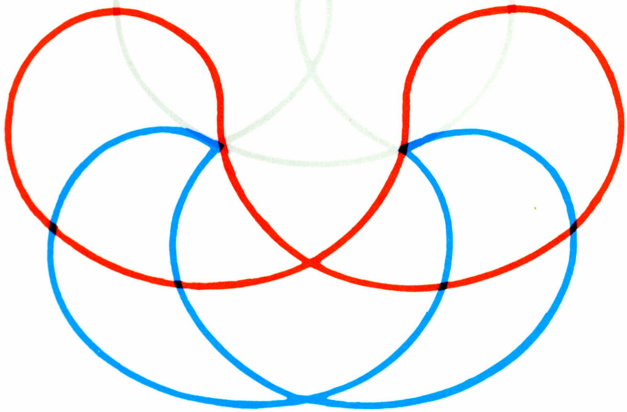
# NIMBUS

## BEGINNING RM LOGO

THE HOMERTON LOGO MANUAL



RM

**Discrepancies in the Beginning RM Logo Manual**

Page 1.6

In the table of colours, the colour for number 0 should be `black`, not white.

Page 7.1

The third line of the procedure ray should be `backward 50`.

Page 10.4

If you type: `add 3 20 50 30`, the result `100` will be printed on the screen because it is not wanted by another procedure.

Page 10.5

The final line on the page should be:

```
repeat (add 3 20 50 30)[square 50 right 40]
```

Page 13.3

`first` returns the first digit of a number. rest returns all but the first digit of the integer part of a number.

Page 17.2

The third line of the procedure in project 3 should be:

```
right :angle
```

Page 18.3

The final line of the procedure randwd should be:

```
result :listn # pick count :listn
```

# BEGINNING RM LOGO
## THE HOMERTON LOGO MANUAL

PN14393

Beginning RM Logo
The Homerton Logo Manual
PN 14393

Hilary Shuard is Deputy Principal of Homerton College.

Fred Daly is Director of Computing and Curriculum Development at Homerton College.

Homerton College is the College of Education in the University of Cambridge.

The Homerton Logo Manual was first written in 1983 as part of the Homerton Logo Project. It has been used with a range of in-service and pre-service courses with teachers who have little or no experience of computers. Through these courses and with the helpful advice of others, the manual has been successively refined and augmented to take account of the developments in technology.



Customers comments are of great value to us in improving our computer systems, publications and services. If you would like to make any comments, please use the reply-paid form at the back of the Handbook.

Edited by Nicola Bourdillon and Cathy M. Hand.

Illustrations by Jane A. Hannah and Inkwell Studios, Bicester, Oxon.

Typeset by direct transfer from Research Machines Network to Linotron 202 at Oxford Publishing Services, Oxford.

Printed by The Hazell Press, Wembley.

Research Machines Limited, Mill Street, Oxford OX2 0BW.
Tel: Oxford (0865) 249866.

# Introduction

This book is intended to show beginners how to use Logo in a satisfying way. It is structured to build your confidence that you understand the major features of Logo.

When Seymour Papert developed Logo, he wanted to provide a rich environment for children to explore, both giving them a knowledge of computer programming, and fascinating the beginner.

Some readers of this book will be teachers who are working through Logo, preparing to use it with children. We do not suggest that you introduce children to Logo in the structured way used in this book. We recommend a much more exploratory approach, encouraging children to set and solve their own problems, introducing new features of Logo from time to time. Use this book to help children find easier ways of tackling their projects.

Other readers will be learning Logo for fun, or for a satisfying introduction to the power of computer programming.

Whoever you are, we hope you will have fun as you explore Logo.

An RM Logo Reference book is available from Research Machines, PN 14394. It gives an introduction to using RM Logo on Nimbus and a reference section of Logo primitives listed alphabetically; special Logo characters; Logo keywords and signals. A quick reference card is also included.

# Contents

# Chapter 1
# Using Logo for Drawing Pictures

## Getting Started

Your Nimbus is either a Standalone or Network Nimbus. If you are unsure which your Nimbus is, look at the back. A Network Nimbus has a cable attached to the socket marked 'Network'.

### On a Network Nimbus

We assume everything has been prepared by the Network Manager for you to use RM Logo, and you can join the steps for a Standalone Nimbus where they specify you type:

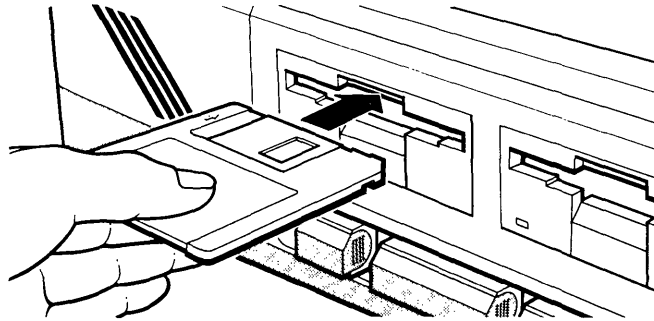
```
logo <ENTER>
```

If you have a copy of the RM Logo Disk, you can treat your Network Nimbus like a Standalone and follow the steps below.

### On a Standalone Nimbus

Switch on the computer and the monitor. The screen will say `Welcome` and then tell you it is looking for an operating system.

The operating system it is looking for, together with the Logo language, is on the RM Logo Disk, so place this in the disk drive (the left-hand drive if you have two drives). One corner of the disk has an arrow on it, to show which way the disk goes in. The arrow should be *top left* as you put the disk in, and there will be a satisfying firm click as the disk fits snugly in to the drive.



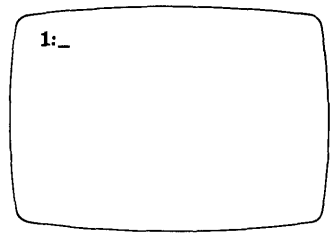
After a few moments, Nimbus will ask you for the date. To get into Logo quickly, ignore this by simply pressing the <ENTER> key. It is towards the right hand side of the keyboard.

In this book, the sign <ENTER> means pressing the <ENTER> key.

Next, ignore Nimbus's request to tell it the time by pressing <ENTER> again; you are then ready to load Logo. To do this, merely type:

```
Logo <ENTER>
```

The red light on the disk drive shows that Logo is being copied from the disk into the computer. When Logo has finished loading, you see the following screen and can start using Logo.



## Some Primitive Commands

The Logo language can program Nimbus to do many tasks. Young children start by using Logo to make the computer draw pictures. Adults can use Logo to program Nimbus to carry out complex tasks, just as they might use another computer language such as Basic or Pascal.

When you enter Logo, Nimbus already understands a few words of English. You can use these words to tell the computer how to draw pictures. Type:

`cs` <ENTER>

The command `cs` stands for `C`lear `S`creen.

*Note*:

- In this book, Logo commands are written in a `different type.`
- As you can type more in a line on the screen than can be written in a line in this book, any lines indented in this book are a continuation of the line above.
- Nimbus writes them on the screen in lower-case letters, and you should also use lower-case letters.

You should now see an arrow in the middle of the screen pointing upwards, and a rectangular border round the edge of the drawing screen. The arrow is called the *screen turtle* (turtle for short). Below the drawing screen is the *prompt*:

```
1:
```

which shows you are talking to *turtle number 1*. The prompt always shows when Logo is waiting for you to type something. You are now ready to command turtle number 1 (which carries a pen with it) to draw on the screen.

Nimbus understands the following commands, and can instruct the turtle to obey them:

```
forward
backward
right
left
```

Each of these commands must be followed by a number. Type:

```
forward 50 <ENTER>
```

Remember that the *zero* key is among the number keys on the top row of the keyboard; it prints on the screen as 0. Do not confuse it with the letter O, or you will get an error message.

Experiment with some commands — make the turtle draw some pictures. Each command must end with <ENTER>.

To clean the screen and start again, use `cs`. Other useful commands for starting again are:

```
clean
```

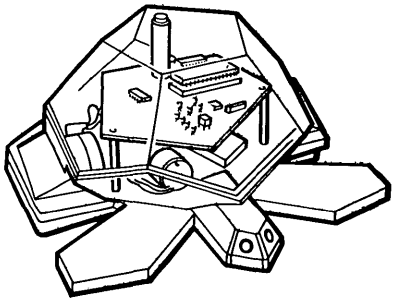
which wipes away the drawing, but leaves the turtle where it is, and:

```
centre
```

which sends the turtle back to its starting point without cleaning away the drawing.

The screen turtle is the little arrow which does the drawing. It is an abstract screen version of the *floor turtle*. The floor turtle is a small computer-controlled robot which Seymour Papert used in his work with children. It carries a fibre-tipped pen, and it understands the Logo language. It draws pictures on paper on the floor, just as the screen turtle draws pictures on the Nimbus screen.

Young children feel more comfortable with the floor turtle, but the floor turtle and the screen turtle can do exactly the same things.



Children as young as 4 or 5 years old can use the floor turtle to make drawings, using only the letter keys <F>, <B>, <R>, <L> to command the turtle. These one-key-press abbreviations are not available on RM Logo when you switch on; but it is possible to program them (see Chapter 12).

## Abbreviated Primitives

It is tiresome, particularly for non-typists, to have to type in the complete command every time, and so most versions of Logo allow abbreviations. In RM Logo some of the abbreviations are:

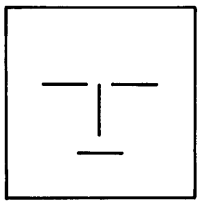
```
fd   forward
bk   backward
rt   right
lt   left
cl   clean
ct   centre
```

Try:

```
fd 60
rt 90
fd 60
```

An alphabetic listing of all Logo primitives and their abbreviations are in the second part of the RM Logo book.

## Raising and Lowering the Pen



Here is a simple drawing of a face.

You would need to lift the pen up from the paper after drawing one eye, so as to move to the other eye. When the turtle is drawing on screen, the commands `lift` and `drop` behave exactly as if the floor turtle were using them on the floor. `lift` lifts the pen, and enables the turtle to move without drawing a line; `drop` lowers the pen.

The turtle has a good range of coloured pens (either eight or sixteen colours depending on your colour monitor). The command `setpc` stands for `set pen colour` and makes the turtle draw with different coloured pens. `setpc` must be followed by a number from the following table:

0 = white
1 = dark blue
2 = dark red
3 = dark purple
4 = dark green
5 = dark sky blue
6 = brown
7 = light grey
8 = dark grey
9 = light blue
10 = light red
11 = light purple
12 = light green
13 = light sky blue
14 = yellow
15 = white

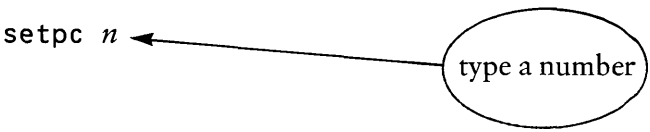
When you first enter Logo, the pen colour is 15.

If you have a black and white monitor, colours come out as various shades of grey. The colours of the turtle and the background can also be changed. Type these commands:

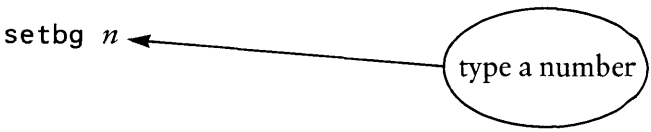
```
setpc 11
setc 2
setbg 1
```

Nothing happens until you type `cs`. Then you get a dark blue screen with a red turtle sitting in the middle of it. If you now type `forward 50` you should get a pinkish purple line. Experiment to find some more tasteful colour combinations.

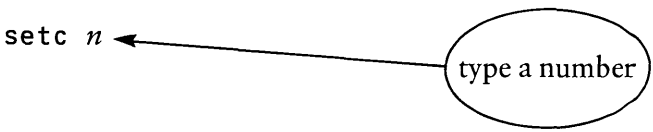
In general, you should use:




to change the colour of the turtle's pen;



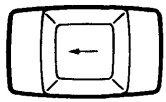

to change the colour of the background. The command:




changes the turtle's colour. `setbg` and `setc` use the same colour numbers as `setpc`, except that `setbg 0` sets the background to *black*, and `setc 0` sets the turtle to the colour of the *background*, making it invisible. The command `setbg` does not operate immediately; you have to follow it by `cs` to make the background colour change.

## Typing Errors

Press the <BACKSPACE> key to rub out the last character you typed.



### Projects

1. Draw some pictures. Enjoy yourself and do not rush this stage; the Logo environment was originally designed to enable beginners to engage in free exploration. Through this exploration they 'learn without teaching'. The possibilities are immense — try some of them.

2. Draw a square, an equilateral triangle, a regular hexagon.

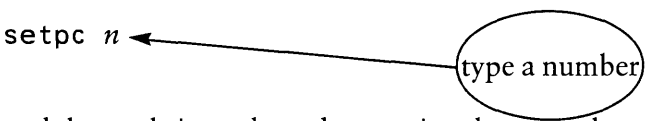
## Rubbing Out

Computer programs, when first made, nearly always contain *bugs*; the program does not do exactly what you intended. If you need to *debug* your drawings, when you have drawn a line in the wrong place, you can rub it out by changing the pen for a rubber. Type:

```
rubber
```

Then you can use the command `backward` to rub out the line you drew wrongly. You will arrive back at the point where you started to draw a line in the wrong place.

To change to a drawing pen again, use a command such as:



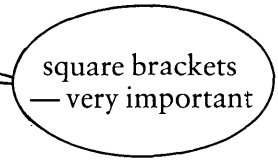

and the turtle is ready to draw, using the pen colour corresponding to the number. For example, if you type `setpc 14` you should get a yellow line.

# Chapter 2
# Text and Numbers

## Printing Titles

Logo can be used to do many other things as well as drawing pictures. You can use it to write on the screen; in this way, you can give your pictures titles. Draw a picture, and then type:

```
label [Drawn by Jane]
```




The words `Drawn by Jane` appear on the screen near where the turtle is. The square brackets `[ ]` are very important: they enclose the *list* of words used by the command `label`. The list of words can be varied at your choice.

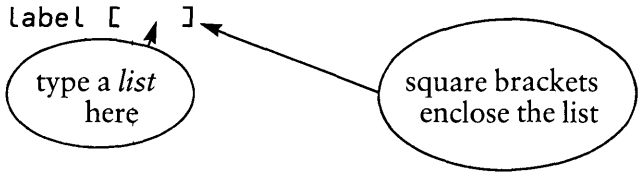
Lists are an important feature of Logo; in the command you have just typed, the expression:

```
[Drawn by Jane]
```

is a list containing the words `Drawn`, `by` and `Jane`. A list is always enclosed in square brackets `[ ]`.

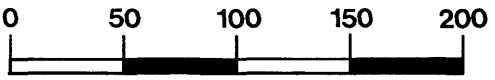
Experiment with using the command `label` to write labels on your pictures; don't forget the square brackets. The command `label` is always followed by a list:

```
label [ ]
```



## Project

1. The turtle takes very small steps as it moves around the screen. Draw a turtle ruler which measures in turtle steps on the screen, like this:




You might use different colours as you draw the ruler.

# Using Logo as a Calculator

Logo can also do arithmetic calculations, so that you can use Nimbus as a calculator. The symbols for addition, subtraction, multiplication and division are very like those used in ordinary arithmetic. The arithmetic symbols are + (add), - (subtract), * (multiply), / (divide).

However, the equals sign is not used in Logo calculations. Instead of writing 3 + 4 =, you write:

```
say 3 + 4 <ENTER>
```

Logo should respond by printing 7. Logo does not even complain if you omit `say` and merely write:

```
3 + 4 <ENTER>
```

Even if you do this, Logo will still obligingly print 7.

You can also do arithmetic within turtle graphics commands. For example, you can use commands such as:

```
forward (30 + 40)  or  backward (110 -40)
```

(The brackets are optional, but they can make your commands easier to read.)

A command such as `10/3` works out the result of the division as a decimal number (correct to 15 significant figures!).

Explore the arithmetic abilities of Logo. If you need more space for calculations on the screen, and do not want to use the turtle for the time being, type:

`textscreen` abbreviation `ts`

This enables the whole screen to be used for calculations. To return to the drawing screen, use `cs`.

# Chapter 3
# Repeats

## Squares

When you drew a square, you probably did something like this:

```
forward 50
right 90
forward 50
right 90
forward 50
right 90
forward 50
right 90
```

Notice these three points:

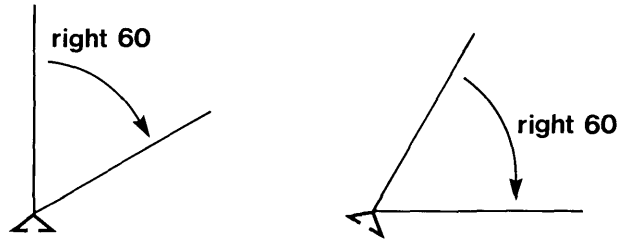
- You may have omitted the last `right 90`.

Its only purpose is to finish the square with the turtle facing the way it started. This is a good idea for more complicated geometric drawings; you should develop the habit of finishing geometric drawings with the turtle in the same place and facing the same way as it started — that way, drawings combine more easily.

- Your square may have turned `left` rather than right, but in either case the *turning number* was 90.

Turning numbers are measured in degrees. Even young children who do not know about degrees can find out for themselves that a turning number of 180 makes the turtle face the opposite direction, and that a turning number of 90 produces a right angled turn.

The turtle steers the same way as a person; it measures the turn starting from the direction it is facing.




This makes it possible for children who are starting to learn Logo to 'play turtle', steering one another by giving turtle directions. People who cannot decide how to draw something can often walk through what they want to do.

- The `forward` number in drawing a square gives the side of the square, measured in *turtle steps*. A turtle step is very small.

### Project

1. Measure the screen in turtle steps. How tall and how wide is it?

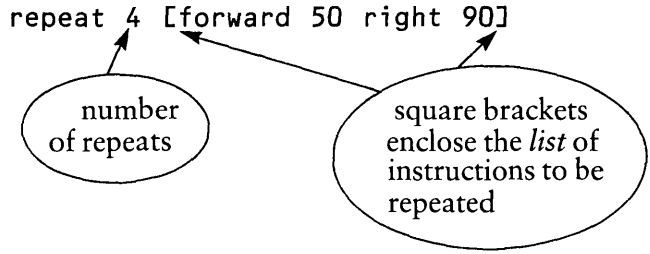
## Repeating Squares

In drawing a square, you typed:

```
forward 50
right 90
```

four times. To avoid this, you can use the `repeat` command.

These commands draw the square:




## Inputs

Different commands require different numbers of inputs. The `cs` command has nothing following it; it has no inputs. The command `forward` needs one number following it, so you have to write something like:

```
forward 50
```

`forward` has one input, which must be a number.

The `repeat` command takes *two inputs*:

- the first input is a *number*, telling the number of repeats
- the second input is a *list*, the list of instructions which are to be repeated

In Logo, lists are always enclosed in *square brackets*. The list which was used as an input to `repeat` was the list of instructions:

```
[forward 50 right 90]
```

## Projects

2. Use `repeat` to draw a square, an equilateral triangle, a regular hexagon.

3. Draw some more regular polygons.

## Fencing in the Turtle

When the drawing gets too big for the screen, the turtle is lost from view. If you don't like this effect, you can change it by the command:

```
fence
```

Then you will get an error message if the command would take the turtle out of the screen area. The command:

```
nofence
```

enables the turtle to go anywhere again. In this mode, the screen is a window on a boundless drawing surface; the turtle goes on drawing when it is off the screen, but you can no longer see it.

### Project

4. Draw some regular polygons with a great many sides.

## Nested Repeats

Commands to `repeat` something can even be nested within each other. Try this:

```
repeat 8 [repeat 4 [forward 40
        right 90] right 45]
```

### Project

5. Explore patterns made by turning some of your regular polygons round repeatedly.

# Chapter 4
# Procedures

## Teaching Nimbus

When you enter Logo, it already understands a few commands, such as `forward`, `cs` and `repeat`. It always remembers these commands, which are called Logo *primitives*. Whenever you ask it to carry out a combination of these commands, such as:

```
repeat 3 [forward 50 right 120]
```

it draws what you ask, and immediately forgets the sequence of commands you gave it.

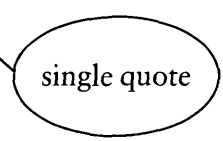
However, one of the most powerful features of Logo is its ability to learn and remember new commands. To teach Logo a new command, you have to define that command by telling Logo how to do it (using what it already knows).

## Extending Logo's Vocabulary

We shall teach Logo how to do `square`. A definition starts with the word `build`.

Type:

```
build 'square
```

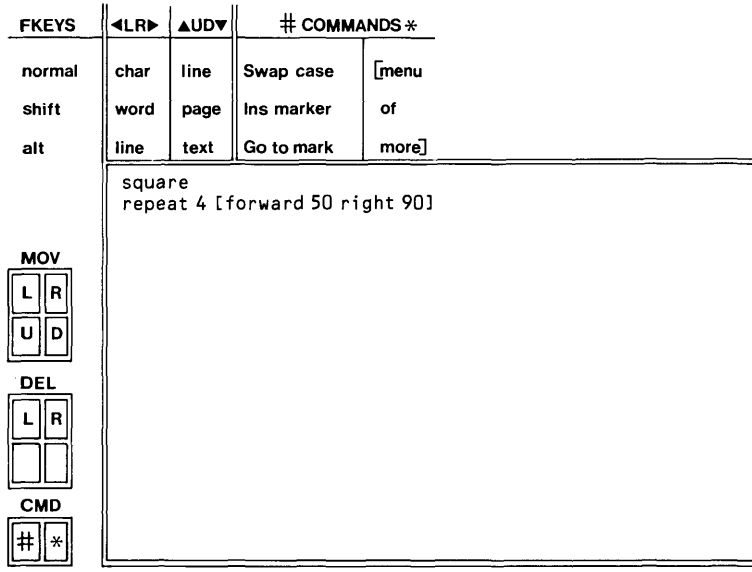



The screen changes completely. You have entered the *editor* and Logo is ready to learn the definition of `square`. You will see that Logo has already written the word `square` on the first line of the editor.

Now type in the definition of `square`:

```
repeat 4 [forward 50 right 90]
```

The screen will look like this:

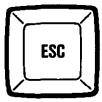



The appearance of the editor screen reminds you that you are making a definition. Beginners are often puzzled by this, because they expect the turtle to draw at once when they type in the commands. What you are doing now is rather like writing the commands down on paper before drawing the picture.

Around the border of the editor screen are instructions for using the *function keys* as an alternative way of moving the cursor around the editor screen. These keys are marked <F1> to <F10> and are found at the left of the keyboard.Experiment with them!

## Making Logo Remember

To exit from the editor, press <ESC>, the key at the top left-hand corner of the main block of keys.




Logo immediately stores the *procedure* for `square` in its memory, and returns to the drawing screen. When you have taught Logo the procedure for `square`, you can type:

```
square
```

and Logo will draw the whole square, starting wherever the turtle is. You can teach Logo as many procedures as you like, and it will remember them all. Its language grows under your control.

In the next chapter, you will find out how to `edit` a procedure which has a *bug* in it, and does not do what you want it to do. For the moment, you can get rid of the procedure that does not work by typing:

```
scrap 'square
```

The command `scrap` deletes the named procedure from the computer's memory. Do not omit the quotation mark `'` from `'square`; Logo will complain if you do. A quotation mark tells Logo that what is coming is a name or *word*; you are asking Logo to scrap the procedure whose name follows.

You can also `scrap` some of your procedures by telling Logo the list of procedures that you no longer want, using a command such as:

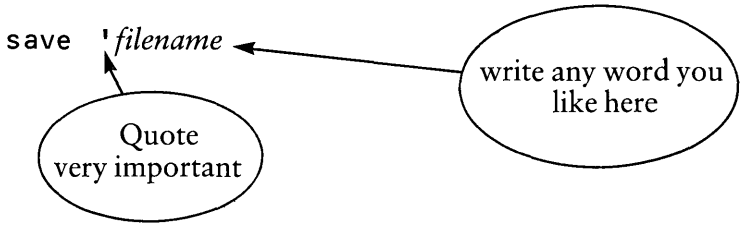
```
scrap [square triangle]
```

In this case, the names of the procedures do not have quotes. The square brackets `[ ]` signal to Logo that a list of words is enclosed in the brackets.

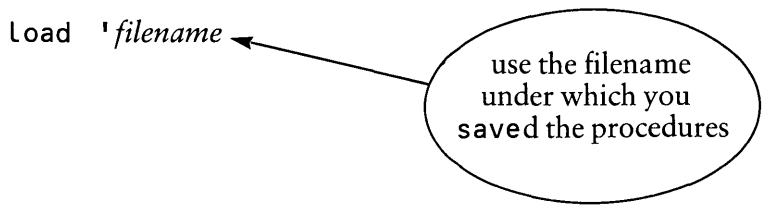
## Saving your Procedures on Disk

Logo will remember your procedures until you turn the computer off. If you want to save them for later use, you should `save` them on disk.

To save procedures on disk, all you have to do is to type:




You can use any name for the set or *file* of procedures you are saving. This command saves *all* the procedures you have taught Logo. To get your procedures back at the beginning of the next session, use the command:




### Projects

1. Teach Logo the procedures `square`, `triangle`, `hexagon`.
2. Draw a `house`. Draw a `street` of houses.
3. Draw anything you like.

## Stopping a Procedure

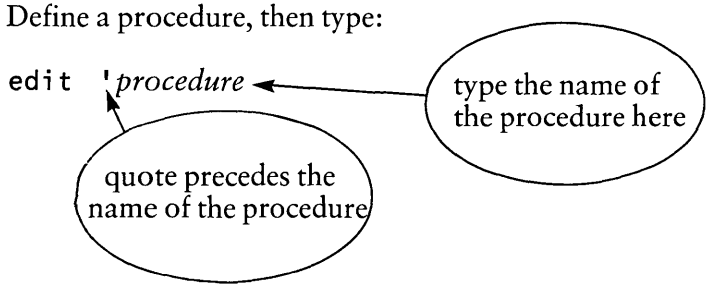
To stop a procedure while it is executing, press the <ESC> key.

# Chapter 5
# Editing

## Editing Your Procedures

You often want to change a procedure when you have seen what it does. The Logo editor makes this very easy.

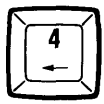
Define a procedure, then type:

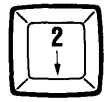


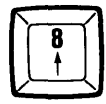

You are now in the editor, and the text of your procedure can be changed on the screen.

## Steering the Cursor

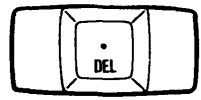
The four arrow keys on the bank of number keys in the right-hand block steer the cursor round the text.

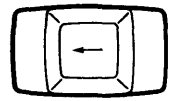
| *action* | *key* |
|---|---|
| moves it to the left |  |
| moves it to the right |  |

moves it down one line 


moves it up one line 


## Changing the Text

To delete a character, move the cursor under the character and press the <DELETE> key. 


Or use the <BACKSPACE> key to delete the character *before* the cursor. 

To insert characters into your text, move the cursor to the *exact* position where you want the new characters, and type them in. Notice that everything else moves over to make room from them, and nothing is lost.

Sometimes you need to insert a new line of text. You can type it in at the appropriate place, and let the next line move over, or you can create an empty line by moving the cursor to where you want the line, and pressing <ENTER>.

When you have finished editing, come out of the edit mode by using the <ESC> key.

## The Function Keys

The function keys in the left-hand bank of keys on the Nimbus can be used to control some operations of the Logo editor.

Some of the more useful actions are:

<F1> moving the cursor one character to the left

<F2> moving the cursor one character to the right

These keys can also be used with the <SHIFT> and <ALT> keys, which make their actions increasingly effective. <SHIFT> and <F1> or <F2> move the cursor one *word* to left or right, <ALT> and <F1> or <F2> move the cursor to the beginning or end of the *line*.

<F3> moving the cursor one line up

<F4> moving the cursor one line down

Again, using <SHIFT> or <ALT> produces more drastic movement.

The keys <F5> and <F6> delete:

<F5> deleting the character left of the cursor

<F6> deleting the character under the cursor

## Projects

1. Draw some stars. Try building a procedure `star`:

   ```
   star
   repeat 8 [forward 80 right 135]
   ```

   Change the angle and see what happens. You may need to change the number of repeats as well.

2. Some stars turn out to be polygons. Find some. Draw a regular pentagon. Draw a five-pointed star.

3. How far has the turtle turned when it has finished drawing a polygon? How far has it turned when it has finished drawing a star? Look for patterns.

4. The turtle has drawn a *closed path* when it is back at the same spot facing the same direction as when it started. What can you say about the *total turn* in a closed path?

## forever

Sometimes it is difficult to decide how many repeats are needed to draw a star. This problem can be avoided by using `forever`.

For example, the command:

```
forever [forward 80 right 115]
```

draws a star with a great many points, and continues to go over and over it for ever. The procedure can be stopped by pressing <ESC>.

# Chapter 6
# The Graphics and Text Screens

## Hiding the Turtle

When you draw a picture, you may feel that the turtle is a blemish on the finished work of art. It can be hidden by using:

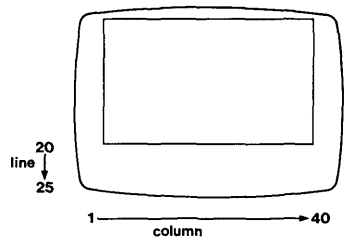
`hideturtle` abbreviation `ht`

and revealed again with:

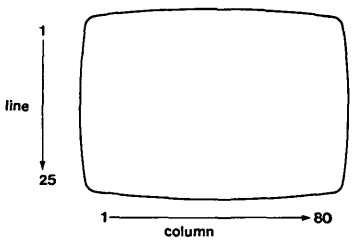
`showturtle` abbreviation `st`

## Screen Modes

Logo has a *graphics mode* and a *text mode*. When you first load Logo, the screen is in text mode, and the whole of it is available for text. When you type `cs`, the screen goes into graphics mode and displays the drawing screen, together with six lines which can be used for text at the bottom of the drawing screen.




*graphics mode*




*text mode*

Sometimes you want to see a lot of text at the same time. The command:

`textscreen` abbreviation `ts`

moves Logo into text mode. In text mode, you can clear the screen with the command:

`cleantext` abbreviation `ctx`

To get back to graphics mode, use `cs`.

## Your Workspace

When you have been working for a little time, you will have taught the computer several procedures, such as:

```
house
square
triangle
```

Your procedures are stored in an area of the computer's memory called the *workspace.* You can see the titles of all the procedures in your workspace by using:

```
titles
```

This shows a list of titles of all the procedures you have taught the computer.

## Projects

1. Draw some windmills. Try building a procedure `squaremill`:

```
squaremill
repeat 8 [square right 45]
```

   Make sure that the procedure `square` is in your workspace before you run this.

   Try some variations, using some of your other procedures.

2. Draw anything you like.

# Chapter 7
# Recursion

## What Recursion Does

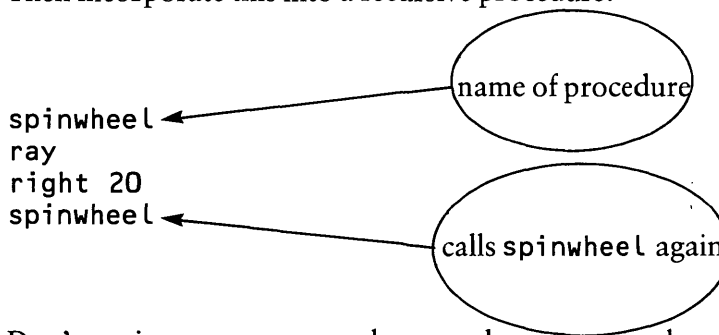
Recursion is a wonderful and powerful feature of Logo; not all computer languages can do recursion. It is enormously labour-saving — except for the turtle!

You know that procedures can call other procedures; `house` calls `square` and `triangle`. A procedure can even call itself (or seem to). This is called *recursion.*

Build a simple procedure `ray`:

```
ray
forward 50
back 50
```

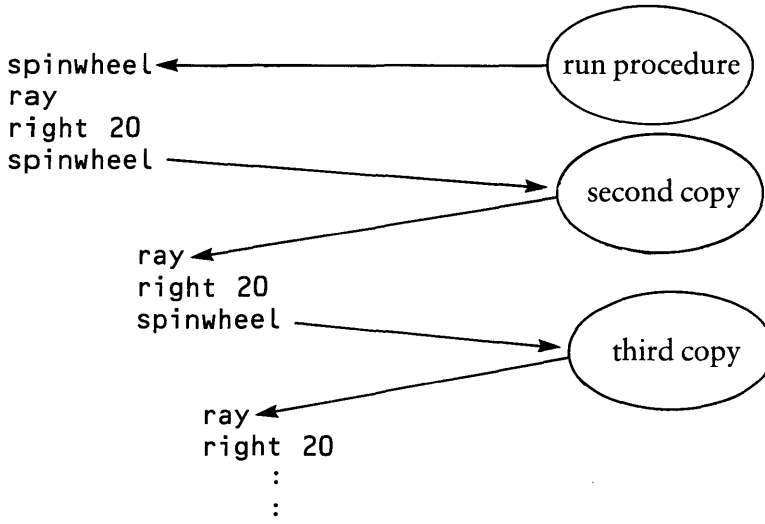
Then incorporate this into a recursive procedure:




Don't panic — you can stop the procedure executing by pressing <ESC>.

## Thinking About Recursion

We can get a clearer picture of recursion if we think of `spinwheel` as making and calling *another copy* of itself. You can see below what would happen if you could do `spinwheel` one step at a time.




In theory, this process will never end; `spinwheel` can always make and call another copy of itself. In practice, the computer may run out of memory eventually, depending on how complicated the recursion is.

### Projects

1. Try replacing the `ray` of `spinwheel` by something more exotic. You may need to change the angle as well.

2. The `ray` procedure finished with the turtle at the same place that it started. Try different effects of `spinwheel` with procedures that do this and with those that do not.

3. Draw a `scribble` something like this:

   Try this very simple recursion:

```
wheel
scribble
wheel
```

4. Try these procedures and adapt them to repeat other messages. You might try messages of more than one line.

```
message
say [Hello, how are you?]

recur
message
recur
```

# Chapter 8
# Polygons, Stars and Circles

## Using Recursion

Recursion is very powerful for drawing polygons, stars and circles. Try this:

```
square
forward 50
right 90
square
```

### Project

1. Draw some other polygons and stars using recursion.

## Heading

The turtle's heading is the direction in which it is pointing. You can set the heading using `seth 90`.

In general, the syntax is:

`seth` *n* ← the number shows the direction of the heading

You can sometimes stop a recursion by *testing* the heading.

```
onesquare
forward 50
right 90
if heading = 0 [stop]
onesquare
```

square brackets for the *list* of commands after `if`

## Project

2. Does this method of stopping recursion work for stars?

   Have you discovered the Turtle Total Trip Theorem for closed paths yet? (The total turning along any closed path is an integer multiple of 360).

# Circles

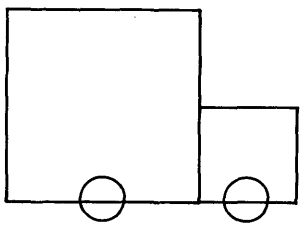
The turtle only draws straight lines, but if you draw a polygon with enough sides, it looks very like a circle.

## Projects

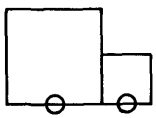
3. Experiment with circles.

   Draw a circle; draw a circle half the size, a quarter of the size ...

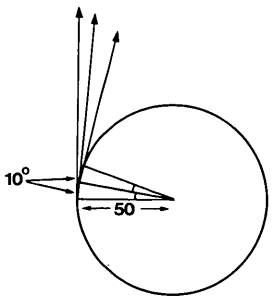
4. Draw a truck.



Draw a small truck — half size?



## Drawing a Circle with Given Radius




This is more advanced, using pi.

Suppose you want to draw a circle of radius 50 by making a 36-sided polygon. The turn will have to be 10 degrees, so you will need:

```
circle
forward ?
right 10
circle
```

how far forward

The problem is how far forward to go. The complete circle (whose circumference is 2 π radius) is to be drawn in 36 steps, so each step must be:

(2 π x 50) / 36 = ( π x 50) / 18

RM Logo has the value of π built in, under the name `pi`. Type:

```
say pi
```

and Logo will print out the value of π :

```
3.14159265358979
```

This is correct to the 15 significant figures to which RM Logo works.

So, to draw a circle, you do:

```
circle
forward 50*pi/18
right 10
circle
```

The centre of this circle is very slightly above where the turtle starts. If you want to line circles up alongside one another, a more accurate effect is produced by:

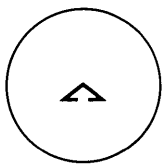
```
circle
right 5
forward 50*pi/18
right 5
circle
```

To see why this is better, draw a hexagon as a *very* approximate circle by each of the following procedures:

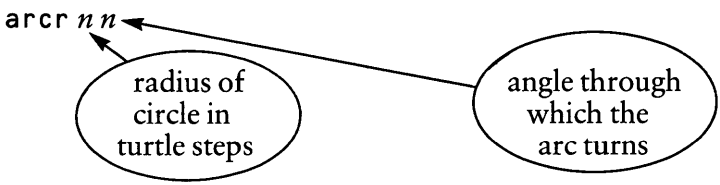
```
hex
forward 50
right 60
hex

hex2
right 30
forward 50
right 30
hex2
```

## Projects

5. Make a procedure to draw a circle of given radius when the turtle is sitting at the centre of it.



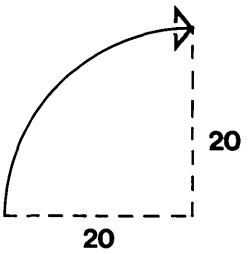
6. RM Logo provides two primitive procedures, `arcr` and `arcl`, which draw arcs, turning to either right or left. Each procedure takes two inputs, and the syntax is:



For example:

```
arcr 20 90
```

does this:



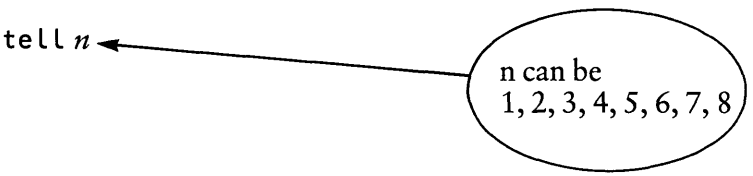

and `arcr 20 360` would draw a complete circle of radius 20 turtle steps.

7. Build a `snake`. 

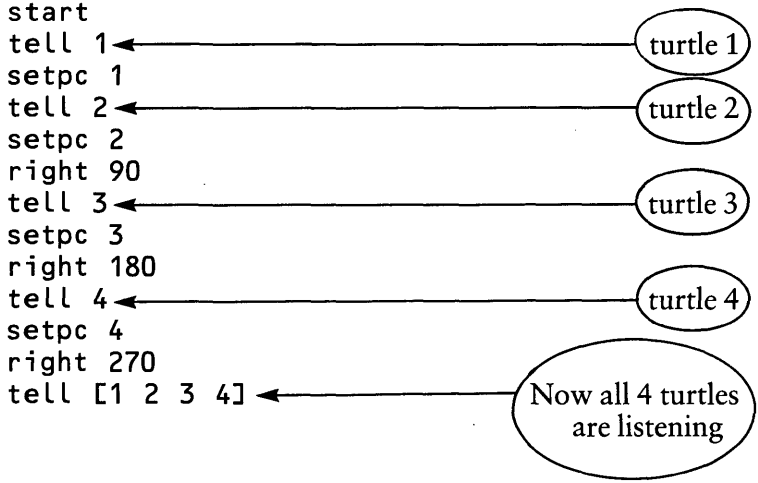
   Use a `snake` instead of a line for drawing some stars.

## Multiple Turtles and tell

RM Logo can talk to just one turtle, or it can talk to up to *eight* different turtles. The command which calls up a particular turtle is:




Try the following procedure which produces four turtles with different pen colours, facing different directions, all ready to draw the same pattern at the same time.




Now type in `square` (make sure that `square` is in your workspace first), and watch all your turtles drawing squares at the same time. Try some other patterns with multiple turtles.

# Chapter 9
# Variables

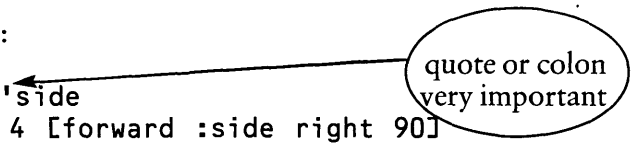
## Varying the Size of a Drawing

At present, if you want to draw a square of a different size, you have to change the procedure. If you use variables, you can input the size of drawing you would like without changing the procedure.

Change the original version of `square` to use a variable. Start with:

```
square
repeat 4 [forward 50 right 90]
```

Edit it to:




```
sqinp "side
repeat 4 [forward :side right 90]
```

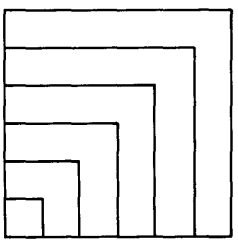
Do not omit the quotes or colons; they are explained later in the chapter. When you try to run `sqinp`, it will complain that it does not have enough inputs. Type:

```
sqinp 50
```

and it will run. You can use any number as the input.

### Project

1. Make a procedure `growsquare` to draw this.

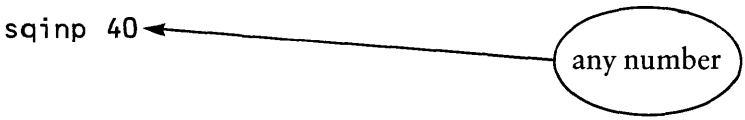


## Procedures Taking Inputs

You will recall from Chapter 3 that some of the Logo primitives take inputs. They include `forward, backward, right, left.`

You have to type `forward 50` and so on. Other Logo primitives do not take inputs. For example:
`cs, rubber`

Until this chapter, none of the procedures you wrote yourself took inputs. Now you have written a procedure `sqinp` that takes an input. You have to type:




A procedure may take several inputs. Try this:

```
rectangle 'height 'width
repeat 2 [forward :height right 90
        forward :width right 90]
```

To run this procedure, you have to give two inputs, so you type something like:

```
rectangle 50 35
```

### Project

1. Build procedures to draw:

   a house of given size

   a truck of given size

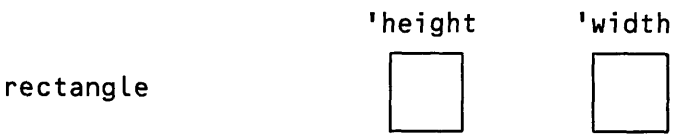
## Colons and Quotes

When you build:

```
rectangle 'height 'width
```

Logo does two things:

- It sets up a procedure called `rectangle`.
- It associates with `rectangle` two empty boxes called *variables*. Logo gives those boxes the *names* `'height` and `'width`.



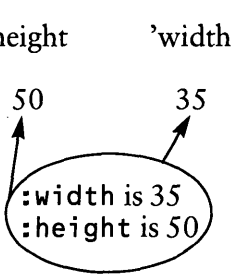

In Logo, names (or *words*) always have a quotation mark ' in front of them. You have used this earlier. To save your workspace, you typed something like:

```
save 'Sally
```

`'sally` is the *name* that you gave your workspace when you saved it. In the same way, the names of the variables in `rectangle` are `'height` and `'width`.

But why are there colons rather than quotes in the rest of the procedure? The answer is that a colon indicates the value which is given to a variable, that is, the number which is put in the box.

When you type `rectangle 50 35`, the situation becomes:




and the values in the boxes are 50 and 35. The procedure is given the title:

```
rectangle 'height 'width
```

telling Logo that `rectangle` will use two named boxes. So:

```
rectangle 50 35
```

tells Logo that the value found in the box `:height` is 50 and the value in `:width` is 35.

Within the procedure, the command `forward :height` instructs the procedure to look in the box named `'height`, and to find and use the *value* of the variable — that is, 50. The punctuation gives Logo important signals; `'height`, `:height` and `height` mean three different things in Logo.

- `'height`
  (pronounced quote-height) is the *name* of a variable

- `:height`
  (pronounced colon-height or dots-height) is the *value* of the variable whose name is `'height`

- `height`
  without a quote or a colon is nothing to do with variables. Logo always thinks that a word without a colon or a quote is a *procedure*

If you write `height` without a quote or colon in:

```
rectangle height 'width
```

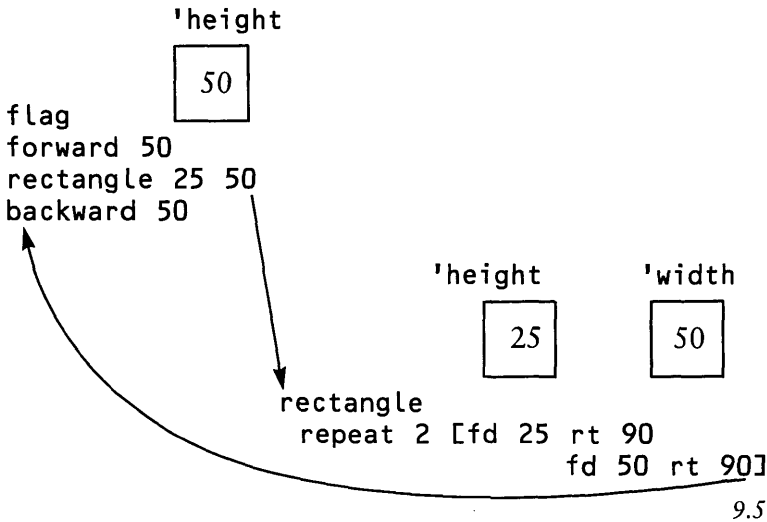
Logo will complain. It has looked for a procedure `height`, and failed to find one.

## Local Variables

The boxes provided for variables in Logo are private; their use is restricted to a particular procedure. They are called *local variables*, and they belong to that procedure only. Because the variables that we have dealt with up until now are local, different procedures can use variables with the same name. Try this:

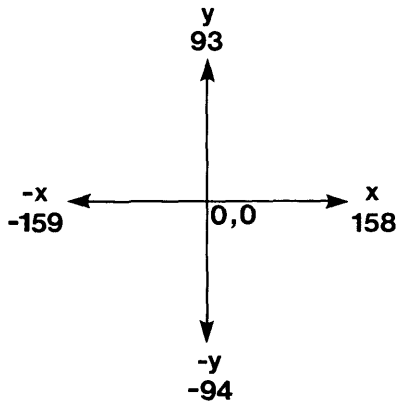
```
flag 'height
forward :height
rectangle (:height/2) :height
backward :height
```

This is what happens when you do `flag 50`:



## Coordinates

Instructions for moving the turtle to a particular position on the screen are often useful. The screen has the coordinate system shown below.




The commands `setx` *n* and `sety` *n* move the turtle to given coordinates. For example:

```
setx 100
sety 50
```

will move the turtle to position (100,50).

You can move to new x and y coordinates at the same time by using:

```
setpos
```

This command needs square brackets to enclose the *list* of the two coordinates to which the turtle is to move, so:

```
setpos [100 50]
```

produces the same effect as the last two commands.

If you want to use variables with `setpos`, different syntax is needed. To move the turtle to the point whose coordinates are stored in the variables named `"along` and `"up`, for example, you have to write:

```
setpos sentence :along :up
```

The reason for this is that Logo does not allow you to write variables inside lists, so it objects to `[:along :up]`. But the command:

`sentence` abbreviation `se`

overcomes this problem by turning its inputs into a list. So:

```
sentence :along :up
```

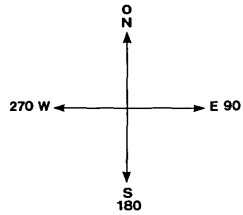
outputs a list whose members are the values of `:along` and `:up`, so that if, for example, `:along` is 50 and `:up` is 35 then `setpos sentence :along :up` is equivalent to `setpos [50 35]`.

## Heading

The turtle's *heading* is the direction in which it is pointing. You can set the heading using:

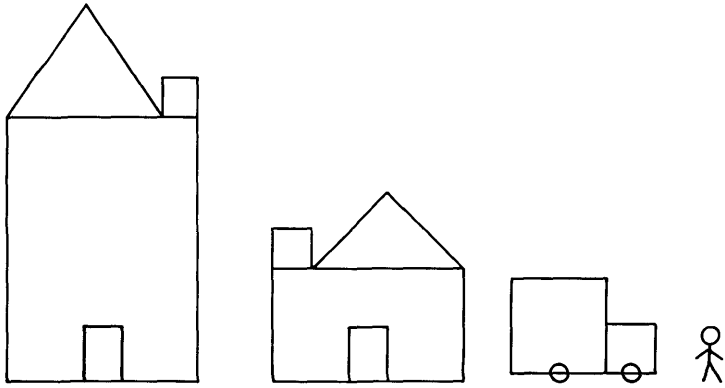
`seth` *n* stands for `setheading`

The heading is measured clockwise from the position where the turtle starts from (ie. facing upwards).



## Project

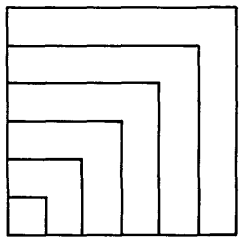
3. You are now ready to draw streets.



and all sorts of pictures.

# Chapter 10
# make, say and result

## make



In Chapter 9 you wrote a procedure `growsquare` to draw this.

Probably you used a combination of two procedures which were something like these:

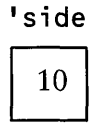
```
square 'side
repeat 4 [forward :side right 90]

growsquare
square 10
square 20
square 30
square 40
square 50
```

This seems a rather tedious way of writing the solution to this problem. You might ask whether there is a way of changing the value of the variable named `'side` automatically within `growsquare`, so that you can avoid typing `square` five times.

The Logo primitive `make` enables you to do this — it is used to put a value in a box such as `'side`. Type:

```
make 'side 10
say :side
```

The command `make 'side 10` tells Logo to create a box whose name is `'side` and put the number 10 in it.




When you ask the computer to `say :side`, it will print 10. The `say` command puts whatever follows it onto the screen.

If a box whose name is `'side` already exists, `make 'side` puts a new value in the box. Logo will even work out the value to put in that box. Try:

```
make 'side (:side + 10)
say :side
```

Did Logo print out 20? It added 10 to the previous value in the `'side` box, which should have been 10.

## Project

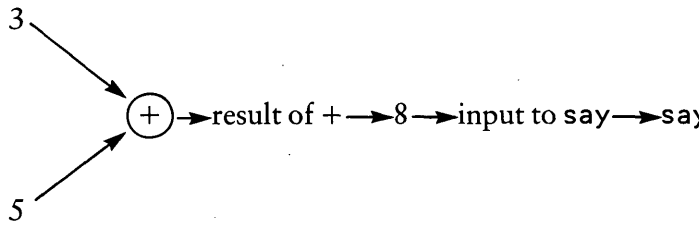
1. Build a revised procedure for `growsquare`, using `make`. A possible procedure follows.

```
growsquare
make 'side 10
repeat 5 [square :side make 'side
     (:side + 10)]
```

## say

The `say` command enables you to print out a result without using the turtle. `say` takes one input, so it needs to be followed by that input, which may be the output of an arithmetic operation, for example.

The following diagram shows how `say 3 + 5` works.




Try:

```
say (60 -10)
say (40 / 6)
```

You can omit `say` in these examples; Logo will respond to `60 -10` and `40 / 6`, provided that you are not working within a procedure. In a procedure, if you omit the `say` and write only `60 -10`, Logo will complain that you didn't tell it what to do with 50.

## Variables in Non-Graphics Procedures

In a turtle drawing procedure we can use variables to control the size of the drawing.

For example, the procedure:

```
hexagon 'side
repeat 6 [forward :side right 60]
```

draws a hexagon of side 60 when you type `hexagon 60`.

In just the same way, we can use variables in non-graphics procedures. For example, we can build the procedure below, called `add3`, to add three numbers; this procedure will take three inputs.

```
add3 'a 'b 'c
say :a + :b + :c
```

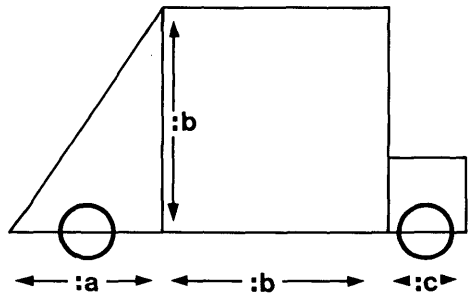
Sometimes, you might want to use the result of adding three numbers in another procedure. Then, you do not want Logo to say the result of the addition, but to send the result to another procedure. The primitive:

```
result
```

enables you to do this. Modify the procedure `add3` to this:

```
add3 'a 'b 'c
result :a + :b + :c
```

Now if you type `add3 20 50 30`, nothing happens on the screen; the `result 100` is waiting to be used by some other command, within another procedure. For example, you might want to draw the baseline of this car:




This could be done with:

```
forward add3 :a :b :c
```

The command `forward add3 20 50 30` is carried out in the following way:

`forward` calls procedure `add3`.

`add3` takes the inputs 20 50 30
and adds them together to get the result 100.

the result 100 is returned from `add3` to `forward`

`forward 100` can be carried out.

So, the primitive `result` takes one input and it does three things:

- it stops the procedure in which it is used, and returns control to the procedure which called it
- at the same time it takes its input, which may itself be a complicated procedure, and works it out
- it passes the `result` of that working out to the procedure which called it

The procedure `add3` is rather trivial, but in more complicated cases there are advantages in using sub-procedures which send back a `result` to the main procedure. For example, the same procedure `add3` can be used in all the following examples without change:

```
say add3 20 50 30

forward add3 20 50 30

repeat (add3 20 50 30) [square right 40]
```

## Projects

In these projects, you will probably want to use `textscreen`, as they do not involve any drawing.

2. Make a procedure to find the area of a rectangle. When you type `rect.area 6 5` it should respond with `The area is 30`

   You can achieve this message by using something like:
   ```
   say sentence [The area is] :a * :b
   ```

   Recall that `sentence` makes a list from its two inputs.

3. Make a procedure to find the perimeter of a rectangle. When you type: `rect.perim 6 5` it should respond `The perimeter is 22.`

4. Make procedures to find the area and perimeter of a square. Use `rect.area` and `rect.perim` as subroutines.

5. Print out a list of the squares of numbers from 1 to 20.

6. `sentence` puts together two inputs to make a list. Build a procedure `se3`, whose result is a list consisting of *three* inputs to the procedure. Use this to build a procedure `signature` which will write titles for your pictures, as follows. When you write:

   ```
   label signature 'Superstar 'Jane
   ```

   it will label your picture

   ```
   Superstar drawn by Jane
   ```

   creating the signature near where the turtle is.

# Chapter 11
# More Commands Using Numbers

## Prefix Arithmetic

There are arithmetic commands in RM Logo which are of a different type from the ordinary arithmetic commands — they are *prefix commands*. They do the same jobs as the ordinary arithmetic commands, but their syntax is different. To add 25 and 12, you can use:

```
add 25 12
```

instead of `25 + 12`. The command `add` is written first, followed by its two inputs: the command is *prefix*ed to the inputs. There are four prefix arithmetic commands which can be used instead of the ordinary arithmetic commands if you wish. They are:

| | |
|---|---|
| `add` | |
| `subtract` | abbreviation `sub` |
| `multiply` | abbreviation `mul` |
| `divide` | abbreviation `div` |

These commands are provided in RM Logo to enable the language to be used consistently: all the non-arithmetic Logo commands are prefix. You write:

```
forward 50
```

putting the command before the inputs. The ordinary arithmetic commands, such as +, are *infix*; you put the command between inputs:

```
say 3 + 4
```

The arithmetic commands are an exception to the general rule that Logo commands are prefixed to their inputs.

## Other Arithmetic Commands

It is sometimes useful to be able to get rid of the decimal part of a number such as 6.66667. This can be done by using the command `int`. For example:

```
say int 6.66667
```

chops off the decimal part and prints `6`. Similarly, `frac` removes the whole number part, leaving only the decimal. For example:

```
say frac 6.6667
```

prints `0.6667` as its result.

Logo can also evaluate square roots; the command is:

`sqt` *number*

For example: `say sqt 225` prints out `15`.

Logo can also do trigonometry. Look in the RM Logo book for details.

## Integer Division

If you type `say 9/4` or `say div 9 4` then Logo will respond with `2.25`. Some young children may not be used to seeing decimal numbers. The command:

```
share 9 4
```

produces the whole number quotient `2`.

You can get the remainder by using:

```
remainder 9 4
```

abbreviation `rem`

which produces `1`.

## pick and random

These are useful facilities for playing games and for drawing unexpected designs. The command:

```
pick
```

takes one input and returns a random whole number. For example:

```
say pick 6
```

prints a random whole number chosen from 1, 2, 3, 4, 5, 6, so `pick 6` can be used to simulate throwing a die. You can also produce random pen colours by choosing a whole number from 0, 1, 2, ..., 14, 15. To do this, use:

```
(pick 16) -1
```

The brackets are necessary here. If you leave them out, and write:

```
pick 16 -1
```

Logo will work out 16 — 1 first, and think that you mean to do:

```
pick 15
```

The command `random` gives a random decimal number between 0 and 1, which is sometimes useful.

## Projects

1. Write a procedure `average` which takes two inputs, so that:

   ```
   say average 12 20
   ```

   prints `16`.

2. Draw a square at a random position on the screen. (You can use a combination of `setpos` and `pick`, for example.) Make the square of random size. Also alter the pen colour randomly. Use a procedure to draw a lot of random squares.

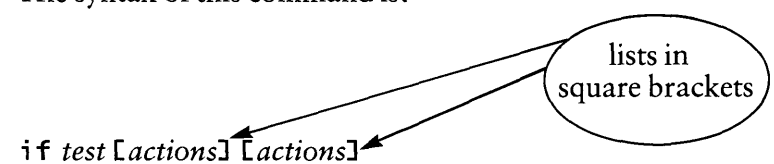
3. Use Pythagoras' Theorem to draw a square of given size and its diagonals.

4. Simulate a die. When you type `toss`, it will draw a picture of the face, showing a random number of spots. Design each face first. You will need a *test* to decide which face is to be displayed; see the next section.

# if ... and Tests

Often, you want Logo to do different actions according to whether some condition is true or not. For example, when drawing random squares on the screen, you may not want to draw a square which will be partly outside the screen area. The command needed to make a choice between actions is `if ...`

The syntax of this command is:



For example, you might use the line:

```
if heading = 0 [right 90 stop] [left 90]
```

The lists of actions must be enclosed in square brackets.

A *test* is a question which returns the answer `'true` or `'false`. If the test returns the answer `'true` then the first list of actions is carried out. If the test returns `'false` then the second list of actions is carried out.

The second list of actions may be omitted, so that the command reads:

`if` *test* `[`*actions*`]`

In this case, if the test returns `'false`, the program goes straight on to the next line.

To make `if` work, we need tests which return `'true` or `'false` answers. A comparison between two numbers using <, > or = gives a way of doing this. For example:

`5 > 2` is `'true`

`5 > 7` is `'false`

Logo is able to make comparisons such as this, so that we can use commands such as:

```
if :x < 90 [setx :x]

if :x > 90 [setx (120 -:x)]
```

# Chapter 12
# Words and Lists

## A Larger Project

Young children can gain much pleasure from using the turtle graphics part of Logo, but they may not have enough skill to use the full computer keyboard, and the amount of typing involved in using the full version of Logo may be too difficult.

Very young children need a simplified version of Logo, which uses a single key to represent each of `forward`, `backward`, `right` and `left`. The space bar should not be needed, and pressing a single number key should produce a larger movement of the turtle than in the full Logo, so that a child will really see that the turtle has moved. Then brightly coloured labels can be stuck over the few keys that are to be used, to help the child to find them.

You will have your own ideas about exactly what features you would want to build in to your own version of Younglogo. One of the exciting features of Logo is that it does not only turtle graphics: it is a high-level computing language which enables you to write software such as Younglogo to your own design.

The next few chapters show you how to use features of Logo which are not needed for turtle graphics; however, you do need them for writing software such as Younglogo and even games programs.

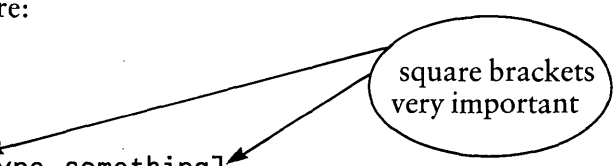
We shall introduce these additional Logo commands as we start you off on designing and writing your own version of Younglogo.

## Input From the Keyboard

In Younglogo, you will want a child to input a single letter such as `f` from the keyboard to the program, and Logo will need to translate that to `forward` and use it to drive the turtle. The command which enables Logo to accept input from the keyboard is:

```
readlist
```

To see how `readlist` works, build this little experimental procedure:




```
try
say [type something]
make 'txt readlist
say :txt
try
```

When you run `try`, the words `type something` are printed on the screen, and the procedure then waits for you to type something in at the keyboard and press <ENTER>. Whatever you type goes into the `'txt` box. The procedure `try` is a recursive procedure, so it will keep asking you to type something in until you press the <ESCAPE> key.

Try different sorts of input when `try` asks you to type something. Include words, phrases, numbers, nothing at all.

The command `readlist` works like this:

- `readlist` stops and waits for you to input some characters at the keyboard.
- `readlist` outputs your typing, stored in a list. So if you type `hello`, the command `readlist` outputs the list `[hello]`.

In `try`, the output of `readlist` goes into the `'txt` box, as if the line were:

```
make 'txt [hello]
```

## Lists

Square brackets are an essential part of Logo; they indicate that you are dealing with some data in the form of a list. A list always has square brackets surrounding it. Here are some examples of lists:

```
[type something]
[hello]
[what do you want?]
[243]
[2 4 3]

[46 ramble 12.76]
[sunday monday tuesday wednesday thursday
        friday saturday]
[]
```

The elements of a list can be *numbers*, *words* or other *lists*. The easiest of these three data types to recognise is a number.

A number can be positive or negative, a whole number or a decimal. You must not put spaces in the middle of a number. These are valid numbers:

254
–254
254.123
–254.123

# Words

A *word* is any sequence of characters which starts with a quotation mark. Logo recognises the beginning of a word by the quotation mark, and it recognises that it has come to the end of a word when it comes to a space. This means that you can mix up letters, figures and punctuation marks in a word. These are valid words:

```
'hello
'line2
'want?
'2+3
'goodb?ye
';;;
```

## When to use Quotes

- The name of a variable is a word (which explains why you use the quotation mark so often in Logo). For example, you might type: `make 'height 50`.
- When you save your workspace with `save 'sally`, then `'sally` is a word.

## When not to use Quotes

- When you command Logo to execute a primitive procedure, for example `lift` or `forward 50`
- When you run a procedure you have built yourself, such as `square 50`
- Procedure names do not have a quote. If you accidently type a word without the quote, for example `hello`, Logo will look for a procedure `hello` and complain if it doesn't have it in its memory.

The idea of using a quote to distinguish a word from a procedure is borrowed from the language Lisp from which Logo is descended. So remember that a quote stops Logo from trying to find and carry out a procedure with that name.

In Logo, brackets also prevent execution. If you type:

```
say [hello how are you]
```

Logo prints the list `hello how are you` *without the brackets*, and does not try to look for procedures called `hello`, `how`, `are` and `you`.

It is important to keep in mind the fact that `say` prints a list without its outer square brackets, and a word without the `'` at the beginning. The primitive `print` does exactly the same as `say`, except that it prints square brackets when they exist. Try these:

```
say [say hello]
print [say hello]
say 'hello
print 'hello
```

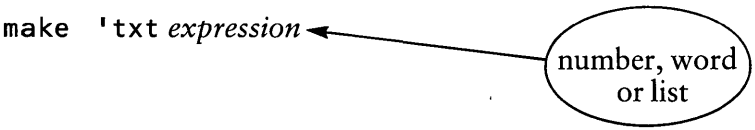
## Project

1. The error messages in Logo vary according to the type of error, as you know. This can help you to distinguish between words and lists. Observe the effect of various commands such as:

   ```
   say 'hello
   print hello
   say [hello]
   print :hello
   say []
   ```

## Variables and their Values

The name of a variable is a word, such as `'txt`. The command:




creates a box labelled `'txt`, and puts the expression into that box as the value of the variable. Logo is extremely kind, compared with other computer languages, about the expressions which it will accept as values of variables. Experiment in putting different types of expression into the `'txt` box.

Did you find that Logo will accept a number, or a word, or a list, and put it in the `'txt` box? Logo's ability to accept any data-type and put it into a variable box is very useful. From now onwards, we shall use *expression* to stand for any of a *number*, a *word* or a *list*.

Logo can also test to see which data-type is in a variable box. Try this procedure:

```
test 'type
if listq :type [say [list]]
if wordq :type [say [word]]
if numberq : type [say [number]]
end
```

Use different values for the variable `'type`. Try:

```
test 20
test [hello there]
test [hello]
```

Three new Logo primitives were used in this procedure. They are:

`listq, wordq, numberq`

Each *tests* to see if its input is of its own type. The syntax is:

`listq` *expression*

`listq` returns `true` if the expression is a list, and `false` otherwise. Hence, `listq` is a *test* which provides the `true` or `false` output which `if ...` needs as its input. The syntax of `wordq` and `numberq` is similar.

## Projects

2. Start to build the procedures which will make up Younglogo. `try` is the basis of Younglogo.

   Design a nice prompt which sits on the screen when Younglogo is waiting for you to type something in.

   Arrange (temporarily) for Younglogo to print `forward` when you press <F>, `backward` when you press <B>, and to ignore any other input. To do this, you need to know that the equals sign = is a test which not only tests numbers for equality; it can also be used to test words or lists to see if they are the same.

3. Amend Younglogo so that it will stop gracefully without using `escape`. It should stop if you type in some special word such as `exit`. If you choose `exit` it will apparently become a primitive command in Younglogo; `exit` in RM Logo takes you out of Logo to the operating system.

## Laying out Text on the Screen

Now that you can use `readlist`, you are able to write an interactive program. For example, a program may ask someone to input a number at the keyboard, and the procedure will then print out the multiplication table for that number.

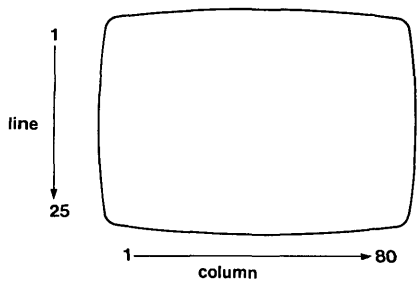
The screen layout may be a bit tiresome, but the following points should help. You can clean the textscreen by using:

`cleantext` abbreviation `ctx`

and start the printing at a particular place on the textscreen by using:

`setcursor`

`setcursor` takes a list with two elements as input; the first element is the line number and the second the column number, as shown in the diagram below.




The command `setcursor [10 2]` puts the cursor in the position shown.

The command `type` is just like `say`, except that the cursor does not move to the next line after printing.

The commands `say` and `type` take only one input. If you need them to take more than one input, you have to use `sentence` to make two inputs into a single list, as in the following example:

```
say sentence [how are you] :name
```

If the variable `'name` contains `Jane`, this will produce the message:

```
how are you Jane
```

The command `say []` produces a blank line of printing.

## Projects

4. Print out some multiplication tables. (You may need to read the next chapter to cope with some of the problems you will encounter.)

5. Write a "drill and practice" arithmetic program.

# Chapter 13
# Looking More Closely at Lists

## Further Development of Younglogo

When Younglogo is running, a child's input from the keyboard will be something like:

```
f7
```

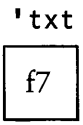
You will want the turtle to respond by doing something like:

```
forward 70
```

In this chapter we look at the commands which enable you to make Younglogo do this.

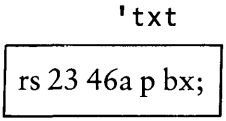
## Tearing Lists Apart

In the second section of Chapter 12, you probably arranged to store the child's input from the keyboard in a box called `'txt`.




A likely method of proceeding with the Younglogo project is to get Logo to do something like this: look at the first character of the list in `'txt` to see if it is `f`, and the second character to verify that it is a number. Then you will need to get Logo to build up the command `forward 70` and tell the turtle to carry out this command.

Logo provides commands which take lists apart. Our list has only one element, the word `'f7`. However, a young child might manage to get a list with many elements into the `'txt` box; for instance lots of keys (including the space bar) pressed randomly would end up with something like this:




This list has five elements. A child may also manage to get the empty list `[]` into `'txt`, by hitting <ENTER> before doing anything else. The empty list has no elements.

The commands `first` and `rest` are used to split off bits of lists and look at them. Try:

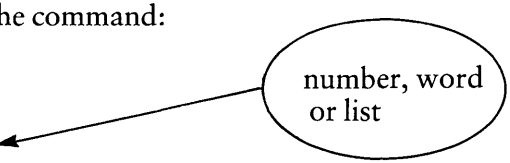
```
print first [hello how are you?]
print rest [hello how are you?]
```

Also try commands such as the following, until you know how `first` and `rest` work with lists, words and numbers:

```
print first 'hello
print first 123.4
print rest 'hello
print rest 123.4
print first []
```

## first and rest

The behaviour of the command:




can be described as follows:

- If *expression* is a *list*, `first` returns the *word* which is the first element of the list. `first` gives an error message if the list is empty.

- If *expression* is a *word*, `first` returns the character which is the first character of the word. `first` gives an error message if the word is empty.

- If *expression* is a number, `first` returns the number.

The behaviour of the command:

`rest` *expression*

is rather similar to that of `first`, except that the result of deleting the first element of a list is a list, and the result of deleting the first element of a word is a word:

- If *expression* is a *list*, `rest` returns a list consisting of all the elements except the first element of the list. It gives an error message if the original list is empty.

- If *expression* is a *word*, `rest` returns a word containing all the characters except the first character of the word. It gives an error message if the original word is empty.

- If *expression* is a *number*, `rest` returns a 0.

You can print the character which is the first character of the first word of a non-empty list stored in `'txt` by doing:

```
print first first :txt
```

For example, if `'txt` contains the list [f7], then `first :txt` produces `f7`, and so `first (first :txt)` produces the character `f`.

Similarly, the second character of the list in `'txt` can be produced by doing:

```
print first rest first :txt
```

Here `rest (first :txt)` produces the list [7], and so `first rest first :txt` produces the character 7.

## Project

1. As the next step towards Younglogo, build a procedure Babylogo, which might be a child's first introduction to Logo. It should do these things:

   — ignore an empty input from the keyboard;

   — move the turtle `forward 10` if <F> is pressed;

   — move the turtle `backward 10` if <B> is pressed;

   — turn the turtle `right 45` if <R> is pressed;

   — turn the turtle `left 45` if <L> is pressed;

   — return to Logo if the keyboard input is `exit`;

   — ignore any other keyboard input.

# Other Commands to Manipulate Lists

The following commands may also be useful:

```
last
butlast                                    abbreviation bl
```

They behave similarly to `first` and `rest`, at the other end of a list.

The command:

`sentence` abbreviation `se`

makes a list out of two inputs. Try this procedure with different inputs until you know how `sentence` works.

```
sent.test
make 'a readlist
make 'b readlist
print sentence :a :b
sent.test
```

## List Pointers

RM Logo has another way of getting at a particular element of a list, instead of using `first` and `rest`. This method is the use of a *list pointer*. Try this procedure:

```
pointer
make 'x [hello how are you?]
print :x # 1
print :x # 2
```

The syntax of the list pointer # is:

```
list # integer
```

The *list* may be a value stored in a box such as `'x`, or you may input it directly. For example:

```
print [hello how are you?] # 4
```

will produce you? (Some people pronounce # 'number', and others call it 'hash'.)

List pointers only work on lists; you cannot use them to get at particular characters in words. Then, you need `first` and `rest`. In the Babylogo procedure, you could obtain the first element of the list in `'typing` by doing:

```
make 'front :typing # 1
```

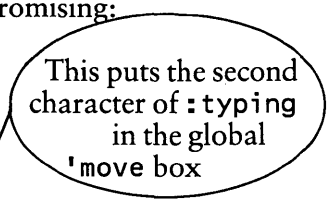
This produces a word, which then has to be split up using `first`.

# Chapter 14
# Single Key Input

## The Primitive key

Babylogo teaches a young child to move the turtle move by pressing one of the keys <F>, <B>, <R> or <L> and then hitting <ENTER>. The amount by which the turtle moves is pre-determined in the procedure.

The next step in a child's learning may be to press <F>, <B>, <R> or <L> followed by a number key which controls how far the turtle moves. It seems quite easy to design a procedure to do this by splitting off the second character of the list in `'typing`, looking at it to see that it is a number, and using it to control the turtle's movement. This procedure looks promising:




```
distance
cs
make 'move first rest first :typing
if not numberq :move [stop]
forward (10 * :move)
```

Unfortunately, the procedure does not work, and the turtle obstinately refuses to move. If a child's input is `f7`, Logo fails to recognise that the 7 (the second character of f7) is a number. When `first` is used to split up a word it treats every character as a letter, even one that looks like a number to us; hence, the answer to the test `numberq` is `false`.

You can verify this by amending the line of `distance` which makes the test:

```
if not numberq :move [print :move stop]
```

and using various lists in `'typing`. You can often detect bugs in your programs by putting temporary `print` statements at suitable places in your procedures.

There is no primitive command in RM Logo which can be used to persuade Logo to recognise the 7 as a number rather than a letter, once it has been embedded inside a word. A different approach must be used to solve this problem.

The command `key` gives this different approach. `key` waits for a *single* key to be pressed, and returns its value. It accepts the input instantly, and does not wait for <ENTER> to be pressed. To see how `key` works, build this procedure and try it out by pressing various keys:

```
input
make 'a key
print :a
print numberq :a
input
```

You will see that `key` can distinguish between a letter and a number. When `key` encounters a character which is not a letter or a number, for example <ENTER> or the space bar, it returns a word made up of a combination of characters such as `\0d` or `\`; these are codes for the characters.

You are now ready to build Younglogo. The first two characters a child inputs can be stored in two different boxes, called `'front` and `'move`, by using the commands:

```
make 'front key
make 'move key
```

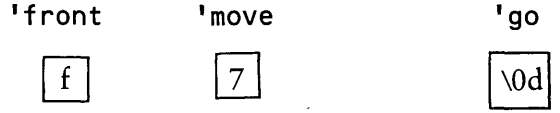
A keyboard input of `f7` will now produce:


You can then look in `'front` to see which command the turtle is to execute, and look in `'move` for the number which is used to calculate how much the turtle is to move.

If you follow this approach, the child will no longer need to press <ENTER>; the turtle will move as soon as two suitable keys are pressed. However, you may still want the child to press <ENTER>, to get used to one of the characteristics of many computer programs — nothing happens until <ENTER> is pressed. If you want to do this, you can arrange for Younglogo to wait for a third character to be input, and put this in the box `'go`, with:

```
make  'front key
make  'move key
make  'go key
```

In this case, an input of `f7` <ENTER> would produce:




Then you can test to see whether the word '\0d is in `'go` before making the turtle move. Note that a word must be preceded by `'` in a command such as:

```
if :go =  '\0è/D [draw]
```

## Project

1. You are now able to complete Younglogo. When finished, it should be able to draw `forward` and `backward`, and to turn `right` and `left` by appropriate multiples of the keyboard input. You may also want to add other features, such as `lift` and `drop`. You may also want it to exit gracefully to Logo when you type some special characters such as `ex`.

# keyq

The command `keyq` tests whether a key has been pressed, and so it can be used to stop Logo doing something. Try this procedure:

```
draw
do [forward 1] until keyq
```

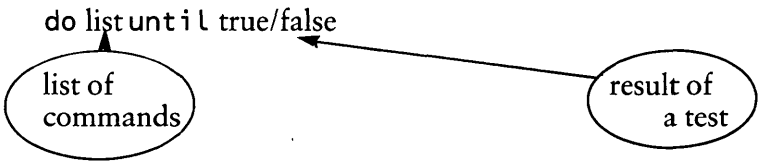
The turtle moves forward until you press a key, and then it stops. `keyq` is a test, and returns `'false` if no key has been pressed, and `'true` if a key has been pressed. `keyq` does not look to see what key has been pressed; if you want Logo to read the key, you can use `key`. Try this addition to the `draw` procedure:

```
draw
do [forward 1] until keyq
if key = 'r [right 90]
draw
```

# do [ ... ] until ...

In the commands `do [...] until ...`, `do` is followed by a *list* of the commands to be carried out, and `until` is followed by the `true` or `false` result of a *test*.

The syntax is:




Another example of how `do [...] until ...` can be used to control a loop is:

```
do [forward 1 right 1] until heading >= 359
```

This draws a circle if the original heading of the turtle is 0 (the starting position).

This method of looping enables a list of commands to be repeated until something happens to stop the loop. Looping stops when the test returns `true`. Note that the list of commands following `do` is always executed at least once, even if the test is `true` the first time.

As an example of this, try these procedures:

```
count2s
make 'x readlist
make 'x first :x
do [make 'x inc :x] until :x > 100
count2s

inc 'x
print :x
result 2 + :x
```

Unfortunately, this has a bug in it; if you input the number 108, it prints 108 before asking you for another number. The problem is that the `do [...] until ...` loop does `inc` once before it tests the number to see whether to stop looping.

## while

Another way of controlling looping is to use `while`. We could replace the looping line of `count2s` by:

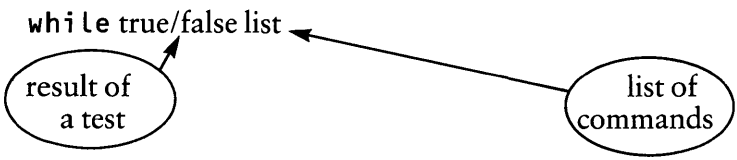
```
while :x < 100 [print 2 + :x]
```

`while` carries on doing the list of commands as long as the test returns `true`. If the test is `false` the first time, the command is never executed.

Try this procedure to see how `while` works.

```
holdkey
while key = 'f [forward 5]
while key = 'r [right 90]
while key = 'l [left 90]
holdkey
```

The syntax of `while` is:




An important difference between `do` [...] `until` ... and `while` [...] is the order in which they do the test:

```
do [list of commands] until test
```

tests *after* carrying out the command, but:

```
while test [list of commands]
```

tests *before* carrying out the command. Hence, if `while` is used in the `count2s` procedure, an input greater than 100 is not printed out.

## Projects

2. Turn `draw` into a piece of software which allows you to sketch on the screen, steering the turtle by pressing different keys.

3. The following procedure shows one way of taking input from the keyboard and storing each letter as an element of a list:

```
letterlist
make 'a key
do [make 'n key make 'a sentence
     :a :n] until :n = '\0d
print :a
```

   Try to use recursion to do the same job. You may want to incorporate this idea into Younglogo.

4. You can now write procedures to print out sequences according to given rules. As examples, make procedures which produce:

```
1, 3, 5, 7, ...
56, 66, 76, 86, ...
1, 2, 4, 8, ...
1, 1, 2, 3, 5, 8, 13, ...
```
(the Fibonacci sequence)

# Chapter 15
# Boolean Primitives

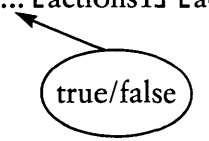
## The Primitive not

In the procedure `distance` which you tried in Chapter 14, the primitive `not` was used in the following way:

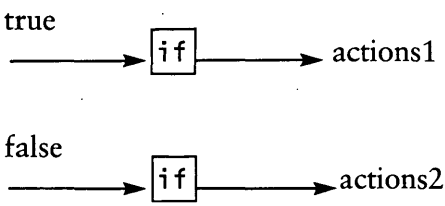
```
if not numberq :move [stop]
```

The command `not` is an example of a Boolean primitive; these primitives are used to carry out 'logical operations'. We recall that the command:

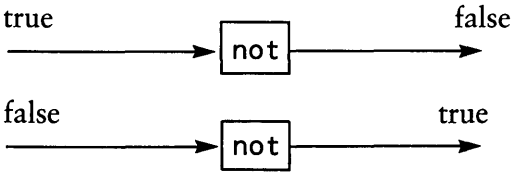
```
if ... [actions1] [actions2]
```




requires either `true` or `false` as its input, as shown below.




The command `not` requires either `true` or `false` as its input, and it also outputs either `true` or `false`; if the input to `not` is `true`, the output is `false`, and vice versa.




To see how this works, try:

```
print not listq [hello there]
```

This will print `'false`, because `[hello there]` is a list, and so the output of `listq [hello there]` is `true`.

In the procedure `distance`, `not` was used to do something if the box `'move` did not contain a number. For example, suppose the box `'move` contained K:


Then the test `numberq :move` would give `false` as its output, but `not numberq :move` would give `true`. So:

```
if not numberq :move [stop]
```

would stop the procedure of which it was part, and return control to the procedure which called it.

## both and either

There are four Boolean primitives; the others are `both`, `either` and `xor`. The command `xor` will be mentioned later; the other commands express the same meaning as the English words from which they are derived. Each of these commands requires *two* inputs, which must be either `true` or `false`.

Their behaviour is as follows:

- The primitive `both` returns `true` if *both* its inputs are true, and `false` otherwise.
- The primitive `either` returns `true` if *either* (or *both*) of its inputs is true, and `false` otherwise.

Each of the commands `both` and `either` is a *prefix* command; it must be written before its inputs. These commands enable you to test combinations of conditions at the same time. For example, the line:

```
if both :a > 0 :a < 10 [stop]
```

stops the procedure if $0 < a < 10$, while the line:

```
if either :a > 10 :b < 0 [stop]
```

stops the procedure if either $a > 10$ or $b < 0$ or both.

The commands `both` and `either` have alternative *infix* forms which may be easier to use. The infix form of `both` is `&`, and the line:

```
if :a > 0 & :a <10 [stop]
```

means the same as:

```
if both :a > 0 :a < 10 [stop]
```

The infix form is likely to be easier for most people to read and to think about, especially because the symbol & is often used as an abbreviation for *and*, and so is a synonym for `both`.

The infix symbol for `either` is not so intuitive; it is `|`. We can write the line:

```
if either :a < 0 :b > 10 [stop]
```

or alternatively as:

```
if :a < 0 | :b > 10 [stop]
```

### xor

The command `xor` is an *exclusive* OR; it returns `true` if *either* of its inputs is true, but not both. An example of using `xor` is given in the primitive section of the RM Logo book.

## Commands using true or false

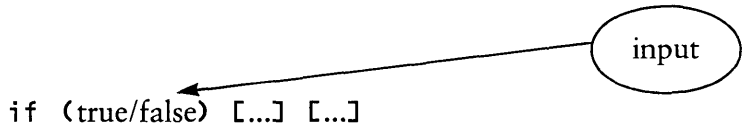
In this book, we have used several commands which *output* either `true` or `false`. They are:

```
>
<
=
listq
numberq
wordq
both
either
not
```

A number of other commands which output `true` or `false` are given in the RM Logo book. These commands include:

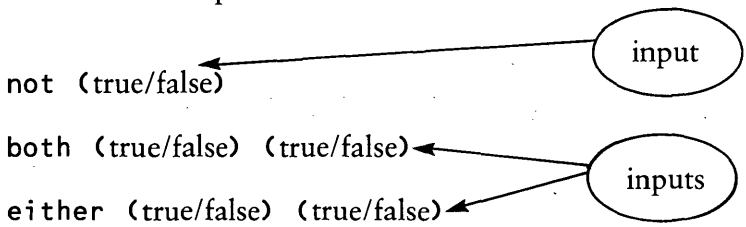
- `emptyq` which tests a word or list to see if it is empty
- `equalq` which is a prefix form of the equals sign
- `memberq` looks to see if its first input is a member of the list which is its second input.

There are also several commands which need `true` or `false` as inputs. These include the control commands:




```
do [...] until (true/false)

while (true) [...]
```

and the Boolean primitives:




The Boolean primitives produce *results* which are also `true` or `false`.

## Building your own Tests

A command which requires `true` or `false` as its input does not care where that input comes from. It may come as the output of another command, or you can type it in yourself, as the *word* `'true` or `'false`. The two lines:

```
do [Babylogo] until (1 = 2)

do [Babylogo] until 'false
```

have exactly the same effect; Babylogo is carried out.

Because you can type in `'true` or `'false`, Logo enables you to build your own test procedures; you can devise tests that were not provided as primitives. For example, the primitive `numberq` tests whether a variable is a number; however, no primitive is provided to test whether a variable is a *whole* number. The procedure given below will do this. It looks to see whether the whole number part of `:x` is equal to `:x` itself; if so, `:x` is a whole number.

```
intq 'x
if (int :x) = :x [result 'true]
        [result 'false]
```

As another example of a home-made test, try the following procedure to test whether a number is even:

```
evenq 'n
if (remainder :n 2) = 0 [result 'true ]
        [result 'false]
```

## Projects

1. Make a procedure to check whether one number is divisible by another.

2. Make procedures to test whether a number is prime. Print out a list of prime numbers.

# Chapter 16
# Multiple Turtles

## Introduction

You first used multiple turtles in Chapter 8. Your increasing command of Logo will now make it easier to create and control multiple turtles. Try these procedures:

```
setup 'n
make 'x 1
repeat :n [tell :x right :x*360/:n
        setc :x+1 setpc :x+2
        make 'x :x+1]
tellall :n

tellall 'n
make 'x 1
make 't []
repeat :n [make 't putlast :t :x
        make 'x (:x+1)]
tell :t
```

The procedure `setup` creates *n* turtles facing out at equal angles from the centre of the screen. `tellall` creates a list of the numbers of these turtles, and tells all the turtles in the list to be ready to carry out to commands. Now if you do:

```
setup 8
arcr 20 180
```

all eight turtles will draw semicircles at the same time. In fact, they all use a little bit of the computer's time, one after another. It happens so fast that all the turtles seem to draw together.

The primitive `putlast` adds an expression to the end of a list, and so is used to build up the list of turtles which are to be commanded. `putlast` will be used again in Chapter 18.

## Project

1. Set up eight turtles in a line at equal distances from each other. Make them all draw squares which touch one another. Fill the drawing screen with a *tessellation* of touching squares.

   Experiment with other shapes. How many of them tessellate (cover the screen leaving no gaps)?

# vanish and turtles

The command `vanish` will make all the turtles you have created disappear. To command them again, you must recreate them with `tell`.

Try the `turtles` command. It returns a list of the turtles which you are commanding at the moment.

RM Logo has given turtle number 1 a name! It is called Seymour, presumably after Seymour Papert who developed the language. In fact, it responds to either `tell 1` or `tell 'seymour`.

You can even name your own turtles with commands such as:

```
tell 'fred
```

Then the turtle knows its name is `'fred`, and will respond to such commands as:

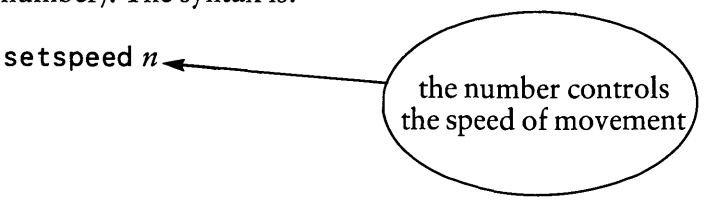
```
tell 'fred arcr 20 180
```

## Parallel Processing

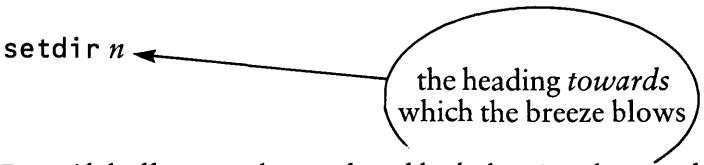
Remove all the turtles except number 1 with `vanish` and `tell 1`. Then type the command:

```
setspeed 3
```

The turtle moves forward at a steady speed, and goes on moving forward at that speed whatever else you make it do at the same time. Build a procedure for continuously drawing squares, and use it with `setspeed 3` (or some other number). The syntax is:




It is as if the turtle were blown up the screen by a gentle breeze as it draws. You can control the direction of the breeze with:




Beautiful effects can be produced by balancing the speed of the turtle against the speed of the wind. Your square was probably drawn too fast for interesting effects to show, but the procedures below will give you some idea of the possibilities.

```
cirwind
setdir 200
setspeed 6
circle

circle
forward 1 right 2
circle
```

Notice that when you use `setspeed` to produce a breeze, Logo automatically goes into `wrap` mode.

Now slow the turtle down. A useful way of doing this is to make the computer count silently to itself before doing the next command. Amend `circle` to this:

```
circle
forward 1 right 2
make 'x 1
repeat 5 [make 'x :x+1]
circle
```

Nothing shows on the screen, but the turtle counts silently to 5 before drawing the next bit of its circle, and this makes it go slower and appear to be battling against a gale.

## Projects

2. Swallows! This project may give you ideas for many other beautiful effects. Imagine six swallows trying to fly in circles from a central starting point, but they are blown off course by a breeze. Use `setup 6`, `setspeed 0.3` and `circle` to produce this effect. Experiment with altering the speeds of the swallows and the wind.

3. Build a Red Arrows flying display. It is possible to give each turtle its own windspeed and wind direction. For example, try this procedure:

```
start2
tell 2
setpos [50 0]
setdir 45
setspeed 6
```

# Chapter 17
# More About Variables

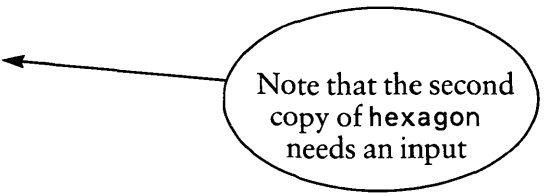
## Polygons with Recursion with Variables

In Chapter 7 you drew some polygons and stars using recursion; for example:

```
hexagon
forward 50
right 60
hexagon
```

You can combine this with the use of a variable to draw a hexagon of any size:

```
hexagon 'side
forward :side
right 60
hexagon :side
```




Here is an all-purpose polygon, star and circle drawing procedure.

```
poly 'side 'angle
forward :side
right :angle
poly :side :angle
```

# Polyspirals

Polyspirals are like polygons, except that they grow.

```
polyspi 'side 'angle
forward :side
right :angle
polyspi (:side + 2) :angle
```

## Projects

1. What different shapes will `poly` draw?

2. Try `polyspi` with different angle inputs. Some of the best designs are produced by angles near, but not equal to, the angles that make polygons and stars. For instance, try:

   ```
   polyspi 1 120
   polyspi 1 118
   polyspi 1 122
   ```

3. Try some variations on `polyspi`, such as:

   ```
   polyspi2 'side 'angle 'inc
   forward :side
   right :side
   polyspi2 (:side + :inc) :angle :inc
   ```

   In this procedure, `inc` stands for *increase*. Also use a `dec` to make the polyspi spiral inwards.

4. `inspi` is a variation which changes the angle, rather than the side (the names `poly`, `polyspi` and `inspi` are traditional in Logo circles).

   ```
   inspi 'side 'angle
   forward :side
   right :angle
   inspi :side (:angle + 10)
   ```

Try `inspi 10 33` and others. Amend `inspi` to use an `inc`, so that you have:

```
inspi 'side 'angle 'inc
```

Experiment with the effects obtained by changing the values of the variables.

## Local Variables

Logo uses two types of variables: *local variables* and *global variables*. To prevent surprises in the ways in which programs behave, you need to know how they differ.

A *local variable* in Logo is a box which belongs only to the procedure in which it occurs. Any other procedures in your workspace cannot find that box or look inside it. This fact is very important, because it means that different procedures can use the same names for variables which mean different things. To see how this works, try these procedures.

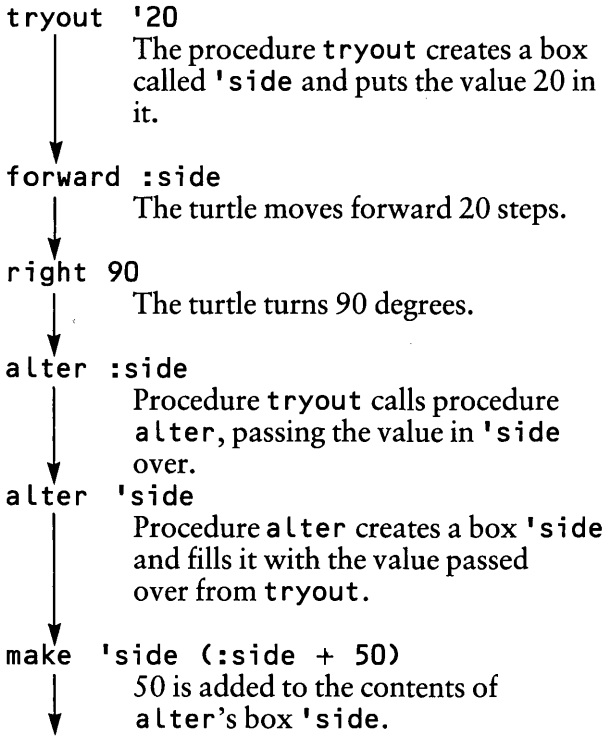
```
tryout 'side
forward :side
right 90
alter :side
forward :side

alter 'side
make 'side (:side + 50)
forward :side
right 90
```

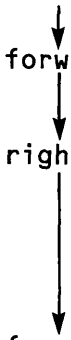
Do `tryout 20`. Were you surprised that the third line you drew was of length 20 rather than 70? In procedure `alter`, `:side` was increased from 20 to 70, but procedure `tryout` drew the third line, and in the `tryout` procedure, `:side` stayed at 20. Changing `:side` in `alter` did not change `:side` in `tryout`.

Here is how the variables work in the two procedures. When the name of a variable appears in the *title line* of a procedure, Logo creates a box to hold the variable. The box only belongs to that particular procedure — no other procedure can find it or look in it. It is *local.*

So in the title line `tryout 'side`, a box whose name is `'side` is created for `tryout`. Similarly, in the title line of `alter 'side`, a box whose name is `'side` is created for the procedure `alter`. These boxes have the same labels, but they are different boxes, and they can contain different values.

If you type `tryout 20` then this is what happens as the procedure runs:





`forward :side`
The turtle turns forward 70 steps.

`right 90`
The turtle turns 90 degrees to the right.

Control returns to `tryout` as there are no more commands in `alter`.

`forward :size`
The turtle moves forward 20 steps.
Procedure `tryout` did not copy the value 70 in `alter's` box `'side`.
The value in `'side` remains 20 in procedure `tryout`.

## Levels

It would be nice to be able to draw `polys` in different colours without having to change the procedure each time. Here is how to do this; start by typing in these procedures:

```
poly 'side 'angle
setpc :colour
repeat (repts :angle)
        [forward :side right :angle]

repts 'angle
result (360 / :angle)
```

When you run `poly`, Logo will complain that `:colour` does not exist. And it is quite right to complain — you have not created a variable whose value is `:colour`. You can fix this bug in one of two ways. First, you already know how to put the variable whose value is to be `:colour` in the title line of a procedure:

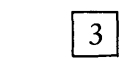
```
poly 'side 'angle 'colour
```

This creates a box for the variable whose name is `'colour` in `poly`, and so makes `'colour` a local variable in `poly`. If you fix the bug in this way, you will have to type in a number for `'colour` each time you run the procedure.

A second method of fixing the bug is to create a box for `'colour`, and put a value in it, direct from the keyboard. The number you put in the box stays there until you change it. You use `make` to create this box. Type:

```
make  'colour 3
```

This creates a *keyboard level* box:



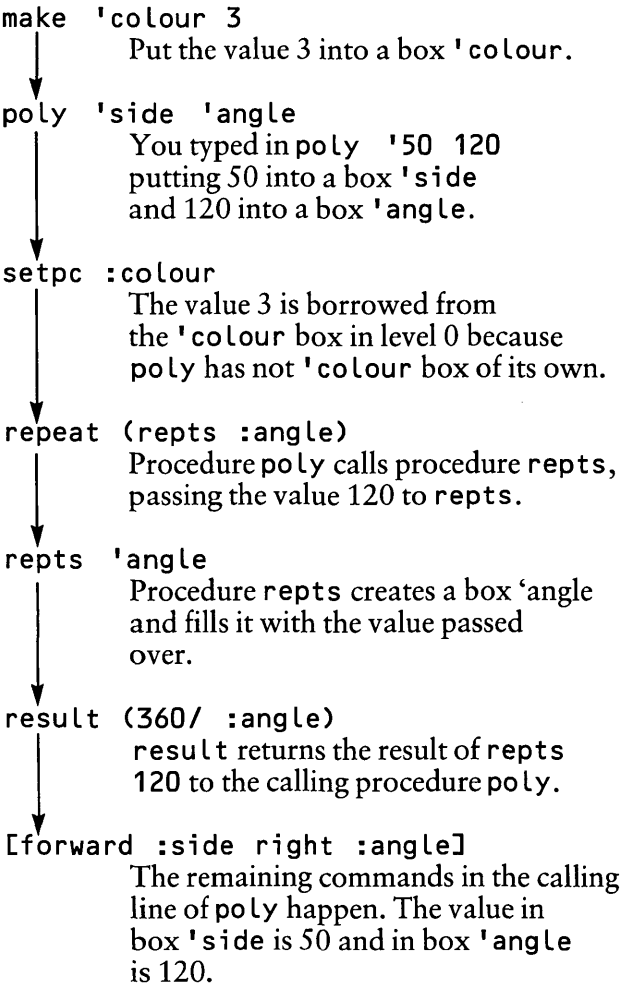

We can think of procedures as being at *lower levels* than the keyboard. For example:

| | |
|---|---|
| Level 0 | keyboard |
| Level 1 | `poly` |
| Level 2 | `repts` |

When you make a box labelled `'colour` direct from the keyboard, it is a level 0 box, and it can be borrowed by any procedures at lower levels which do not have their own `'colour` boxes.

`poly 'side 'angle` is a procedure at the next level down (level 1). You call it from the keyboard by typing a command such as `poly 50 120`. `poly` itself calls the procedure `repts`, which works out how many repetitions of a side of the polygon must be drawn. `repts` is at Level 2, because it is called by the level 1 procedure `poly`. The diagram shows what happens when you run these procedures.

If you type `poly 50 120` this is what happens as the procedure runs. (Remember that a value must be put into a box `'colour` first).




The box labelled `'colour`, created at level 0, can be used by every procedure in the workspace, because whenever a procedure is called, that procedure is below level 0. The

box `'colour` at level 0 is often called a *global variable*, because it can be borrowed by all level 1 procedures, and by all the procedures which they call. Hence, all the procedures in the workspace can borrow a level 0 variable; that is why it is called a global variable.

Now you have made a `'colour` box and put the value 3 into it; the procedure `poly 'side 'angle` will run and draw using the colour 3.

Other procedures can make use of the value you have given to the global variable `'colour`. Try:

```
sqcol 'side
setpc :colour
repeat 4 [forward :side right 90]
```

Logo will not complain about this, because `'colour` is at level 0 and so is a global variable; it has the value 3. You can also make global variables from within procedures. Because Logo can only create boxes for local variables when you write the names of the variables in the title line of a procedure, it follows that if you use make to create a new variable within a procedure, the procedure does not have a box for it, and so Logo makes a new box at Level 0.

For example, if you do:

```
sq 'side
make 'shade 4
setpc :shade
repeat 4 [forward :side right 90]
```

then `'shade` is a new name, so it becomes the name of a global variable and can be used by any procedure. When you have run `sq`, type:

```
print :shade
print :side
```

The variable `'shade` is a global variable, accessible from the keyboard, so Logo replies to `print :shade` with `4`. `'side` is local to the procedure `sq`, and level 0 cannot look into the `'side` box belonging to `sq`. Logo complains that you have not created a variable called `'side`. It is inside the procedure `sq`, so it can never be accessible to a keyboard command to print its value.

## Changing Colours

In many of your drawings, you must have wanted to change colours automatically within a procedure, so as to produce a multi-coloured effect. A `polyspi` would look very handsome if successive sides were in different colours. You can now arrange for this to happen — here is a possible colour changing procedure:

```
chcol
make 'col (:col + 1)
if (:col = 16) [make 'col 1]
setpc :col
```

This procedure will not run by itself. When you ask it to `make 'col (:col + 1)` it needs to know what number `:col` represents, and as you have not told it, it complains. Thus, you must be sure to make a higher level `col` box, either by making it directly from the keyboard at Level 0, or you can make the `colourbox` within a procedure which calls `hcol`. Here is one possibility:

```
tricol 'side
make 'col 1
repeat 3 [forward :side right 120 chcol]
```

### Project

5. Do some colour changes within procedures such as `polyspi`. You might also try random colour changes.

# Chapter 18
# Building Up Lists

## A Random Sentence Generator

Projects generating sentences enable a learner to build up random sentences, and to see whether these sentences seem right; this may help the learner to focus on the grammatical features of writing. In this chapter we shall build a very simple procedure of this type. At each recursion, Nimbus asks you to teach it a noun and a verb, and it then makes a random sentence from the lists of nouns and verbs built up so far.

The basic structure is:

```
message
noun
verb
makesent
message
```

`message` is a *superprocedure*; its only function is to call the procedures `noun` and `verb`, which ask you to teach it words, and then the procedure `makesent`, which makes and prints the random sentence. Recursion is then used to repeat the procedure `message`, so that the lists of nouns and verbs grow every time `message` is called.

The procedures `noun` and `verb` use the command `readlist` to obtain a word as input from the keyboard and store it temporarily as a list in `'inp`.

```
noun
type [Teach me a noun]
make 'inp readlist
make 'nounl putlast :nounl first :inp
```

```
verb
type [Teach me a verb]
make 'inp readlist
make 'verbl putlast :verbl first :inp
```

In order to run these procedures, you will need to create the boxes `'nounl` and `'verbl`, and put empty lists in them. You can do this from the keyboard with:

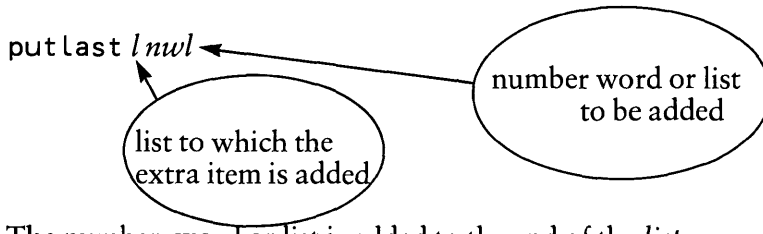
```
make 'nounl []
```

and

```
make 'verbl []
```

## putlast and putfirst

The command `putlast` has been used to add new words to the lists `'nounl` and `'verbl`. The syntax is:




The number, word or list is added to the end of the *list* which is the first input. So, if:

```
'nounl
```

contains the list `[dogs cats]`, and `[rabbits]` is contained in `'inp`, then:

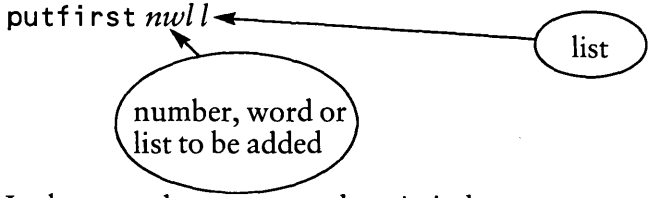
```
make 'nounl putlast :nounl (first :inp)
```

constructs the list:

```
[dogs cats rabbits]
```

in `'nounl`. It is necessary to use `first :inp`, rather than just `:inp`, to be tacked on to the end of `'nounl`, because `:inp` is the list `[rabbits]`, and we only want the word `'rabbits` to be added to the list.

There is another command, `putfirst`, which can be used to add an expression at the *beginning* of a list. Its syntax is:




In the procedures `noun` and `verb`, it does not matter whether the new words which the child teaches Nimbus are added at the beginning or the end of the lists of nouns and verbs, because the next move will be to select random words from the lists. In other programs, it may be extremely important to know at which end of a list a new expression is put.

## Choosing a Random Member of a List

The following procedure chooses a random word from the list of words stored in the list `'listn`. The word is returned as a `result`.

```
randwd  'listn
if emptyq :listn [result []]
result :listn # pick count :list
```

The primitive `count` counts the number of elements in a list, or the number of letters in a word. `pick` chooses a random whole number in the range 1 to the number input. We then have to pick out the chosen element of the list.

RM Logo has the operator # which selects an element from a list, for example `:noun1 # 3` gives the third word in the list `:noun1`

## Making the Random Sentence

To build the random sentence, we choose a random word from `'nounl` and a random word from `'verbl`, enclose each one in a list, and then join the two lists together into a single list by using the primitive `sentence`, as follows:

```
makesent
make 'subj randwd :nounl
make 'verb randwd :verbl
make 'sent sentence (sentence 'the :subj) :verb
say :sent
```

Now the procedure `message`, given at the beginning of this chapter, can be used to put together the procedures into the complete program. A superprocedure, `writer`, can be used to avoid you having to create `'nounl` and `'verbl` at the keyboard at the start of a session:

```
writer
make 'nounl []
make 'verbl []
message
```

### Projects

1. Improve `message`. You might want longer sentences, or you might want the user to be able to to view and change the word lists. You might want to avoid having to teach Nimbus new words at every round.

2. Write some (pseudo-Japanese) Haiku poetry using random choices from word lists. A Haiku poem has three lines, and is structured like the following example:

   Late cool showers fall.
   Tiny blossoms open and
   greet the new warm sun.

   The structure is:

   Adjective adjective noun verb.
   Adjective noun verb *and*
   verb *the* adjective adjective verb.

   Haiku poems are always about nature and its beauties, so you will need lists of appropriate nouns, adjectives and verbs to go in each place in the poem.

   You could also work out how many different Haiku poems could be made from the word lists which you have supplied to Logo.

# Chapter 19
# More About Recursion

## Drawing a Tree

A very simple procedure draws two branches of a tree, and finishes with the turtle facing the way it started.

```
vee 'length
left 45
forward :length
backward :length
right 90
forward :length
backward :length
left 45
```

We can make a tree by drawing a smaller `vee` at the tip of each branch, and continuing to do this recursively.

```
tree 'length
left 45
forward :length          <-------------------- draw a branch
tree :length/2 <-------- draw small tree at end of branch
backward :length
right 90
forward :length          <-------------------- draw a branch
tree :length/2 <-------- draw small tree at end of branch
backward :length
left 45
```

Unfortunately, this remains in the left-hand branch, and continues until the computer runs out of memory. To see why, think about how the procedure `tree` can stop.

We need to stop the recursion at an appropriate point when the branches are short.

Put in the stop line:

```
if :length < 2 [stop]
```

after the title line, so that you have:

```
tree 'length
if :length < 2 [stop]
left 45
forward :length
tree :length/2
backward :length
right 90
forward :length
tree :length/2
backward :length
left 45
```

The best place to put a stop line in recursion is usually after the title. Notice that the command `stop` stops execution of the procedure in which it occurs, and returns control to the procedure which called it. If you ever want to stop completely and return control to the keyboard, use the command `escape`.

After this change, the whole tree neatly draws itself.

## Tracing through Programs

RM Logo has some facilities for tracing the progress of your programs, which may help you to follow what is going on in recursion. Type:

```
walk 'tree
```

Then when you execute `tree`, Logo will print out what it is doing at each line, and pause for you to press , <ENTER> before carrying out the command. To get rid of `walk`, use: `unwalk 'tree`

## Variations on tree

You can make one branch of `vee` longer than the other. You can vary the angle between branches (use a variable to do this).

You can use different stop rules. The one below uses the variable `depth` — each branch grows a tree of depth one less than the previous branch.

```
tree1 'length 'depth
if :depth = 0 [stop]
left 45
forward :length
tree1 :length/2 :depth-1
backward :length
right 90
forward :length
tree1 :length/2 :depth-1
backward :length
left 45
```

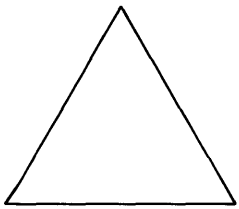
## Snowflake Curves

A Snowflake Curve, as well as being very beautiful, has the strange mathematical property that if it could be drawn completely if would be of infinite length, while fitting in to the area of a basic hexagon which encloses the snowflake. Of course the computer cannot actually draw a curve of infinite length, but Logo enables us to draw a good enough approximation to see how it would work ideally, and so help us to visualise a curve of infinite length fitting in to a finite area.

Curves of this type were discovered late in the nineteenth century, and for some time they were regarded as tiresome mathematical paradoxes. Drawing curves of this type, and working out the lengths of their perimeters, may enable you to realise how the perimeter of a shape may be greatly

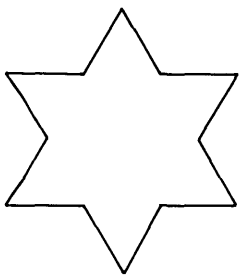
increased without making much difference to the area. People often find this concept very difficult, because we have little experience which would enable us to form the idea.

We start building the snowflake by drawing an equilateral triangle of convenient size; a good length for each side is either 81 or 192 turtle steps, because we are going to do a lot of dividing by 3.

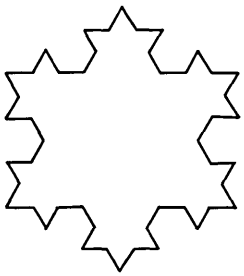


The way to build the Snowflake is to replace the middle third of each side by a 'tooth' which sticks out from the side:

This produces a new shape, which looks like this:



It has a longer perimeter — putting the 'tooth' on each side increases the length of that side by one third. In the following example, the middle third of the line was of length 27, and has been replaced by 2 lines of length 27 each. The length of the side of the triangle has been multiplied by 4/3. The Snowflake curve is drawn by continuing to do this — whenever we see a line, we replace the middle third of it by a 'tooth', and so multiply its length by 4/3. The next step is shown in the following diagram:



We could, in theory, go on doing this for ever, and recursion is an appropriate way of programming it. However, in practice a stopping method is needed; we shall stop when the size of a 'tooth' is less than a limiting short length such as 1 unit.

The basic pattern for drawing a tooth is:

```
tooth 'length
forward :length/3
left 60
forward :length/3
right 120
forward :length/3
left 60
forward :length/3
```

This procedure can be adapted as the basis for a recursive procedure to draw a side of the snowflake, like this:

```
side 'length 'lim
if :length < :lim [forward :length stop]
side :length/3 :lim
left 60
side :length/3 :lim
right 120
side :length/3 :lim
left 60
side :length/3 :lim
```

Try typing `side 81 2`

The whole snowflake can now be drawn:

```
snowflake 'length 'lim
side :length :lim
right 120
snowflake :length :lim
```

You could experiment drawing snowflakes with different stopping conditions, such as:

```
snowflake 81 27
snowflake 81 9
snowflake 81 3
snowflake 81 1
```

## Project

1. The C-curve is another recursively drawn curve. The idea here is to replace a line by an 'elbow' consisting of two lines drawn at right angles.

   The basic procedure to replace a line of given length by an 'elbow' is:

```
elbow 'length
left 45
forward :length / sqt 2
right 90
forwaard :length / sqt 2
left 45
```

   Incorporate this into a recursive procedure with a stop when `:length < :limit`. A possible method follows:

```
elbow 'length 'lim
if :length < :lim [forward :length stop]
left 45
elbow :length / sqt 2 :lim
right 90
elbow  :length / sqt 2 :lim
left 45
```

The 'Dragon' curve is a similar curve, which has elbows pointing alternately on the two sides of the previous line. Experiment with this.

# Chapter 20
# List of Logo Primitives

In this chapter, the Logo primitives which have been used in this book are listed in groups (not in alphabetical order). These primitives are the greater part of the complete set of RM Logo primitives. A complete list will be found in the RM Logo reference book and the RM Logo reference card.

The commands are grouped according to their function.

## Graphics Commands

## Words and Lists



## Conditionals



## Control

## User Input and Output



## Variables



## Information and Files

What a wonderful book! I can now make my Nimbus a part of my imagination. I'll draw castles with flags fluttering in the breeze, and draw the treasures that are stored within.

I'll jump in a truck and drive to the sea and watch the yachts racing. When the wind gets up they'll blow off course, and finally be blown from the screen.

When I look up in the sky I think about the planets and I can map them out on my Nimbus with Logo. I can come back to earth and draw birds and trees and a sun that smiles all day long.

Logo earns its keep: it holds lots of information for me so I can always find it.

Logo is magical: I can draw what I please in the colour I choose.

For instance I can draw the front cover picture in lots of colours and watch it go on growing, just by typing: `swallow`

```
swallow
cs
star 6 0
tell [1 2 3 4 5 6] setx -60
setdir 90 setspeed 0.3
forever [fd 1 1t 2]

star 'number 'angle
if eqq :number 0 [stop]
tell :number setpc :number setc 8 + :number seth :angle
star ( :number - 1 ) ( :angle + 60)
```