Logo
Extensions and examples

ACORNSOFT

Acknowledgements

Most of the examples in this book were produced by Nick Benton. Some of the ideas were suggested by John and Audrey Laski.

Copyright © Acornsoft Limited 1984

All rights reserved

First published in 1984 by Acornsoft Limited

No part of this book may be reproduced by any means without the prior permission of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act, or for the purpose of review, or in order for the software herein to be entered into a computer for the sole use of the owner of this book.

FIRST EDITION

ISBN 0 907876 77 3

Note: British Broadcasting Corporation has been abbreviated to BBC in this publication.

Acornsoft Limited, Betjeman House, 104 Hills Road,
Cambridge CB2 1LQ, England. Telephone 316039

Contents

1 Introduction

This manual describes the contents of the tape and disc supplied with Acornsoft Logo for the BBC Microcomputer.

There are two types of program: extensions and examples.

Extensions add features to Logo, for example to control a particular printer, or to support multiple turtles. When an extension has been loaded the new features can be used by any Logo program.

The examples are written in Logo and generally show some of the less obvious things that Logo can do.

1.1 The programs on tape

The tape starts with the extensions in alphabetical order, followed by the examples. However, when an example uses a particular extension the code of that extension is repeated after the example since this simplifies the loading of the example.

The order of files on the tape is as follows:

BUGGY
CALC
CLEPSON
EPSON
JESSOP
MOS
MULT
OLIV
PROP
SECT
VALIANT

EDSHAPE
MOS (used by EDSHAPE)
GRAVITY
SECT (used by GRAVITY)
HAND
LETTERS
SECT (used by LETTERS)

LOGIC
LOGOP
MAP
MAPIRE (data on Ireland)
MAPSCOT (data on Scotland)
MAZE
MAZEDAT (a picture of a trial maze)
MIRROR
MULT (used by MIRROR)
PLANTS
SCATTER
STORY
TOOLBOX
WAFFLE
PROP (used by WAFFLE)

2 Logo examples

A few of the examples (namely EDSHAPE, LETTERS, MAP and SCATTER) may be of use in your Logo projects.

The others should be fun to use for a time, but they have served their purpose best if you find them a useful source of ideas for your own projects.

You may choose to add to or improve the programs supplied, or you may use them as a source of ideas on how to achieve some of the effects you want.

The examples supplied are as follows:

EDSHAPE – editing character shapes
GRAVITY – moving turtles
HAND – left and right game
LETTERS – for posters
LOGIC – artificial intelligence
LOGOP – a pattern for floor turtles
MAP – drawing maps
MAZE – edit and solve mazes
MIRROR – turtles reflecting each other
PLANTS – the computer learns
SCATTER – spreading objects
SPIRAL – pattern drawing
STORY – help the computer write a story
TOOLBOX – a conversational program
TRI – space filling by recursion
WAFFLE – the computer gossips

2.1 Loading examples

Examples are loaded as for any other program. However, as explained below, it is safest to restart Logo before loading an example. Thus:

```
*LOGO
LOAD "LOGOP
```

All the examples will run automatically when loaded. If an extension is required this will also be loaded automatically provided that it is on the same disc or follows the example on the tape.

Several Logo examples change the Logo system by the extensions they load, by burying procedures or by redefining primitives. If these changes are still in effect when another example is loaded it may not work correctly. For this reason it is always desirable to be careful to restart Logo before running another program.

EDSHAPE

Introduction

This is a utility program which allows you to define and edit user-definable characters. These can then be incorporated into your own programs.

The program is simple to operate because it uses only 5 keys (the cursor arrows and RETURN) to control everything.

How to use the program

When the program is `LOAD`ed, you will be shown an 8x8 grid of boxes with four separate boxes to the right, labelled `CHAR`, `DEFINE`, `EDIT` and `END`. In the top left-hand corner of the large grid you will see an arrow. This is the cursor and it can be moved about with the arrow keys. At the top right of the screen you will see a number and a character. This is the 'current character' – one of the user-defined characters in the range 224 to 249.

Designing a new shape

To design a new shape, move the cursor about the grid and use the RETURN key to 'flip' the squares (ie change an empty square to a filled-in square, and vice versa). When a design looks right, you can copy it into the current character (shown at the top right of the screen) by moving the cursor into the `DEFINE` box and pressing RETURN. The cursor changes into an 'hourglass' and the `VDU 23` codes needed to define the shape in a program are displayed at the bottom of the screen.

Changing the current character

To choose which user-defined character is to be the current character, move the cursor into the `CHAR` box and press RETURN. The cursor will become a double-headed vertical arrow. Pressing the up arrow key will cause the current character number to be increased, whilst pressing the down arrow will decrease it. When you find the correct character, you can select it by pressing RETURN. This restores the cursor's arrow shape.

Editing the definition of the current character

To edit an existing character definition, the definition of the current character can be copied onto the defining grid. This is done by moving the cursor to the `EDIT` box and pressing RETURN. The cursor changes into an hourglass to indicate a short delay whilst the definition is read, converted into the program's internal format and displayed on the grid.

As the definition is read the `VDU 23` codes for the current character are displayed at the bottom of the screen. The shape on the grid can then be edited in the same way as a new shape is defined.

Leaving the program

If the cursor is moved into the `END` box and RETURN pressed, the program will end. It can be restarted by typing `START` (assuming you haven't altered the program or its data in the meantime).

Over to you

Once you have defined a character, such as number 225, you might like to use it as the turtle shape by, in this case, `SETSH 225`.

There are several ways in which you might like to extend the program as presented here. One of the most obvious is to write a program which allows the definition of several characters on screen at once. This would make the design of multi-character shapes (such as those used in the program to draw the hourglass) much simpler. Other possible facilities include displaying the characters in different screen modes or colours.

GRAVITY

Introduction

This program allows you to control a turtle moving in a plane under the laws of Newtonian mechanics. You may try to land the turtle within a small region, or you may have a course which you can try to follow. You can also decide on the acceleration due to gravity and whether or not the turtle should leave a trail.

How to use the program

When the program is `LOAD`ed, you are shown a summary of the instructions and then the screen goes into graphics mode when the Space Bar is pressed. In the text area you will be asked whether or not you want a 'course' on the screen. If you press 'Y' in response to this question, a curving track will be drawn, starting on the left of the screen and ending on the right. This course is only for guidance as nothing actually happens if you stray off it. If you press anything except 'Y' in response to the course question, the landing pad will be drawn at the bottom of the screen.

The next thing you have to decide is how much gravity you want. This is a number which should be between 0 (no gravity) and 1 (gravity exactly balances the turtle's rocket). Enter the number and press RETURN.

You will then be asked whether or not the turtle should leave a trail. If you press 'Y', the turtle will draw a line of dots behind it to mark its path. If you press any other key, no trail will be left.

Having answered the questions, the program proper will begin. Use the 'Z' key to rotate the turtle left and the 'X' key to rotate it right. To turn the turtle's rocket on, press ':' (the state of the rocket is printed in the text area). To turn the rocket off, press '/'.

If you hit a wall, a message will be printed in the text area to tell you that you have crashed and the game will end. There are two exceptions to this: if you have a course and you pass through its end (at the right of the screen) you will be congratulated. If you do not have a course and you hit the landing pad (at the bottom of the screen) you will be given a message of success or failure depending on your vertical speed when you landed.

HAND

Introduction

This is a game designed to improve spatial awareness and recognition of left and right. A picture of a man with a flag in one hand is displayed and you have to press (as quickly as possible) a key corresponding to the hand in which the flag is held.

How to use the program

When the program has been loaded, instructions will be displayed on the screen. When you are ready, you can start the game by pressing the Space Bar.

You will be shown a picture of a man facing either towards or away from you and rotated by 0, 90, 180 or 270 degrees. You can tell which way he is facing because his face is visible when he faces towards you. In one of his hands, he will be holding a flag and your job is to decide which of his hands it is in. If you think it is in his left hand, press the cursor left key. Similarly, if you think it is in his right hand, press the the cursor right key.

When the game has finished it can be restarted by typing `START`.

LETTERS

Introduction

This example allows you to create posters on the screen using letters of different sizes. It also provides some ideas to help you if you would like to create your own letters, numbers or symbols.

`LETTERS` will not work with floor turtles.

How to use the program

The example supports procedures called `A`, `B`, `C`, etc, up to `Z`. These draw the appropriate letter at the turtle position and heading with the current turtle pen, and then move the turtle to a position for the next letter.

The supporting procedures are:

`BS`	move turtle left by one character space
`SP`	move turtle right by one character space
`UC`	letters are to be drawn in upper case (Capitals)
`LC`	letters are to be drawn in lower case
`LF`	move down a line

`SETSIZE <size>` this sets the size for letters. Useful sizes are from 1 to 10 and the initial size is 3.

If you wish to put letters on a picture you have drawn, it may be best to save the picture using `SAVEPICT` then develop a procedure to place the letters as you want them.

To place letters on successive lines you should remember the start of each line, so that you can return to this point and then call `LF`. For example:

```
SETPOS [ -400 300]
MAKE "OLDPOS POS
C A T
```
writes `CAT` on the first line
```
SETPOS :OLDPOS
LF
MAKE "OLDPOS POS
S A T
```

writes SAT on the second line

```
SETPOS :OLDPOS
LF
MAKE "OLDPOS POS
O N
```

and so on.

If you wish to create your own letters or numbers in a similar way you will find that the procedures in the package use the extension SECT and the procedures

`ARC <radius> <angle> <width>` to draw an arc
`ARM <side1> <side2>` to draw an arm of a letter
`MR <fwd> <right>` to move without drawing
`OPA <fwd> <width>` to draw a pair of parallel lines
`PA <side1> <angle> <side2>` to draw a parallelogram
`TZ <side1> <angle> <side2> <side3>` to draw a trapezium

Each letter calls LETSTART before drawing and LETEND at the end to move to the next position.

LOGIC

Introduction

This program is a much simplified version of the logic programming language Prolog. In fact it resembles true Prolog to about the same extent that a turtle graphics package resembles a true Logo implementation.

The program allows you to input facts and rules about facts which are stored in a 'database' (a collection of data). These are then used to answer queries from the user. The program will cope with most of the simple database query applications used to introduce Prolog, but has none of the built-in functions or list processing capabilities of the real language.

How to use the program

When the program is loaded, you will be presented with a short summary of the main commands available and then the `LOGIC` program prompt `&..`

To clear the program's memory (removing any facts from the database), type `CLEAR` and press the RETURN key.

Formulating assertions

Now we can make some assertions. Assertions are usually of the form:

```
[<relationship> <individual> <individual>]
```

or

```
[<class> <individual>]
```

Here are some English assertions and the form in which they are required by the program:

English → *Logic*

Mary likes John. → `LIKES MARY JOHN`
Steak is served with chips. → `SERVED.WITH STEAK CHIPS`
Bill is a man. → `MAN BILL`
Fred's parents are Jim and Sheila → `PARENTS FRED JIM SHEILA`
The borogroves are mimsy. → `MIMSY BOROGROVES`

The reason why the relationships are put at the front (in so-called 'prefix form') is that the database is indexed on the first word of each assertion so as to speed searches. This representation also leads to a more consistent way of expressing assertions. You should notice that when a relationship name has more than one word (as in 'served with'), it has to be made into one word; usually by using an underline character or full stop to separate two English words.

Adding new assertions to the database

To actually add an assertion to the database, you use the built in command ADD:

```
ADD [[LIKES MARY JIM]]
ADD [[MALE BILL]]
ADD [[HUMAN BILL]]
ADD [[LIKES JIM FRED]]
```

The assertions are surrounded by double list brackets because some assertions can contain more than one clause (as we shall see later).

Listing the assertions

A list of all the assertions which have been entered can be obtained by using the command DISPALL (for DISPlay ALL). Alternatively, a listing of all the assertions under a particular index or indices can be obtained by the use of the command DISP, as in DISP "LIKES to obtain a listing of all the LIKES relationships or DISP [LIKES HUMAN] to obtain a listing of both the LIKES and HUMAN assertions.

Asking yes / no questions

So far, we have put information into the database and got a listing of it, but we have not actually asked any questions of the database. We can ask if a particular assertion is in the database by using the command DOES:

```
DOES [[LIKES MARY JIM]]    ('Does Mary like Jim?')
YES

DOES [[HUMAN BILL]]    ('Is Bill human?')
YES

DOES [[HUMAN MARY]]    ('Is Mary human?')
NO
```

The program operates on what is called a 'closed-world assumption'. That is to say that all true statements are either in the database or are derivable from statements in the database by rules which are in the database. Thus, the answer to the last question is NO because the program has not been told and cannot show that MARY is HUMAN.

The wildcard character

We may also want to know if a particular kind of assertion is in the database. For example, to ask 'Does Mary like anybody ?' We want to see if there are any assertions of the form:

```
[[LIKES MARY <anything>]]
```

where any word will match against `<anything>` in the query. This is done by using the underline character (under the £ sign on the keyboard) as a 'wildcard' character:

```
DOES [[LIKES MARY _]]
YES
```

```
DOES [[EATS JIM _]]
NO
```

Asking 'which' questions

It is far more likely that we will want to know 'Who does Mary like ?' than 'Does Mary like anybody ?'. To do this we need a way of 'getting back' the word which was matched with the wildcard character in the above example. This is done by using variables and the `WHICH` command:

```
WHICH [?X] [[LIKES MARY ?X]]
```
('Which x is such that Mary likes x?')
```
Answer is: ?X=JIM
No (more) answers.
```

```
WHICH [?X ?Y] [[LIKES ?X ?Y]]
Answer is: ?X=MARY, ?Y=JIM
Answer is: ?X=JIM, ?Y=FRED
No (more) answers.
```

Notice that whilst there can only be one answer to a `DOES` query, there may be many ways of satisfying a `WHICH` query, all of which are found by the system. If you don't want all the possibilities, use `ONE` instead of `WHICH`. This prints each answer as it is found and waits for you to press a key. If you press RETURN then it will carry on to find the next answer, otherwise it will stop.

A variable is one of `?V`, `?W`, `?X`, `?Y` or `?Z`. This is not as restrictive as it sounds since variables are always completely local to the assertion or query in which they appear.

We can also ask compound queries, that is to say queries having several different goals to be satisfied simultaneously. For example, using the following database:

```
ADD [[BIG LORRY]]
ADD [[BIG HOUSE]]
ADD [[COLOUR LORRY RED]]
ADD [[COLOUR HOUSE WHITE]]
```

We could ask 'What is big and white ?' like this:

```
WHICH [?X] [[BIG ?X] [COLOUR ?X WHITE]]
```

('which x is such that x is big and x is white ?')

```
Answer is: ?X=HOUSE
No (more) answers.
```

Assertions containing variables

So far, the only information which we have got out of the database has been explicitly entered – the system only knows that Mary likes Jim because we told it so. A more powerful kind of information can also be entered into the database which will allow the system to infer new facts without being explicitly told them. For example, one of the classic forms of syllogism is often stated as:

1 Socrates is human

2 All humans are mortal

therefore Socrates is mortal.

Assertion 1 is the sort we have met already. We could enter it as:

```
ADD [[HUMAN SOCRATES]]
```

Assertion 2, however, is of a different kind. We are introducing a general rule which could be stated: 'If x is human then we can deduce that x is also mortal' or, alternatively, as this: 'To solve the problem of who is mortal, first solve the problem of who is human'.

This rule would be added to our database as follows:

```
ADD [[MORTAL ?X] [HUMAN ?X]]
```

We could then ask the system 'Who is mortal ?' like this:

```
WHICH [?X] [[MORTAL ?X]]
Answer is: ?X=SOCRATES
Answer is: ?X=BILL
No (more) answers.
```

There is no relationship between the `?X` in the assertion and the `?X` in the question; we could equally well have used `?X` in the rule and `?Z` in the question, or asked a question with no variable in it, like this:

```
DOES [[MORTAL SOCRATES]]
```

In the same way as we can have compound queries, we can make compound assertions. For example, we might want to store some data on a family tree and to do this we may have a rule corresponding to '(x is the brother of y) if (x is male) and (the parents of x are v and w) and (the parents of y are also v and w)':

```
ADD [[BROTHER ?X ?Y] [MALE ?X] [PARENTS ?X ?V ?W]
[PARENTS ?Y ?V ?W]]   (type as one line)
```

Notice that this rule can make somebody his own brother (since he has the same parents as himself) but that this will only be the case for those males who have parent data stored:

```
ADD [[MALE FRED]]
ADD [[FEMALE SHEILA]]
ADD [[MALE JIM]]
ADD [[PARENTS JIM FRED SHEILA]]
ADD [[PARENTS SALLY FRED SHEILA]]
ADD [[FEMALE SALLY]]
WHICH [?X ?Y] [[BROTHER ?X ?Y]]
Answer is: ?X=JIM, ?Y=SALLY
Answer is: ?X=JIM, ?Y=JIM
No (more) answers.
```

So `JIM` is his own brother, but `FRED` isn't since we cannot show that he has the same parents as himself.

Recursive rules and the danger of infinite loops

Rules can be recursive (they can invoke themselves), but you have to be very careful about not allowing infinite loops to arise. As an example of a legal kind of recursion, consider a database concerning some blocks on a table:

```
ADD [[DON A TABLE]]   (A is directly on the table)
ADD [[DON E TABLE]]   (E is directly on the table)
ADD [[DON B A]]       (B is directly on A)
ADD [[DON C A]]       (C is directly on A)
ADD [[DON D C]]       (D is directly on C)
```

Now we want to define `SUPPORTS` so that 'x supports y if y is directly on top of x OR y is directly on top of some z such that x supports z'. This is quite legal:

```
ADD [[SUPPORTS ?X ?Y] [DON ?Y ?X]]
ADD [[SUPPORTS ?X ?Y] [DON ?Y ?Z] [SUPPORTS ?X ?Z]]
DOES [[SUPPORTS TABLE D]]
YES
```

(Notice how the OR part of the definition is implemented by having two rules starting with '`[[SUPPORTS ?X ?Y]`').

It would NOT be a good idea to define our second `SUPPORTS` rule like this:

```
[[SUPPORTS ?X ?Y] [SUPPORTS ?X ?Z] [DON ?Y ?Z]]
```

because the interpreter works from left to right trying to satisfy each goal, backing up when some goal fails. In this case it would respond to a query like:

```
WHICH [?X ?Y] [[SUPPORTS ?X ?Y]]
```

by first trying to find x and z such that x supports z. This would lead to a recursive call of the same rule which would carry on going deeper and deeper until the program was interrupted or ran out of stack space.

Infinite looping can occur quite simply:

```
ADD [[LIKES BILL JANE]]
ADD [[LIKES JANE ?X] [LIKES ?X JANE]]
```

This was intended to say that Jane likes anybody who likes her, but will run into problems because the program cannot decide whether or not Jane likes herself – if she does then she does but if she doesn't then she doesn't!

For example:

```
WHICH [X?] [[LIKES JANE ?X]]
```

will give

```
Answer is: ?X = BILL
```

and then go into an infinite loop which will continue until ESCAPE or CTRL ESCAPE is pressed.

Over to you

Obviously it is tempting to try and expand this simple system into a much fuller implementation of Prolog. This is not really practicable, given the constraints of memory and speed which are imposed by using Logo (Logo would, however, be a reasonable environment in which to test your ideas before translating them into machine code or a compiled language such as BCPL). There are some simple improvements which would add to the utility of the program (though probably at the expense of some execution time):

1 Add a NOT function. This would be very useful in queries and could be used like this:

```
WHICH [?X] [[LARGE ?X] [NOT HEAVY ?X]]  or
ADD [[EQUAL ?X ?X]]   (defines EQUAL)
ADD [[NE ?X ?Y] [NOT EQUAL ?X ?Y]]   (defines NOT EQUAL)
```

Note that negative information cannot be entered – the mere absence of the positive version has the same interpretation

2 Add some simple numerical predicates – a built in 'less than' relationship is not too difficult (and is very useful). The arithmetic ones require a little more thought because they have to work in both directions:

```
WHICH [?X] [[SUM 2 3 ?X]]   (which x is equal to 2+3?)
WHICH [?X] [[SUM 2 ?X 5]]   (which x is such that 2+x is equal
                             to 5?)
```

LOGOP

This example draws a word in a way that can be used either on the screen or by a floor turtle. For example, to draw with the Jessop turtle, place the turtle on a large piece of paper on the floor so that it can move at least 600 mm forward and 800 mm right. Then type:

```
LOAD "LOGOP
```
Hold CTRL and press ESCAPE
```
LOAD "JESSOP
FLOOR
START
```

Type `SCREEN` to drive the screen turtle again.

The word can be drawn again by typing:

```
START
```

MAP

Introduction

This set of procedures is for drawing maps. The maps are held as lists of longitude and latitude pairs which are joined together to draw the outline of a country, the course of a river or whatever other feature you wish to map. You can map all or selected features of a country at any desired scale and about any centre. Two examples of using the maps in simple games are also included.

Using the program

The main procedure is called `MAP` and is used in the form:

```
MAP :<country> <featurelist>
```

For example:

```
MAP :ENGLAND [BORDERS]
```

maps an outline of England;

```
MAP :ENGLAND [BORDERS TOWNS]
```

maps the borders and towns of England;

```
MAP :ENGLAND []
```

maps all features of England.

Note that to enable more than one map to be drawn on the screen at once, `MAP` does not clear the screen before drawing, so that `CLEAN` may be necessary between maps.

Scales and centres

In addition to the lists of features, each country list also has a default centre and scale. These are used if none is specified by the user before plotting. The effect of this is that if you (say) `MAP` England and then `MAP` Wales, England will be centred and Wales will be plotted in its correct position relative to England. If, however, you were to `MAP` Wales first and then `MAP` England, the map would be centred on Wales and would have a much larger scale (a lot of England would be off the screen).

If you want to override the defaults, use

```
MAKE "MAPSCA <number>
```

and

```
MAKE "MAPCENT <list>
```

before MAPping a country. The map scale is in units of screen units per kilometre whilst the centre is a position on the Earth's surface given as longitude and latitude in hundredths of a degree (the best way to learn how this works is to experiment).

Plotting grids

Two grid plotting procedures are supplied:

```
DEGGRID <n>
```

places a grid over the map at <n>/100 degree intervals whilst

```
KMGRID <xkm> <ykm>
```

overlays the map with a grid of rectangles <xkm> by <ykm> kilometres.

Games

Two very simple examples of the use of the map drawing procedures as a basis for other programs are also included. These are RALLY and DISTGAME.

DISTGAME will draw a map of England with a 100km grid and mark on some towns. Then it asks ten questions of the form 'WHAT IS THE DISTANCE BETWEEN <town1> AND <town2> ?' Each time it gets a response it tells you the correct answer and then, after ten questions, it gives a percentage score.

RALLY is also based on a map of England. Each of the towns is labelled with a letter and the user chooses a route which visits each town once and once only. This route is plotted on the map as it is entered and the total distance is printed out when all the towns have been visited.

Loading other map data files

Data is stored with the program on England and Wales. More data on Scotland is stored in the file called MAPSCOT, and more on Ireland in MAPIRE. See the 'How the program works' section for details on how to put your own maps into the program from an atlas.

To load the data from the MAPIRE or MAPSCOT files, you must first make enough space for it. You can delete all the map data stored in the program with the command ERDATA. This does not delete the map centre and scale. Thus, you can MAP one lot of data and then erase the data, load some new data (using LOAD "MAPSCOT or LOAD "MAPIRE) and MAP that, all on the screen at once and to the same scale. You can also free some space by deleting the games (RALLY and DISTGAME). This is done by calling ERGAMES.

How the program works

The data for each country is stored in several pieces to make it easier to enter. The data structure for England looks like this:

```
"ENGLAND is [ CENTRE [-146 5291]
              SCALE   1
              BORDERS [OUTER BWALES OUTER2 ... ]
              TOWNS   [LONDON BIRMINGHAM ... ]
              RIVERS  [TRENT THAMES SEVERN ... ] ]
"OUTER is   [ [-200 5578] [-143 5561] ... ]
"LONDON is  [ [-7 5090] ]
```

etc.

The data structure for any country is a list of pairs: the name of a feature type followed by a list of names which make up that feature type. The first pair must be the default CENTRE for that country and the second the default SCALE. The order of the actual plottable features is not important.

Each individual feature (eg "OUTER) is a list of coordinates which were entered directly from a map. The coordinates are in hundredths of a degree longitude east (west is negative) and latitude north (south is negative). The way in which these are transformed into screen coordinates is dependent on the current values of "MAPSCA (the scale of the map in screen units per kilometre) and "MAPCENT (the longitude and latitude of the point which will be at the middle of the screen).

When the program is given a list of coordinate values to plot, these are joined by straight lines plotted in the current pen state. When, on the other hand, the object to be plotted consists of a single coordinate (as does "LONDON above), then a special symbol (plotted by the procedure CITYMARK) is put at that position instead.

Also stored with the data on England and Wales is the data required to drive the RALLY and DISTGAME procedures.

Over to you

Putting in your own data from an atlas is not difficult. Simply delete the data in the program (see 'How to use the program') and then enter your own data in the format described above. The features which are described in the example data are BORDERS, TOWNS and RIVERS, but you can add more if you wish (motorways or railways, perhaps). New data should be entered using the Logo editor to edit the coordinate lists and the country data lists, since many of them will be more than 255 characters long and could not, therefore, be entered directly in response to the Logo prompt.

To save your data, you can either resave the program along with your data, or you can save just the data by using:

```
SAVE <filename> []
```

MAZE

Introduction

This program allows you to create simple mazes on the computer screen. You can then either attempt to solve the maze or write a program for a maze running 'robot' to use in solving the maze. Mazes can also be saved to and loaded from disc or tape.

How to use the program

When you load the `"MAZE` file, a `LOADINIT` procedure explains the basis of using the procedures to generate and save your own mazes. Then the screen goes into `DRAW` mode with a small menu in the text window. This gives you the option of:

1 Making a maze

2 Trying a maze

3 Running a maze robot program

4 Loading a maze

5 Saving a maze

6 Returning to command level

Constructing a maze

Option 1 allows you to create a maze from scratch. You are given a grid of cells with a little man in the bottom left hand corner. The little man can be moved about the maze with the cursor arrow keys. If you press DELETE, the man becomes a hammer. This can be moved about in the same way, but instead of walking through the walls of the maze it will knock down existing walls and build new ones where old ones have previously been knocked down. Pressing DELETE again will change the hammer back to the man.

To mark the start of the maze, press the 'S' key (this will remove any previously defined start). To mark the finish, press the 'F' key (this will remove any previously defined finish). To exit the maze definition procedure, press the 'Q' key. The system will not allow you to end the construction of a maze unless start and finish have been defined.

Trying to solve a maze

Option 2 lets you try to solve the maze which is currently on the screen (either just made or loaded from disc or tape). A smiling face appears in the start position of the maze and this can be moved about using the cursor arrow keys. Attempting to move through walls generates an 'ouch' message and a 'boinng' noise. When the face reaches the goal position of the maze, a message of congratulation is printed and a little tune is played. The 'Q' key allows you to give up before finishing.

Running a maze solving procedure

Option 3 executes a previously defined maze running procedure. The maze running robot is represented by a thick arrow and is initially placed in the start position of the maze facing upwards. Control is then passed to the user's procedure which has the following sub-procedures at its disposal:

1 `GOALQ` returns `"TRUE` if the robot is at the end of the maze, otherwise it returns `"FALSE`.

2 `MOVEF` moves the robot forward if there is no wall in the way.

3 `WALLF`, `WALLB`, `WALLR` and `WALLL` return `"TRUE` if there is a wall in front of, behind, to the right of and to the left of the robot, respectively. Otherwise, they return `"FALSE`.

4 `TURNR` and `TURNL` turn the robot through 90 degrees right and left respectively.

Your procedure should simply `STOP` or `END` when it finds the goal or when it gives up.

There is already one simple maze running procedure loaded which you might like to try. It is called `RANWALK` and is not very bright! Pressing 'Q' will return you to the menu.

Loading a maze

Option 4 will load a maze from tape or disc and there is then a short delay whilst the procedure identifies the start and finish of the maze. Try loading `MAZEDAT` from the tape or disc.

Saving a maze

Option 5 simply saves the maze which is currently on the screen to tape or disc.

Quitting the program

Selecting option 6 from the menu just passes control back to command level. To return to the menu type `START`.

Over to you

One of the purposes of this program is to provide an environment in which you can write your own maze solving procedures. Experiment with different strategies – you may find that different strategies work best in different kinds of maze. Can you write a procedure that will solve any maze? You will probably find it helpful to keep track of where your robot has been already, to stop it getting trapped and going round in circles.

MIRROR

Introduction

This program is intended to demonstrate the use of multiple turtles. It also shows the way in which you can redefine the basic Logo primitives to 'customise' the system to a certain extent. The basic idea is that the screen is divided vertically into two halves and there is a turtle in each section of the screen; one on the left and one on the right. Any turtle graphics commands which are given (either directly or from within a procedure) are then executed directly by the turtle on the right and mirrored by the turtle on the left. Because this program redefines primitives and makes use of buried procedures it is advisable to reinitialise the language by typing `*LOGO` before running any of the other examples.

How to use the program

When you `LOAD` the program, you will be presented with a page of instructions and then the screen will clear to leave two turtles, each centred in its half of the screen and separated by a vertical line. There is a short turtle graphics procedure called `DEMO` which you might try at this stage. To return to having just one turtle, type `SINGLE`. The procedure `MIRROR` will then restore the mirror turtles. Apart from these two commands, you should just use turtle graphics as before (but avoid using commands like `SETPOS`, which are based on Cartesian coordinates).

The only thing to bear in mind is this: whilst you normally control the right hand turtle directly and the left hand one mirrors its actions, if the turtle leaves its half of the screen then it may wrap round. Thus, your turtle will be on the left and the mirror turtle on the right.

How the program works

When the program is first loaded, the old definitions of `DRAW`, `LEFT`, `RIGHT` and `HOME` are 'saved' by `COPYDEF`ing them. Then new versions are substituted which direct different commands to the two different turtles. The advantage of doing things this way is that a procedure can work in a normal one-turtle environment (for example, `DEMO` will work in `SINGLE` mode as well) and then work unchanged when there are two turtles responding in different ways.

Over to you

This simple idea has a lot of potential for further experiments with different kinds of symmetry. You could start by varying some or all of the following:

1 The initial states of the turtles

2 The number of turtles on the screen

3 Scaling factors for the two turtles (one turtle moves with bigger 'steps' than the other)

4 The definition of a straight line – try distorting turtle paths around a point for example (this is more complicated).

PLANTS

Introduction

This simple program allows you to 'teach' the computer how to distinguish between members of a set of objects. Initially, the computer 'knows' very little but it gradually 'learns' more from the person using it.

How to use the program

The way in which this program works is to ask you to think of a member of the class of objects (such as plants). It then attempts to guess what you are thinking of by asking you a series of YES/NO questions. The computer will then say what it thinks is the answer, and ask if it is correct. If not, it will ask for the name of the object of which you were thinking, and a new question which it can use in future so as not to make the same mistake twice.

Here is a sample session with the 'plants' database which is initially in memory when the program is loaded:

```
START "PLANTS
```
(this starts the program off)

```
THINK OF A PLANT
IS IT A TREE ? N
IS IT A WEED ? N
IS IT A VEGETABLE ? Y
I THINK IT IS A CARROT. AM I RIGHT ? N
WHAT IS IT THEN ? A LETTUCE
PLEASE GIVE ME A QUESTION WHICH I CAN
USE TO DISTINGUISH A CARROT FROM A LETTUCE
? DOES IT GROW UNDERGROUND ?
SO IF I ASKED YOU:
DOES IT GROW UNDERGROUND ?
AND YOU WERE THINKING OF A LETTUCE
WHAT WOULD THE ANSWER BE ? N
OK

THINK OF A PLANT
IS IT A TREE ? N
IS IT A WEED ? N
IS IT A VEGETABLE ? Y
DOES IT GROW UNDERGROUND ? N
```

```
   (the new question)
I THINK IT IS A LETTUCE. AM I RIGHT ? Y
Whoopee!

THINK OF A PLANT
IS IT A TREE ? Q
    (Q for QUIT)
?                            (command level)
```

If you save the program after a session, the new state of its 'knowledge' will be saved with it. You can then keep building up its expertise over several sessions.

Starting a new tree

The structure which is used to hold the information in this program is called a 'binary tree' ('binary' because there are two choices at each question and 'tree' because the structure resembles the branches of a tree when it is drawn on paper, although it is normally drawn to look more like the roots.

To initialise a tree on a new subject, type `NEW` from command level. First you will be asked what you want to call the tree. This name should be the subject about which the tree is going to hold information; for example, `CARS`, `PEOPLE`, `ANIMALS`, `PLANETS` or `COMPUTERS`. The procedure will next ask you for the singular of your name; for example: `CAR`, `PERSON`, `ANIMAL`. Then you have to say whether or not your subject will require an 'article'. This piece of information is used when the program makes a guess, so that it will say `I THINK IT IS A MINI` when you're talking about cars, but `I THINK IT IS SHAKESPEARE` (rather than `A SHAKESPEARE`) when you're talking about people. The final question which you must answer to create a new tree is an example of your subject. This is needed to start the tree off.

Erasing a tree

If you want to wipe a tree from the computer's memory (either to make more room for another tree or so you can start a new tree with the same name), type

```
WIPE <treename>
```

For example:

```
WIPE "PLANTS
```

You will be asked to confirm your decision before the tree is actually destroyed as there is no way to get it back if it has not been saved.

How the program works

The information for this program is held in a single Logo list for each topic. This list contains three elements: the first is TRUE or FALSE depending on whether or not the names are proper names; the second is the singular form of the tree name, and the third is a classification tree.

A classification tree is held as a single Logo list with the following structure: a tree-list is either an atom or

```
[[question list]
  [tree-list for yes]
  [tree-list for no]]
```

If the chosen sub-list in the above structure is an atom (that is to say, a Logo word rather than a list), the atom is interpreted as a terminal node (sometimes called a leaf, for obvious reasons) and is printed out as the computer's guess.

For example, say you have just created a new tree concerning CARS and you have given it BUGATTI as an example, then :CARS is FALSE CAR BUGATTI. If you then think of a 2CV and add the question:

```
IS IT FRENCH
```

to the database, :CARS is:

```
FALSE CAR [[IS IT FRENCH] 2CV BUGATTI]
```

The addition of the new car CHEVROLET and the question:

```
IS IT AMERICAN
```

then makes :CARS equal to:

```
FALSE CAR [[IS IT FRENCH] 2CV [[IS IT AMERICAN]
 CHEVROLET BUGATTI]]
```

The procedure which actually does the questioning is called TRAVERSE; it prints out a question (the first element of its input list) and then calls itself on the second or third element of the list, depending on the answer. TRAVERSE also keeps a list of the sequence of 'Y's and 'N's which you type. This is used to speed up the REPLACE procedure which modifies the tree when a new question is to be added.

Over to you

There are many interesting additions which you could make to the basic program as presented here. A fairly simple one would be a function to print out the structure of a tree. Such a function is called a 'pretty printer' and it should produce something like this:

```
"CARS is:

[[IS IT FRENCH]
     2CV
     [[IS IT AMERICAN]
          CHEVROLET
          BUGATTI]]
```

Another function which would be rather nice is an editor which allows changes to the tree structure – especially useful when you make a mistake. The program at present does not check for inconsistent input: you can have the same object at different places in the tree (which is obviously wrong). Since the mistake might well be in either occurrence of the name, detecting this kind of error is not very useful until you have implemented some sort of editor, so that you can correct the list structure (you can, of course, use `EDN`, but this works on the list as a string of characters, rather than a nested structure, and is therefore fairly inconvenient).

SCATTER

Introduction

This is a set of procedures which is designed to be used with your own programs. It allows you to spread your pictures about the screen in various ways. The images may be of fixed or variable size or they may be plotted with diminishing size to give the effect of perspective.

How to use the procedures

Your procedures should take just one parameter to control size. For example:

```
TO TREE :S
IF :S<16 [STAMP STOP]
FD :S RT 30
TREE :S/1.5
LEFT 60
TREE :S/1.5
RT 30 BACK :S
END
```

Images are plotted on the screen within a rectangular area (initially set to the whole screen) which you can define using the procedure call:

```
LIMITS <low x> <low y> <high x> <high y>
```

(these are the limits within which the start of a drawing may lie – your procedure may well cause the turtle to exceed these boundaries).

To plot an image on the screen after you have defined the procedure, the basic command is:

```
PLACE <proc name> <max size>
```

This will plot one `<proc name>` at a random position.

The size of the image plotted by `PLACE` is defined by the `<max size>` input and also by which one of three possible options has been selected before doing the `PLACE`. If the procedure `FIXED` has been run, then the size of the resulting image will be exactly `<max size>`. If `VARY` has been run first, the size of the resulting image will be a random number between 0 and `<max size>`. The other option is `PERSPECTIVE` which causes the size to vary with Y-coordinate from `<max size>` in the foreground to 0 on the horizon.

If you want to PLACE several images on the screen (for example, to create a garden of TREEs), you can use the SCATTER command; this will PLACE a random number of drawings within the defined window. For example:

```
SCATTER 20 "TREE 150
```

will PLACE between 1 and 20 TREEs with <max size> 150 on the screen.

If you require a demonstration of the different commands, type HELP. The best way to learn how to use the procedures is simply to experiment with them.

SPIRAL

This example is so short that it has been left for you to type in.

`SPIRAL` shows how a very simple Logo program can be used to create a large variety of interesting patterns. It is also an example of tail recursion and can be used to illustrate this powerful idea.

The program to be typed in is as follows:

```
TO SPIRAL :FWD :ANGLE :INC
FORWARD :FWD
RIGHT :ANGLE
SPIRAL :FWD (:ANGLE + :INC) :INC
END
```

Having entered this procedure you could try out the variety of possible effects, by entering commands such as:

```
CS SPIRAL 20 6 9
```

If you would like to study how these effects are produced, press ESCAPE while the program is running, then type:

```
TRACE 15 CO
```

The program will now run step by step, waiting at each stage for you to press a key, and you will be able to see each line before it is executed, then each command with the actual data to the command.

Less comprehensive traces are possible. For example, press ESCAPE and type:

```
TRACE 10 CO
```

This now shows each call to the procedure `SPIRAL`.

To exit from tracing press ESCAPE and type:

```
TRACE 0 CO
```

STORY

Introduction

This program is for writing a kind of computer/human generated story. The computer decides on sentence structures at random and then allows you to choose the actual words from lists of words which it 'thinks' would be suitable.

How to use the program

When the program is `LOAD`ed, you are shown a page of instructions and then the screen clears. The story will appear word by word at the top of the screen whilst your choices are made at the bottom.

Choices are made by pressing the RETURN key to cycle through the list of words and then pressing the space bar to select the desired word. This is then printed at the top of the screen.

Press CTRL and ESCAPE together to exit from the program.

Over to you

After you have used the program for a while, you will begin to see several improvements which could be made. For example, there is no way of 'unmaking' a choice and there is no way of getting hard copy of your story. Both of these changes are fairly straightforward to make (although you'll have to back up on the sentence structure to undo a choice).

At a more fundamental level, the way in which the program decides on a sentence structure and then sticks to it can be slightly limiting. An alternative way of generating a sentence would be to allow the structure to develop at the same time as the sentence is actually being built up. For example, if you had selected 'Once upon a time' as the start of your sentence, the structure would still be completely undecided. Then you might select 'the', and this choice would then rule out all the sentence structures which start with a place or an adverb (eg `UNDER THE SHOEBOX, ...` or `STIFFLY, ...`). This sort of program would allow you much more control over the way in which the story develops (although the program would be quite a bit more complicated than this one).

The example program `WAFFLE` shows one possible approach to making the computer generate the sentences completely on its own, but making the sentences hang together into a coherent (or even semi-coherent) whole is very difficult without some kind of internal model of what it is that the sentences are describing.

TOOLBOX

Introduction

This program is intended to show how the computer can be made to 'understand' a very small set of English questions about a limited domain and respond in an 'intelligent' manner.

The idea is that the computer has a box containing several objects, each of which has several properties like size and colour. Whilst you can't just ask the computer what is in the box (that would be too easy), you can ask it simpler questions such as:

```
IS THERE A RED SCREWDRIVER IN THE BOX ?
HOW MANY SPANNERS ARE THERE ?
WHAT COLOUR IS THE SMALL HAMMER ?
WHAT TYPE IS THE BIG BLUE THING ?
ARE THERE ANY BIG OBJECTS IN THE BOX ?
```

How to use the program

Using the program is really just a matter of typing the questions. When the program is loaded you are presented with a page of instructions and then the colon (`:`) prompt appears. Type a question like one of those above then press RETURN. The computer will either respond with an answer or it will print '`EH?`' to signify that it could not understand your input.

Press CTRL and ESCAPE together to exit from the program.

The computer answers questions with full sentences which tell you what question it has actually answered. This prevents it from appearing to tell lies when it has really misunderstood you. For example: '`HOW MANY TRUFFLES ARE THERE ?`' would be interpreted as '`HOW MANY anythings ARE THERE ?`' because the word '`TRUFFLES`' in the input is not in any of the computer's lists. The computer would therefore respond with something like:

```
THERE ARE THREE OBJECTS IN THE BOX.
```

Changing the contents of the box is very easy, just type `PONS` from the command level and you will see the structure of the data. Edit this to change the objects, their properties or whatever.

How the program works

The heart of the program is a general purpose pattern matching procedure called MATCH (a very similar function does most of the work in the LOGIC program). This procedure is used to 'analyse' the input sentence and to call the appropriate routine. Property words like BIG or BLUE in the input are used to construct a filter to decide which objects interest us. This filter is applied to each object in turn and if it matches, the appropriate action (printing 'yes there is one' or adding one to the count of objects satisfying the conditions) is taken.

Some fairly simple procedures then generate the program's response, such as:

```
THERE ARE FOUR BIG BLUE THINGS IN THE BOX.
I DON'T KNOW WHICH BIG RED OBJECT YOU MEAN.
THERE IS AT LEAST ONE SMALL SCREWDRIVER IN THE BOX.
THE BIG SCREWDRIVER IS GREEN.
```

Over to you

The area of natural language processing by computer is a very large and complicated one. If you want to write more powerful natural language programs, you should read a book on artificial intelligence. However, you can go some way by just adding to this sort of system. The program could be extended to deal with a wider variety of sentence constructs and possibly with relationships between objects (for example, 'How many things are bigger than the red brick?' or 'What is on top of the book?'). Systems of this kind are very useful for allowing non-specialists to interact with information stored on computers.

You can make the computer appear more intelligent by adding a fairly simple natural-language 'front-end' to a program which performs some other pseudo-intelligent task (such as working out how to stack bricks on a table).

TRI

This example is so short that it has been left for you to type in.

This shows how a simple Logo program can be used to fill an area in an attractive way. It is also an example of full recursion, and can be used to illustrate this powerful idea.

The program to be typed in is as follows:

```
TO TRI :SIDE :DEPTH
IF :DEPTH=0 [STOP]
REPEAT 3 [TRI :SIDE/2 (:DEPTH - 1) FORWARD :SIDE LEFT 120 ]
END
```

Having entered this procedure, you might like to experiment with commands such as:

```
CS TRI 240 5

CS SETPOS [-300 -300] RIGHT 90 CLEAN TRI 600 6

CS SETNIB 80 SETPT 3 SETPOS [-300 -300] RIGHT 90
FD 0 CLEAN TRI 600 4
```

The program can be understood as a generalisation of the triangle drawing procedure:

```
TO TRIANGLE :SIDE
REPEAT 3 [ FORWARD :SIDE LEFT 120 ]
END
```

In TRI, however, a smaller triangle is to be drawn at the start of each side. This is achieved by calling TRI again with a smaller :SIDE. The :DEPTH parameter is needed to stop the program trying to draw smaller and smaller triangles for ever.

Tracing the program

You may find it useful in understanding this idea to trace the execution of TRI. For example, type:

```
(DRAW 12)
TRACE 15
TRI 240 5
```

and watch the successive calls to `TRI`. Remember to press a key (such as the Space Bar) when you see the prompt '! '.

At any stage you can see the way `TRI` is calling itself by pressing ESCAPE and then typing:

```
TC
```

The program can then be resumed by typing:

```
CO
```

Tracing is removed by pressing ESCAPE and then typing:

```
TRACE Ø CO
```

You can restore the original number of text lines by typing:

```
(DRAW Ø)
```

Over to you

You may like to experiment with other similar space filling programs, for example one that fills space with squares.

WAFFLE

Introduction

This program is designed to generate grammatically correct English sentences by choosing at random from several lists of words and some of their properties.

`WAFFLE` is a complicated program and so can take over a minute to build each sentence.

The program generates sentences of the following form (called 'type 1 sentences' in this explanation):

```
<subject noun phrase> <verb> (<optional object noun
 phrase>).
```

For example:

```
THE BIG DOG ATE THE SMALL GREEN BOOK.
THE GIRL SLEPT.
```

The program can also generate 'type 2 sentences', which have the following form:

```
<object noun phrase> WAS <verb> BY <subject noun
 phrase>.
```

For example:

```
THE CUP WAS BROKEN BY THE BIG CLUMSY MAN.
```

Noun phrases may be either simple:

```
THE BIG BLUE CARPET
```

or they may be complex:

```
THE STEAK WHICH WAS EATEN BY FIDO
```

(based on a type 2 sentence)

```
THE BOOK WHICH DISGUSTED MUMMY
```

(based on a type 1 sentence)

```
THE BOY WHO CRIED
```

(these complex phrases are called 'fancy nouns' in the program).

How to use the program

To generate and print out a single random sentence, run the procedure `GENSENT`. If you want the machine to carry on generating sentences (until you press the ESCAPE key), run the procedure called `DRIVEL`.

`GENSENT` chooses between type 1 and type 2 sentences and will generate either simple or fancy nouns as subjects and objects. For example:

```
THE BLUE TABLE WHICH IMPRESSED THE MAN DISGUSTED THE
 QUEEN WHO WAS LIKED BY BILL.
```

```
MARY DISLIKED THE SMELLY HOUSE WHICH BELONGED TO THE
 FAT MAN.
```

Changing the vocabulary

If you want to add new words to the program's vocabulary, you need to appreciate the way in which the program uses certain properties of words so that it does not choose completely at random from lists of nouns, verbs and adjectives. The details of the classifying scheme are explained below, but you should remember that this is not the only way to approach the problem; it was chosen because it appeared to work quite well and you may be able to think of a better one.

When you understand the classification system and you want to add new words (or delete old ones), use `EDN` on `"NOUNS`, `"VERBS` or `"ADJECTIVES` along with `PPROP` or `REMPROP` as appropriate.

How the program works

There are certain restrictions placed on the words that may be chosen in each position. These aim to prevent you generating completely nonsensical sentences such as 'Colourless green ideas sleep the skyscraper.' The way in which this works for a type 1 sentence with simple nouns is (roughly):

1 There are three global word lists

```
"NOUNS is [BOY GIRL BOOK VIRTUE FIDO MARY ...]
"VERBS is [ATE LIKED AMUSED BROKE ...]
"ADJECTIVES is [BIG HEAVY BLUE GREEN ...]
```

2 Each noun has a list of qualities which are stored in a property list under the name `"CLASS`. These are used to decide whether or not the noun is a sensible one to use in the current context. For example:

```
PPROP "DOG "CLASS [COMMON COUNT ANIMATE ~HUMAN]
```

defines the `"CLASS` of the word `"DOG` to be `COMMON` (ie it's not a proper noun), COUNT (ie it's an actual object), ANIMATE (ie it can perform actions) and ~HUMAN which means that it is non-human (the tilde ~ represents negation).

3 Each verb has a property called "SUBJC which is a list of CLASS conditions that must be satisfied by a noun for that noun to be used as a subject to the verb. For example:

```
PPROP "EMBRACED "SUBJC [HUMAN]
```

means that only nouns with the property HUMAN can be used as subjects to the verb 'embraced'.

4 Each verb has a property called "OBJCL which is a list of alternative lists of CLASS conditions that must be satisfied by nouns for use as objects to that verb. For example:

```
PPROP "ATE "OBJCL [[COUNT] [~COMMON ANIMATE] [@]]
```

means that the verb ATE can take as an object any countable noun (THE BLUE TABLE or THE FAT MAN), any proper animate noun (FIDO), or no object at all (the [@] represents the intransitive use of the verb).

5 Each adjective has a property called "ADJCL which is a list of class conditions that must be satisfied by any noun before the adjective can be applied to the noun. For example:

```
PPROP "BLUE "ADJCL [COUNT ~ANIMATE]
```

limits the application of the adjective BLUE to nouns which are both countable and inanimate (like CAR).

6 The procedure GENSENT will generate and print out a random sentence in the following way:

(a) Decide on a subject for the sentence. This will be a noun group generated at random with no restrictions

(b) Decide on a main verb for the sentence. This will be a verb picked at random from those which will accept the noun generated in (a) as subject

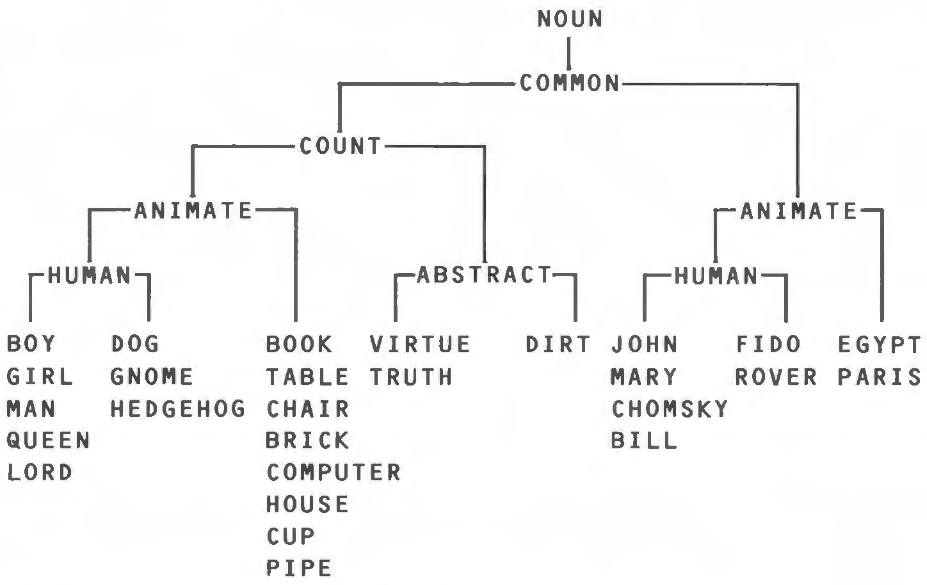
(c) Decide on a class type for the object. This will be a random member of the verb's "OBJCL property

(d) If the object class type is not [@] (ie if we are going to use the verb transitively), pick an object for the sentence. This will be a noun picked at random subject to the restriction that it must satisfy the object class condition decided in (c)

(e) Print out the sentence (subject) followed by (verb) followed by (object)

(adjectives and articles are added as appropriate by the noun generating procedure)

7 The way in which the nouns are subdivided into CLASSes is shown by the following tree diagram:



(left – property positive, right – property negative)

For example:

TRUTH's "CLASS is [COMMON ~COUNT ABSTRACT]
FIDO's "CLASS is [~COMMON ANIMATE ~HUMAN]

Some of the verbs have an extra property, called "PASTP, which stores a verb's past participle if it is different from its past indicative (eg ATE, EATEN or BROKE, BROKEN). This is stored for use in the type 2 sentence (JOHN ATE FIDO but FIDO WAS EATEN BY JOHN) or complex noun phrases built up on the type 2 sentence structure (THE CUP WHICH WAS BROKEN BY THE GIRL).

There are further minor complications such as the need to say 'THE CUP WHICH WAS ...' but 'THE GIRL WHO WAS ...' and the fact that generating a fancy noun to satisfy some list of conditions is more complicated than generating a simple noun to comply with the same restrictions.

Over to you

There are many possible ways in which the program could be extended. These include:

1 Preventing the same adjective from being used twice on the same noun

2 Giving adjectives a 'binding power' which determines how strongly they attach to the noun (so that you get `THE BIG BLUE HOUSE` rather than `THE BLUE BIG HOUSE`)

3 Adding new sentence structures (eg allowing indirect objects, adverbs, negations or questions)

4 Adding a wider range of tenses (this will require the addition of lookup tables for irregular verbs)

5 Allowing nouns to form proper plurals

(You may find that you need to add extra properties to some word classes in order to make certain enhancements)

3 Logo extensions

The extensions supplied with the package are as follows:

BUGGY – Supports BBC BUGGY
CALC – Mathematical functions
CLEPSON – Prints coloured screen on EPSON-compatible printer
EPSON – Prints screen on EPSON-compatible printer
JESSOP – Supports JESSOP floor turtle
MOS – Operating system primitives
MULT – Multiple turtles extension
OLIV – Prints screen on Olivetti-compatible printer
PROP – Property lists extension
SECT – `SECT` primitive to draw sectors
VALIANT – Supports VALIANT floor turtle

3.1 Loading extensions

Load any required extension as you would a program, for example:

```
LOAD "CALC
```

Do not load extensions repeatedly, since each time you use up more workspace.

If you want your program to have access to a particular extension include a command such as the following:

```
IF NOT PRIMITIVEQ "ASN [LOAD "CALC]
```

Extensions are not saved with your program.

BUGGY

This is the extension for the BBC BUGGY. It supports standard procedures for Floor Turtles, namely:

```
BACK(BK)   EXPLORE   FLOOR   FORWARD(FD)   HOOT   LEFT(LT)
PENDOWN(PD)   PENUP(PU)   PENUPQ   RIGHT(RT)   SCREEN
SENSE
```

`FORWARD 1` moves approximately 1 mm.

The action of `SENSE` is as follows:

`SENSE 1`	reads the LDR and returns 0 to 32760
`SENSE 2`	reads the barcode sensor and returns 0 to 32760
`SENSE 3`	reads an optional input to the ADC and returns 0 to 32760
`SENSE 4`	reads an optional input to the ADC and returns 0 to 32760
`SENSE 5`	returns `"TRUE` if the tilt sensor is set
`SENSE 6`	returns `"TRUE` if the right bumper is touching
`SENSE 7`	returns `"TRUE` if the left bumper is touching

`EXPLORE` stops if either `SENSE 6` or `SENSE 7` is detected or ESCAPE is pressed

`HOOT` sounds on the computer's speaker

CALC

This contains the following mathematical functions:

```
ASN   EXP   LN   PI   TAN
```

Other operations can be derived from these, for example:

```
TO POWER :NO :EXPONENT
OUTPUT EXP :EXPONENT * LN :NO
END
```

CLEPSON

This contains PRSCREEN and it prints a coloured screen on an Epson-compatible printer. The printout is shaded to indicate the different colours.

This gives a much slower print than is provided by EPSON.

EPSON

This contains PRSCREEN and it prints a screen on an Epson-compatible printer. Any part of the screen which is not in logical colour 0 is printed in black.

This gives a much faster print than is provided by CLEPSON.

JESSOP

This is the extension for the JESSOP floor turtle. It supports standard procedures for Floor Turtles, namely:

`BACK(BK)   EXPLORE   FLOOR   FORWARD(FD)   HOOT   LEFT(LT)   PENDOWN(PD)   PENUP(PU)   PENUPQ   RIGHT(RT)   SCREEN   SENSE`

`FORWARD 1` moves approximately 1 mm.

There are, in fact, no sense functions and so `SENSE <n>` always returns `"FALSE`.

`EXPLORE` goes forward the given distance or until ESCAPE is pressed. In the latter case it reports the distance actually travelled.

MOS

This contains the machine code interface functions:

`CALL   DASIZE   DATAAREA   HEX`

MULT

This is the multiple turtle extension. It contains the following:

`ALIVEQ   FORGET   HATCH   TELL   TURTLES   WHO`

OLIV

This contains PRSCREEN and prints a screen on an Olivetti-compatible printer. Any part of the screen which is not in logical colour 0 is printed in black.

This is a much faster print than is provided by CLEPSON.

PROP

This is the property list extension. It contains the following:

ERPLIST ERPLISTS GPROP PLIST PPALL PPROP PPS REMPROP

SECT

This contains the primitive `SECT`. `SECT` has the following syntax:

```
SECT <radius> <angle> <width>
```

It draws a sector through the specified `<angle>`. `<radius>` is the distance from the turtle to the centre of curvature; a positive `<radius>` means that the centre is to the right of the turtle. `<width>` specifies the separation of the two lines of the arc; a positive `<width>` means that the second line is to the right of the turtle. If `<angle>` is positive the turtle moves forward; if negative it moves backwards.

The turtle finishes at the other end of the line from its starting point.

If the nib has been set to 80, the space between the lines is filled.

The following example draws an annulus:

```
SETNIB 80
SECT 100 360 50
```

VALIANT

This is the extension for the VALIANT floor turtle. It supports standard primitives for floor turtles, namely:

`BACK(BK)  EXPLORE  FLOOR  FORWARD(FD)  HOOT  LEFT(LT)  PENDOWN(PD)  PENUP(PU)  PENUPQ  RIGHT(RT)  SCREEN  SENSE`

and two special primitives:

`SELECT  SELECTED`

`FORWARD 1` moves approximately 1 mm.

`HOOT` uses the computer's speaker.

There are no sensors and so `SENSE <n>` returns `"FALSE`.

`EXPLORE <distance>` runs until ESCAPE is pressed or the `<distance>` is covered; it returns the length covered.

It is possible to drive up to four turtles. The turtle selected initially is number 0. Other turtles can be selected by typing `SELECT <n>` (where `<n>` is in the range 0 to 3).

`SELECTED` returns the number (0 to 3) of the VALIANT turtle currently selected.