try to make x be how many children the human has:

```
Console.WriteLine("How many children do you have?");
int x = Convert.ToInt32(Console.ReadLine());
```

Those lines ask the human "How many children do you have?" then wait for the human's response then try to convert that string to an integer (such as 2 or 0) and call it x. But what happens if the human does *not* input an integer? What if human inputs a number that includes a decimal point? What if the human types a word, such as "none" or "one" or "many"? What if the human types a phrase, such as "not sure" or "too many" or "none of your business" or "my girlfriend was pregnant but hasn't told me yet whether she got an abortion"? In those errant situations (which are called **exceptions**), the computer can't do Convert.ToInt32 and will instead abort the program, show the human all the program's lines, and highlight the problematic line. Then the human will be upset and confused!

To avoid upsetting people, change those lines to this group of lines instead:

```
AskAboutKids:
    Console.WriteLine("How many children do you have?");
try
{
    int x = Convert.ToInt32(Console.ReadLine());
}
catch
{
    Console.WriteLine("Please type an integer");
    go to AskAboutKids;
}
```

The group begins with a label (AskAboutKids) and makes the computer ask "How many children do you have?" Then the computer will **try** to do this line:

```
int x = Convert.ToInt32(Console.ReadLine());
```

If the computer fails to do that line (because what the person typed can't be converted to an integer), the computer won't gripe; instead, it will **catch** the error and do the lines indented under "catch". Those lines are called the **catch block** (or **exception handler**). They make the computer say "Please type an integer" then go back to the beginning of AskAboutKids, to give the human another opportunity to answer the question correctly.

If the human doesn't know what an "integer" is, phrase the advice differently: make the computer write "Please type a simple number without a decimal point".

Comments

To put a comment in your program, begin the comment with the symbol `//`. The computer ignores everything that's to the right of `//`. Here's an example:

```
// This program is fishy
// It was written by a sick sailor swimming in the sun
Console.WriteLine("Our funny God"); // notice the religious motif
Console.WriteLine("invented cod"); // said by a nasty flounder
```

The computer ignores all the comments, which are to the right of `//`.

While you type the program, the computer makes each `//` and each comment turn green. Then the computer ignores everything that's turned green, so the computer writes just:

```
Our funny God
invented cod
```

Subroutines

Like most other languages, C# lets you invent subroutines and give them names. For example, here's how to invent a subroutine called "insult" and use it in the Main routine:

<u>Program</u>	<u>Meaning</u>
<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Threading.Tasks; namespace Joan { class Program { static void Main(string[] args) { Console.WriteLine("We all know..."); insult(); Console.WriteLine("...and yet we love you."); Console.ReadKey(); } static void insult() { Console.WriteLine("You are stupid!"); Console.WriteLine("You are ugly!"); } } }</pre>	<p>Here's the main routine:</p> <p>write "We all know..." do the insult write the ending</p> <p>Here's how to insult:</p> <p>write "You are stupid!" write "You are ugly!"</p>

The computer will write:

```
we all know...
You are stupid!
You are ugly!
...and yet we love you.
```

In that program, the lines beginning with "static void Main(string[] args)" define the Main routine. The bottom few lines, beginning with "static void insult()", define the subroutine called "insult".

Whenever you write a subroutine's name, you must put parentheses afterwards, like this: `insult()`. Those parentheses tell the computer: `insult`'s a subroutine, not a variable.

To write a subroutine's definition simply, begin the definition by saying "static void".

Here's another example of a main routine and subroutine:

<u>Routines</u>	<u>Meaning</u>
<pre>static void Main(string[] args) { laugh(); Console.ReadKey(); } static void laugh() { for (int i = 1; i <= 100; ++i) Console.Write("ha "); }</pre>	<p>Here's the main routine:</p> <p>main routine says to laugh</p> <p>Here's how to laugh:</p> <p>write "ha ", 100 times</p>

The Main routine says to laugh. The subroutine defines "laugh" to mean: write "ha " a hundred times.

Let's create a more flexible subroutine, so that whenever the Main routine says laugh(2), the computer will write "ha ha" and Enter; whenever the Main routine says laugh(5), the computer will write "ha ha ha ha" and Enter; and so on. Here's how:

Routines	Meaning
<pre>static void Main(string[] args) { Console.WriteLine("Here is a short laugh: "); laugh(2); Console.WriteLine("Here is a longer laugh: "); laugh(5); Console.ReadKey(); } static void laugh(int n) { for (int i = 1; i <= n; ++i) Console.WriteLine("ha "); Console.WriteLine(); }</pre>	<p>Here's the main routine:</p> <p>do laugh(2), so write "ha ha"</p> <p>do laugh(5), so write "ha ha ha ha"</p> <p>Here's how to laugh(n):</p> <p>write "ha", n times then press Enter</p>

The computer will print:

```
Here is a short laugh: ha ha
Here is a longer laugh: ha ha ha ha ha
```

Average Let's define the "average" of a pair of integers, so that "average(3, 7)" means the average of 3 and 7 (which is 5), and so a Main routine saying "i = average(3, 7)" makes i be 5.

This subroutine defines the "average" of all pairs of integers:

```
static int average(int a, int b)
{
    return (a + b) / 2;
}
```

The top line says, "Here's how to find the average of any two integers, a and b, and make the average be an integer." The next line says, "Return to the main routine, with this answer: (a + b) / 2."

Here's a complete program:

Program	Meaning
<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Threading.Tasks; namespace Joan { class Program { static void Main(string[] args) { int i; i = average(3, 7); Console.WriteLine(i); Console.ReadKey(); } static int average(int a, int b) { return (a + b) / 2; } } }</pre>	<p>Here's the main routine:</p> <p>make i be an integer make i be average(3, 7) write i</p> <p>Here's how to compute average(a, b):</p> <p>return this answer: (a + b) / 2</p>

In that program, the Main routine is:

```
int i;
i = average(3, 7);
Console.WriteLine(i);
Console.ReadKey();
```

make i be an integer
make i be average(3, 7)
write i

You can shorten it, like this:

```
int i = average(3, 7);
Console.WriteLine(i);
Console.ReadKey();
```

make the integer i be average(3, 7)
write i

You can shorten it further, like this:

```
Console.WriteLine(average(3, 7));
Console.ReadKey();
```

write average(3, 7)

To make that program handle double-precision numbers instead of integers, change each int to double. After changing to double, the program will still work, even if you don't change 3 to 3.0 and don't change 7 to 7.0.

Windows forms

Like Visual Basic, C# lets you easily create Windows forms. Here's how.

Start C# (by typing "vi" in the Windows 10 Search box, then clicking "Visual Studio 2015: Desktop app" then "New Project" then "Visual C#").

Click "Windows Forms Application".

Double-click in the Name box (which is near the screen's bottom). Type a name for your project (such as Joymaker). At the end of your typing, press the Enter key.

You see an **object**, called the **Form1** window. Double-click in that window (below "Form1"). That tells the computer you want to write a program (subroutine) about that window.

The computer starts writing the **subroutine** for you. The computer writes:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Joymaker
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
        }
    }
}
```

The line saying "private void Form1_Load" is the subroutine's **header**. Below that, between the **braces** (the symbols "{" and "}"), insert lines that tell the computer what to do when Form1 is **loaded** (appears). The lines you insert are called the subroutine's **body**.

Simplest example

Let's make the Form1 window say "I love you". To do that, type this line —

```
Text = "I love you";
```

The computer automatically indents that line for you, so the subroutine becomes:

```
private void Form1_Load(object sender, EventArgs e)
{
    Text = "I love you";
}
```

To run your program, click "Start" (which is at the screen's top center). Then you see the Form1 window again; but instead of saying "Form1", it tries to say the text:

```
I love you
```

(To see all that, maximize that Form1 window by clicking its **Maximize button**, which is left of its X.)

When you've finished admiring the Form1 window, stop the program by clicking the Form1 window's X button. Then you see the subroutine again:

```
private void Form1_Load(object sender, EventArgs e)
{
    Text = "I love you";
}
```

If you wish, edit the subroutine. For example, try changing the Text line to this:

```
Text = "I hate cabbage";
```

Math

The Text line can include math calculations, but you must convert the answer to a string, since Text must be a string. For example, to make the computer write the answer to 4 + 2, type this line:

```
Text = Convert.ToString(4 + 2);
```

Message box

To create a message box saying "Your hair is messy", type this line:

```
MessageBox.Show("Your hair is messy");
```

To create a message box saying the answer to 4 + 2, type this line:

```
MessageBox.Show(Convert.ToString(4 + 2));
```

Property list

Click the "Form1.cs [Design]" tab, so you see the Form1 window itself. Then click (just once) in the middle of the Form1 window.

Then the screen's bottom-right corner shows a list whose title is:

```
Properties
Form1 System.Windows.Forms.Form
```

That list is called **Form1's main property list** (or **property window**). It looks the same as if you were using Visual Basic. To explore it, reread my chapter about Visual Basic.

See the toolbox

Click the "**Form1.vb [Design]**" tab then "**View**" (which is near the screen's top-left corner) then "**Toolbox**". Then you see 10 **toolbox categories**:

```
All Windows Forms
Common Controls
Containers
Menus & Toolbas
Data
Components
Printing
Dialogs
WPF Interoperability
General
```

(If you don't see that whole list yet, scroll down.)

The toolbox looks the same way as if you were using Visual Basic. To explore how to use its tools, reread pages 607-615, starting with "See common controls".

Exotic languages

The previous 6 chapters explained 5 popular computer languages: **Basic** (especially the version called BBC Basic), **Visual Basic**, **Python**, **JavaScript** (especially the JScript version), and **Visual C#**.

Those 5 languages are just the tip of the iceberg. Programmers have invented *thousands* of others.

This table shows how to give popular commands in 26 languages:

Language	Assign variable	Condition	Start a counting loop	Output	Declare an array	Comment
Basic	j=k+2	if x=4.3	for i=5 to 17	print k	dim x(4)	'wow
Visual Basic	j = k + 2	If x = 4.3 Then	For i = 5 To 17	Console.WriteLine(k)	Dim x(4)	'wow
Java	j=k+2	if (x==4.3)	for (int i=5; i<=17; ++i)	System.out.println(k)	double[] x=new double[4]	//wow
JavaScript	j=k+2	if (x==4.3)	for (int i=5; i<=17; ++i)	document.write(k)	x=Array(4)	//wow
C	j=k+2	if (x==4.3)	for (i=5; i<=17; ++i)	printf("%d",k)	float x[4]	/* wow */
C++ & Visual C++	j=k+2	if (x==4.3)	for (int i=5; i<=17; ++i)	cout <<k	double x[4]	//wow
Visual C#	j = k + 2	if (x==4.3)	for (int i = 5; i <=17; ++i)	Console.WriteLine(k)	double[] x = new double[4]	//wow
Perl	\$j=\$k+2	if (\$x==4.3)	for (\$i=5; \$i<=17; ++\$i)	print \$k	@x=(1..4)	#wow
PHP	\$j=\$k+2	if (\$x==4.3)	for (\$i=5; \$i<=17; ++\$i)	echo \$k	\$x=range(1,4)	//wow
Algol	J:=K+2	IF X=4.3 THEN	FOR I := 5 STEP 1 UNTIL 17 DO	PRINT(K)	REAL ARRAY X[1:4]	COMMENT WOW
Pascal & Delphi	J:=K+2	IF X=4.3 THEN	FOR I := 5 TO 17 DO	WRITELN(k)	X: ARRAY[1..4] OF REAL	{WOW}
Modula	J:=K+2	IF X=4.3 THEN	FOR I := 5 TO 17 DO	WRITEINTEGER(K,6)	X: ARRAY[1..4] OF REAL	(*WOW*)
Ada	J:=K+2	IF X=4.3 THEN	FOR I IN 5..17 LOOP	PUT(K)	X: ARRAY(1..4) OF FLOAT	--WOW
Python	j=k+2	if x==4.3:	for i in range(5,18):	k	x=zeros(4)	#wow
Ruby	j=k+2	if x==4.3	for i in 5..17	puts k	x=(1..4)	#wow
Fortran	J=K+2	IF (X .EQ. 4.3)	DO 10 I=5,17	PRINT *, K	DIMENSION X(4)	C WOW
PL/I	J=K+2	IF X=4.3 THEN	DO I = 5 TO 17	PUT LIST(K)	DECLARE X(4)	/* WOW */
Cobol	COMPUTE J = K + 2	IF X = 4.3	PERFORM L VARYING I FROM 5 BY 1 UNTIL I>17	DISPLAY K	X OCCURS 4 TIMES	*WOW
dBase	J=K+2	IF X=4.3	not available	? K	DECLARE X[4]	&&WOW
Easy	let j=k+2	if x=4.3	loop i from 5 to 17	say k	prepare x(4)	'wow
Snobol	J = K + 2	EQ(X,4.3) :S(not available	OUTPUT = K	X = ARRAY(4)	*WOW
Pilot	C:#J=#K+2	(#X=4.3)	not available	T:#K	DIM:#X(4)	R:WOW
Lisp	(SETQ J (PLUS K 2))	IF :X=4.3	not available	K	(ARRAY ((X (4) LIST)))	;WOW
Logo	MAKE "J :K+2	IF :X=4.3	not available	PRINT :K	DEFAR "X 4 1	!WOW

That table clumps the languages into groups. For example, the first group includes Basic and Visual Basic.

In each group, I list the languages in the order they were invented. For example, in the first group, Basic was invented before Visual Basic, so Basic is listed first.

The bottom 4 (Snobol, Pilot, Lisp, and Logo) differs wildly from the others. They're called **radical languages**; the other 22 languages are called **mainstream**.

Each of those 26 languages is flexible enough to program anything. Which language you choose is mainly a matter of personal taste.

Other languages are more specialized. For example, a language called **GPSS** is designed specifically to analyze how many employees to hire, to save your customers from waiting in long lines for service. **Dynamo** analyzes social interactions inside your company and city and throughout the world then graphs your future. **Prolog** lets you store answers to your questions and act as an **expert system**.

This table reveals more details about all those languages:

Name	What the name stands for	Original use	Version 1 arose at	When	Names of new versions
Mainstream languages					
Fortran	Formula Translating	sciences	IBM	1954-1957	Fortran 2018, Lahey Fortran
Algol	Algor ithmic Language	sciences	international	1957-1958	Algol W, Algol 68, Balgol
Cobol	Common Business-Oriented Language	business	Defense Department	1959-1960	Cobol 2014
Basic	B eginners A ll-purp. S ymbolic I nstruc. C ode	sciences	Dartmouth College	1963-1964	GW Basic, QBasic, QB64, BBC Basic
PL/I	P rogramming Language O ne	general	IBM	1963-1966	PL/I Optimizer, PL/C, Ansi PL/I
Pascal	Blaise Pascal	general	Switzerland	1968-1970	Turbo Pascal, Delphi
Modula	M odular programming	systems programming	Switzerland	1975	Modula-2, Oberon
C	beyond B	systems programming	AT&T's Bell Labs	1971-1973	Ansi C, Objective-C
Ada	A da Lovelace	military equipment	France	1977-1980	Ada final version
dBase	D ata B ase	database management	Jet Prop'n Lab & Ashton-T.	1978-1980	dBase 2019, Visual FoxPro 9
Easy	E asy	general	Secret Guide	1972-1982	Easy
C++	C increased	systems programming	AT&T's Bell Labs	1979-1983	Borland C++, ISO C++
Perl	P ractical E xtraction and R eport Language	systems programming	Unisys	1987	Perl 5
Python	as fun as Monty P ython's Flying Circus	systems programming	Netherlands	1989	Python 3
Java	as stimulating as J ava coffee	Web-page animation	Sun Microsystems	1990-1995	Java 7, JBuilder, Visual J++, J#
Visual Basic	B asic for creating Windows V isually	Windows-form design	Microsoft	1991	Visual Basic 2019
Visual C++	C ++ for creating Windows V isually	Windows-form design	Microsoft	1993	Visual C++ 2019
Delphi	oracle at D elphi	general	Borland	1993-1995	Delphi 10, Oxygene
Ruby	the birthstone beyond Pearl	systems programming	Japan	1993-1996	Ruby 3, Ruby on Rails
PHP	P ersonal H ome P age	Web-page design	Canada	1994-1995	PHP 8
JavaScript	J ava for creating simple s cripts	Web-page calculations	Netscape	1996	JScript
Visual C#	C sharp for creating Windows V isually	Windows-form design	Microsoft	1999-2000	Visual C# 2019
Radical languages					
Lisp	L ist Processing	artificial intelligence	MIT	1958-1960	Common Lisp
Snobol	S tring-Oriented symbolic Language	string processing	AT&T's Bell Labs	1962-1963	Snobol 4
Logo	L ogo	general	Bolt Beranek Newman	1967	Terrapin Logo, LCSI MicroWorlds Pro
Pilot	P rogrammed I nquiry, L earning, O r T eaching	tutoring kids	U. of Cal. at San Francisco	1968	Atari Pilot
Specialized languages					
Dynamo	D ynamic Models	simulation	MIT	1959	Dynamo 4, Stella
GPSS	G eneral- P urpose S imulation S ystem	simulation	IBM	1961	GPSS 5, GPSS/H, GPSS/PC
Prolog	P rogramming in L ogic	artificial intelligence	France	1972	Arity Prolog, Turbo Prolog

Within each category ("mainstream", "radical", and "specialized"), I listed the languages in chronological order.

Of those 29 languages, 6 were invented in Europe (Algol, Pascal, Modula, Ada, Python, and Prolog), 1 in Japan (Ruby), and 1 in Canada (PHP). The other 21 were invented in the USA.

3 were invented at IBM (Fortran, PL/I, and GPSS), 3 at Microsoft (Visual Basic, Visual C++, and Visual C#), 3 at AT&T's Bell Labs (C, C++, and Snobol), 2 at MIT (Lisp and Dynamo), and 2 by Professor Niklaus Wirth in Switzerland (Pascal and Modula). The rest were invented by geniuses elsewhere.

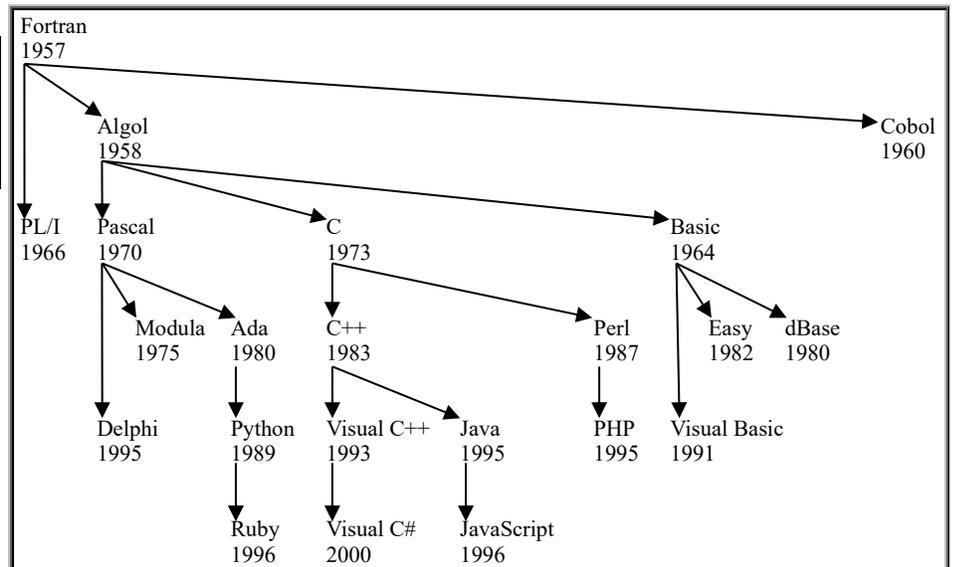
Mainstream languages

The first mainstream languages were **Fortran**, **Algol**, and **Cobol**. They were the **big 3**.

IBM invented **Fortran**, which appealed to engineers.
An international committee invented **Algol**, which appealed to logicians.
A committee based at the Pentagon invented **Cobol**, which appealed to government bureaucrats and business managers.

Beyond the big 3

Other mainstream languages came after the big 3 and were just slight improvements of the big 3. This family tree shows how the mainstream languages influenced each other:



In that tree, a vertical line means "a direct influence" (like a parent); a slanted line means "an indirect influence" (like an aunt or uncle). For each language, I show the year when the language's first version was complete. As each language grew, it stole features from other languages (just like English stole the word "restaurant" from French); the tree shows just history's main thrust.

The tree's third row has 4 languages: PL/I, Pascal, C, and Basic. Here's why they were invented....

Why PL/I? After inventing **Fortran** and further improvements (called Fortran II, Fortran III, Fortran IV, and Fortran V), IBM decided to invent the "ultimate" improvement: a language that would include all the important words of Fortran V and **Algol** and **Cobol**. At first, IBM called it "Fortran VI"; but since it included the best of everything and was the first complete language ever invented, IBM changed its name to **Programming Language One** (written as **PL/I**).

IBM bragged that PL/I was eclectic, but most programmers considered it a confusing mishmash and continued using the original 3 languages (Fortran, Algol, and Cobol), which were pure and simple.

Why Pascal? Among the folks who disliked PL/I was Niklaus Wirth, who preferred **Algol**. At a Swiss university, he invented an improved Algol and called it **Pascal**. Then he invented **Modula**, which he thinks is even better, but critics disagree. Pascal is the most popular of that trio. (Hardly anybody uses the original Algol anymore, and Modula is considered a controversial experiment.)

A company called **Borland** became famous by developing **Turbo Pascal** (a Pascal version that runs fast on DOS) then **Delphi** (which resembles Turbo Pascal but run on Windows and lets you create your own windows).

The Department of Defense happily used Cobol to run the military's paperwork bureaucracy but needed a more science-oriented language, to control missiles and other military equipment. The Department held a contest to develop such a language and said it wanted the language to resemble PL/I, Algol, and Pascal. (It didn't know about Modula, which was still being developed.) The winner was a French company. The Department adopted that company's language and called it **Ada**. It resembled Modula but included more commands — and therefore consumed more RAM and was more expensive. Critics complain that Ada, like PL/I, is too big and complex. But Ada inspired **Python** and **Ruby**, which are smaller and popular.

Why Basic? Two professors at Dartmouth College combined Fortran with Algol, to form **Basic**. It was designed for students, not professionals: it included just the *easiest* parts of Fortran and Algol. Students liked it because it was easy to learn, but professionals complained it lacked advanced features.

Basic's first version ran on a maxicomputer. Later, Digital Equipment Corporation (DEC) invented versions for minicomputers, and Microsoft invented many microcomputer versions, such as **QBasic**.

After Microsoft invented Windows, Microsoft invented **Visual Basic**, which runs on Windows, lets you create your own windows, and includes advanced features.

Basic inspired me to invent a language called **Easy**, which is even easier to learn than Basic but hasn't yet been put on any computer fully.

Inspired by languages such as Basic and PL/I, Wayne Ratliff invented **dBase**. Like Basic, dBase is easy. What makes dBase unique is its wonderful commands for manipulating databases.

Why C? Fancy languages, such as PL/I and Modula, require lots of RAM. At AT&T's Bell Labs, researchers needed a language small enough to fit in the tiny RAM of a minicomputer or microcomputer. They developed the ideal tiny language and called it **C**. Like PL/I, it borrows from Fortran, Algol, and Cobol; but it lacks PL/I's frills. It's "lean and mean" and runs very fast.

Later, Bell Labs invented an improved C, called **C++**, which includes extra commands. Microsoft invented **Visual C++**, which adds commands for manipulating windows. Then Anders

Hejlsberg (the Danish programmer who developed Turbo Pascal and Delphi at Borland) moved to Microsoft, where he invented **Visual C#**, which tries to combine the best features of Visual C++, Turbo Pascal, and Delphi.

Sun Microsystems invented a C++ variant called **Java**, to handle Web-page programming (such as animation). Netscape invented **JavaScript**, which resembles Java but is simpler and more limited. C also led to **Perl & PHP**, which handle Web-page programming and compete against Java & JavaScript.

Look back! Let's take a closer look at the oldest of those mainstream languages, the ones invented up through 1983....

Fortran

During the early 1950's, the only available computer languages were specialized or awkward. **Fortran** was the first computer language good enough to be considered mainstream. Algol and Cobol came shortly afterwards. Fortran, Algol, and Cobol were so good they made all earlier languages obsolete.

Fortran's tortures On pages 477-527, I explained how to program in Basic. Fortran resembles Basic but is weirder — because Fortran was invented before programmers learned how to make programming languages pleasant. Here's how Fortran tortures you.

For example, suppose you want to add 2+2. In Basic, you can say just:

```
PRINT 2+2
```

In Fortran, you must lengthen the program, so it looks like this instead:

```
N=2+2
PRINT *, N
END
```

Here's why:

Fortran requires the program's bottom line to say END.

Fortran requires each line to be indented 6 spaces.

Fortran is too stupid to do math in the middle of a PRINT statement, so you must do the math first, in a separate line (N=2+2).

Fortran expects you to comment about *how* to print the answer. If you have no comment on that topic, put an asterisk and comma in the PRINT statement. The asterisk and comma mean: no comment.

That's how the typical version of Fortran works. Some versions are different. For example, some versions require you to say STOP above END, like this:

```
N=2+2
PRINT *, N
STOP
END
```

Some versions want you to say TYPE instead of PRINT.

Some old versions won't accept "no comment" about printing. They require you to say:

```
N=2+2
PRINT 10, N
10 FORMAT (1X,I1)
END
```

That PRINT line means: PRINT, using the FORMAT in line 10, the value of N. In line 10, the 1X means "normal"; the I1 means "an integer that's just one digit". Those details drive beginners nuts, but experienced Fortran programmers are used to such headaches and take them in stride, just like Frenchmen are used to conjugating French verbs and Germans are used to conjugating German adjectives. Yuck!

Like Basic, Fortran lets you do math by using these symbols:

```
+ - * /
```

But Fortran is harder to learn than Basic:

To divide 399 by 100 in Basic, you can write 399/100 to get the correct answer, 3.99. But in Fortran, requesting 399/100 makes the computer assume you don't care about decimal points (since you didn't mention any), so it says just 3; if you want the computer to say 3.99 instead, you must insert a decimal point into the original problem, by asking for 399.0/100.0 (or at least asking for 399./100, if you're lazy).

Basic lets you use the symbol < to mean "less than". Fortran is afraid to use fancy symbols (since ancient keyboards didn't have them), so Fortran wants you to write .LT. instead, like this....

Basic: IF x < 4.3 THEN

Fortran: IF (X .LT. 4.3) THEN

Likewise, Fortran requires you to say .GT. instead of > for "greater than", say .LE. for "less than or equal to", say .GE. for "greater than or equal to", and, for consistency, say .EQ. for "equals" in an IF statement....

Basic: IF x = 4.3 THEN

Fortran IF (X .EQ. 4.3) THEN

In Basic, the symbol ^ means exponents (for example, 4.7 ^ 3 means "4.7 times 4.7 times 4.7"). Since Fortran's afraid of fancy symbols, Fortran uses ** instead of ^ (like this: 4.7 ** 3).

In Basic, a variable can be any letter of the alphabet (such as n) or a longer name (up to 40 characters long). In Fortran, each variable's name must be short (no longer than 6 characters), because Fortran is supposed to run even on primitive old computers having little memory.

Basic assumes each variable is single-precision real (unless you specifically indicate otherwise, such as by putting a \$ at the end of the variable's name to indicate the variable's a string). Fortran is more complicated: it assumes any variable whose name begins with I, J, K, L, M, or N is an integer, and all other variables are single-precision real (unless you indicate otherwise). Since Fortran assumes that variables beginning with I, J, K, L, M, or N are integers, Fortran programmers purposely misspell variable names. For example, if a Fortran variable's purpose is to count, call it KOUNT (rather than COUNT) to make it an integer. If you want a Fortran variable to be an integer that measures a position, call it LOCATN (rather than POSITN) to make it an integer. If a Fortran variable measures an object's mass as a real number, call it AMASS (rather than MASS) to make it a real.

Since Fortran's purpose was just to do math, Fortran's original version didn't include any string variables. Later, many manufacturers added string commands, but they're much more awkward than Basic's.

Fortran's advantages Fortran can handle complex numbers (such as the square root of -1). Fortran's ability to handle complex numbers make it better for advanced math & engineering than Basic.

Fortran did well at handling math functions (such as square roots) and subroutines (for handling statistics, calculus computations, and other math challenges). Many programmers created Fortran math functions and subroutines, organized them into libraries, and sold them to other programmers. A whole culture developed of programmers writing Fortran routines. If you didn't know Fortran, you weren't part of the "in" crowd.

How Fortran arose In 1954, an IBM committee planned a new computer language to help engineers make the computer handle math formulas. The committee called the language **Fortran**, to emphasize that the language would be particularly good for **translating formulas** into computer notation.

Those original plans for Fortran were modest:

They did *not* allow long variable names, subroutines, long function definitions, double precision, complex numbers, or apostrophes. A variable's name had to be short: just two letters. A function's definition had to fit on a single line. To print 'PLEASE KISS ME', the programmers had to write that string as 14HPLEASE KISS ME instead of 'PLEASE KISS ME'; the 14H warned the computer that a 14-character string was coming.

Then came improvements:

Fortran's first working version (1957) allowed longer variable names: 6 characters. Fortran 2 (1958) allowed subroutines and long function definitions. IBM experimented with Fortran 3 but never released it to the public. Fortran 4 (1962) allowed double precision and complex numbers. Apostrophes around strings weren't allowed until later.

The original plans said you'd be able to add an integer to a real. That didn't work in Fortran 1, Fortran 2, and Fortran 4 but works now.

The original plans said an IF statement would compare any two numbers. Fortran 1 and Fortran 2 required the second number to not be zero, but Fortran 4 removed that restriction.

IBM tried to convince everyone that Fortran was easier than previous methods of programming. IBM succeeded: Fortran became popular. Fortran was easy enough so that, for the first time, engineers who weren't computer specialists could write programs.

Other manufacturers sold their own variations of IBM's Fortran. Those variations annoyed engineers, who wished manufacturers would all use a single, common version of Fortran. So the engineers turned to the **American National Standards Institute (Ansi)**, which is a non-profit group of engineers that sets standards.

"Ansi" is pronounced "an see". It sets standards for practically all equipment in your life. For example, Ansi sets the standard for screws: to tighten a screw, you turn it clockwise, not counterclockwise.

In 1966, Ansi decided on a single version of Fortran 4 to be used by all manufacturers. Afterwards, each manufacturer obeyed the Ansi standard but also added extra commands, to try to outclass the other manufacturers. Engineers asked Ansi to meet again and develop a common standard for those extras. Ansi finished developing the standard in 1977 and called it **Fortran 77**. Then came **Fortran 90**, **Fortran 95**, **Fortran 2003**, **Fortran 2008**, and **Fortran 2018**.

Fortran's popularity. During the 1960's and 1970's, Fortran was the most popular computer language among engineers, scientists, mathematicians, and college students. Colleges required all freshman computer-science majors to learn Fortran.

But at the end of the 1970's, Fortran's popularity began to drop.

Engineers switched to newer languages, such as Basic (which is easier), Pascal (more logical), and C (faster and consuming less RAM). Though Fortran 77 included extra commands to make Fortran resemble Basic and Pascal, those commands were "too little, too late": Fortran's new string commands weren't quite as good as Basic's, and Fortran's new IF command wasn't quite as good as Pascal's.

Now kids study Basic, Python, Pascal, Java, C++, or C#, and hardly anybody studies Fortran. People who still program in Fortran are called "old-fashioned".

But in these ways, Fortran's still the best for engineering:

Fortran includes more commands for handling "complex numbers".

Fortran programmers have developed libraries containing *thousands* of Fortran subroutines, which you can use in your own Fortran programs. Such large libraries haven't been developed for other languages yet.

Algol

In 1955, a committee in Germany began inventing a computer language. Though the committee spoke German, it decided the computer language should use English words instead, since English was the international language for science.

In 1957 those Germans invited Americans to join them. In 1958 other European countries joined also, to form an international committee, which proposed a new computer language, called "IAL" (International Algebraic Language).

The committee eventually changed the language's name to **Algol 58** (the **Algorithmic language** invented in **1958**), then created an improved version called **Algol 60**, then created a further revision called **Algol 60 Revised**, and disbanded. Today, programmers who mention "Algol" usually mean the committee's last report, Algol 60 Revised.

Algol differs from Fortran in many little ways....

How to end a statement At the end of each statement, Fortran requires you to press the Enter key. Algol requires you to type a semicolon instead.

Algol's advantage: you can type many statements on the same line, by putting semicolons between the statements. Algol's disadvantage: those ugly semicolons are a nuisance to type and make your program look cluttered.

Integer variables To tell the computer that a person's AGE is an integer (instead of a real number), Fortran expects you to put the letter I, J, K, L, M, or N before the variable's name, like this: IAGE. Algol requires you to insert a note saying "INTEGER AGE" at the top of your program instead.

Algol's advantage: it doesn't encourage you to write unpronounceable gobbledygook such as "IAGE". Algol's disadvantage: whenever you create a new variable, Algol forces you to go back to the program's top and insert a line saying "INTEGER" or "REAL".

Assignment statements In Fortran, you can say J=7. In Algol, you must insert a colon and say J:=7 instead. To increase K by 1 in Fortran, you say K=K+1. In Algol, you say K:=K+1.

Algol's disadvantage: the colon is a nuisance to type. Fortran's disadvantage: according to the rules of algebra, it's impossible for K to equal K+1, so the Fortran command K=K+1 looks like an impossibility.

Algol's beauty Here's how Algol avoids Fortran's ugliness:

In Algol, a variable's name can be practically as long as you like. In Fortran, a variable's name must be short: no more than 6 characters.

Algol lets you write 2 instead of 2.0, without affecting the computer's answer. In Fortran, if you write 1/2 instead of 1/2.0, you get 0 instead of .5; and if you write SQRT (9) instead of SQRT (9.0), you get nonsense.

Algol's IF statement is very flexible: it can include the words ELSE, BEGIN, and END, and it lets you insert as many statements as you want between BEGIN and END. Algol even lets you put an IF statement in the middle of an equation, like this: X:=2+(IF Y<5 THEN 8 ELSE 9). The IF statement in Fortran I, II, III, and IV was very limited; the IF statement in Fortran 77 copies some of Algol's power, but not yet all.

Algol's FOR statement is very flexible. To make X be 3.7, then be Y+6.2, then go from SQRT(Z) down to 5 in steps of .3, you can say "FOR X:=3.7, Y+6.2, SQRT(Z) STEP -.3 UNTIL 5 DO". Fortran's DO is more restrictive; some versions of Fortran even insist that the DO statement contain no reals, no negatives, and no arithmetic operations.

At the beginning of a Fortran program, you can say DIMENSION X(20) but not DIMENSION X(N). Algol permits the "DIMENSION X(N)" concept; in Algol you say ARRAY X[1:N].

Algol's popularity When Algol was invented, programmers loved it. Europeans began using Algol more than Fortran. The American computer association (called the **Association for Computing Machinery, ACM**) said all programs in its magazine would be in Algol.

But since IBM refused to put Algol on its computers, most American programmers couldn't use Algol.

That created a ridiculous situation: American programmers programmed in Fortran but submitted Algol translations to the ACM's magazine, which published the programs in Algol, which the magazine's readers had to translate back to Fortran to run on IBM computers. IBM computers eventually swept over Europe, so even Europeans had to use Fortran instead of Algol.

In 1966 the ACM gave in and agreed to publish programs in Fortran; but since Algol was prettier, everybody continued to submit Algol versions anyway. IBM gave in also and put Algol on its computers; but IBM's version of Algol was so limited and awkward that nobody took it seriously, and IBM stopped selling it. In 1972 Stanford University invented **Algol W** (a better Algol for IBM computers), but Algol W came too late: universities and businessmen had already tired of waiting for a good IBM Algol and committed themselves to Fortran.

Critics blamed IBM for Algol's demise. But here's IBM's side of the story:

IBM had invested 25 man-years to develop the first version of Fortran. By the time the Algol committee finished the report on Algol 60 Revised, IBM had also developed Fortran II and Fortran III and made plans for Fortran IV. IBM was proud of its Fortrans and wanted to elaborate on them. Moreover, IBM realized that **computers run Fortran programs faster than Algol**.

When asked why it didn't support Algol, IBM replied that the committee's description of Algol was incomplete. IBM was right; the Algol 60 Revised Report had 3 loopholes:

The report didn't say what words to use for input and output, because the committee couldn't agree. So computers differ. If you want to transfer an Algol program from one computer to another, you must change all the input and output instructions.

The report uses symbols such as + and ^, which most keyboards lack. The report underlines keywords; most keyboards can't underline. To type Algol programs on a typical keyboard, you must substitute other symbols for +, ^, and underlining. Manufacturers differ in what to substitute. To transfer an Algol program to different manufacturer, you must change symbols.

Some features of Algol are hard to teach to a computer. Even now, no computer understands all of Algol. When a manufacturer says its computer "understands Algol", you must ask, "Which features of Algol?"

Attempts to improve Algol Long after the original Algol committee wrote the Algol 60 Revised Report, two other Algol committees were formed.

One committee developed suggestions on how to do input and output, but its suggestions were largely ignored.

The other committee tried to invent a much fancier Algol. That committee wrote its preliminary report in 1968 and revised it in 1975. Called **Algol 68 Revised**, that weird report requires you to spell words backwards: to mark the end of the IF statement, you say FI; to mark the end of the DO statement, you say OD. The committee's decision was far from unanimous: several members refused to endorse the report.

Algol now Few programmers still use Algol, but many use Pascal (which is very similar to Algol 60 Revised) and Basic (which is a compromise between Algol and Fortran).

Cobol

If you're going to give a speech or write a paper, teachers recommend you organize your thinking by creating an outline. Back in the 1950's, managers of computer departments got together and decided programmers should organize programs in the same way: create an outline before writing the program, especially since a well-organized program is easier for the company to analyze and improve when the original programmer gets fired.

Those managers invented a computer language that lets the programmer just fill in an outline and feed the outline to the computer. The outline itself acts as the program. No further programming is necessary.

That outline-oriented computer language is used for handling tough programming problems in business accounting (such as payroll, inventory, accounts payable, and accounts receivable), so it was named the **Common Business-Oriented Language** (whose abbreviation is **Cobol**, which is pronounced "koe ball"). But cynics complain that "Cobol" also stands for **Completely Obsolete Business-Oriented Language and Compiles Only Because Of Luck**.

4 parts To write a program in Cobol, just fill in an outline that has 4 parts:

In the first part, called the **IDENTIFICATION DIVISION**, you give your name (so your boss knows who to fire when the program doesn't work) and comments about when you wrote the program, the program's name, and security (who's allowed to see this program). The computer ignores everything you say in the IDENTIFICATION DIVISION, but writing that stuff makes your boss happy.

In the second part, called the **ENVIRONMENT DIVISION**, you say what kind of environment you wrote the program for: which computer it runs on, which devices the program's files use (disks? tapes? printers? punched cards?), and whether decimal points should be printed as commas instead (since people in France, Italy, and Germany want you to do that).

In the third part, called the **DATA DIVISION**, you list all the program's variables. For each numeric variable, you must say how many digits it should store (to the left and right of the decimal point) and how to format the number (for example, say whether to print a dollar sign before the number). For example, if you want N to be a 3-digit integer (from 000 to 999) without special formatting, say N PICTURE IS 999 (which means N is a variable whose picture is at most the number 999). If you want N to be a 7-digit integer (from 0000000 to 9999999), say N PICTURE IS 9999999. If you want N to be a 7-character string, say N PICTURE IS XXXXXXXX. You can abbreviate: you can say just PIC instead of PICTURE IS, and you can say X(7) instead of XXXXXXXX.

In the fourth and final part, called the **PROCEDURE DIVISION**, you finally write the procedures you want the computer to perform, using commands such as READ, WRITE, DISPLAY, ACCEPT, IF, GO TO, SORT, MERGE, and PERFORM. Each command's an English sentence that includes a verb and ends in a period. You organize the PROCEDURE DIVISION into paragraphs, invent a name for each paragraph, treat each paragraph as a separate procedure/subroutine, and tell the computer in what order to PERFORM the paragraphs. One line in the PROCEDURE DIVISION must say "STOP RUN": when the computer encounters that line, the computer stops running the program.

Unfortunately, that idea of dividing a program into 4 divisions is wrong-headed: when you write or read a Cobol program, your eye must keep hopping between the PROCEDURE DIVISION (where the action is) and the DATA DIVISION (which tells what the variables mean), while taking an occasional peek at the ENVIRONMENT DIVISION (which tells what devices are involved). Other programming languages, developed later, use better ways to organize thoughts.

How Cobol arose During the 1950's, several organizations developed languages to solve problems in business. The most popular business languages were IBM's **Commercial Translator** (developed from 1957-1959), Honeywell's **Fact** (1959-1960), Sperry Rand's **Flow-matic** (1954-1958), and the Air Force's **Aimaco** (1958).

In April 1959, a group of programmers and manufacturers met at the University of Pennsylvania, decided to develop a *single* business language for *all* computers, and asked the Department of Defense to sponsor the research. The Department agreed.

In a follow-up meeting held at the Pentagon in May, the group tentatively decided to call the new language "CBL" (for "Common Business Language") and created 3 committees.

The Short-Range Committee would meet immediately to develop a temporary language. A Medium-Range Committee would meet later to develop a more thoroughly thought-out language. Then a Long-Range Committee would develop the ultimate language.

The Short-Range Committee met immediately and created a language nice enough so the Medium-Range and Long-Range Committees never bothered to meet.

The Short-Range Committee wanted a more pronounceable name for the language than "CBL". At a meeting in September 1969, the committee members proposed 6 names:

"BUSY" (BUiness SYstem)
"BUSYL" (BUiness SYstem Language)
"INFOSYL" (INFORmation SYstem Language)
"DATASYL" (DATA SYstem Language)
"COSYL" (COmmon SYstem Language)
"COCOSYL" (COmmon COmputer SYstem Language)

But the next day, a member of the committee suggested "Cobol" (**Common Business-Oriented Language**), and the rest of the committee agreed.

I wish they'd have kept the name "BUSY", because it's easier to pronounce and remember than "Cobol". Today, Cobol programmers are still known as "BUSY bodies".

From Sperry Rand's Flow-matic, the new language (called "Cobol") borrowed 2 rules:

Begin each statement with an English verb.
Put data descriptions in a different program division than procedures.

From IBM's Commercial Translator, Cobol borrowed fancy IF statements, COMPUTE formulas, PICTURE symbols (for showing how to format the numbers and strings), and group items (called 01 and 02, which let a variable stand for a whole collection of data).

Compromises On some issues, the committee's members had to compromise.

For example, some members wanted Cobol to let programmers write mathematical formulas by using these symbols:

+ - * / = ()

But other members of the committee disagreed: they said that since Cobol is for stupid businessmen who fear formulas, Cobol should use the words ADD, SUBTRACT, MULTIPLY, and DIVIDE instead. The committee compromised:

When you write a Cobol program, you can use the words ADD, SUBTRACT, MULTIPLY, and DIVIDE. You can use a formula instead but just if you warn the computer by putting the word COMPUTE before the formula.

Can Cobol handle long numbers? How long? The committee decided that Cobol would handle any **number up to 18 digits long** and handle any **variable name up to 30 characters long**. So the limits of Cobol are "18 digits, 30 characters". Here's why:

Some manufacturers wanted "16 digits, 32 characters", because their computers were based on the numbers 16 and 32; but other manufacturers wanted other combinations (such as "24 digits, 36 characters"). The committee, hunting for a compromise, chose "18 digits, 30 characters" because *nobody* wanted it, and so it would give no manufacturer an unfair advantage over competitors. Yes, Cobol was designed to be equally terrible for everybody! That's politics!

Cobol's popularity In 1960, the Defense Department announced it would buy just computers that understand Cobol, unless a manufacturer can demonstrate why Cobol isn't helpful. In 1961, Westinghouse Electric Corp. made a similar announcement. Other companies followed. Cobol became the most popular computer language.

Today it's still the most popular computer language for maxicomputers, though programmers on minicomputers and microcomputers have switched to newer languages.

Improvements The original version of Cobol was finished in 1960 and called **Cobol 60**. Then came an improvement, called **Cobol 61**. In 1962, the verb SORT and a "Report Writer" feature were added. Then came **Cobol 65**, **Cobol 68**, **Cobol 74**, **Cobol 85**, **Cobol 2002**, and **Cobol 2014**.

Cobol's most obvious flaw Cobol requires you to put info about file labeling into the **data division's** FD command. But since file labeling describes the environment, not the data, Cobol should have put file labeling in the **environment division** instead.

Jean Sammet, who headed some of the Short-Term Committee's subcommittees, admits her group goofed when it put file labeling in the data division. But Cobol's too old to change now.

Basic

The first version of **Basic** was developed in 1963 and 1964 by a genius (**John Kemeny**) and his friend (**Tom Kurtz**).

How the genius grew up John Kemeny was a Jew born in Hungary in 1926. In 1940, he & his parents fled the Nazis and came to America. When he began high school in New York City, he knew hardly any English; but he learned enough so he graduated as the top student in the class. 4 years later, he graduated from Princeton **summa cum laude** even though he had to spend 1½ of those years in the Army, where he helped solve equations for the atom bomb.

2 years after his B.A., Princeton gave him a Ph.D. in mathematics *and* philosophy, because his thesis on symbolic logic combined both fields.

While working for the Ph.D., he was Einstein's youngest assistant. He told Einstein he wanted to quit math and instead hand out leaflets for world peace, but Einstein said leafletting would waste his talents: the best way for him to help world peace would be to become a famous mathematician, so people would *listen* to him, as they had to Einstein. He took Einstein's advice and stayed with math.

After getting his Ph.D., he taught symbolic logic in Princeton's philosophy department. In 1953, most of Dartmouth College's math professors were retiring, so Dartmouth asked him to come to Dartmouth, chair the department, and bring his friends. He accepted the offer and brought his friends. That's how Dartmouth stole Princeton's math department.

At Dartmouth, Kemeny invented several new branches of math. Then Kemeny's department got General Electric to sell Dartmouth a computer at a 90% discount, in return for which his department had to invent programs for that computer and let General Electric use them.

To write the programs, Kemeny invented his own little computer language in 1963 and showed it to his colleague Tom Kurtz, who knew less about philosophy but more about computers. Kurtz added features from Algol & Fortran and called the combination **Basic**.

After inventing Basic, Kemeny got bored and thought of quitting Dartmouth. Then Dartmouth asked him to become the college's president. He accepted.

Later, when the 3-Mile Island nuclear power plant almost exploded, President Jimmy Carter told Kemeny to head the investigation, because of Kemeny's reputation for philosophical & scientific impartiality. Kemeny's report was impartial — and sharply critical of the nuclear industry.

Basic versus Algol & Fortran Basic is simpler than both Algol and Fortran in 2 ways:

In Algol and Fortran, you must tell the computer which variables are integers and which are reals. In Algol, you do that by saying INTEGER or REAL. In Fortran, you do that by choosing an appropriate first letter for the variable's name. **In Basic, the computer assumes all variables are real**, unless you specifically say otherwise.

In Algol and Fortran, output is a hassle. In Fortran, you must worry about FORMATS. In Algol, each computer handles output differently — and in most cases strangely. **Basic's PRINT statement automatically invents a good format.**

Is Basic closer to Algol than to Fortran?

On the one hand, Basic uses the Algol words FOR, STEP, and THEN and the Algol symbol ↑ (or ^).

On the other hand, Basic uses the Fortran words RETURN and DIMENSION (abbreviated DIM); and Basic's "FOR I = 1 TO 9 STEP 2" puts the step size at the *end* of the statement, like FORTRAN's "DO 30 I = 1,9,2" and unlike Algol's "FOR I:=1 STEP 2 UNTIL 9".

Basic versus Joss Basic is *not* the simplest computer language. **Joss**, which was developed a year earlier by the Rand Corporation, is simpler to learn but runs slower, requires more memory, lacks string variables, and doesn't let you name your programs: you must give each program a number instead).

A few programmers still use Joss and 3 of its variants, called **Aid, Focal, and Mumps**.

Aid appealed to high-school kids; Focal appealed to scientists; Mumps appealed to doctors designing databases of patient records. Mumps *does* have string variables and other modern features but was replaced by newer database languages, such as dBase.

6 versions Kemeny & Kurtz finished the **original version** of Basic in May 1964. It included just 11 kinds of statements:

PRINT, GO TO, IF...THEN, FOR...NEXT, DATA...READ, GOSUB...RETURN, DIM, LET (for commands such as LET X=3), REM (for REMarks and comments), DEF (to DEFine your own functions), and END

In that version, the only punctuation allowed in the PRINT statement was the comma.

Version 2 of Basic (October 1964) added the semicolon.

Version 3 (1966) added the words INPUT, RESTORE, and MAT. (The word MAT helps you manipulate a "MATrix", which means an "array". Now most versions of Basic omit the word MAT because its definition consumes too much RAM.)

All those versions, let you use numeric variables. (A numeric variable is a letter that stands for a number. For example, you could say LET X=3.)

Version 4 (1967) added a new concept: string variables (such as AS). That version also added TAB (to improve printing), RANDOMIZE (to improve RND), and "ON...GO TO".

Version 5 (1970) added data files (sequential access and random access).

Version 6 (1971) added PRINT USING (to format the printing) and a sophisticated way to handle subroutines.

How Basic became popular During the 1960's and 1970's, Kemeny & Kurtz worked on Basic with a fervor that was almost religious.

They believed *every* college graduate should know how to program a computer and become as literate in Basic as in English.

They convinced Dartmouth to spend as much on its computer as on the college library. They put computer terminals in most college buildings, even the dorms. Altogether, the campus had about 300 terminals. Over 90% of all Dartmouth students used Basic before they graduated.

Dartmouth let all the town's children come onto campus and use the terminals. Dartmouth trained high-school teachers to use Basic. Many New England colleges, high schools, and prep schools had terminals connected to Dartmouth's computer by phone.

Dartmouth's computer was built by General Electric, which eventually quit making computers and sold its computer factory to Honeywell. The National Science Foundation funded Dartmouth's research on Basic, so Basic got in the public domain and could be used by other computer makers without paying royalties.

DEC The first company to copy Dartmouth's ideas was **Digital Equipment Corporation (DEC, pronounced "deck")**.

DEC put Basic on DEC's first popular minicomputer, the PDP-8. DEC invented fancier minicomputers (the PDP-11 and Vax) and maxicomputers (the DECsystem-10 and DECsystem-20) and put Basic on them. The versions put on the PDP-8 were primitive (almost as bad as Dartmouth's first edition), but the versions put on DEC's fancier computers were sophisticated. Eventually, DEC put decent versions of Basic even on the PDP-8.

DEC's best version of Basic was **Vax Basic**, which worked just on Vax computers. DEC's second-best version of Basic was **Basic-Plus-2**, which worked on the Vax, the PDP-11, and the DECsystem-20. DEC's third-best version of Basic was **Basic-Plus**, which works just on the PDP-11.

HP Soon after DEC started putting Basic on its computers, Hewlett-Packard (HP) did likewise.

HP put Basic on the HP-2000 computer then put a better version on the HP-300 computer. But HP's Basic was more awkward than DEC's. On HP computers, each time you used a string you had to write a "DIM statement" that warned the computer how many characters the string would contain.

How Microsoft Basic arose The first popular microcomputer was the Altair 8800, which used a version of Basic invented by a 20-year-old kid named **Bill Gates**. His version imitated DEC's. That Altair computer was manufactured by a company called **Mits**, which didn't treat Bill Gates fairly, so he broke away from Mits and formed his own company, called **Microsoft**.

Bill Gates and his company, Microsoft, invented many versions of Basic.

First came **4K Basic**, which consumed just 4K of memory chips (RAM or ROM). Then came **8K Basic** (which included a bigger vocabulary) then **Extended Basic** (which included an even bigger vocabulary and consumed 14K). All those versions were for primitive microcomputers that used tapes instead of disks. Finally came **Disk Basic**, which came on a disk and included all commands for handling disks.

All those versions of Basic were written for computers that contained an 8080 or Z-80 CPU. Simultaneously, he wrote **6502 Basic**, for computers containing a 6502 CPU.

The Apple 2 version of 6502 Basic was called **Applesoft BASIC**. Commodore's version of 6502 Basic was called **Commodore BASIC**.

Unfortunately, 6502 Basic was primitive, resembling his 8K Basic.

After writing 6502 Basic, Bill wrote a souped up version of it, called **6809 Basic**, just for Radio Shack's Color Computer. Radio Shack called it **Extended Color Basic**.

Texas Instruments (TI) asked Bill to write a version of Basic for TI computers. Bill said "yes"; but when TI told Bill what kind of Basic it wanted, Bill's company (Microsoft) found 90 ways that TI's desires would contradict Microsoft's traditions. Microsoft convinced TI to change its mind and remove 80 of those 90 contradictions, but TI stood firm on the other 10.

So TI Basic (which was on the TI-990 and TI-99/4A computers) contradicted all other versions of Microsoft Basic in 10 ways. For example, in TI Basic, the INPUT statement used a colon instead of a semicolon, and a multi-statement line uses a double colon (::) instead of a single colon.

Because of those differences, TI's computers became unpopular, and TI stopped making them. Moral: if you contradict Bill, you die!

Later, Bill invented an amazingly wonderful version of Basic, better than all earlier versions. He called it **Gee-Whiz Basic (GW Basic)**. It ran just on the IBM PC and clones.

When you bought PC-DOS from IBM, you typically got GW Basic at no extra charge. (IBM called it **BasicA**.) When you bought MS-DOS for an IBM clone, the typical dealer included GW Basic at no extra charge.

Beyond GW Basic GW Basic was the last version of Basic that Bill developed personally. All Microsoft's later improvements were done by his assistants.

They created **Microsoft Basic for the Mac**, **Amiga Microsoft Basic** (for Commodore's Amiga computer), **Quick Basic** (for the IBM PC & clones), **QBasic** (which you got instead of GWBasic when you bought MS-DOS version 5 or 6), and **Visual Basic** (which lets you create Windows programs, so the human can use a mouse and pull-down menus).

Those Basics are harder to learn than GW Basic but run faster, have a better editor, include more words from Algol and Pascal, and produce fancier output.

While developing those versions of Basic, Microsoft added 3 new commands: SAY, END IF, and SUB.

The SAY command makes the computer talk by using a voice synthesizer. For example, to make the computer's voice say "I love you", type this command:

```
SAY TRANSLATE$("I LOVE YOU")
```

That makes the computer translate "I love you" into phonetics then say the phonetics. That command works just on Amiga computers.

The END IF command lets you make the IF statement include many lines, like this:

```
IF AGE<18 THEN
    PRINT "YOU ARE STILL A MINOR."
    PRINT "AH, THE JOYS OF YOUTH!"
    PRINT "I WISH I COULD BE AS YOUNG AS YOU!"
END IF
```

The SUB command lets you give a subroutine a name.

Divergences Several microcomputer manufacturers tried to invent their own versions of Basic, to avoid paying royalties to Bill Gates. They were sorry!

Radio Shack hired somebody else to write Radio Shack's Basic. That person quit in the middle of the job, so Radio Shack's original Basic was never finished. Nicknamed "Level 1 Basic", it was a half-done mess. Radio Shack, like an obedient puppy dog, then went to Bill, who finally wrote a decent version of Basic for Radio Shack; Bill's version was called "Level 2".

Apple's original attempt at Basic was called "Apple Integer Basic". It was written by Steve Wozniak and terrible: it couldn't handle decimals and made the mistake of imitating HP instead of DEC (because he'd worked at HP). Eventually, he wised up and hired Bill, who wrote Apple's better Basic, called **Applesoft** (which means "Apple Basic by Microsoft"). Applesoft was intended for tapes, not disks. Later, when Steve Wozniak wanted to add disks to the Apple 2 computer, he made the mistake of not rehiring Bill — which is why the Apple 2's disk system was worse than Radio Shack's.

Atari made the mistake of hiring the inventor of Apple's disastrous DOS. That guy's Basic, called **Atari Basic**, resembles HP's Basic. Like Apple's DOS, it looks pleasant at first glance but turns into a nightmare when you try to do advanced programming. As a result, Atari's computers became less popular than Atari hoped, and the Atari executive who "didn't want to hire Bill" was fired. Atari finally hired Bill's company, Microsoft, which wrote **Atari Microsoft Basic version 2**.

2 other microcomputer manufacturers — **North Star Computers** and **APF** — developed their own versions of Basic to avoid paying royalties to Bill. Since their versions of Basic were lousy, they went out of business.

While DEC, HP, Microsoft, and idiots were developing their own Basic versions, professors at Dartmouth College were still tinkering with Dartmouth Basic version 6. In 1976, Professor Steve Garland added more commands from Algol, PL/I, and Pascal to Dartmouth Basic. He called his version **Structured Basic (SBasic)**.

One of Basic's inventors, Professor Tom Kurtz, became chairman of an Ansi committee to standardize Basic. His committee published two reports:

The 1977 report defined **Ansi Standard Minimal Basic**, a minimal standard that all advertised versions of "Basic" should live up to. That report was reasonable; everybody agreed to abide by it. (Microsoft's old Basic versions were written before that report came out. Microsoft Disk Basic version 5 was Microsoft's first version to obey that standard.)

In 1985, Ansi created a more ambitious report, to standardize Basic's most advanced features. The report said Basic's advanced features should closely imitate SBasic. But Bill Gates, who invented Microsoft Basic and was also on the committee, disliked SBasic and quit the committee. (He was particularly annoyed by the committee's desire to include Dartmouth's MAT commands, which consume lots of RAM.) He refused to follow the committee's recommendations.

That left 2 standards for advanced Basic: the "official" standard (by the Ansi committee) and the "de facto" standard (by Microsoft, such as GW Basic).

For example, in GW Basic you say:

```
10 INPUT "WHAT IS YOUR NAME"; A$
```

But in Ansi Basic, you must say this instead:

```
10 INPUT PROMPT "WHAT IS YOUR NAME? ": A$
```

Yes, Ansi Basic makes you to type the word PROMPT, a question mark, a blank space after the question mark, and a colon instead of a semicolon.

Tom Kurtz (who chaired the Ansi committee) and John Kemeny (who invented Basic with Tom Kurtz) put Ansi Basic onto Dartmouth's computer. So Ansi Basic became Dartmouth's 7th official version of Basic. Then Kurtz & Kemeny left Dartmouth and formed their own company, which invented **True Basic** (an Ansi Basic version for the IBM PC & Mac).

Since Microsoft's Basic versions had become the de facto standard and True Basic wasn't much better, hardly anybody bothered switching from Microsoft Basic to True Basic.

Comparison Here are 9 commands in advanced Basic:

USING, LINE, CIRCLE, SOUND, PLAY, SAY, ELSE, END IF, SUB

Here's what they accomplish:

"USING" lets you control how many digits print after the decimal point.
"LINE" makes the computer draw a diagonal line across the screen.
"CIRCLE" makes the computer draw a circle as big as you wish.
"SOUND" and "PLAY" make the computer create music.
"SAY" makes the computer talk.
"ELSE" and "END IF" let you create fancy IF statements.
"SUB" lets you name subroutines.

This list shows which Basics understood those 9 commands:

IBM PC with QBasic (and Visual Basic's version 2 and later) understood **8** of the commands (all except SAY)

Commodore Amiga with Microsoft Basic understood **8** of the commands (all except PLAY)

Apple Mac with Quick Basic understood **7** of the commands (all except SAY and PLAY)

IBM PC (with GW Basic), Commodore 128, and Radio Shack TRS-80 Color understood **6** of the commands (all except SAY, END IF, and SUB)

Atari ST understood **5** of the commands (all except PLAY, SAY, END IF, and SUB)

Atari XE (or XL) with Microsoft Basic understood just **4** commands (USING, LINE, SOUND, and ELSE)

Radio Shack TRS-80 Model 3, 4, 4P, and 4D understood just **2** commands (USING and ELSE)

Apple 2, 2+, 2e, 2c, 2c+, and 2GS understood just **1** command (LINE)

Commodore 64 and Vic-20 understood **no** commands

Notice that the Commodore 128 and Radio Shack TRS-80 Color Computer understood 6 of the commands, while the more expensive Apple 2c understood just 1 command. If schools would've bought Commodore 128 and Radio Shack TRS-80 Color Computers instead of Apple 2c's, students would have become better programmers!

PL/I

During the early 1960's, IBM sold two kinds of computers: one kind for scientists, the other kind for business bookkeepers.

For the scientific kind of computer, the most popular language was Fortran.
For the business kind of computer, the most popular language was Cobol.

In 1962, IBM secretly began working on a project to create a single, big computer that could be used by everybody: scientists and businesses. IBM called it the **IBM 360**, because it could handle the full circle of applications.

What language should the IBM 360 be programmed in? IBM decided to invent a single language that could be used for both science and business.

IBM's first attempt at such a language was Fortran 5. It ran all Fortran 4 programs but added commands for handling strings and data-file fields.

Instead of announcing Fortran 5, IBM began working in 1963 on Fortran 6, which would be more powerful and modern (and hence incompatible). It would include all important features of Cobol & Algol.

As work on Fortran 6 progressed, IBM realized it differed so much from traditional Fortran that it should get a different name. In 1964, IBM changed the name to "NPL" (New Programming

Language), since the language was intended for the IBM 360 and the rest of IBM's New Product Line. But IBM discovered the letters "NPL" already stood for the National Physics Laboratory in England, so IBM changed the language's name to **Programming Language One (PL/I)**, to brag it was the first good programming language and all predecessors were worth zero by comparison.

The committee inventing PL/I had to meet just on weekends and just in hotel rooms in California & New York State. & Ca hard time. The first meeting was in October 1963. IBM wanted the language's design to be finished in 2 months, but the committee took 4 months, finishing in February 1964.

After the design was finished, the language still had to be put on the computer. That took 2½ more years of programming and polishing, so the language wasn't available for sale to IBM's customers until August 1966.

That was too late.

It was *after* IBM began shipping the IBM 360. The 360's customers continued using Fortran and Cobol, since PL/I wasn't available yet. After those customers bought, installed, and learned how to use Fortran and Cobol on the 360, they refused to take the trouble to switch to PL/I, especially since PL/I was expensive (requiring twice as much RAM as Cobol, 4 times as much RAM as Fortran) and ran slowly (1½ times as long to compile as Cobol, twice as long as Fortran). Most programmers already knew Fortran or Cobol, were satisfied with those languages, and weren't willing to spend time to learn PL/I.

Some programmers praised PL/I for being amazingly powerful, but others called it just a scheme to make people buy more RAM. Critics call it a disorganized mess, an "ugly kitchen sink of a language", thrown together by a committee that was too rushed.

Since PL/I is so big, hardly anybody understands it all.

As a PL/I programmer, you study just the part of the language you plan to use. But if you make a mistake, the computer might not gripe: instead, it might think you're trying to give a different PL/I command you never studied. Instead of griping, the computer will perform an instruction that wasn't what you meant.

Stripped versions In 1972, Cornell University developed a stripped-down version of PL/I for students. That version, **PL/Cornell (PL/C)**, is a compromise between PL/I's power and Algol's pure simplicity.

In 1975, the University of Toronto developed an even *tinier* stripped-down PL/I version, called **SP/k**. It ran faster and printed messages that were more helpful. SP/k came in several sizes: the tiniest was SP/1; the largest was SP/8.

Stripped-down versions of PL/I stayed popular in universities until about 1980, when universities switched to Pascal.

Digital Research invented a tiny version of PL/I for microcomputers and called it **PL/M**. It couldn't handle decimals.

Full PL/I is still used on big IBM computers, because full PL/I is the only language including enough commands to let programmers unleash IBM's full power.

PL/I's statements are borrowed from Fortran, Algol, and Cobol.

from Fortran:	FORMAT, STOP, CALL, RETURN, DO
from Cobol:	OPEN, CLOSE, READ, WRITE, DISPLAY, EXIT
from Algol:	IF, GO, PROCEDURE, BEGIN, END

Like Algol, PL/I requires a semicolon at the end of each statement.

Pascal

In 1968, a European committee invented "Algol 68," which was strange: it even required you to spell some commands backwards. Some committee members disagreed with the majority and thought Algol 68 was nuts. One of those dissidents, Niklaus Wirth, quit the committee and created his own Algol version, which he called **Pascal**. Now most computerists feel he was right: Pascal is better than Algol 68.

He wrote Pascal in Switzerland on a CDC maxicomputer. His version of Pascal couldn't handle video screens, couldn't handle random-access data files, and couldn't handle strings well. Those 3 limitations were corrected in later Pascal versions, especially the one invented at the **University of California at San Diego (UCSD)**.

Apple's Pascal Apple Computer Company got permission to sell an Apple version of UCSD Pascal. Apple ran full-page ads bragging that the Apple 2 was the only popular microcomputer that could handle Pascal.

Apple Computer Company sold an Apple 2 add-on called the **Apple Language System**, whose \$495 price included disks for Pascal & advanced Basic, plus 16K of extra RAM. Many people who bought that system were disappointed when they realized Pascal is *harder* to learn than Basic.

Pascal is helpful just if the program you're writing is long. Pascal helps you organize and dissect long programs more easily than Basic. But the average Apple 2 owner never wrote long programs and never needed Pascal. Many of Apple's customers felt "ripped off", since they spent \$495 uselessly.

Pascal's rise Many programmers who wrote big Fortran programs for big computers switched to Pascal, because Fortran is archaic and Pascal helps organize long programs. Many programmers who used PL/I switched to Pascal, because Pascal consumes less RAM than PL/I and fits in smaller computers. Many colleges required freshman computer-science majors to learn Pascal, so the College Entrance Examination Board's **Advanced Placement Test in Computer Science** required knowing Pascal. High-school students studied Pascal to pass that test and prepare for college.

Pascal's fall Basic improved by incorporating many features from Pascal, so Pascal stopped having much advantage over Basic. Now students skip Pascal: after learning Basic or Python, they skip past Pascal to tougher languages: Java, C++, or C#. Now the Advanced Placement Test in Computer Science requires knowing Java instead of Pascal.

Pascal is ignored.

Modula

After Niklaus Wirth invented Pascal, he designed a more ambitious language, called **Modula**. He designed the Modula's first version in 1975, then **Modula-2** in 1979. When today's programmers discuss "Modula", they mean Modula-2.

Modula-2 resembles Pascal but lets programs look briefer. Though it still requires each program's main routine to begin with the word BEGIN, it does *not* require you to say BEGIN after DO WHILE or IF THEN:

Pascal	Modula-2
IF AGE<18 THEN BEGIN WRITELN('YOU ARE STILL A MINOR'); WRITELN('AH, THE JOYS OF YOUTH'); END ELSE BEGIN WRITELN('GLAD YOU ARE AN ADULT'); WRITELN('WE CAN HAVE ADULT FUN'); END;	IF AGE<18 THEN WRITESTRING("YOU ARE STILL A MINOR"); WRITESTRING("AH, THE JOYS OF YOUTH"); ELSE WRITESTRING("GLAD YOU ARE AN ADULT"); WRITESTRING("WE CAN HAVE ADULT FUN") END;

That example shows 4 ways Modula-2 differs from Pascal: Modula-2 says **WRITESTRING** instead of WRITELN, uses **regular quotation marks (")** instead of apostrophes, lets you **omit the word BEGIN** after IF ELSE (and WHILE DO), and lets you **omit the word END** before ELSE.

Advanced programmers like Modula-2 more than Pascal because Modula-2 includes extra commands for handling subroutines.

C

Many programmers use **C**.

How C arose In 1963 England, researchers (at Cambridge University and the University of London) developed a "practical" version of Algol and called it the **Combined Programming Language (CPL)**. In 1967 at Cambridge University, Martin Richards invented a stripped-down simpler version of CPL and called it **Basic CPL (BCPL)**.

In 1970 at Bell Labs, Ken Thompson developed a version that was even more stripped-down and simpler; since it included just the most critical part of BCPL, he called it **B**. But Ken had stripped down the language *too* much. It no longer contained enough commands to do practical programming. In 1971, his colleague Dennis Ritchie added a few commands to B, to form a more extensive language, which he called **New B**.

Then he added even more commands and called the result **C**, because it came after B.

Most of C was invented in 1972. In 1973, it improved enough so it was used for something practical: developing a new version of the **Unix** operating system. (The original version of Unix had been created at Bell Labs by using B. Beginning in 1973, Unix versions were created using C.)

So C is a souped-up version of New B, which is a souped-up version of B, which is a stripped-down version of BCPL, which is a stripped-down version of CPL, which is a "practical" version of Algol.

C's peculiarities Like B, C is a tiny language.

C doesn't even include any words for input or output. When you buy C, you also get a **library** of routines that can be added to C. The library includes words for output (such as printf), input (such as scanf), math functions (such as sqrt), and other goodies.

When you write a program in C, you can choose whichever parts of the library you need: the other parts of the library don't bother to stay in RAM. So if your program uses just a *few* of the library's functions, running it will consume very little RAM. It will consume less RAM than if the program were written in Basic or Pascal.

In Basic, if you reserve 20 RAM locations for X (by saying DIM X(20)) and then say X(21)=3.7, the computer will gripe, because you haven't reserved a RAM location for X(21). If you use C instead, the computer will *not* gripe about that kind of error; instead, the computer will store the number 3.7 in the RAM location immediately after X(20), even if that location's already being used by another variable, such as Y. As a result, Y will get messed up. Moral: **C programs run quickly and dangerously, because in C the computer never bothers to check your program's reasonableness.**

In your program, which variables are integers? Which are real?

Basic assumes all variables are real.

Fortran & PL/I assume all variables beginning with I, J, K, L, M, and N are integers and the rest are real.

Algol & Pascal make no assumptions; they require you to declare "integer" or "real" for each variable.

C assumes all variables are integers, unless you specifically say otherwise.

Ada

In 1975, the U.S. Department of Defense decided it wanted a better computer language and wrote requirements for it.

The original requirements was called the Strawman Requirements (1975). Then came improved versions, called Woodenman (1975), Tinman (1976), Ironman (1978), and finally Steelman (1979).

While the Department was moving from Strawman to Steelman, it checked whether any existing computer language could meet such requirements. The Department decided no existing language came close to meeting the requirements, so a new language must be invented. The Department required the new language to resemble Pascal, Algol 68, or PL/I but be better.

Contest In 1977, the Department held a contest, to see which software company could invent a language meeting such specifications (which were in the process of changing from Tinman to Ironman).

16 companies entered the contest.

The Department selected 4 semifinalists and paid them to continue their research for 6 more months. The semifinalists were CII-Honeywell-Bull (which is French and owned partly by Honeywell), Intermetrics (in Cambridge, Massachusetts), SRI International, and Softech.

In 1978, the semifinalists submitted improved designs, which were all souped-up versions of Pascal. To prevent bribery, the judges weren't told which design belonged to which company. The 4 designs were called "Green", "Red", "Yellow", and "Blue".

Yellow and Blue lost. The winning designs were Green (designed by CII-Honeywell-Bull) and Red (designed by Intermetrics).

The Department paid those 2 winning companies to continue their research for another year. In 1979, those 2 companies submitted their improved versions. The winner was the Green language, designed by CII-Honeywell-Bull.

The Department made the Green language be called **Ada** to honor Ada Lovelace, the woman who was the world's first programmer. So Ada is a Pascal-like language developed by a French company (CII-Honeywell-Bull) under contract to the U.S. Department of Defense.

Ada's too big to be practical. Researchers make computers understand just *part* of Ada.

dBase

dBase was invented by Wayne Ratliff because he wanted to bet on which football teams would win the 1978 season.

To bet wisely, he needed to know how each team scored in previous games, so every Monday he clipped pages of football scores from newspapers. Soon those clippings covered his room. To reduce the clutter, he decided to write a data-management program to handle all the statistics.

He worked at the **Jet Propulsion Laboratory (JPL)**. His coworkers had invented a data-management system called the **JPL Display and Information System (JPLDIS)**, which imitated IBM's **Retrieve**. Unfortunately, Retrieve and JPLDIS required maxicomputers. Working at home, he invented **Vulcan**, a stripped-down version of JPLDIS small enough to run on the CP/M microcomputer in his house and good enough to compile football statistics — though by then he'd lost interest in football and was more interested in the theory of data management and business applications.

In 1979, he advertised his Vulcan data-management system in Byte Magazine. The mailman delivered so many orders to his house that he didn't have time to fill them all — especially since he still had a full-time job at JPL. He stopped advertising, to give himself a chance to catch up filling the orders.

In 1980, the owners of Discount Software phoned him, visited his home, examined Vulcan, and offered to market it. He agreed. Since "Discount Software" was the wrong name to market Vulcan, Discount Software's owners — Hal Lashlee and George Tate — thought of marketing Vulcan under the name "Lashlee-Tate Software". But since "Lashlee" sounded wimpy, they changed the name to *Ashton-Tate* Software. Instead of selling Vulcan's original version, Ashton-Tate Software decided to sell Wayne's further improvement, called **dBase 2**.

At Ashton-Tate, George Tate did the managing. Hal Lashlee was a silent partner who just contributed capital.

Ad George Tate hired Hal Pawluck to write an ad for dBase 2. Hal's clever ad showed a photo of a bilge pump (which removes water from a ship's bilge). The ad's headline was: "dBase versus the Bilge Pump". The ad went on to say that most database systems are like bilge pumps: they suck! That explicit ad ran in *Infoworld*, a weekly newspaper read by all computer experts. Suddenly, all experts knew that dBase was the database-management system that claimed not to suck.

The ad generated just one big complaint — from the company that manufactured the bilge pump!

George Tate offered to add a footnote saying "*This bilge pump does not suck*". The pump manufacturer didn't like that either but stopped complaining.

Beyond dBase 2 The original dBase 2 ran on computers using the CP/M operating system. It worked well. When IBM began selling the IBM PC, Wayne invented an IBM PC version of dBase 2, but it was buggy.

He created those early versions of dBase by using assembly language. By using C instead, he finally created an IBM PC version that worked reliably, included extra commands, and was called **dBase 3**.

dBase 2 and dBase 3 were sold as programming languages, but many people wanting databases didn't want to learn programming, so Ashton-Tate created a new version, called **dBase 3 Plus**, which you can control by using menus instead of typing programming commands; but those menus are hard to learn how to use and incomplete: they don't let you tap dBase 3 Plus's full power, which requires you to learn programming.

In 1988, Ashton-Tate shipped **dBase 4**, which includes better menus and extra programming commands. Some of those extra commands were copied from a database language called **Structured Query Language (SQL)**, which IBM invented for mainframes. Unfortunately, Ashton-Tate priced dBase 4 high: \$795 for the plain version, \$1295 for the "developer's" version.

Over the years, Ashton-Tate became a stodgy bureaucracy. George Tate died, Wayne Ratliff quit, the company's list price for dBase grew ridiculously high, and the company was callous to dBase users.

In 1991, Borland bought Ashton-Tate. In 1994, Borland began selling **dBase 5**, then further improvements.

In 1999, Borland gave up trying to sell dBase; Borland transferred all dBase rights to **KSoft**, which sold **Visual dBase 7.5** and tried to develop **dBase 2000 (DB2K)**. The newest version of dBase is **dBase 2019** (for Windows), published by dBase LLC (30 Charles St., Binghamton NY 13905, phone 607-729-0960, dBase.com).

Other companies made dBase clones that worked better than dBase itself! The most popular clone was **Visual FoxPro 9**: it ran faster than dBase, includes extra commands, and was marketed by Microsoft until 2007. But Microsoft doesn't bother selling it anymore.

Easy

Easy is a language I developed several years ago. It combines the best features of all other languages. It's easy to learn, because it uses just these 12 keywords:

say & get	let
repeat & skip	here
if & pick	loop
prepare & data	how

Here's how to use them...

The computer will say the answer:

4

Say Easy uses the word “say” instead of Basic’s word PRINT, because “say” is briefer. If you want the computer to say the answer to 2+2, give this command:

```
say 2+2
```

Whenever the computer prints, it automatically prints a blank space afterwards but does *not* press the Enter key. So if you run this program —

```
say "love"  
say "hate"
```

the computer will say:

```
Love hate
```

Here’s a fancier example:

```
say "love" as 3 at 20 15 trim !
```

The “as 3” is a format: it makes the computer print just the first 3 letters of “love”. The “at 20 15” makes the computer begin printing “love” at the screen’s pixel whose X coordinate is 20 and whose Y coordinate is 15. The computer usually prints a blank space after everything, but the word “trim” suppresses that blank space. The exclamation point makes the computer press the Enter key afterwards.

Here’s another example:

```
say to screen printer harry
```

It means that henceforth, whenever you give a “say” command, the computer will print the answer simultaneously onto your screen, onto your printer, and onto a disk file named “harry”. If you ever want to cancel that “say to” command, give a “say to” command that contradicts it.

Get Easy uses the word “get” instead of Basic’s word INPUT, because “get” is briefer. The command “get x” makes the computer wait for you to input the value of x. Above the “get” command, you typically put a “say” command that makes the computer ask a question.

You can make the “get” command fancy, like this:

```
get x as 3 at 20 15 wait 5
```

The “as 3” tells the computer that x will be just 3 characters; the computer waits for you to type just 3 characters and doesn’t require you to press the Enter key afterwards. The “at 20 15” makes the computer move to pixel 20 15 before your typing begins, so your input appears at that part of the screen. The “wait 5” makes the computer wait just 5 seconds for your response. If you reply within 5 seconds, the computer sets “time” equal to how many seconds you took. If you do *not* reply within the 5 seconds, the computer sets “time” equal to -1.

Let The “let” statement resembles Basic’s. For example, you can say:

```
let r=4
```

To let r be a random decimal, type:

```
let r=random
```

To let r be a random integer from 1 to 6, type:

```
let r=random to 6
```

To let r be a random integer from -3 to 5, type:

```
let r=random from -3 to 5
```

Repeat If you put the word “repeat” at your program’s bottom, the computer will repeat the entire program again and again, forming an infinite loop.

Skip If you put the word “skip” in your program’s middle, the computer will skip the program’s the bottom part. “Skip” is like Basic’s END or STOP.

Here In your program’s middle, you can say:

```
here is fred
```

An earlier line can say “skip to fred”. A later line can say “repeat from fred”. The “skip to” and “repeat from” are like Basic’s GO TO.

If In your program, a line can say:

```
if x<3
```

Underneath that line, you must put some indented lines, which the computer will do if x<3.

Suppose you give a student a test on which the score can be between 0 and 100. If the student’s score is 100, let’s make the computer say “perfect”; if the score is below 100 but at least 70, let’s make the computer say the score and also say “okay though not perfect”; if the score is below 70, let’s make the computer say “you failed”. Here’s how:

```
if score=100  
  say "perfect"  
if score<100 and score>=70  
  say score  
  say "okay though not perfect "  
if score<70  
  say "you failed"
```

To shorten the program, use the words “not” and “but”:

```
if score=100  
  say "perfect"  
if not but score>=70  
  say score  
  say "okay though not perfect "  
if not  
  say "you failed"
```

The phrase “if not” is like Basic’s ELSE. The phrase “if not but” is like Basic’s ELSE IF.

Pick You can shorten that example even further, by telling the computer to pick just the first “if” that’s true:

```
pick score  
if 100  
  say "perfect"  
if >=70  
  say score  
  say "okay though not perfect"  
if not  
  say "you failed"
```

Loop If you put the word “loop” above indented lines, the computer will do those lines repeatedly. For example, this program makes the computer say the words “cat” and “dog” repeatedly:

```
loop  
  say "cat"  
  say "dog"
```

This program makes the computer say 5, 8, 11, 14, and 17:

```
loop i from 5 by 3 to 17  
  say i
```

That “loop” statement is like Basic’s “FOR I = 5 TO 17 STEP 3”. If you omit the “by 3”, the computer will assume “by 1”. If you omit the “from 5”, the computer will assume “from 1”. If you omit the “to 17”, the computer will assume “to infinity”.

To make the computer count down instead of up, insert the word “down”, like this:

```
loop i from 17 down by 3 to 5
```

Prepare To do an unusual activity, you should “prepare” the computer for it. For example, if you want to use subscripted variables such as x(100), you should tell the computer:

```
prepare x(100)
```

In that example, “prepare” is like Basic’s “dim”.

Data Easy’s “data” statement resembles Basic’s. But instead of saying READ X, say:

```
let x=next
```

How In Easy, you can give any command you wish, such as:

```
pretend you are human
```

If you give that command, you must also give an explanation that begins with the words:

```
how to pretend you are human
```

Interrelated features In a loop’s middle, you can abort the loop. To skip out of the loop (and progress to the rest of the program), say “skip loop”. To hop back to loop’s beginning (to do the next iteration of loop), say “repeat loop”.

Similarly, you can say “skip if” (which makes the computer skip out of an “if”) and “repeat if” (which makes the computer repeat the “if” statement, and thereby imitate Pascal’s WHILE).

Apostrophe Like Basic, Easy uses an apostrophe to begin a comment. The computer ignores everything to the right of an apostrophe, unless the apostrophe is between quotation marks or in a “data” statement.

Comma If two statements begin with the same word, you can combine them into a single statement, by using a comma.

For example, instead of saying —

```
let x=4
let y=7
```

you can say:

```
let x=4, y=7
```

Instead of saying —

```
pretend you are human
pretend God is dead
```

you can say:

```
pretend you are human, God is dead
```

More info I stopped working on Easy in 1982 but hope to continue development again. To get on my mailing list of people who want details and updated info about Easy, phone me at 603-666-6644.

C++

An improved C, called **C++**, was invented in 1985 at Bell Labs by **Bjarne Stroustrup**.

He was born in Denmark (where he studied at Aarhus University). Then he moved to England (where he got a Ph.D. from Cambridge University). Then he moved to New Jersey (to work at Bell Labs, where he invented C++).

To pronounce his name, say “Bee-ARE-nuh STRAH-stroop”, but say the “Bee” and “STRAH-stroop” fast, so it sounds closer to “BYAR-nuh STROV-strup”.

C++ uses the same fundamental commands as C but adds extra commands. Some of those extra commands are for advanced programming; others make regular programming more pleasant. Unlike C, C++ lets you use **object-oriented programming (OOP)**, in which you define “objects” and give those objects “properties”.

For input and output, C++ offers different commands than C. C++’s input/output commands are more pleasant. Most C programmers have switched to C++ or a further improvement, called **C#**.

C++ became the most popular language for creating advanced programs. The world’s biggest software companies switched to C++ from assembly language, though many are starting to go a step further and switch to C#.

If you become an expert C++ or C# programmer, you can help run those rich software companies and get rich yourself!

Radicals

Let’s examine the radical languages, beginning with the oldest radical — the oldest hippie — Lisp.

Lisp

Lisp is the only language made specially to handle lists of concepts. It’s the most popular language for research into artificial intelligence.

It’s the father of Logo, which is “oversimplified Lisp” and the most popular language for young children. It inspired Prolog, which is a Lisp-like language that lets you make the computer imitate a wise expert and become an **expert system**.

Beginners love to play with Logo and Prolog, which are easier and more fun than Lisp. But professionals keep using Lisp because it’s more powerful than its children.

Lisp’s original version was called **Lisp 1**. Then came **Lisp 1.5** (which wasn’t different enough from LISP 1 to rate the title “LISP 2”). Then came **Lisp 1.6**. Lisp’s newest version, called **Common Lisp**, runs on maxicomputers, minicomputers, and microcomputers.

I’ll explain “typical” Lisp, which is halfway between Lisp 1.6 and Common Lisp.

Typical Lisp uses these symbols:

Basic	Lisp
5+2	(PLUS 5 2)
5-2	(DIFFERENCE 5 2)
5*2	(TIMES 5 2)
5/2	(QUOTIENT 5 2)
5^2	(EXPT 5 2)
"LOVE"	'LOVE old versions say (QUOTE LOVE)

If you want the computer to add 5 and 2, just type:

```
(PLUS 5 2)
```

When you press the Enter key at the end of that line, the computer will print the answer. (You do *not* have to say PRINT or any other special word.) The computer will print:

```
7
```

If you type —

```
(PLUS 1 3 1 1)
```

the computer will add 1, 3, 1, and 1 and print:

```
6
```

If you type —

```
(DIFFERENCE 7 (TIMES 2 3))
```

the computer will find the difference between 7 and 2*3 and print:

```
1
```

If you type —

```
'LOVE
```

the computer will print:

```
LOVE
```

Note you must type an apostrophe before LOVE but must *not* type an apostrophe afterwards. The apostrophe is called a **single quotation mark** (or a **quote**).

You can put a quote in front of a word (such as 'LOVE) or in front of a parenthesized list of words, such as:

```
'(LAUGH LOUDLY)
```

That makes the computer print:

```
(LAUGH LOUDLY)
```

Lisp 1, Lisp 1.5, and Lisp 1.6 don’t understand the apostrophe. On those old versions of Lisp, say (QUOTE LOVE) instead of 'LOVE, and say (QUOTE (LAUGH LOUDLY)) instead of '(LAUGH LOUDLY).

The theory of lists Lisp can handle lists. Each list must begin and end with a parenthesis.

Here’s a list of numbers: (5 7 4 2).

Here’s a list of words: (LOVE HATE WAR PEACE DEATH).

Here’s a list of numbers and words:

(2 WOMEN KISS 7 MEN).

That list has 5 items:

2, WOMEN, KISS, 7, and MEN.

Here’s a list of 4 items:

(HARRY LEMON (TICKLE MY TUBA TOMORROW AT TEN) RUSSIA).

The first item is HARRY; the second is LEMON; the third is a list; the fourth is RUSSIA.

In a list, **the first item is called the CAR, and the rest of the list is called the CDR** (pronounced “could er” or “cudder” or “coo der”). For example, the CAR of (SAILORS DRINK WHISKEY) is SAILORS, and the CDR is (DRINK WHISKEY).

To make the computer find the CAR of (SAILORS DRINK WHISKEY), type this:

```
(CAR '(SAILORS DRINK WHISKEY))
```

The computer will print:

```
SAILORS
```

If you type —

```
(CDR '(SAILORS DRINK WHISKEY))
```

the computer will print:

```
(DRINK WHISKEY)
```

If you type —

```
(CAR (CDR '(SAILORS DRINK WHISKEY)))
```

the computer will find the CAR of the CDR of (SAILORS DRINK WHISKEY). Since the CDR of (SAILORS DRINK WHISKEY) is (DRINK WHISKEY), whose CAR is DRINK, the computer will print:

```
DRINK
```

You can insert an extra item at the beginning of a list, to form a longer list. For example, you can insert MANY at the beginning of (SAILORS DRINK WHISKEY), to form (MANY SAILORS DRINK WHISKEY). To do that, tell the computer to CONSTRUCT the longer list, by typing:

```
(CONS 'MANY '(SAILORS DRINK WHISKEY))
```

The computer will print:

```
(MANY SAILORS DRINK WHISKEY)
```

Notice that CONS is the opposite of CAR and CDR. The CONS combines MANY with (SAILORS DRINK WHISKEY) to form (MANY SAILORS DRINK WHISKEY). The CAR and CDR break down (MANY SAILORS DRINK WHISKEY), to form MANY and (SAILORS DRINK WHISKEY).

Variables To make X stand for the number 7, say:

```
(SETQ X 7)
```

Then if you say —

```
(PLUS X 2)
```

the computer will print 9.

To make Y stand for the word LOVE, say:

```
(SETQ Y 'LOVE)
```

Then if you say —

```
Y
```

the computer will say:

```
LOVE
```

To make STOOGES stand for the list (MOE LARRY CURLEY), say:

```
(SETQ STOOGES '(MOE LARRY CURLEY))
```

Then if you say —

```
STOOGES
```

the computer will say:

```
(MOE LARRY CURLEY)
```

To find the first of the STOOGES, say:

```
(CAR STOOGES)
```

The computer will say:

```
MOE
```

Your own functions You can define your own functions. For example, you can define (DOUBLE X) to be 2*X, by typing this:

```
(DEFUN DOUBLE (X)
  (TIMES 2 X)
)
```

Then if you say —

```
(DOUBLE 3)
```

the computer will print:

```
6
```

REPEAT Let's define REPEAT to be a function so that (REPEAT 'LOVE 5) is (LOVE LOVE LOVE LOVE LOVE), and (REPEAT 'KISS 3) is (KISS KISS KISS), and (REPEAT 'KISS 0) is ().

If N is 0, we want (REPEAT X N) to be ().

If N is larger than 0, we want (REPEAT X N) to be a list of N X's.

```
That's X followed by N-1 more X's.
That's the CONS of X with a list of N-1 more X's.
That's the CONS of X with (REPEAT X (DIFFERENCE N 1)).
That's (CONS X (REPEAT X (DIFFERENCE N 1))).
That's (CONS X (REPEAT X (SUB1 N))), since (SUB1 N) means N-1 in LISP.
```

You can define the answer to (REPEAT X N) as follows: if N is 0, the answer is (); if N is *not* 0, the answer is (CONS X (REPEAT X (SUB 1 N))). Here's how to type that definition:

```
(DEFUN REPEAT (X N)
  (COND
    ((ZEROP N) ())
    (T (CONS X (REPEAT X (SUB1 N))))
  )
)
```

The top line says you're going to Define a FUNCTION called REPEAT (X N). The next line says the answer depends on CONDitions. The next line gives one of those conditions: *if N is ZERO*, the answer is (). The next line says: *otherwise*, the value is (CONS X (REPEAT X (SUB1 N))). The next line closes the parentheses opened in the second line. The bottom line closes the parentheses opened in the top line.

Then if you type —

```
(REPEAT 'LOVE 5)
```

the computer will print:

```
(LOVE LOVE LOVE LOVE LOVE)
```

The definition is almost circular: the definition of REPEAT assumes you already know what REPEAT is. For example:

```
(REPEAT 'KISS 3) is defined as the CONS of KISS with the following:
(REPEAT 'KISS 2), which is defined as the CONS of KISS with the following:
(REPEAT 'KISS 1), which is defined as the CONS of KISS with the following:
(REPEAT 'KISS 0), which is defined as ().
```

That kind of definition, which is almost circular, is called **recursive**.

You can say "The definition of REPEAT is **recursive**", or "REPEAT is **defined recursively**", or "REPEAT is **defined by recursion**", or "REPEAT is **defined by induction**", or "REPEAT is a **recursive function**".

Lisp was the first popular language that allowed recursive definitions.

When the computer uses a recursive definition, the computer refers to the definition *repeatedly* before getting out of the circle. Since the computer repeats, it's performing a loop. In traditional Basic and Fortran, the only way to make the computer perform a loop is to say GO TO or FOR or DO. Although Lisp contains a go-to command, Lisp programmers avoid it and write recursive definitions instead.

ITEM As another example of recursion, let's define the function ITEM so (ITEM N X) is the Nth item in list X, and so (ITEM 3 '(MANY SAILORS DRINK WHISKEY)) is the 3rd item of (MANY SAILORS DRINK WHISKEY), which is DRINK.

```
If N is 1, (ITEM N X) is the first item in X, which is the CAR of X, which is (CAR X).
If N is larger than 1, (ITEM N X) is the Nth item in X. That's the (N-1)th item in the CDR of X. That's (ITEM (SUB1 N) (CDR X)).
```

So define (ITEM N X) as follows:

```
If N is 1, the answer is (CAR X).
If N is not 1, the answer is (ITEM (SUB 1 N) (CDR X)).
```

Here's what to type:

```
(DEFUN ITEM (N X)
  (COND
    ((ONEP N) (CAR X))
    (T (ITEM (SUB1 N) (CDR X)))
  )
)
```

If your computer doesn't understand (ONEP N), say (EQUAL 1 N) instead.

Snobol

Snobol lets you analyze strings more easily than any other language. It can handle numbers also.

Simple example Here's a simple Snobol program:

```
A = -2
B = A + 10.6
C = "BODY TEMPERATURE IS 9" B
OUTPUT = "MY " C
END
```

When you type the program, indent each line except END. Indent *at least* one space; you can indent more spaces if you wish. Put spaces around the symbols = and + and other operations.

The first line says A is the integer -2. The next line says B is the real number 8.6. The next line says C is the string "BODY TEMPERATURE IS 98.6". The next line makes the computer print:

```
BODY TEMPERATURE IS 98.6
```

In Snobol, a variable's name can be short (like A or B or C) or as long as you wish. The variable's name can even contain periods, like this:

```
NUMBER.OF.BULLIES.I.SQUIRTED
```

Loop This program's a loop:

```
FRED OUTPUT = "CAT"
      OUTPUT = "DOG" : (FRED)
END
```

The first line (whose name is FRED) makes the computer print:

```
CAT
```

The next line makes the computer print —

```
DOG
```

and then go to FRED. Altogether the computer will print:

```
CAT
DOG
CAT
DOG
CAT
DOG
etc.
```

Replace Snobol lets you replace a phrase easily.

```
X = "SIN ON A PIN WITH A DIN"
X "IN" = "UCK"
OUTPUT = X
END
```

The first line says X is the string "SIN ON A PIN WITH A DIN". The next line says: in X, replace the first "IN" by "UCK". So X becomes "SUCK ON A PIN WITH A DIN". The next line says the output is X, so the computer will print:

```
SUCK ON A PIN WITH A DIN
```

That program changed the *first* "IN" to "UCK". Here's how to change *every* "IN" to "UCK":

```
X = "SIN ON A PIN WITH A DIN"
X "IN" = "UCK"
X "IN" = "UCK"
X "IN" = "UCK"
OUTPUT = X
END
```

The first line says X is "SIN ON A PIN WITH A DIN". The second line replaces an "IN" by "UCK", so X becomes "SUCK ON A PIN WITH A DIN". The next line replaces another "IN" by "UCK", so X becomes "SUCK ON A PUCK WITH A DIN". The next line replaces another "IN", so X becomes "SUCK ON A PUCK WITH A DUCK", which the next line prints.

This program does the same thing:

```
X = "SIN ON A PIN WITH A DIN"
LOOP X "IN" = "UCK" :S(LOOP)
      OUTPUT = X
END
```

Here's how it works:

The first line says X is "SIN ON A PIN WITH A DIN". The next line replaces "IN" successfully, so X becomes "SUCK ON A PIN WITH A DIN". At the line's end, the :S(LOOP) means: if Successful, go to LOOP. So the computer goes back to LOOP. The computer replaces "IN" successfully again, so X becomes "SUCK ON A PUCK WITH A DIN", and the computer goes back to LOOP. The computer replaces "IN" successfully again, so X becomes "SUCK ON A PUCK WITH A DUCK", and the computer goes back to LOOP. The computer does not succeed, so the computer ignores the :S(LOOP) and proceeds instead to the next line, which prints: SUCK ON A PUCK WITH A DUCK

Delete This program deletes the first "IN":

```
X = "SIN ON A PIN WITH A DIN"
X "IN" =
OUTPUT = X
END
```

The second line says to replace an "IN" by nothing, so the "IN" gets deleted. X becomes "S ON A PIN WITH A DIN", which the computer will print.

This program deletes *every* "IN":

```
X = "SIN ON A PIN WITH A DIN"
LOOP X "IN" = :S(LOOP)
      OUTPUT = X
END
```

The computer will print:

```
S ON A P WITH A D
```

Count Let's count how often "IN" appears in "SIN ON A PIN WITH A DIN". To do that, delete each "IN"; but each time you delete one, increase the COUNT by 1:

```
X = "SIN ON A PIN WITH A DIN"
COUNT = 0
LOOP X "IN" = :F(ENDING)
      COUNT = COUNT + 1 : (LOOP)
ENDING OUTPUT = COUNT
END
```

The third line tries to delete an "IN": *if successful*, the computer proceeds to the next line, which increases the COUNT and goes back to LOOP; *if failing* (because no "IN" remains), the computer goes to ENDING, which prints the COUNT. The computer will print:

```
3
```

How Snobol developed At MIT

during the 1950's, Noam Chomsky invented a notation called **transformational-generative grammar**, which helps linguists analyze English and translate between English and other languages. His notation was nicknamed "linguist's algebra", because it helped linguists just like algebra helped scientists. (A decade later, he became famous for also starting the rebellion against the Vietnam War.)

Chomsky's notation was for pencil and paper. In 1957 and 1958, his colleague Victor Yngve developed a computerized version of Chomsky's notation; the computerized version was a language called **Comit**. It was nicknamed "linguist's Fortran" because it helped linguists just as Fortran helped engineers.

Comit manipulated strings of *words*. In 1962 at Bell Telephone Laboratories (Bell Labs), Chester Lee invented a variant called **Symbolic Communication Language (SCL)**, which manipulated strings of *math symbols* instead of words and helped mathematicians do abstract math.

A team at Bell Labs decided to invent a simplified SCL that would also include features from Comit. The team called their new language "SCL7" then renamed it "Sexi" (which stands for **String Expression Interpreter**); but Bell Labs' management didn't like sex. Then, as a joke, the team named it **Snobol**, using the flimsy excuse that Snobol stands for **String-Oriented symbolic Language**; but here's the real reason it got named "Snobol": the team feared it didn't have "a snowball's chance in hell" of success.

Snobol was used mainly to write programs that translate between computer languages. (For example, you could write a Snobol program that translates Fortran to Basic.)

Which is better: Comit or Snobol?

People who like Chomsky's notation (such as linguists) prefer Comit. People who like algebra (such as scientists) prefer Snobol.

Snobol's supporters were more active than Comit's: they produced Snobol 2, Snobol 3, Snobol 4, and Snobol 4B, put Snobol on newer computers, wrote books about Snobol, and emphasized that Snobol can solve *any* problem about strings, even if the problem has nothing to do with linguistics. They won: more people use Snobol than Comit.

Most new versions of Snobol are named after baseball pitching methods — such as Fasbol, Slobol, and Spitbol. (Spitbol stands for **Speedy Implementation of Snobol**.)

Logo

Logo began in 1967, during an evening at Dan Bobrow's home in Belmont, Massachusetts. He'd gotten his Ph.D. from MIT and was working for a company called **Bolt, Beranek, and Newman (BBN)**. In his living room were 3 of his colleagues from BBN (Wally Feurzeig, Cynthia Solomon, and Dick Grant) and an MIT professor: Seymour Papert. BBN had tried to teach young kids how to program by using BBN's own language (Telcomp), which was a variation of Joss. BBN had asked Professor Seymour Papert for his opinion. The group was all gathered in Dan's house to hear Seymour's opinion.

Seymour chatted with the group, which agreed with Seymour on these points:

Telcomp was *not* a great language for kids. It placed too much emphasis on math formulas. Instead of struggling with math, kids should have fun by programming the computer to handle strings instead.

The group also agreed that the most sophisticated language for handling strings was Lisp, but that Lisp was too complex for kids. The group concluded that a new, simplified Lisp should be invented for kids and called **Logo**.

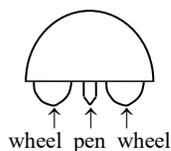
That's how Logo began. Seymour Papert was the guiding light, and all other members of the group gave helpful input during the conversation.

That night, after his guests left, Dan went to his bedroom, where he started writing a program (in Lisp) to make the computer understand Logo.

That's how Logo was born. Work on Logo continued. The 3 main researchers who continued improving Logo were Seymour (the MIT guru), Wally (from BBN), and Cynthia (also from BBN). Logo resembled Lisp but required fewer parentheses.

After helping BBN for a year, Seymour returned to MIT. Cynthia and several other BBN folks worked with him at MIT's Artificial Intelligence Laboratory to improve Logo.

Turtles At first, Logo was as abstract and boring as most other computer languages. But in the spring of 1970, a strange creature walked into the Logo lab. It was a big yellow mechanical turtle. It looked like "half a grapefruit on wheels" and had a pen in its belly:



It also had a horn, feelers, and several other fancy attachments. To use it, you put paper all over the floor then programmed it to roll across the paper. As it rolled, the pen in its belly drew pictures on the paper. The turtle was controlled remotely by a big computer programmed in Logo.

Suddenly, Logo became a fun language whose main purpose was to control the turtle. Kids watching the turtle screamed with delight and wanted to learn how to program it. Logo became a favorite programming game for kids. Even kids who were just 7 years old started programming in Logo. Those kids were barely old enough to read, but reading and writing were *not* prerequisites for learning how to program in Logo. All the kids had to know was:

FD 3 makes the turtle go forward 3 steps
RT 30 makes the turtle turn to the right 30 degrees

As for the rest of Logo — all that abstract stuff about strings, numbers, and Lisp-like lists — the kids ignored it. They wanted to use just the commands "FD" and "RT" that moved the turtle.

The U.S. Government's National Science Foundation donated money, to help MIT improve Logo further. Many kids came into the Logo lab to play with the turtles.

The turtles were expensive, and so were the big computers that controlled them. But during the early 1970's, computer screens got dramatically cheaper; so to save money, MIT stopped building mechanical turtles and instead bought cheap computer screens that showed pictures of turtles. Those pictures were called "mock turtles".

Cheaper computers Logo's first version was done on BBN's expensive weird computer (the MTS 940). Later versions were done on the PDP-1, the PDP-10, and finally on a cheaper computer: the PDP-11 minicomputer (in 1972).

At the end of the 1970's, companies such as Apple and Radio Shack began selling microcomputers, which were even cheaper. MIT wanted to put Logo on microcomputers but ran out of money to pay for the research.

Texas Instruments (TI) came to the rescue....

TI Logo TI agreed to pay MIT to research how to put Logo on TI's microcomputer (the TI-99/4).

TI and MIT thought that would be easy, since MIT already wrote a Pascal program that could make computers understand Logo, and since TI already wrote a version of Pascal for the CPU chip inside the TI-99/4. MIT worried because its Pascal program running on MIT's PDP-10 computer handled Logo too slowly; but TI claimed TI's Pascal was faster than the PDP-10's and so Logo would run fast enough on the TI.

TI was wrong. TI's Pascal didn't make Logo run fast enough, and TI's Pascal also required too much RAM. So TI had to take MIT's research (on the PDP-10) and laboriously translate it into TI's assembly language, by hand. The hand translation went slower than TI expected. TI became impatient and took a shortcut: it omitted parts of Logo, such as decimals. TI began selling its version of Logo, which understood just integers.

MIT Apple Logo After TI started selling its Logo, the MIT group invented a version of Logo for the Apple. The Apple version included decimals but omitted **sprites** (animated creatures that carry objects across the screen) because Apple's hardware couldn't handle sprites fast enough.

MIT wanted to sell the Apple version, since more schools owned Apples than TI computers. But if MIT were to make money from selling the Apple version, MIT might get into legal trouble, since MIT was supposed to be non-profit. And anyway, who "owned" Logo? Possible contenders were:

MIT, which did most of the research
BBN, which trademarked the name "Logo" and did the early research
Uncle Sam, whose National Science Foundation paid for much research
TI, which also paid for much research

Eventually, MIT solved the legal problems and sold the rights for "MIT Apple Logo" to two companies: Krell and Terrapin.

Krell was strictly a marketing company. It sold MIT Apple Logo to schools but made no attempt to improve Logo further.

Terrapin, on the other hand, was a research organization that had built mechanical turtles for several years. Terrapin hired MIT graduates to improve Logo further.

LCSI versus competitors Back when MIT was waiting for its lawyers to determine who owned Apple Logo, a group of MIT's faculty and students (headed by Cynthia Solomon) left MIT and formed a company called **Logo Computer Systems Incorporated (LCSI)**. That company invented its own version of Logo for the Apple. LCSI became successful and was hired by Apple, IBM, Atari, and Microsoft to invent Logo versions for those systems. Commodore hired Terrapin instead.

For the Apple 2c (and 2e and 2+), you could buy 3 Logo versions:

official Apple Logo (sold by Apple Computer Inc. and created by LCSi)
"Terrapin Logo for the Apple" (sold by Terrapin)
original "MIT Logo for the Apple" (sold by Krell)

Krell became unpopular, leaving Terrapin and LCSi as the main Logo versions. LCSi's versions were daring (resulting from wild experiments), while Terrapin's versions were conservative (closer to the MIT original).

The two companies had different styles:

Terrapin was small & friendly and charged little.
 LCSi was big & rude and charged more.

Terrapin's owners had financial difficulties and sold the company to **Harvard Associates** (which had invented a Logo version called "PC Logo"). So Terrapin became part of Harvard Associates (run by Bill Glass, who's friendly).

To find out about his Terrapin Logo, look at his Web site (TerrapinLogo.com) then phone him at 800-774-Logo (or 508-487-4141) or write to 955 Massachusetts Ave. #365, Cambridge MA 02139-3233.

LCSi's newest, daring version of Logo is **MicroWorlds Pro**.

To find out about it, look at LCSi's Web site (LCSi.ca) then phone LCSi at 800-321-5646. LCSi is based in Montreal, Canada but accepts U.S. mail at PO Box 162, Highgate Springs VT 05460.

Logo versus Basic Most of Logo's designers *hate* Basic and want to eliminate Basic from schools altogether. They believe Logo's easier to learn than Basic, encourages a kid to be more creative, and lets a kid think in a more organized fashion. They also argue that since Logo is best for little kids, and since switching languages is difficult, kids should continue using Logo until they graduate from high school and never use Basic.

That argument is wrong, for 2 reasons:

Knowing Basic is *essential* to understanding our computerized society. Most programs are still written in Basic, not Logo, because Basic consumes less RAM and because Basic's newest versions contain many practical features (for business, science, and graphics) that Logo lacks.

Logo suffers from awkward notation. For example, Basic lets you type a formula such as —

A=B+C

but in Logo you must type:

MAKE "A :B+:C

Notice how ugly the Logo command looks! You must put a quotation mark *before* the A but *not afterwards!* Look at those frightful colons! Anybody who thinks such notation is great for kids is a fool.

Extensible One of Logo's nicest features is: you can modify Logo and turn it into your *own* language, because Logo lets you invent your own commands and add them to the Logo language.

A language (such as Logo) that lets you

invent your own commands is called an **extensible language**. Though some earlier languages (such as Lisp) were extensible also, Logo is *more* extensible and pleasanter.

Pilot

Pilot was invented in 1968 by John Starkweather at the University of California's San Francisco branch. It's easier to learn than Basic but intended to be programmed by teachers, not students. Teachers using Pilot can easily make the computer teach students about history, geography, math, French, and other schoolbook subjects.

For example, suppose you're a teacher and want to make the computer chat with your students. Here's how to do it in Pilot:

Basic program

```
PRINT "I AM A COMPUTER"
INPUT "DO YOU LIKE COMPUTERS";A$
IF A$="YES" OR A$="YEAH" OR A$="YEP" OR A$="SURE" OR A$="SURELY" OR A$="I SURE
DO" THEN PRINT "I LIKE YOU TOO" ELSE PRINT "TOUGH LUCK"
```

Pilot program

```
T:I AM A COMPUTER
T:DO YOU LIKE COMPUTERS?
A:
M:YE, SURE
TY:I LIKE YOU TOO
TN:TOUGH LUCK
```

What the computer will do

Type "I AM A COMPUTER".
 Type "DO YOU LIKE COMPUTERS?"
 Accept the human's answer.
 Match. (See whether answer contains "YE" or "SURE".)
 If there was a match, type "I LIKE YOU TOO".
 If no match, type "TOUGH LUCK".

The Pilot program is briefer than Basic.

Atari, Apple, and Radio Shack sold versions of Pilot including commands to handle graphics. Atari's version is the best, since it includes the fanciest graphics & music and even a Logo-like turtle, and it's also the easiest version to learn how to use.

Though Pilot's easier than Basic, most teachers prefer Basic because it's available on more computers, costs less, and accomplishes a greater variety of tasks. Hardly anybody uses Pilot.

Specialists

For specialized applications, use a special language.

Dynamo

Dynamo uses these symbols:

Symbol	Meaning
.J	a moment ago
.K	now
.JK	during the past moment
.KL	during the next moment
DT	how long "a moment" is

For example, suppose you want to explain to the computer how population depends on birth rate. If you let P be the population, BR be the birth rate, and DR be the death rate, here's what to say in Dynamo:

P.K=P.J+DT*(BR.JK-DR.JK)

The equation says: Population now = Population before + (how long "a moment" is) times (Birth Rate during the past moment - Death Rate during the past moment).

World Dynamics The most famous Dynamo program is the **World Dynamics Model**, which Jay Forrester programmed at MIT in 1970. His program has 117 equations that describe 112 variables about our world.

Here's how the program begins:

```
* WORLD DYNAMICS
L P.K=P.J+DT*(BR.JK-DR.JK)
N P=PI
C PI=1.65E9
R BR.KL=P.K*FIFGE(BRN, BRN1, SWT1, TIME.K)*BRFM.K*BRMM.K*BRCM.K*BRPM.K
etc.
```

Here's why:

The first line gives the program's title. The next line defines the Level of Population, in terms of Birth Rate and Death Rate.

The second equation defines the initial Population to be PI (Population Initial). The next equation defines the Constant PI to be 1.65e9, because the world's population was 1.65 billion in 1900.

The next equation says the Rate BR.KL (the Birth Rate during the next moment) is determined by the Population now and several other factors, such as the BRFM (Birth-Rate-from-Food Multiplier), the BRMM (Birth-Rate-from-Material Multiplier), the BRCM (Birth-Rate-from-Crowding Multiplier), and the BRPM (Birth-Rate-from-Pollution Multiplier). Each of those factors is defined in later equations.

When you run the program, the computer automatically solves all the equations simultaneously and draws graphs showing how the population, birth rate, etc. will change during the next several decades.

The graphs show the quality of life will decrease (because of the overpopulation, pollution, and dwindling natural resources). Although the material standard of living will improve for a while, it too will eventually decrease, as will industrialization (capital investment).

The bad outlook is caused mainly by dwindling natural resources. Suppose scientists suddenly make a “new discovery” that lets us reduce our usage of natural resources by 75%. Will our lives be better? The computer predicted that if the “new discovery” were made in 1970, this would happen:

People will live well, so in 2030 the population is almost 4 times what it was in 1970. But the large population generates too much pollution. In 2030, the pollution is being created faster than it can dissipate. From 2040 to 2060, a pollution crisis occurs: the pollution increases until it’s 40 times as great as in 1970 and kills most people on earth, so the world’s population in 2060 is a sixth of what it was in 2040. After the crisis, the few survivors create little pollution and enjoy a very high quality of life.

Forrester tried other experiments on the computer. To improve the quality of life, he tested the effect of requiring birth control, reducing pollution, and adopting other strategies. Each of those simple strategies backfired. The graphs showed that the only way to maintain a high quality of life is to adopt a *combination* strategy immediately:

reduce natural resource usage by 75%
 reduce pollution generation by 50%
 reduce the birth rate by 30%
 reduce capital-investment generation by 40%
 reduce food production by 20%

Other popular applications

Although the World Dynamics Model is Dynamo’s most famous program, Dynamo’s been applied to many other problems.

The first Dynamo programs were aimed at helping managers run companies. Plug your policies about buying, selling, hiring, and firing into the program’s equations; when you run the program, the computer draws a graph showing what will happen to your company during the coming months and years. If you dislike the computer’s prediction, change your policies, put them into the equations, and see whether the computer’s graphs are more optimistic.

How Dynamo developed

Dynamo developed from research at MIT. At MIT in 1958, Richard Bennett invented a language called **Simple**, which stood for “Simulation of Industrial Management Problems with Lots of Equations”. In 1959, Phyllis Fox and Alexander Pugh III invented Dynamo as an improvement on Simple. At MIT in 1961, Jay Forrester wrote a book called *Industrial Dynamics*, which explained how Dynamo can help you manage a company.

MIT is near Boston, whose mayor from 1960 to 1967 was John Collins. When his term as mayor ended, he became a visiting professor at MIT. His office happened to be next to Forrester’s. He asked Forrester whether Dynamo could solve the problems of managing a city. Forrester organized a conference of urban experts and got them to turn urban problems into 330 Dynamo equations involving 310 variables.

Forrester ran the program and made the computer graph the consequences. The results were surprising:

The graph showed that if you try to help the underemployed (by giving them low-cost housing, job-training programs, and artificially created jobs), the city becomes better for the underemployed — but then more underemployed people move to the city, the percentage of the city that’s underemployed increases, and the city is worse than before the reforms were begun. So socialist reform just backfires.

Another example: free public transportation creates *more* traffic, because it encourages people to live farther from their jobs.

The graphs show the only long-term solution to the city’s problems is to do this instead: knock down slums, fund new “labor-intensive export” businesses (businesses that will hire many workers, occupy little land, and produce goods that can be sold outside the city), and let the underemployed fend for themselves in this new environment.

Another surprise: any city-funded housing program makes matters *worse* (regardless of whether the housing is for the underemployed, the workers, or the rich) because more housing creates less space for industry, so fewer jobs.

If you ever become a mayor or President, use the computer’s recommendations cautiously: they’ll improve the cities, but only by driving the underemployed out to the suburbs, which will worsen.

In 1970 Forrester created the World Dynamics Model to help “The Club of Rome”, a private club of 75 people who try to save the world from ecological calamity.

GPSS

A **queue** is a line of people who are waiting. **GPSS** analyzes queues. For example, let’s use GPSS to analyze the customers waiting in “Quickie Joe’s Barbershop”:

Joe’s the only barber in the shop, and he spends exactly 7 minutes on each haircut. (That’s why he’s called “Quickie Joe”.)

About once every 10 minutes, a new customer enters the barbershop. More precisely, the number of minutes before another customer enters is a random number between 5 and 15.

To make the computer imitate the barbershop and analyze what happens to the first 100 customers, type this program:

```
SIMULATE
GENERATE 10,5      A new customer comes every 10 minutes ± 5 minutes.
QUEUE      JOEQ    He waits in the queue, called JOEQ.
SEIZE      JOE      When his turn comes, he seizes JOE,
DEPART     JOEQ    which means he leaves the JOEQ.
ADVANCE    7        After 7 minutes go by,
RELEASE    JOE      he releases JOE (so someone else can use JOE)
TERMINATE  1        and leaves the shop.
START      100     Do all that 100 times.
END
```

Indent so the word SIMULATE begins in column 8 (preceded by 7 spaces) and the “10,5” begins in column 19.

When you run the program, the computer will tell you this:

Joe was working 68.5% of the time. The rest of the time, his shop was empty and he was waiting for customers.

There was never more than 1 customer waiting. “On the average”, .04 customers were waiting. There were 101 customers. (The 101st customer stopped the experiment.) 79 of them (78.2% of them) obtained Joe immediately and didn’t have to wait.

The “average customer” had to wait in line .405 minutes. The “average not-immediately-served customer” had to wait in line 1.863 minutes.

Alternative languages For most problems about queues, GPSS is the easiest language to use. But if your problem is complex, you might have to use **Simscrip** (based on Fortran) or **Simula** (an elaboration of Algol) or **Simpl/I** (an elaboration of PL/I).

Prolog

In 1972, **Prolog** was invented in France at the **University of Marseilles**. In 1981, a different version of Prolog arose in Scotland at the **University of Edinburgh**. In 1986, **Turbo Prolog** was created in California by Borland International (which also created Turbo Pascal).

Those versions of Prolog are called **Marseilles Prolog**, **Edinburgh Prolog**, and **Turbo Prolog**.

Prolog programmers call Marseilles Prolog the “old classic”, Edinburgh Prolog the “current standard”, and Turbo Prolog the “radical departure”.

Turbo Prolog has two advantages over its predecessors: it runs programs extra-fast, and it uses English words instead of weird symbols. On the other hand, it requires extra lines at the beginning of your program, to tell the computer which variables are strings.

The ideal Prolog would be a compromise, incorporating the best features of Marseilles, Edinburgh, and Turbo. Here’s how to use the ideal Prolog and how the various versions differ from it...

Creating the database Prolog analyzes relationships. Suppose Alice loves tennis and sailing, Tom loves everything that Alice loves, and Tom also loves football (which Alice does *not* love). To feed all those facts to the computer, give these Prolog commands:

```
loves(alice,tennis).
loves(alice,sailing).
loves(tom,X) if loves(alice,X).
loves(tom,football).
```

The top two lines say Alice loves tennis and sailing. In the third line, the “X” means “something”, so that line says: Tom loves something if Alice loves it. The bottom line says Tom loves football.

When you type those lines, be careful about capitalization.

You must capitalize variables (such as X). You must *not* capitalize specifics (such as tennis, sailing, football, alice, tom, and love).

At the end of each sentence, put a period.

That’s how to program by using ideal Prolog. Here’s how other versions of Prolog differ...

For *Edinburgh Prolog*, type the symbol “:-” instead of the word “if”.

For *Marseilles Prolog*, replace the period by a semicolon, and replace the word “if” by an arrow (->), which you must put in every line:

```
loves(alice,tennis)->;
loves(alice,sailing)->;
loves(tom,X) -> loves(alice,X);
loves(tom,football)->;
```

For *Turbo Prolog*, you must add extra lines at the top of your program, to warn the computer that the person and sport are strings (“symbols”), and the word “loves” is a verb (“predicate”) that relates a person to a sport:

```
domains
    person,sport=string
predicates
    loves(person,sport)
clauses
    loves(alice,tennis).
    loves(alice,sailing).
    loves(tom,X) if loves(alice,X).
    loves(tom,football).
```

(To indent, press the Tab key. To stop indenting, press the left-arrow key.) When you’ve typed all that, press the Esc key (which means Escape) then the R key (which means Run).

Simple questions After you’ve fed the database to the computer, you can ask the computer questions about it.

Does Alice love tennis? To ask the computer that question, type this:

```
loves(alice,tennis)?
```

The computer will answer:

```
yes
```

Does Alice love football? Ask this:

```
loves(alice,football)?
```

The computer will answer:

```
no
```

That’s how the ideal Prolog works. Other versions differ. *Marseilles Prolog* is similar to the ideal Prolog. *Turbo Prolog* omits the question mark, says “true” instead of “yes”, and says “false” instead of “no”. *Edinburgh Prolog* puts the question mark at the beginning of the sentence instead of the end, like this:

```
?-loves(alice,tennis).
```

Advanced questions What does Alice love? Does Alice love something? Ask this:

```
loves(alice,X)?
```

The computer will answer:

```
X=tennis
X=sailing
2 solutions
```

What does Tom love? Does Tom love something? Ask:

```
loves(tom,X)?
```

The computer will answer:

```
X=tennis
X=sailing
X=football
3 solutions
```

Who loves tennis? Ask:

```
loves(X,tennis)?
```

The computer will answer:

```
X=alice
X=tom
2 solutions
```

Does anybody love hockey? Ask:

```
loves(X,hockey)?
```

The computer doesn’t know of anybody who loves hockey, so the computer will answer:

```
no solution
```

Does Tom love something that Alice doesn’t? Ask:

```
loves(tom,X) and not (loves(alice,X))?
```

The computer will answer:

```
X=football
1 solution
```

That’s ideal Prolog.

Turbo Prolog is similar to ideal Prolog. For *Marseilles Prolog*, replace the word “and” by a blank space.

For *Edinburgh Prolog*, replace the word “and” by a comma. After the computer finds a solution, type a semicolon, which tells the computer to find others; when the computer can’t find any more solutions, it says “no” (which means “no more solutions”) instead of printing a summary message such as “2 solutions”.

Prolog’s popularity After being invented in France, Prolog quickly became popular throughout Europe.

Its main competitor was Lisp, which was invented in the United States before Prolog. Long after Prolog’s debut, Americans continued to use Lisp and ignored Prolog.

In the 1980’s, the Japanese launched the **Fifth Generation Project**, which was an attempt to develop a more intelligent kind of computer. To develop that computer’s software, the Japanese used Prolog instead of Lisp, because Prolog was non-American and therefore furthered the project’s purpose, which was to one-up the Americans.

When American researchers heard that the Japanese chose Prolog as a software weapon, the Americans got scared and launched a counter-attack by learning Prolog also.

Speed When Borland — an American company — developed Turbo Prolog, American researchers were thrilled, since Turbo Prolog ran faster than any other Prolog. It ran faster on a cheap IBM PC than Japan’s Prolog ran on Japan’s expensive maxicomputers!

Assembler

Let's look deeper into the computer to see how circuits can "think".

Number systems

Most humans use the **decimal system**, which consists of ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), because humans have ten fingers. The computer does not have fingers, so it prefers other number systems instead. Here they are....

Binary

Look at these powers of 2:

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$

Now try an experiment. Pick your favorite positive integer, and try to write it as a sum of powers of 2.

For example, suppose you pick 45; you can write it as $32+8+4+1$. Suppose you pick 74; you can write it as $64+8+2$. Suppose you pick 77. You can write it as $64+8+4+1$. *Every* positive integer can be written as a sum of powers of 2.

Let's put those examples in a table:

Original number	Written as sum of powers of 2	Does the sum contain...						
		64?	32?	16?	8?	4?	2?	1?
45	$32+8+4+1$	no	yes	no	yes	yes	no	yes
74	$64+8+2$	yes	no	no	yes	no	yes	no
77	$64+8+4+1$	yes	no	no	yes	yes	no	yes

To write those numbers in the **binary system**, replace "no" by 0 and "yes" by 1:

Decimal system	Binary system
45	0101101 (or simply 101101)
74	1001010
77	1001101

The **decimal system** uses the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 and uses these columns:

thousands	hundreds	tens	units
-----------	----------	------	-------

For example, the decimal number 7105 means "7 thousands + 1 hundred + 0 tens + 5 units".

The **binary system** uses just the digits 0 and 1, and uses these columns:

sixty-fours	thirty-twos	sixteens	eights	fours	twos	units
-------------	-------------	----------	--------	-------	------	-------

For example, the binary number 1001101 means "1 sixty-four + 0 thirty-twos + 0 sixteens + 1 eight + 1 four + 0 twos + 1 unit". In other words, it means seventy-seven.

In elementary school, you were taught how to do arithmetic in the decimal system. You had to memorize the addition and multiplication tables:

		DECIMAL ADDITION									
		0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9	
1	1	2	3	4	5	6	7	8	9	10	11
2	2	3	4	5	6	7	8	9	10	11	12
3	3	4	5	6	7	8	9	10	11	12	13
4	4	5	6	7	8	9	10	11	12	13	14
5	5	6	7	8	9	10	11	12	13	14	15
6	6	7	8	9	10	11	12	13	14	15	16
7	7	8	9	10	11	12	13	14	15	16	17
8	8	9	10	11	12	13	14	15	16	17	18
9	9	10	11	12	13	14	15	16	17	18	19

		DECIMAL MULTIPLICATION									
		0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	
2	0	2	4	6	8	10	12	14	16	18	
3	0	3	6	9	12	15	18	21	24	27	
4	0	4	8	12	16	20	24	28	32	36	
5	0	5	10	15	20	25	30	35	40	45	
6	0	6	12	18	24	30	36	42	48	54	
7	0	7	14	21	28	35	42	49	56	63	
8	0	8	16	24	32	40	48	56	64	72	
9	0	9	18	27	36	45	54	63	72	81	

In the binary system, the only digits are 0 and 1, so the tables are briefer:

BINARY ADDITION

0	0
0	1
1	0
1	1

because two is written "10" in binary

BINARY MULTIPLICATION

0	0
0	1
1	0
1	1

If society had adopted the binary system instead of the decimal system, you'd have been spared many hours of memorizing!

Usually, when you ask the computer to perform a computation, it converts your numbers from the decimal system to the binary system, performs the computation by using the binary addition and multiplication tables, and then converts the answer from the binary system to the decimal system, so you can read it. For example, if you ask the computer to print $45+74$, it will do this:

45	converted to binary is	101101	
+74	converted to binary is	+1001010	
		1110111	converted to decimal is 119
			because $1+1=10$

The conversion from decimal to binary and then back to decimal is slow. But the computation itself (in this case, addition) is quick, since the binary addition table is so simple. The only times the computer must convert is during input (decimal to binary) and output (binary to decimal). The rest of the execution is performed quickly, entirely in binary.

You know fractions can be written in the decimal system, by using these columns:

units	point	tenths	hundredths	thousandths
-------	-------	--------	------------	-------------

For example, $1\frac{5}{8}$ can be written as 1.625, which means "1 unit + 6 tenths + 2 hundredths + 5 thousandths".

To write fractions in the binary system, use these columns instead:

units	point	halves	fourths	eighths
-------	-------	--------	---------	---------

For example, $1\frac{5}{8}$ is written in binary as 1.101, which means "1 unit + 1 half + 0 fourths + 1 eighth".

You know $\frac{1}{3}$ is written in the decimal system as 0.333333..., which unfortunately never terminates. In the binary system, the situation is no better: $\frac{1}{3}$ is written as 0.010101.... Since the computer stores just a finite number of digits, it can't store $\frac{1}{3}$ accurately — it stores just an approximation.

A more distressing example is $\frac{1}{5}$. In the decimal system, it's .2, but in the binary system it's .0011001100110011.... So the computer can't handle $\frac{1}{5}$ accurately, even though a human can.

Most of today's microcomputers and minicomputers are inspired by a famous maxicomputer built by DEC and called the DECsystem-10 (or PDP-10). Though DEC is no longer in business, its influence lives on!

Suppose you run this Basic program on a DECsystem-10 computer:

```
10 PRINT "MY FAVORITE NUMBER IS";4.001-4
20 END
```

The computer will try to convert 4.001 to binary. Unfortunately, it can't be converted exactly; the computer's binary approximation of it is slightly too small. The computer's final answer to 4.001-4 is therefore slightly less than the correct answer. Instead of printing MY FAVORITE NUMBER IS .001, the computer will print MY FAVORITE NUMBER IS .000999987.

If your computer isn't a DECsystem-10, its approximation will be slightly different. To test your computer's accuracy, try 4.0001-4, and 4.00001-4, and 4.000001-4, etc. You might be surprised at its answers.

Let's see how the DECsystem-10 handles this:

```
10 FOR X = 7 TO 193 STEP .1
20 PRINT X
30 NEXT X
40 END
```

The computer will convert 7 and 193 to binary accurately, but will convert .1 to binary just approximately; the approximation is slightly too large. The last few numbers it should print are 192.8, 192.9, and 193, but because of the approximation it will print slightly more than 192.8, then slightly more than 192.9, and then stop (since it is not allowed to print anything over 193).

There are just two binary digits: 0 and 1. A binary digit is called a **bit**. For example, .001100110011 is a binary approximation of $\frac{1}{5}$ that consists of twelve bits. A sixteen-bit approximation of $\frac{1}{5}$ would be .0011001100110011. A bit that is 1 is called **turned on**; a bit that is 0 is **turned off**. For example, in the expression 11001, three bits are turned on and two are off. We also say three of the bits are **set** and two are **cleared**.

In a computer, all info is coded as bits:

Location	What's 1?	What's 0?
wire	high voltage	low voltage
flashing light	the light is on	the light is off
punched paper tape	a hole in the tape	no hole
punched IBM card	a hole in the card	no hole
magnetic drum	a magnetized area	not magnetized
core memory	a core (iron doughnut) magnetized clockwise	counterclockwise

For example, to represent 11 on part of a punched paper tape, the computer punches two holes close together. To represent 1101, the computer punches two holes close together, and then another hole farther away.

Octal

Octal is a shorthand notation for binary:

Octal	Meaning
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Each octal digit stands for three bits. For example, the octal number 72 is short for this:

```
111010
 7  2
```

To convert a binary integer to octal, divide the number into chunks of three bits, starting at the right. For example, here's how to convert 11110101 to octal:

```
11110101
 3  6  5
```

To convert a binary real number to octal, divide the number into chunks of three bits, starting at the decimal point and working in both directions:

```
10100001.10011
 2  4  1 . 4  6
```

Hexadecimal

Hexadecimal is another short-hand notation for binary:

Hexadecimal	Meaning
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

For example, the hexadecimal number 4F is short for this:

```
01001111
 4  F
```

To convert a binary number to hexadecimal, divide the number into chunks of 4 bits, starting at the decimal point and working in both directions:

```
11010110100.1111111
 6  B  4  .  F  E
```

Character codes

To store a character in a string, the computer uses a code.

Ascii

The most famous code is the **American Standard Code for Information Interchange (Ascii)**, which has 7 bits for each character. Here are examples:

Character	Ascii code	Ascii code in hexadecimal
space	0100000	20
!	0100001	21
"	0100010	22
#	0100011	23
\$	0100100	24
%	0100101	25
&	0100110	26
'	0100111	27
(0101000	28
)	0101001	29
*	0101010	2A
+	0101011	2B
,	0101100	2C
-	0101101	2D
.	0101110	2E
/	0101111	2F
0	0110000	30
1	0110001	31
2	0110010	32
etc.		
9	0111001	39
:	0111010	3A
;	0111011	3B
<	0111100	3C
=	0111101	3D
>	0111110	3E
?	0111111	3F
@	1000000	40
A	1000001	41
B	1000010	42
C	1000011	43
etc.		
Z	1011010	5A
[1011011	5B
\	1011100	5C
]	1011101	5D
^	1011110	5E
_	1011111	5F

“Ascii” is pronounced “ass key”.
Many variants of Ascii were invented.

Most **terminals** (which connect to a maxicomputer or minicomputer) use that 7-bit Ascii, unmodified.

Most **microcomputers** and **PDP-11** minicomputers use an “8-bit Ascii” formed by putting a 0 before 7-bit Ascii.

PDP-8 minicomputers use mainly a “6-bit Ascii” (formed by eliminating 7-bit Ascii’s leftmost bit) but can also handle an “8-bit Ascii” formed by putting a 1 before 7-bit Ascii.

PDP-10 maxicomputers use mainly 7-bit Ascii but can also handle a “6-bit Ascii” formed by eliminating Ascii’s second bit. For example, the 6-bit Ascii code for the symbol \$ is 0 00100.

Bytes

Nowadays, a “byte” usually means 8 bits. For example, here’s a byte: 10001011.

For old computers using 7-bit Ascii, programmers sometimes define a byte to be 7 bits instead of 8. For old computers using 6-bit Ascii, programmers sometimes define a byte to be 6 bits. So if someone tries to sell you an old computer whose memory can hold “16,000 bytes”, he probably means 16,000 8-bit bytes but might mean 7-bit bytes or 6-bit bytes.

To be more precise, some computerists call 8 bits an **octet** instead of a byte.

Nibbles

A **nibble** is 4 bits. It’s half of an 8-bit byte. Since a hexadecimal digit stands for 4 bits, a **hexadecimal digit stands for a nibble**.

Unicode

To store a character, the Internet uses **Unicode**, which lets each character’s code contain *many* bits. Unlike Ascii, which handles just English, Unicode can also handle European languages (which have accents above and below the characters) and the wild characters used in Greek, Russian, Chinese, Japanese, and Korean.

To store each character as bits, Unicode uses a trick called the **Unicode Transformation Format’s 8-bit version (UTF-8)**. Here’s how the computer reads bits sent by another computer using UTF-8.

Byte beginning with 0 If a byte begins with 0, the next 7 bits are the same as 7-bit Ascii code.

For example, suppose the computer receives this byte: 00100100. Since it begins with 0, the next 7 bits (0100100) are a 7-bit Ascii code. The computer looks up 0100100 in a table and discovers 0100100 is the Ascii code for a dollar sign (\$), so the computer puts a dollar sign on your screen.

Byte beginning with 1 If a byte begins with 1, the computer does this analysis:

If the byte begins with 110,	the code will be 2 bytes long.
If the byte begins with 1110,	the code will be 3 bytes long.
If the byte begins with 11110,	the code will be 4 bytes long.

For example, suppose the computer receives this byte: **11100010**. Since the byte begins with 1110, the code will be 3 bytes long, so the computer must look at 3 bytes altogether to read the character.

Suppose the 3 bytes are **11100010 10000010 10101100**. The first byte begins with this warning: **1110** (which means “the code will be 3 bytes long”). The second byte begins with this warning: **10** (which means “this byte is continuing a code that was started earlier”). The third byte begins with the same warning: **10** (which means “this byte is continuing a code that was started earlier”).

The parts of the 3-byte code that are *not* warnings are: 0010 000010 101100. The computer looks up that in a table and discovers 0010 000010 101100 is the Unicode for a Euro sign (€), so the computer puts a Euro sign on your screen.

Unicode was created to be efficient:

The most popular characters (the Ascii characters) consume just 1 byte.
Characters that are somewhat less popular consume 2 or 3 bytes.
Characters that are used rarely consume 4 bytes.

If the computer encounters a byte that begins with **10**, the computer knows that the byte is *not* a character’s *first* byte, so the computer must hunt back to find the character’s first byte — or request that another computer retransmit the character.

EBCDIC

Instead of using Ascii, IBM mainframes use the **Extended Binary-Coded-Decimal Interchange Code (EBCDIC)**, which has 8 bits for each character. Here are examples:

Character	EBCDIC code in hexadecimal	Character	EBCDIC code in hexadecimal
space	40	A	C1
€	4A	B	C2
<	4C	C	C3
(4D	etc.	
+	4E	I	C9
	4F	J	D1
&	50	K	D2
!	5A	L	D3
\$	5B	etc.	
*	5C	R	D9
)	5D	S	E2
;	5E	T	E3
┌	5F	U	E4
-	60	etc.	
/	61	Z	E9
,	6B	0	F0
%	6C	1	F1
	6D	2	F2
>	6E	etc.	
?	6F	9	F9
:	7A		
#	7B		
@	7C		
'	7D		
=	7E		
"	7F		

“EBCDIC” is usually pronounced “ebb sih Dick,” though programmers hating it say “Ed sucks Dick.”

IBM 360 computers can also handle an “8-bit Ascii”, formed by copying Ascii’s first bit after the second bit. For example, the 8-bit Ascii code for the symbol \$ is 01000100. But IBM 370 computers (which are newer than IBM 360 computers) don’t bother with Ascii: they stick strictly with EBCDIC.

80-column IBM cards use **Hollerith code**, which resembles EBCDIC but has 12 bits instead of 8. 96-column IBM cards use a 6-bit code that’s an abridgement of Hollerith code.

Here’s a program written in old Basic:

```
10 IF "9"<"A" THEN 100
20 PRINT "CAT"
30 STOP
100 PRINT "DOG"
110 END
```

Which will the computer print: CAT or DOG? The answer depends on whether the computer uses Ascii or EBCDIC.

Suppose the computer uses 7-bit Ascii. Then the code for “9” is hexadecimal 39, and the code for “A” is hexadecimal 41. Since 39 is less than 41, the computer considers “9” to be less than “A”, so the computer prints DOG.

But if the computer uses EBCDIC instead of Ascii, the code for “9” is hexadecimal F9, and the code for “A” is hexadecimal C1; since F9 is greater than C1, the computer considers “9” to be greater than “A”, so the computer prints CAT.

Sexy assembler

In this chapter, you’ll learn the fundamental concepts of assembly language, quickly and easily.

Unfortunately, different CPUs have different assembly languages.

I’ve invented an assembly language that combines the best features of all the other assembly languages. My assembly language is called **Sexy Ass**, because it’s a **S**imple, **E**xcellent, **Y**ummy **A**ssembler.

After you study the mysteries of the Sexy Ass, you can easily get your rear in gear and become the dominant master of the assemblers sold for IBM, Apple, and competitors. Mastering them will become so easy that you’ll say, “Assembly language is a piece of cheesecake!”

Bytes in my ASS

Let’s get a close-up view of the Sexy Ass...

CPU registers The computer’s guts consist of two main parts: the brain (which is called the **CPU**) and the **main memory** (which consists of RAM and ROM).

Inside the CPU are many electronic boxes, called **registers**. Each register holds several electrical signals; each signal is called a **bit**; so each register holds several bits. Each bit is either 1 or 0.

A “1” represents a high voltage; a “0” represents a low voltage.

If the bit is 1, the bit is said to be **high** or **on** or **set** or **true**.

If the bit is 0, the bit is said to be **low** or **off** or **cleared** or **false**.

The CPU’s most important register is called the **accumulator (A)**. In the Sexy Ass system, the accumulator consists of 8 bits, which is 1 byte. (Later, I’ll explain how to make the CPU handle several bytes simultaneously; but the accumulator itself holds just 1 byte.)

Memory locations Like the CPU, the main memory consists of electronic boxes. The electronic boxes *in the CPU* are called **registers**, but the electronic boxes *in the main memory* are called **memory locations** instead. Because the main memory acts like a gigantic post office, the memory locations are also called **addresses**. In the Sexy Ass system, each memory location holds 1 byte. There are many *thousands* of memory locations; they’re numbered 0, 1, 2, 3, etc.

Number systems When using Sexy Ass, you can type numbers in decimal, binary, or hexadecimal. (For Sexy Ass, octal isn't useful.) For example, the number "twelve" is written "12" in decimal, "1100" in binary, and "C" in hexadecimal. To indicate which number system you're using, **put a percent sign in front of each binary number, and put a dollar sign in front of each hexadecimal number.** For example, in Sexy Ass you can write the number "twelve" as either 12 or %1100 or \$C. (In that respect, Sexy Ass copies the 6502 assembly language, which also uses the percent sign and the dollar sign.)

Most of the time, we'll be using hexadecimal, so let's quickly review what hexadecimal is all about. **To count in hexadecimal, just start counting as you learned in elementary school** (\$1, \$2, \$3, \$4, \$5, \$6, \$7, \$8, \$9); **but after \$9, you continue counting by using the letters of the alphabet** (\$A, \$B, \$C, \$D, \$E, and \$F). **After \$F (which is fifteen), you say \$10** (which means sixteen), then say \$11 (which means seventeen), then \$12, then \$13, then \$14, etc., until you reach \$19; then come \$1A, \$1B, \$1C, \$1D, \$1E, and \$1F. Then come \$20, \$21, \$22, etc., up to \$29, then \$2A, \$2B, \$2C, \$2D, \$2E, \$2F, \$30. Eventually, you get up to \$99, then \$9A, \$9B, \$9C, \$9D, \$9E, and \$9F. Then come \$A0, \$A1, \$A2, etc., up to \$AF. Then come \$B0, \$B1, \$B2, etc., up to \$BF. You continue that pattern, until you reach \$FF. Get together with your friends, and try counting up to \$FF. (Don't bother pronouncing the dollar signs.) Yes, you too can count like a pro!

Since each hexadecimal digit represents 4 bits, an 8-bit byte requires *two* hexadecimal digits. So a byte can be anything from \$00 to \$FF.

Main segment I said the main memory consists of *thousands* of memory locations, numbered 0, 1, 2, etc. The main memory's most important part is called the **main memory bank** or **main segment**: that part consists of 65,536 memory locations (64K), which are numbered from 0 to 65,535. Programmers usually number them in hexadecimal; the hexadecimal numbers go from \$0000 from \$FFFF. (\$FFFF in hexadecimal is the same as 65,535 in decimal.) Later, I'll explain how to use other parts of the memory; but for now, let's restrict our attention to just 64K main segment.

How to copy a byte Here's a simple, one-line program, written in the SEXY ASS assembly language:

```
LOAD    $7000
```

It makes the computer copy one byte, from memory location \$7000 to the accumulator. So after the computer obeys that instruction, the accumulator will contain the same data as the memory location. For example, if the memory location contains the byte %01001111 (which can also be written as \$4F), so will the accumulator.

Notice the wide space before and after the word LOAD. To make the wide space, press the TAB key.

The word LOAD tells the computer to copy from a memory location to the accumulator. The opposite of the word LOAD is the word STORE: it tells the computer to copy from the accumulator to a memory location. For example, if you type —

```
STORE  $7000
```

the computer will copy a byte from the accumulator to memory location \$7000.

Problem: write an assembly-language program that copies a byte from memory location \$7000 to memory location \$7001. Solution: you must do it in two steps. First, copy from memory location \$7000 to the accumulator (by using the word LOAD); then copy from the accumulator to memory location \$7001 (by using the word STORE). Here's the program:

```
LOAD   $7000
STORE  $7001
```

Arithmetic

If you say —

```
INC
```

the computer will **increment** (increase) the number in the accumulator, by adding 1 to it. For example, if the accumulator contains the number \$25, and you then say INC, the accumulator will contain the number \$26. For another example, if the accumulator contains the number \$39, and you say INC, the accumulator will contain the number \$3A (because, in hexadecimal, after 9 comes A).

Problem: write a program that increments the number that's in location \$7000; for example, if location \$7000 contains \$25, the program should change that data, so that location \$7000 contains \$26 instead. Solution: copy the number from location \$7000 to the accumulator, then increment the number, then copy it back to location \$7000....

```
LOAD   $7000
INC
STORE  $7000
```

That example illustrates the fundamental rule of assembly-language programming, which is: **to manipulate a memory location's data, copy the data to the accumulator, manipulate the accumulator, and then copy the revised data from the accumulator to memory.**

The opposite of INC is DEC: it **decrements** (decreases) the number in the accumulator, by subtracting 1 from it.

If you say —

```
ADD    $7000
```

the computer will change the number in the accumulator, by adding to it the number that was in memory location \$7000. For example, if the accumulator had contained the number \$16, and memory location \$7000 had contained the number \$43, the number in the accumulator will change and become the sum, \$59. The number in memory location \$7000 will remain unchanged: it will still be \$43.

Problem: find the sum of the numbers in memory locations \$7000, \$7001, and \$7002, and put that sum into memory location \$7003. Solution: copy the number from memory location \$7000 to the accumulator, then add to the accumulator the numbers from memory locations \$7001 and \$7002, so that the accumulator to memory location \$7003....

```
LOAD   $7000
ADD    $7001
ADD    $7002
STORE  $7003
```

The opposite of ADD is SUB, which means SUBtract. If you say SUB \$7000, the computer will change the number in the accumulator, by subtracting from it the number in memory location \$7000.

Immediate addressing

If you say —

LOAD	#\$25
------	-------

the computer will put the number \$25 into the accumulator. The \$25 is the data. In the instruction “LOAD #\$25”, the symbol “#” tells the computer that the \$25 is the data instead of being a memory location.

If you were to omit the #, the computer would assume the \$25 meant memory location \$0025, and so the computer would copy data from memory location \$0025 to the accumulator.

An instruction that contains the symbol # is said to be an **immediate** instruction; it is said to use **immediate** addressing. Such instructions are unusual.

The more usual kind of instruction, which does *not* use the symbol #, is called a **direct** instruction.

Problem: change the number in the accumulator, by adding \$12 to it. Solution:

ADD	#\$12
-----	-------

Problem: change the number in memory location \$7000, by adding \$12 to that number. Solution: copy the number from memory location \$7000 to the accumulator, add \$12 to it, and then copy the sum back to the memory location....

LOAD	\$7000
ADD	#\$12
STORE	\$7000

Problem: make the computer find the sum of \$16 and \$43, and put the sum into memory location \$7000. Solution: put \$16 into the accumulator, add \$43 to it, and then copy from the accumulator to memory location \$7000....

LOAD	#\$16
ADD	#\$43
STORE	\$7000

Video RAM

The video RAM is part of the computer's RAM and holds a copy of what's on the screen.

For example, suppose you're running a program that analyzes taxicabs, and your computer's screen shows information about various cabs. If the upper-left corner of the screen shows the word CAB, the video RAM contains the Ascii code numbers for the letters C, A, and B. Since the Ascii code number for C is 67 (which is \$43), and the Ascii code number for A is 65 (which is \$41), and the Ascii code number for B is 66 (which is \$42), the video RAM contains \$43, \$41, and \$42. The \$43, \$41, and \$42 represent the word CAB.

Suppose that the video RAM begins at memory location \$6000. If the screen's upper-left corner shows the word CAB, memory location \$6000 contains the code for C (which is \$43); the next memory location (\$6001) contains the code for A (which is \$41); and the next memory location (\$6002) contains the code for B (which is \$42).

Problem: assuming that the video RAM begins at location \$6000, make the computer write the word CAB onto the screen's upper-left corner. Solution: write \$43 into memory location \$6000, write \$41 into memory location \$6001, and write \$42 into memory location \$6002....

LOAD	#\$43
STORE	\$6000
LOAD	#\$41
STORE	\$6001
LOAD	#\$42
STORE	\$6002

The computer knows that \$43 is the code number for “C”. When you're writing that program, if you're too lazy to figure out the \$43, you can simply write “C”; the computer will understand. So you can write the program like this:

LOAD	#"C"
STORE	\$6000
LOAD	#"A"
STORE	\$6001
LOAD	#"B"
STORE	\$6002

That's the solution if the video RAM begins at memory location \$6000. On *your* computer, the video RAM might begin at a different memory location instead. To find out about *your* computer's video RAM, look at the back of the technical manual that came with your computer. There you'll find a **memory map**: it shows which memory locations are used by the video RAM, which memory locations are used by other RAM, and which memory locations are used by the ROM.

Flags

The CPU contains **flags**. Here's how they work.

Carry flag A byte consists of 8 bits. The smallest number you can put into a byte is %00000000. The largest number you can put into a byte is %11111111, which in hexadecimal is \$FF; in decimal, it's 255.

What happens if you try to go higher than %11111111? To find out, examine this program:

LOAD	##10000001
ADD	##10000010

In that program, the top line puts the binary number %10000001 into the accumulator. The next line tries to add %10000010 to the accumulator. But **the sum, which is %10000011, contains 9 bits instead of 8, and therefore can't fit into the accumulator.**

The computer splits that sum into two parts: the left bit (1) and the remaining bits (0000011). The left bit (1) is called the carry bit; the remaining bits (0000011) are called the tail. Since the tail contains 8 bits, it fits nicely into the accumulator; so the computer puts it into the accumulator. **The carry bit is put into a special place inside the CPU; that special place is called the carry flag.**

So that program makes the accumulator become 00000011, and makes the carry flag become 1.

Here's an easier program:

LOAD	##1
ADD	##10

The top line puts %1 into the accumulator; so the accumulator's 8 bits are %00000001. The bottom line adds %10 to the number in the accumulator; so the accumulator's 8 bits become %00000011. Since the numbers involved in that addition were so small, there was no need for a 9th bit — no need for a carry bit. To emphasize that no carry bit was required, the carry flag automatically becomes 0.

Here's the rule: if an arithmetic operation (such as ADD, SUB, INC, or DEC) gives a result that's too long to fit into 8 bits, the carry flag becomes 1; otherwise, the carry flag becomes 0.

Negatives The largest number you can fit into a byte %11111111, which in decimal is 255. Suppose you try to add 1 to it. The sum is %100000000, which in decimal is 256. But since %100000000 contains 9 bits, it's too long to fit into a byte. So the computer sends the leftmost bit (the 1) to the carry flag, and puts the tail (the 00000000) into the accumulator. As a result, the accumulator contains 0.

So in assembly language, if you tell the computer to do %11111111+1 (which is 255+1), the accumulator says the answer is 0 (instead of 256).

In assembly language, %11111111+1 is 0. In other words, %11111111 solves the equation $x+1=0$.

According to high school algebra, the equation $x+1=0$ has this solution: $x=-1$. But we've seen that in the assembly language, the equation $x+1=0$ has the solution $x=%11111111$. Conclusion: in assembly language, -1 is the same as %11111111.

Now you know that -1 is the same as %11111111, which is 255. Yes, -1 is the same as 255. Similarly, -2 is the same as 254; -3 is the same as 253; -4 is the same as 252. Here's the general formula: -n is the same as 256-n. (That's because 256 is the same as 0.)

%11111111 is 255 and is also -1. Since -1 is a shorter name than 255, we say that %11111111 is *interpreted as* -1. Similarly, %11111110 is 254 and also -2; since -2 is a shorter name than 254, we say that %11111110 is interpreted as -2. At the other extreme, %00000010 is 2 and is also -254; since 2 is a shorter name than -254, we say that %11111110 is interpreted as 2. Here's the rule: if a number is "almost" 256, it's interpreted as a negative number; otherwise, it's interpreted as a positive number.

How high must a number be, in order to be "almost" 256, and therefore to be interpreted as a negative number? The answer is: if the number is at least 128, it's interpreted as a negative number. Putting it another way, if the number's leftmost bit is 1, it's interpreted as a negative number.

That strange train of reasoning leads to this definition: **a negative number is a byte whose leftmost bit is 1.**

A byte's leftmost bit is therefore called the **negative bit** or the **sign bit**.

Flag register You've seen that the CPU contains a register called the **accumulator**. The CPU also contains a second register, called the **flag register**. In the Sexy Ass system, the flag register contains 8 bits (one byte). Each of the 8 bits in the flag register is called a **flag**; so the flag register contains 8 flags.

Each flag is a bit: it's either 1 or 0. If the flag is 1, the flag is said to be **up** or **raised** or **set**. If the flag is 0, the flag is said to be **down** or **lowered** or **cleared**.

One of the 8 flags is the carry flag: it's raised (becomes 1) whenever an arithmetic operation requires a 9th bit. (It's lowered whenever an arithmetic operation does *not* require a 9th bit.)

Another one of the flags is **the negative flag: it's raised whenever the number in the accumulator becomes negative**. For example, if the accumulator becomes %11111110 (which is -2), the negative flag is raised (i.e. the negative flag becomes 1). It's lowered whenever the number in the accumulator becomes *non-negative*.

Another one of the flags is **the zero flag: it's raised whenever the number in the accumulator becomes zero**. (It's lowered whenever the number in the accumulator becomes *non-zero*.)

Jumps

You can give each line of your program a name. For example, you can give a line the name FRED. To do so, put the name FRED at the beginning of the line, like this:

FRED	LOAD	\$7000
------	------	--------

The line's name (FRED) is at the left margin. The command itself (LOAD \$7000) is indented by pressing the TAB key. In that line, FRED is called the **label**, LOAD is called the **operation** or **mnemonic**, and \$7000 is called the **address**.

Languages such as BASIC let you say "GO TO". **In assembly language, you say "JUMP" instead of "GO TO"**. For example, to make the computer GO TO the line named FRED, say:

JUMP	FRED
------	------

The computer will obey: it will JUMP to the line named FRED.

You can say —

JUMPN	FRED
-------	------

That means: JUMP to FRED, if the Negative flag is raised. So the computer will JUMP to FRED if a negative number was recently put into the accumulator. (If a *non-negative* number was recently put into the accumulator, the computer will *not* jump to FRED.)

JUMPN means "JUMP if the Negative flag is raised." JUMPC means "JUMP if the Carry flag is raised." JUMPZ means "JUMP if the Zero flag is raised."

JUMPNL means "JUMP if the Negative flag is Lowered." JUMPCL means "JUMP if the Carry flag is Lowered." JUMPZL means "JUMP if the Zero flag is Lowered."

Problem: make the computer look at memory location \$7000; if the number in that memory location is negative, make the computer jump to a line named FRED. Solution: copy the number from memory location \$7000 to the accumulator, to influence the Negative flag; then JUMP if Negative....

LOAD	\$7000
JUMPN	FRED

Problem: make the computer look at memory location \$7000. If the number in that memory location is negative, make the computer print a minus sign in the upper-left corner of the screen; if the number is positive instead, make the computer print a plus sign instead; if the number is zero, make the computer print a zero. Solution: copy the number from memory location \$7000 to the accumulator (by saying LOAD); then analyze that number (by using JUMPN and JUMPZ); then LOAD the Ascii code number for either "+" or "-" or "0" into the accumulator (whichever is appropriate); finally copy that Ascii code number from the accumulator to the video RAM (by saying STORE)....

	LOAD	\$7000
	JUMPN	NEGAT
	JUMPZ	ZERO
	LOAD	#"+"
NEGAT	JUMP	DISPLAY
	LOAD	#"-"
	JUMP	DISPLAY
ZERO	LOAD	#"0"
DISPLAY	STORE	\$6000

Machine language

I've been explaining assembly language. **Machine language** resembles assembly language; what's the difference?

To find out, let's look at a machine language called **Sexy Macho** (because it's a **S**imple, **E**xcellent, **Y**ummy **M**achine-language **O**riginal).

Sexy Macho resembles Sexy Ass; here are the main differences:

In Sexy Ass assembly language, you use words such as LOAD, STORE, INC, DEC, ADD, SUB, and JUMP. Those words are called *operations* or *mnemonics*. In Sexy Macho machine language, you replace those words by code numbers: the code number for LOAD is 1; the code number for STORE is 2; INC is 3; DEC is 4; ADD is 5; SUB is 6; and JUMP is 7. The code numbers are called the **operation codes** or **op codes**.

In Sexy Ass assembly language, the symbol “#” indicates immediate addressing; a lack of the symbol “#” indicates direct addressing instead. In Sexy Macho machine language, you replace the symbol “#” by the code number 1; if you want direct addressing instead, you must use the code number 0.

In Sexy Macho, all code numbers are hexadecimal.

For example, look at this Sexy Ass instruction:

```
ADD    #$43
```

To translate that instruction into Sexy Macho machine language, just replace each symbol by its code number. Since the code number for ADD is 5, and the code number for # is 1, the Sexy Macho version of that line is:

```
5143
```

Let's translate STORE \$7003 into Sexy Macho machine language. Since the code for STORE is 2, and the code for direct addressing is 0, the Sexy Macho version of that command is:

```
207003
```

In machine language, you can't use words or symbols: you must use their code numbers instead. To translate a program from assembly language to machine language, you must look up the code number of each word or symbol.

An **assembler** is a program that makes the computer translate from assembly language to machine language.

The CPU understands just machine language: it understands just numbers. It does *not* understand assembly language: it does not understand words and symbols. **If you write a program in assembly language, you must buy an assembler, which translates your program from assembly language to machine language**, so that the computer can understand it.

Since assembly language uses English words (such as LOAD), assembly language seems more “human” than machine language (which uses code numbers). Since programmers are humans, programmers prefer assembly language over machine language. Therefore, the typical programmer writes in assembly language then uses an assembler to translate the program to machine language, which is the language that the CPU ultimately requires.

Here's how the typical assembly-language programmer works:

The programmer types the assembly-language program and uses a word processor to help edit it. The word processor puts the assembly-language program onto a disk.

Then the programmer uses the assembler to translate the assembly-language program into machine language. The assembler puts the machine-language version of the program onto the disk.

Now the disk contains *two* versions of the program: the disk contains the original version (in assembly language) and also contains the translated version (in machine language). The original version (in assembly language) is called the **source code**; the translated version (in machine language) is called the **object code**.

Finally, the programmer gives a command that makes the computer copy the machine-language version (the object code) from the disk to the RAM and run it.

Here's how the assembler translate “JUMP FRED” into machine language:

The assembler realizes that FRED is the name for a line in your program. The assembler hunts through your program, to find the line labeled FRED.

When the assembler finds that line, it analyzes that line, to figure out where that line will be in the RAM after the program is translated into machine language and running. For example, suppose the line that's labeled FRED will become a machine-language line which, when the program is running, will be in the RAM at memory location \$2053. Then “JUMP FRED” must be translated into this command: “jump to the machine-language line that's in the RAM at memory location \$2053”. So “JUMP FRED” really means:

```
JUMP    $2053
```

Since the code number for JUMP is **7**, and the addressing isn't immediate (and so has code **0** instead of 1), the machine-language version of JUMP FRED is:

```
702053
```

System software

The computer's main memory consists of RAM and ROM. In a typical computer, the first few memory locations (\$0000, \$0001, \$0002, etc.) are ROM: they permanently contain a program called the **bootstrap**, which is written in machine language.

When you turn on the computer's power switch, the computer automatically runs the bootstrap program. If your computer uses disks, the bootstrap program makes the computer start reading information from the disk in the main drive. In fact, it makes the computer copy a machine-language program from the disk to the RAM. The machine-language program that it copies is called the **disk operating system (DOS)**.

After the DOS has been copied to the RAM, the computer starts running the DOS program. The DOS program makes the computer print a message on the screen (such as “Welcome to CP/M” or “Welcome to MS-DOS” or “Windows XP”), print a symbol on the screen (such as “A>” or a Start button), and then wait for you to give a command.

That whole procedure is called **bootstrapping** (or **booting up**), because of the phrase “pull yourself up by your own bootstraps”. By using the bootstrap program, the computer pulls itself up to new intellectual heights: it becomes a CP/M machine or MS-DOS machine or Windows machine.

After booting up, you can start writing programs in Basic. But how does the computer understand the Basic words, such as PRINT, INPUT, IF, THEN, and GO TO? Here's how:

While you're using Basic, the computer is running a machine-language program, that makes the computer *seem* to understand Basic. That machine-language program, which is in the computer's ROM or RAM, is called the **Basic language processor** or **Basic interpreter**. If your computer uses **Microsoft** Basic, the Basic interpreter is a machine-language program that was written by Microsoft Incorporated.

How assemblers differ

In a microcomputer, the CPU is a single chip, called the **microprocessor**. The most popular microprocessors have been the **8088**, the **68000**, and the **6502**.

The **8088**, designed by Intel, hides in the IBM PC and clones. (The plain version is called the 8088; souped-up versions are called the **80286**, the **386**, the **486**, and the **Pentium**.)

The **68000**, designed by Motorola, hides in older computers that rely on mice: the Apple Mac, Commodore Amiga, and Atari ST. (The plain version is called the 68000; a souped-up version, called the **68020**, is in the Mac 2; an even fancier version, called the **68030**, is in fancier Macs.)

The **6502**, designed by MOS Technology (which has become part of Commodore), hides in old-fashioned cheap computers: the Apple 2 family, the Commodore 64 & 128, and the Atari XL & XE.

Let's see how their assemblers differ from Sexy Ass.

Number systems Sexy Ass assumes all numbers are written in the decimal system, unless preceded by a dollar sign (which means hexadecimal) or percent sign (which means binary).

68000 and 6502 assemblers resemble Sexy Ass, except they don't understand percent signs and binary notation. Some stripped-down 6502 assemblers don't understand the decimal system either: they require all numbers to be in hexadecimal.

The 8088 assembler comes in two versions:

The full version of the 8088 assembler is called the **Microsoft Macro Assembler (Masm)**. It lists for \$150, but discount dealers sell it for just \$83. It assumes all numbers are written in the decimal system, unless followed by an H (which means hexadecimal) or B (which means binary). For example, the number twelve can be written as 12 or as 0CH or as 1100B. It requires each number to begin with a digit: so to say twelve in hexadecimal, instead of saying CH you must say 0CH.

A stripped-down 8088 assembler, called the **Debug mini-assembler**, is part of classic Dos; so you get it at no extra charge when you buy classic Dos. It requires all numbers to be written in hexadecimal. For example, it requires the number twelve to be written as C. Do *not* put a dollar sign or H next to the C.

Accumulator Each microprocessor contains *several* accumulators, so you must say *which* accumulator to use. The main 8-bit accumulator is called "A" in the 6502, "AL" in the 8088, and "D0.B" in the 68000.

Labels Sexy Ass and the other full assemblers let you begin a line with a label, such as FRED. For the 8088 full assembler (Masm), add a colon after FRED. Mini-assemblers (such as 8088 Debug) don't understand labels.

Commands Here's how to translate from Sexy Ass to the popular assemblers:

Computer's action	Sexy Ass	6502	68000	8088 Masm
put 25 in accumulator	LOAD #\$25	LDA #\$25	MOVE.B #\$25,D0	MOV AL,25H
copy location 7000 to accumulator	LOAD \$7000	LDA \$7000	MOVE.B \$7000,D0	MOV AL,[7000H]
copy accumulator to location 7000	STORE \$7000	STA \$7000	MOVE.B D0,\$7000	MOV [7000H],AL
add location 7000 to accumulator	ADD \$7000	ADC \$7000	ADD.B \$7000,D0	ADD AL,[7000H]
subtract location 7000 from acc.	SUB \$7000	SBC \$7000	SUB.B \$7000,D0	SUB AL,[7000H]
increment accumulator	INC	ADC #\$1	ADDQ.B #1,D0	INC AL
decrement accumulator	DEC	SBC #\$1	SUBQ.B #1,D0	DEC AL
put character C in accumulator	LOAD #'C'	LDA #'C'	MOVE.B #'C',D0	MOV AL,"C"
jump to FRED	JUMP FRED	JMP FRED	JMP FRED	JMP FRED
jump, if negative, to FRED	JUMPN FRED	BMI FRED	BMI FRED	JS FRED
jump, if carry, to FRED	JUMPC FRED	BCS FRED	BCS FRED	JC FRED
jump, if zero, to FRED	JUMPZ FRED	BEQ FRED	BEQ FRED	JZ FRED
jump, if neg. lowered, to FRED	JUMPNL FRED	BPL FRED	BPL FRED	JNS FRED
jump, if carry lowered, to FRED	JUMPCL FRED	BCC FRED	BCC FRED	JNC FRED
jump, if zero lowered, to FRED	JUMPSL FRED	BNE FRED	BNE FRED	JNZ FRED

Notice that in 6502 assembler, each mnemonic (such as LDA) is 3 characters long.

To refer to an Ascii character, Sexy Ass and 8088 Masm put the character in quotes, like this: "C". 68000 assembler uses apostrophes instead, like this: 'C'. 6502 assembler uses just a single apostrophe, like this: 'C'.

Instead of saying "jump if", 6502 and 68000 programmers say "branch if" and use mnemonics that start with B instead of J. For example, they use mnemonics such as BMI (which means "Branch if MInus"), BCS ("Branch if Carry Set"), and BEQ ("Branch if EQual to zero").

To make the 68000 manipulate a byte, put ".B" after the mnemonic. (If you say ".W" instead, the computer will manipulate a 16-bit word instead of a byte. If you say ".L" instead, the computer will manipulate long data containing 32 bits. If you don't specify ".B" or ".W" or ".L", the assembler assumes you mean ".W".)

8088 assemblers require you to put each memory location in brackets. So whenever you refer to location 7000 hexadecimal, you put the 7000H in brackets, like this: [7000H].

Inside the CPU

Let's peek inside the CPU and see what lurks within!

Program counter

Each CPU contains a special register called the **program counter**.

The program counter tells the CPU which line of your program to do next. For example, if the program counter contains the number 6 (written in binary), the CPU will do the line of your program that's stored in the 6th memory location.

More precisely, here's what happens if the program counter contains the number 6:

A. The CPU moves the content of the 6th memory location to the CPU's **instruction register**. (That's called **fetching** the instruction.)

B. The CPU checks whether the instruction register contains a complete instruction written in machine language. If not — if the instruction register contains just *part* of a machine-language instruction — the CPU fetches the content of the 7th memory location also. (The instruction register is large enough to hold the content of memory locations 6 and 7 simultaneously.) If the instruction register still doesn't contain a complete instruction, the CPU fetches the content of the 8th memory location also. If the instruction register still doesn't contain a complete instruction, the CPU fetches the content of the 9th memory location also.

C. The CPU changes the number in the program counter. For example, if the CPU has fetched from the 6th and 7th memory locations, it makes the number in the program counter be 8; if the CPU has fetched from the 6th, 7th, and 8th memory locations, it makes the number in the program counter be 9. (That's called **updating the program counter**.)

D. The CPU figures out what the instruction means. (That's called **decoding** the instruction.)

E. The CPU obeys the instruction. (That's called **executing** the instruction.) If it's a "GO TO" type of instruction, the CPU makes the program counter contain the address of the memory location you want to go to.

After the CPU completes steps A, B, C, D, and E, it looks at the program counter and moves on to the next instruction. For example, if the program counter contains the number 9 now, the CPU does steps A, B, C, D, and E again, but by fetching, decoding, and executing the 9th memory location instead of the 6th.

The CPU repeats steps A, B, C, D, and E again and again; each time, the number in the program counter changes. Those five steps form a loop, called the **instruction cycle**.

Arithmetic/Logic unit

The CPU contains two parts: the **control unit** (which is the boss) and the **arithmetic/logic unit (ALU)**. When the control unit comes to step D of the instruction cycle, and decides some arithmetic or logic needs to be done, it sends the problem to the ALU, which sends back the answer.

Here's what the ALU can do:

Operation's name	Example	Explanation
plus, added to, +	$\begin{array}{r} 10001010 \\ +10001001 \\ \hline 100010011 \end{array}$	add, but remember that 1+1 is 10 in binary
minus, subtract, -	$\begin{array}{r} 10001010 \\ -10001001 \\ \hline 00000001 \end{array}$	subtract, but remember that 10-1 is 1 in binary
negative, -, the two's complement of	$\begin{array}{r} -10001010 \\ \hline 01110110 \end{array}$	<i>left of the rightmost 1</i> , do this: replace each 0 by 1, and each 1 by 0
not, ~, the complement of, the one's complement of	$\begin{array}{r} \sim 10001010 \\ \hline 01110101 \end{array}$	replace each 0 by 1, and each 1 by 0
and, &, ^	$\begin{array}{r} 10001010 \\ \wedge 10001001 \\ \hline 10001000 \end{array}$	put 1 wherever both original numbers had 1
or, inclusive or, v	$\begin{array}{r} 10001010 \\ \vee 10001001 \\ \hline 10001011 \end{array}$	put 1 wherever some original number had 1
eXclusive OR, XOR, v	$\begin{array}{r} 10001010 \\ \nabla 10001001 \\ \hline 00000011 \end{array}$	put 1 wherever the original numbers differ

Also, the ALU can shift a register's bits. For example, suppose a register contains 10111001. The ALU can shift the bits toward the right:

before	10111001
after	01011100

It can shift the bits toward the left:

before	10111001
after	01110010

It can rotate the bits toward the right:

before	10111001
after	11011100

It can rotate the bits toward the left:

before	10111001
after	01110011

It can shift the bits toward the right **arithmetically**:

before	10111001
after	11011100

It can shift the bits toward the left arithmetically:

before	10111001
after	11110010

Doubling a number is the same as shifting it left arithmetically. For example, doubling six (to get twelve) is the same as shifting six left arithmetically:

six	00000110
twelve	00001100

Halving a number is the same as shifting it right arithmetically. For example, halving six (to get three) is the same as shifting six right arithmetically:

six	00000110
three	00000011

Halving negative six (to get negative three) is the same as shifting negative six right arithmetically:

negative six	11111010
negative three	11111101

Using the ALU, the control unit can do operations such as:

- Find the number in the 6th memory location, and move its negative to a register.
- Change the number in a register, by adding to it the number in the 6th memory location.
- Change the number in a register, by subtracting from it the number in the 6th memory location.

Most computers require each operation to have one source and one destination. In operations A, B, and C, the source is the 6th memory location; the destination is the register.

The control unit *cannot* do a command such as "add together the number in the 6th memory location and the number in the 7th memory location, and put the sum in a register", because that operation would require two sources. Instead, you must give two shorter commands:

- Move the number in the 6th memory location to the register.
- Then add to that register the number in the 7th memory location.

Flags

The CPU contains a **flag register**, which comments on what the CPU is doing. In a typical CPU, the flag register has 6 bits, named as follows:

- the Negative bit
- the Zero bit
- the Carry bit
- the Overflow bit
- the Priority bit
- the Privilege bit

When the CPU performs an operation (such as addition, subtraction, shifting, rotating, or moving), the operation has a source and a destination. The number that goes into the destination is the operation's **result**. The CPU automatically analyzes that result.

Negative bit If the result is a negative number, the CPU turns on the **Negative bit**. In other words, it makes the Negative bit be 1. (If the result is a number that's *not* negative, the CPU makes the Negative bit be 0.)

Zero bit If the result is zero, the CPU turns on the **Zero bit**. In other words, it makes the Zero bit be 1.

Carry bit When the ALU computes the result, it also computes an extra bit, which becomes the **Carry bit**.

For example, here's how the ALU adds 7 and -4:

7 is	00000111
-4 is	11111100
binary addition gives	10000011

So the result is 3, and the Carry bit becomes 1.

Overflow bit If the ALU can't compute a result correctly, it turns on the **Overflow bit**.

For example, in elementary school you learned that 98+33 is 131; so in binary, the computation should look like this:

	128	64	32	16	8	4	2	1
98 is		1	1	0	0	0	1	0
33 is			1	0	0	0	0	1
the sum is	1	0	0	0	0	0	1	1, which is 131

But here's what an 8-bit ALU will do:

	sign	64	32	16	8	4	2	1
98 is	0	1	1	0	0	0	1	0
33 is		0	0	1	0	0	0	1
the sum is	0	1	0	0	0	0	1	1

↑ **Carry**
↓ **result**

Unfortunately, the result's leftmost 1 is in the position marked **sign**, instead of the position marked 128; so the result looks like a negative number.

To warn you that the result is incorrect, the ALU turns on the **Overflow bit**. If you're programming in a language such as Basic, the interpreter or compiler keeps checking whether the **Overflow bit** is on; when it finds that the bit's on, it prints the word **OVERFLOW**.

Priority bit While your program's running, it might be interrupted. Peripherals might interrupt, in order to input or output the data; the **real-time clock** might interrupt, to prevent you from hogging too much time, and to give another program a chance to run; and the computer's sensors might interrupt, when they sense that the computer is malfunctioning.

When something wants to interrupt your program, the CPU checks whether your program has priority, by checking the **Priority bit**. If the **Priority bit** is on, your program has priority and cannot be interrupted.

Privilege bit On a computer that's handling several programs at the same time, some operations are dangerous: if your program makes the computer do those operations, the other programs might be destroyed. Dangerous operations are called **privileged instructions**; to use them, you must be a **privileged user**.

When you walk up to a terminal attached to a large computer, and type HELLO or LOGIN, and type your user number, the operating system examines your user number to find out whether you are a privileged user. If you are, the operating system turns on the **Privilege bit**. When the CPU starts running your programs, **it refuses to do privileged instructions unless the Privilege bit is on**.

Microcomputers omit the **Privilege bit** and can't prevent you from giving dangerous commands. But since the typical microcomputer has just one terminal, the only person your dangerous command can hurt is yourself.

Levels of priority & privilege Some computers have *several* levels of priority and privilege.

If your priority level is "moderately high", your program is immune from most interruptions, but not from all of them. If your privilege level is "moderately high", you can order the CPU to do most of the privileged instructions, but not all of them.

To allow those fine distinctions, large computers devote *several* bits to explaining the priority level, and *several* bits to explaining the privilege level.

Where are the flags? The bits in the flag register are called the **flags**. To emphasize that the flags comment on your program's status, people sometimes call them **status flags**.

In the CPU, the program counter is next to the flag register. Instead of viewing them as separate registers, some programmers

consider them to be parts of a single big register, called the **program status word**.

Tests You can give a command such as, "Test the 3rd memory location". The CPU will examine the number in the 3rd memory location. If that number is negative, the CPU will turn on the **Negative bit**; if that number is zero, the CPU will turn on the **Zero bit**.

You can give a command such as, "Test the difference between the number in the 3rd register and the number in the 4th. The CPU will adjust the flags according to whether the difference is negative or zero or carries or overflows.

Saying "if" The CPU uses the flags when you give a command such as, "If the **Negative bit** is on, go do the instruction in memory location 6".

Speed

Computers are fast. To describe computer speeds, programmers use these words:

Word	Abbreviation	Meaning
millisecond	msec or ms	thousandth of a second; 10 ⁻³ seconds
microsecond	μsec or μs	millionth of a second; 10 ⁻⁶ seconds
nanosecond	nsec or ns	billionth of a second; 10 ⁻⁹ seconds
picosecond	psec or ps	trillionth of a second; 10 ⁻¹² seconds

1000 picoseconds is a nanosecond; 1000 nanoseconds is a microsecond; 1000 microseconds is a millisecond; 1000 milliseconds is a second.

On page 671 I explained that the **instruction cycle** has five steps:

- | |
|--|
| <ul style="list-style-type: none"> A. Fetch the instruction. B. Fetch additional parts for the instruction. C. Update the program counter. D. Decode the instruction. E. Execute the instruction. |
|--|

To do that entire instruction cycle, an old-fashioned computer takes about a microsecond; a modern computer takes about a nanosecond. The exact time depends on the quality of the CPU, the quality of the main memory, and the difficulty of the instruction.

Here are 5 ways to make a computer act faster:

Method	Meaning
multiprocessing	The computer holds more than one CPU. (All the CPUs work simultaneously. They share the same main memory. The operating system decides which CPU works on which program. The collection of CPUs is called a multiprocessor .)
instruction lookahead	While the CPU is finishing an instruction cycle (by doing steps D and E), it simultaneously begins working on the next instruction cycle (steps A and B).
array processing	The CPU holds at least 16 ALUs. (All the ALUs work simultaneously. For example, when the control unit wants to solve 16 multiplication problems, it sends each problem to a separate ALU; the ALUs compute the products simultaneously. The collection of ALUs is called an array processor .)
parallel functional units	The ALU is divided into several functional units: an addition unit, a multiplication unit, a division unit, a shift unit, etc. All the units work simultaneously; while one unit is working on one problem, another unit is working on another.
pipeline architecture	The ALU (or each ALU functional unit) consists of a "first stage" and a "second stage". When the control unit sends a problem to the ALU, the problem enters the first stage, then leaves the first stage and enters the second stage. But while the problem is going through the second stage, a new problem starts going through the first stage. (Such an ALU is called a pipeline processor .)

Parity

Most large computers put an extra bit at the end of each memory location. For example, a memory location in the PDP-10 holds 36 bits, but the PDP-10 puts an extra bit at the end, making 37 bits altogether. The extra bit is called the **parity bit**.

If the number of ones in the memory location is even, the CPU turns the parity bit on. If the number of ones in the memory location is odd, the CPU turns the parity bit off.

For example, if the memory location contains these 36 bits —

```
000000000100010000000110000000000000
```

there are 4 ones, so the number of ones is even, so the CPU turns the parity bit on:

```
0000000001000100000001100000000000001
      (content)                               (parity)
```

If the memory location contains these 36 bits instead —

```
000000000100010000000100000000000000
```

there are 3 ones, so the number of ones is odd, so the CPU turns the parity bit off:

```
0000000001000100000001000000000000000
      (content)                               (parity)
```

Whenever the CPU puts data into the main memory, it also puts in the parity bit. Whenever the CPU grabs data from the main memory, it checks whether the parity bit still matches the content.

If the parity bit doesn't match, the CPU knows there was an error, and tries once again to grab the content and the parity bit. If the parity bit disagrees with the content again, the CPU decides that the memory is broken, refuses to run your program, prints a message saying PARITY ERROR, and then sweeps through the whole memory, checking the parity bit of every location; if the CPU finds another parity error (in your program or anyone else's), the CPU shuts off the whole computer.

Cheap microcomputers (such as the Apple 2c and Commodore 64) lack parity bits, but the IBM PC has them.

UAL

Universal Assembly Language (UAL) is a notation I invented that makes programming in assembly language easier.

UAL uses these symbols:

Symbol	Meaning
M5	the number in the 5 th memory location
R2	the number in the 2 nd register
P	the number in the program counter
N	the Negative bit
Z	the Zero bit
C	the Carry bit
V	the oVerflow bit
PRIORITY	the PRIORITY bits
PRIVILEGE	the PRIVILEGE bits
F	the content of the entire flag register
F[5]	the 5 th bit in the flag register
R2[5]	the 5 th bit in R2
R2[LEFT]	the left half of R2; in other words, the left half of the data in the 2 nd register
R2[RIGHT]	the right half of R2
M5 M6	long number whose left half is in 5 th memory location, right half is in 6 th location

Here are the UAL statements:

Statement	Meaning
R2=7	Let number in the 2 nd register be 7 (by moving 7 into the 2 nd register).
R2=M5	Copy the 5 th memory location's contents into the 2 nd register.
R2= M5	Exchange R2 with M5. (Put 5 th location's content into 2 nd register and vice versa.)
R2=R2+M5	Change the integer in 2 nd register, by adding to it the integer in 5 th location.
R2=R2-M5	Change the integer in 2 nd register, by subtracting the integer in 5 th location.
R2=R2*M5	Change the integer in 2 nd register, by multiplying it by integer in 5 th location.
R2 REM R3=R2/M5	Change R2, by dividing it by the integer M5. Put division's remainder into R3.
R2=-M5	Let R2 be the negative of M5.
R2=NOT M5	Let R2 be the one's complement of M5.
R2=R2 AND M5	Change R2, by performing the AND operation.
R2=R2 OR M5	Change R2, by performing the OR operation.
R2=R2 XOR M5	Change R2, by performing the XOR operation.
SHIFTL R2	Shift left.
SHIFTR R2	Shift right.
SHIFTRA R2	Shift right arithmetically.
SHIFTR3 R2	Shift right, 3 times.
SHIFTR (R7) R2	Shift right, R7 times.
ROTATEL R2	Rotate left.
ROTATER R2	Rotate right.
TEST R2	Examine number in 2 nd register, and adjust flag register's Negative and Zero bits.
TEST R2-R4	Examine the difference between R2 and R4, and adjust the flag register.
CONTINUE	No operation. Just continue on to the next instruction.
WAIT	Wait until an interrupt occurs.
IF R2<0, P=7	If the number in the 2 nd register is negative, put 7 into the program counter.
IF R2<0, M5=3, P=7	If R2<0, do both of the following: let M5 be 3, and P be 7.

M5 can be written as M(5) or M(2+3). It can be written as M(R7), if R7 is 5 — in other words, if register 7 contains 5.

Addressing modes

Suppose you want the 2nd register to contain the number 6. You can accomplish that goal in one step, like this:

```
R2=6
```

Or you can accomplish it in two steps, like this:

```
M5=6  
R2=M5
```

Or you can accomplish it in three steps, like this:

```
M5=6  
M3=5  
R2=M(M3)
```

Or you can accomplish it in an even weirder way:

```
M5=6  
R3=1  
R2=M(4+R3)
```

Each of those methods has a name. The first method ($R2=6$), which is the simplest, is called **immediate addressing**. The second method ($R2=M5$), which contains the letter M, is called **direct addressing**. The third method ($R5=M(M3)$), which contains the letter M twice, is called **indirect addressing**. The fourth method ($R5=M(4+R3)$), which contains the letter M and a plus sign, is called **indexed addressing**.

In each method, the 2nd register is the destination. In the last three methods, the 5th memory location is the source. In the fourth method, which involves R3, the 3rd register is called the **index register**, and R3 itself is called the **index**.

Each of those methods is called an **addressing mode**. So you've seen four addressing modes: immediate, direct, indirect, and indexed.

Program counter To handle the program counter, the computer uses other addressing modes instead.

For example, suppose P (the number in the program counter) is 2073, and you want to change it to 2077. You can accomplish that goal simply, like this:

```
P=2077
```

Or you can accomplish it in a weirder way, like this:

```
P=P+4
```

Or you can accomplish it in an even weirder way, like this:

```
R3=20  
P=R3 77
```

The first method ($P=2077$), which is the simplest, is called **absolute addressing**.

The second method ($P=P+4$), which involves addition, is called **relative addressing**. The "+4" is the **offset**.

The third method ($P=R3 77$) is called **base-page addressing**. R3 (which is 20) is called the **page number** or **segment number**, and so the 3rd register is called the **page register** or **segment register**.

Intel's details

The first **microprocessor** (CPU on a chip) was invented by Intel in 1971 and called the **Intel 4004**. Its accumulator was so short that it held just 4 bits! Later that year, Intel invented the **Intel 8008**, whose accumulator held 8 bits. In 1973 Intel invented the **Intel 8080**, which understood more op codes, contained more registers, could handle more RAM (64K instead of 16K), and ran faster. Drunk on the glories of that 8080, Microsoft adopted the phone number VAT-8080, and the Boston Computer Society adopted the soberer phone number DOS-8080.

In 1978 Intel invented the **8086**, which had a 16-bit accumulator and handled even more RAM & ROM (totalling 1 megabyte). Out of the 8086 came 16 wires (called the **data bus**), which transmitted 16 bits simultaneously from the accumulator to other computerized devices, such as RAM and disks. Since the 8086 had a 16-bit accumulator and 16-bit data bus, Intel called it a **16-bit CPU**.

But computerists complained that the 8086 was impractical, since nobody had developed RAM, disks, or other devices for the 16-bit data bus yet. So in 1979 Intel invented the **8088**, which understands the same machine language as the 8086 but has an 8-bit data bus. To transmit 16-bit data through the 8-bit bus, the 8088 sends 8 of the bits first, then sends the other 8 bits shortly afterwards. That technique of using a few wires (8) to imitate many (16) is called **multiplexing**.

When 16-bit data buses later became popular, Intel invented a slightly souped-up 8086, called the **80286** (nicknamed the **286**).

Then Intel invented a 32-bit version called the **80386** (nicknamed **386**). Intel also invented a multiplexed version called the **386SX**, which understands the same machine language as the 386 but transmits 32-bit data through a 16-bit bus (by sending 16 of the bits first, then sending the other 16). The letters "SX" mean "SiXteen-bit bus". The original 386, which has a 32-bit bus, is called the **386DX**; the letters "DX" mean "Double the siXteen-bit bus".

Then Intel invented a slightly souped-up 386DX, called the **486**. It comes in two versions: the fancy version (called the **486DX**) includes a **math coprocessor**, which is circuitry that understands commands about advanced math; the stripped-down version (called the **486SX**) lacks a math coprocessor.

Finally, Intel invented a souped-up 486DX, called a **Pentium**.

Here's how to use the 8088 and 8086. (The 286, 386, 486, and Pentium include the same features plus more.)

Registers

The CPU contains fourteen 16-bit registers:

```
accumulator (AX), base register (BX), count register (CX), data register (DX)  
flag register (which UAL calls F)  
program counter (which UAL calls P but Intel calls "instruction pointer" or IP)  
stack pointer (which UAL calls S but Intel calls SP), base pointer (BP)  
source index (SI), destination index (DI)  
code segment (CS), data segment (DS), stack segment (SS), extra segment (ES)
```

In each of those registers, the sixteen bits are numbered from right to left, so the rightmost bit is called **bit 0** and the leftmost bit is called **bit fifteen**.

The AX register's low-numbered half (bits 0 through 7) is called **A low** (or **AL**). The AX register's high half (bits 8 through fifteen) is called **A high** (**AH**).

In the flag register, bit 0 is the carry flag (which UAL calls **C**), bit 2 is for parity, bit 6 is the zero flag (**Z**), bit 7 is the negative flag (which UAL calls **N** but Intel calls **sign** or **S**), bit eleven is the overflow flag (**V**), bits 4, 8, 9, and ten are special (**auxiliary carry**, **trap**, **interrupts**, and **direction**), and the remaining bits are unused.

Memory locations

Each memory location contains a byte. In UAL, the 6th memory location is called **M6** or **M(6)**. The pair of bytes M7 M6 is called **memory word 6**, which UAL writes as **MW(6)**.

Instruction set

This page shows the set of instructions that the 8088 understands. For each instruction, I've given the assembly-language mnemonic and its translation to UAL, where all numbers are hexadecimal.

The first line says that INC (which stands for INCRement) is the assembly-language mnemonic that means $x=x+1$. For example, INC AL means $AL=AL+1$.

The eighth line says that IMUL (which stands for Integer Multiply) is the assembly-language mnemonic that means $x=x*y$. For example, IMUL AX,BX means $AX=AX*BX$.

In most equations, you can replace the x and y by registers, half-registers, memory locations, numbers, or more exotic entities. To find out what you can replace x and y by, experiment!

For more details, read the manuals from Intel and Microsoft. They also explain how to modify an instruction's behavior by using flags, segment registers, other registers, and three **prefixes**: REPEAT, SEGMENT, and LOCK.

Math	
INCRement	$x=x+1$
DECRement	$x=x-1$
ADD	$x=x+y$
ADD Carry	$x=x+y+C$
SUBtract	$x=x-y$
SuBtract Borrow	$x=x-y-C$
MULTiPLY	$x=x*y$ UNSIGNED
Integer MULTiPLY	$x=x*y$
DIVide	$AX=AX/x$ UNSIGNED
Integer DIVide	$AX=AX/x$
NEGate	$x=-x$
Decimal Adjust Add	IF $AL[RIGHT]>9$, $AL=AL+6$ IF $AL[LEFT]>9$, $AL=AL+60$
Decimal Adjust Subtr	IF $AL[RIGHT]>9$, $AL=AL-6$ IF $AL[LEFT]>9$, $AL=AL-60$
Ascii Adjust Add	IF $AL[RIGHT]>9$, $AL=AL+6$, $AH=AH+1$ $AL[LEFT]=0$
Ascii Adjust Subtract	IF $AL[RIGHT]>9$, $AL=AL-6$, $AH=AH-1$ $AL[LEFT]=0$
Ascii Adjust Multiply	$AH\ REM\ AL=AL/OA$
Ascii Adjust Divide	$AL=AL+(OA*AH)$ $AH=0$
Logic	
AND	$x=x\ AND\ y$
OR	$x=x\ OR\ y$
XOR	$x=x\ XOR\ y$
CoMplement Carry	$C=NOT\ C$
SHift Left	$SHIFTL(y)\ x$
SHift Right	$SHIFTR(y)\ x$
Shift Arithmetic Right	$SHIFTRA(y)\ x$
ROtate Left	$ROTATEL(y)\ x$
ROtate Right	$ROTATER(y)\ x$
Rotate Carry Left	$ROTATEL(y)\ C\ x$
Rotate Carry Right	$ROTATER(y)\ C\ x$
CLear Carry	$C=0$
CLear Direction	$DIRECTION=0$
CLear Interrupts	$INTERRUPTS=0$
SeT Carry	$C=1$
SeT Direction	$DIRECTION=1$

SeT Interrupts	$INTERRUPTS=1$
TEST	$TEST\ x\ AND\ y$
CoMPare	$TEST\ x-y$
SCAn String Byte	$TEST\ AL-M(DI); DI=DI+1-(2*DIRECTION)$
SCAn String Word	$TEST\ AX-MW(DI); DI=DI+2-(4*DIRECTION)$
CoMPare String Byte	$TEST\ M(SI)-M(DI)$ $SI=SI+1-(2*DIRECTION)$ $DI=DI+1-(2*DIRECTION)$
CoMPare String Word	$TEST\ MW(SI)-MW(DI)$ $SI=SI+2-(4*DIRECTION)$ $DI=DI+2-(4*DIRECTION)$
Moving bytes	
MOVE	$x=y$
Load AH from F	$AH=F[RIGHT]$
Store AH to F	$F[RIGHT]=AH$
Load register and DS	$x=MW(y); DS=MW(y+2)$
Load register and ES	$x=MW(y); ES=MW(y+2)$
LOaD String Byte	$AL=M(SI); SI=SI+1-(2*DIRECTION)$
LOaD String Word	$AX=MW(SI); SI=SI+2-(4*DIRECTION)$
STOre String Byte	$M(DI)=AL; DI=DI+1-(2*DIRECTION)$
STOre String Word	$MW(DI)=AX; DI=DI+2-(4*DIRECTION)$
MOVE String Byte	$M(DI)=M(SI);$ $DI=DI+1-(2*DIRECTION)$ $SI=SI+1-(2*DIRECTION)$
MOVE String Word	$MW(DI)=MW(SI)$ $DI=DI+2-(4*DIRECTION)$ $SI=SI+2-(4*DIRECTION)$
Convert Byte to Word	$AH=-AL[7]$
Convert Word to Dbl	$DX=-AX[0F]$
PUSH	$S=S-2; MW(S)=x$
PUSH F	$S=S-2; MW(S)=F$
POP	$x=MW(S); S=S+2$
POP F	$F=MW(S); S=S+2$
IN	$x=PORT(y)$
OUT	$PORT(x)=y$
ESCape	$BUS=x$
eXCHanGe	$x=y$
XLATe	$AL=M(BX+AL)$
Load Effective Address	$x=ADDRESS(y)$
Program counter	
JuMP	$P=x$
Jump if Zero	IF $Z=1$, $P=x$
Jump if Not Zero	IF $Z=0$, $P=x$
Jump if Sign	IF $N=1$, $P=x$
Jump if No Sign	IF $N=0$, $P=x$
Jump if Overflow	IF $V=1$, $P=x$
Jump if Not Overflow	IF $V=0$, $P=x$
Jump if Parity	IF $PARITY=1$, $P=x$
Jump if No Parity	IF $PARITY=0$, $P=x$
Jump if Below	IF $C=1$, $P=x$
Jump if Above or Eq	IF $C=0$, $P=x$
Jump if Below or Eq	IF $C=1\ OR\ Z=1$, $P=x$
Jump if Above	IF $C=0\ AND\ Z=0$, $P=x$
Jump if Greater or Eq	IF $N=V$, $P=x$
Jump if Less	IF $N<V$, $P=x$
Jump if Greater	IF $N=V\ AND\ Z=0$, $P=x$
Jump if Less or Equal	IF $N<V\ OR\ Z=1$, $P=x$
Jump if CX Zero	IF $CX=0$, $P=x$
LOOP	$CX=CX-1; IF\ CX<>0$, $P=x$
LOOP if Zero	$CX=CX-1; IF\ CX<>0\ AND\ Z=1$, $P=x$
LOOP if Not Zero	$CX=CX-1; IF\ CX<>0\ AND\ Z=0$, $P=x$
CALL	$S=S-2; MW(S)=P; P=x$
RETurn	$P=MW(S); S=S+2$
INTerrupt	$S=S-6; MW(S)=P; MW(S+2)=CS;$ $MW(S+4)=F; P=MW(4*x); CS=MW(4*x+2)$ $INTERRUPTS=0; TRAP=0$
INTerrupt if Overflow	IF $V=1$, $S=S-6$, $MW(S)=P$, $MW(S+2)=CS$, $MW(S+4)=F$, $P=MW(10)$, $CS=MW(12)$, $INTERRUPTS=0$, $TRAP=0$
Interrupt RETurn	$P=MW(S); CS=MW(S+2); F=MW(S+4); S=S+6$
No Operation	CONTINUE
HaLT	WAIT
WAIT	WAIT FOR COPROCESSOR