



A

C

O

M

Turtle Graphics Programming System

Help File

Turtle Graphics Programming System

Peter Millican, University of Leeds

This booklet contains the entire contents of the Help file for the Turtle Graphics Programming System (Version 7.01, as at November, 2003). Cross-reference “hot links” within the file are shown doubly underlined, and followed by the corresponding page reference in square brackets.

TURTLE GRAPHICS PROGRAMMING SYSTEM	1
Introduction	1
Getting Acquainted – Read This First!	1
Overview of the System.....	1
The System Menus	2
Exercises and Illustrative Programs – Start Learning Here!.....	2
The Canvas	2
The Program	2
The Turtle Standalone Run-Time System	3
Teaching Aims: Deep Understanding of Programming Concepts	4
Current Version of the Turtle Software	5
Hardware Requirements and Issues	7
MENUS IN THE TURTLE GRAPHICS SYSTEM	8
File menu	8
File New program	8
File Load program	8
File Save program.....	8
File Save As	8
File Image to Clipboard	8
File Image to File.....	8
File Exit Turtle Graphics.....	8
Edit menu	9
Edit Undo	9
Edit Redo	9
Edit Cut.....	9
Edit Copy	9
Edit Paste insert.....	9
Edit paste Over	9
Edit Indent program lines.....	10
Edit Unindent program lines.....	10
Edit Auto-format program.....	10
Layout menu	11
Layout Canvas (0,0) to (1000,1000)	11

Layout Canvas (-1000,-1000) to (1000,1000).....	11
Layout Display editor and Canvas.....	11
Layout Display Editor (full-width) only	11
Layout 10-point font for program.....	11
Layout 9-point font for program.....	11
Layout Courier New font for program.....	11
Layout Ariel Narrow font for program.....	11
Compile menu.....	12
Compile Compile to PCode.....	12
Compile Run program.....	12
Compile Halt program	12
Compile Analysis tables.....	13
Compile Trace on run	14
Compile Trace Stack height instead of Stack location 3	14
Compile Save PCode file	12
Compile Turtle machine using separate Return Stack.....	14
Compile Turtle machine using separate Heap Control Stack	14
Compile Turtle machine using Procedure Register Stack	14
Options menu	15
Options Auto-Compile on loading.....	15
Options Auto-Run on loading.....	15
Options Auto-Format on loading.....	15
Options Auto-save PCode on compiling.....	15
Options Upper/lower case on auto-format.....	16
Options Strip comments on auto-format	16
Options Blank Canvas on RUN.....	16
Options Update every 100,000 cycles.....	16
Options Update every 300,000 cycles.....	16
Options Update every million cycles	16
Options Never force updating (may hang)	16
Options Independent load/save directories.....	17
Options Linked load/save directories.....	17
Options Flexible load/save directory handling	17
Options Load settings from options file.....	17
Options Save settings to options file.....	17
Help menu	18
Help Turtle Help.....	18
Help Quick reference	18
Help Illustrative programs.....	18
Help Recursion factory	18
Help Exercises.....	18
TURTLE GRAPHICS PROGRAMMING	19
Programming Quick Reference	19
Program Structure	19
Variable Assignment, Arithmetical and Boolean Operators and Constants.....	20
Turtle Movement – Relative (i.e. relative to the <i>turtle</i> 's current position).....	20
Turtle Movement – Absolute (i.e. to a specific location on the Canvas)	20
Shape Drawing.....	21
Drawing Control.....	21
Control Structures	22

Programming Essentials.....	23
Program Structure and Declarations	23
Indentation, Capitalisation, and Comments.....	24
Command Statements	25
Variable Assignment, Arithmetical and Boolean Operators and Constants	25
Turtle Movement – Relative (i.e. relative to the turtle’s current position)	25
Turtle Movement – Absolute (i.e. to a specific location on the Canvas).....	26
Shape Drawing	26
Drawing Control.....	27
Structural Statements.....	29
Looping Structures.....	29
Procedures.....	30
Conditional Structures.....	31
 Introduction to Pascal Syntax	 32
 Pascal Syntax Reference.....	 33
Identifiers.....	33
Reserved Words (Keywords)	33
 Procedures and Parameters	 34
Introducing Simple Value Parameters	34
Recursion	35
Scope	36
Formal and Actual Parameters	36
Value and Reference Parameters.....	37
 The Recursion Factory	 39
Quick-Start Guide – Playing with Patterns	39
Recursion Factory Reference	40
 THE VISUAL COMPILER AND THE TURTLE MACHINE	 42
 An Introduction to PCode	 42
The PCode Display.....	42
 Technical Note on Variables, Procedures and Parameters.....	 44
The Heap and its Maintenance Mechanisms	44
Dealing with Reference Parameters.....	48
 PCode Reference Guide.....	 49
Null Command	49
Loading and Storage of Variables.....	49
Flow Control and Procedure Handling.....	50
Turtle Commands Adjusting Run-Time Flags.....	51
Other Turtle Commands Having No Stack Effect	52
Turtle Movement and Colour Commands, Taking 1 Value from Stack.....	52
Turtle Shape-Drawing Commands, Taking 1 Value from Stack	52
Turtle Commands Taking 2 or More Values from Stack	53
Unary Numeric Operators, Replacing Top Value on Stack.....	53
Unary Boolean Operators, Replacing Top Value on Stack	53
Binary Numeric Operators, Overall Reducing Stack by 1.....	53
Binary Boolean Operators, Overall Reducing Stack by 1	54
Binary Comparison Operators, Overall Reducing Stack by 1	54
Instructions to Provide Stack Variations on the Turtle Machine	55

Exercises	56
Before You Start the Exercises.....	56
Exercise 1.....	56
Exercise 2.....	57
Exercise 3.....	57
Exercise 4.....	58
Exercise 5.....	58
Exercise 6.....	59
Exercise 7.....	59
Exercise 8.....	60
Exercise 9.....	60
Exercise 10.....	61
Exercise 11.....	61
Exercise 12.....	61

Turtle Graphics Programming System

Introduction

Getting Acquainted – Read This First!

The Turtle Graphics Programming System has many features, some very sophisticated, and there is a great deal of information in this comprehensive Help file. So if you are just beginning, it is important not to confuse yourself by following too many links from topic to topic! Probably the best sequence for the newcomer is as follows:

1. Read through the “Overview of the System” that follows, and the material on the “System Menu”, but without following any of the links.
2. Then follow the advice under the final heading “Exercises and Illustrative Programs – Start Learning Here!”, and go straight to the Exercises page.
3. As you work through the Exercises, make use of the relevant illustrative programs if you find them helpful, and refer to the [Programming Essentials](#) and/or [Programming Quick Reference](#) sections for detail on the various commands as they are introduced.
4. Having worked right through the Exercises, come back to read this section again, this time following any links that interest you, to see what other features the system provides.

If you do get lost at all, the Help system’s “Contents” page (which you get to by clicking the “Contents” tab while Help is running, or by selecting “Turtle Help” from the Turtle system’s [Help Menu](#) [17]) will show you how the Help is organised.

Overview of the System

This system is designed to teach programming concepts in a simple, intuitive way, based on the idea of *Turtle Graphics* (made famous through the programming language LOGO). An invisible *turtle* moves around the [Canvas](#) [2] according to the commands specified in the [Program](#) [2]. As the *turtle* moves, it leaves a coloured trail which builds up a picture, and it can also draw shapes (e.g. circles) on the way: by giving appropriate commands, in an appropriate order, you can create pictures and abstract patterns far more complex than you could possibly produce by hand, and which you can then save to disk or copy for use in documents or Web pages. The Program consists of a series of “statements”, most of which are simple commands (e.g. FORWARD, RIGHT, CIRCLE), while others create structures (e.g. procedures and “REPEAT ... UNTIL” loops) which determine the order in which the commands they contain are executed. The available statements are briefly listed in the [Programming Quick Reference](#) [19], and covered in more detail in [Programming Essentials](#) [23]. The latter also summarises the syntactic rules of Pascal, the well-known programming language which forms the basis for the system. If you find you need more detailed help on program layout, see the [Introduction to Pascal Syntax](#) [32].

Turtle graphics can seem child’s play to start with (and indeed quite young children are able to begin programming in this way), but the system is much more than a toy. Its most distinctive feature is the inclusion of a visual compiler, which translates the Program into a form of machine code – called “PCode” – for a “Turtle virtual machine” whose state can be inspected both during compilation and when the Program is run (i.e. when the PCode into which it has been compiled is executed). For a brief outline of these more sophisticated features and their educational motivation, see the section on [Teaching Aims: Deep Understanding of Programming Concepts](#) [4].

The System Menus

Details of the various system instructions are grouped according to the menu used to access them – there are six such menus, the [File Menu](#) [8], the [Edit Menu](#) [9], the [Layout Menu](#) [11], the [Compile Menu](#) [12], the [Options Menu](#) [15], and the [Help Menu](#) [17]. The [File](#) menu is used to load and save programs and to transfer images. The [Edit](#) menu provides various block-handling utilities (e.g. Cut/Copy/Paste and indenting), and an “auto-format” facility which neatly indents the entire program to reflect its structure. The [Layout](#) menu provides a choice of two different settings for the Canvas, and four different font sizes for the program editor, as well as an option to hide the Canvas in favour of a full-width editor. Other system settings are collected together in the [Options](#) menu; these enable programs to be processed immediately on loading (by compiling, running, and/or auto-formatting) and determine the precise behaviour of the auto-formatter, but also included are commands to save and load the various settings in the [Layout](#), [Compile](#), and [Options](#) menus. The [Compile](#) menu will be of interest to those who want to know something about the “innards” of the programming system – how the Program is translated into a form of “Machine Code” called [PCode](#) [42], and how this PCode determines what actually goes on when the Program is run. Finally the [Help](#) menu gives easy access to this Help file, and to a number of illustrative programs.

Exercises and Illustrative Programs – Start Learning Here!

Exercises to get you started are available through the [Exercises Page](#) [56]. If you are learning to program on a course which is based around this Turtle Graphics system, then you will be expected to work through these exercises, but even if you are not required to do them, these exercises provide a straightforward step-by-step way into the system. Also take a look at the illustrative programs available through the [Help Menu](#) [17] (but don't worry if you find some of these hard to understand until you've worked through the relevant parts of this Help system).

The Canvas

The Canvas is the square area on which drawing takes place as the *turtle* moves around. The *turtle* may move off the area of the Canvas (e.g. `forward(5000)` would normally have this effect), but no drawing is done beyond this area. Standardly the Canvas measures 1000 units by 1000 units, with the point (0,0) being at the top left – this follows the usual convention for computer graphics systems (in which, unlike mathematical co-ordinate geometry, the y-axis points downwards rather than upwards). Thus the *turtle*'s initial position in the middle of the Canvas has coordinates (500,500). This setup may be changed using the [Layout Menu](#) [11], which provides an option for a 2000x2000 Canvas with (0,0) in the middle. It is also possible to set the size and position of the Canvas within a program using the `CANVAS` command.

The Program

The Program is a sequence of commands – called “statements” – in the programming language *Pascal*, which you type into the Programming Area on the left of the screen, and which are executed when you click on the “RUN” button (or select “Run program” from the [Compile Menu](#) [12]). These commands then determine what is drawn on the Canvas (the square area on the right of the screen) – initially, you might find it helpful to think of them as commands being given to a small (and invisible) *turtle* which moves around the Canvas, leaving a coloured trail and drawing shapes as it goes.

The easiest way to see how all this works is to go to the [Help Menu](#) [17] at the top of the screen, select “Illustrative programs” and click on the first of these, called “Simple drawing with pauses”. This will load the following short program into the Programming Area:

```

PROGRAM drawpause;
BEGIN
  colour(green);
  blot(100);
  pause(1000);
  colour(red);
  forward(450);
  pause(1000);
  right(90);
  thickness(9);
  colour(blue);
  pause(1000);
  forward(300)
END.

```

If you now click on “RUN” you will see the effect of this program on the Canvas. Here is a brief explanation of each line of the program, which should be sufficient to enable you to write similar simple programs of your own:

Every program has to have a single word name (here *drawpause*), which is given after the word *PROGRAM* and followed by a semicolon; then the commands of the program itself are “bracketed” between *BEGIN* and *END* and separated from each other by semicolons (note that all the capitalisation is entirely optional – *PROGRAM*, *BEGIN* and *END* are shown in upper-case only for emphasis, and it would make no difference if you changed them to lower-case and/or capitalised other words). The final *END* must be followed by a full-stop, which signifies the very end of the program. In this example the first command given to the *turtle* is *colour(green)*, which specifies the drawing colour as green; then the next command *blot(100)* accordingly draws a green “blot” (i.e. a filled-in circle) of radius 100 units around the initial position of the *turtle* in the centre of the Canvas (standardly, the Canvas measures 1000 units square, so this blot will then have a diameter of one fifth of the Canvas). Next we have a *pause(1000)* command, which tells the *turtle* to pause for 1000 milliseconds (i.e. 1 second). Then after another *colour* command we have *forward(450)*, which instructs the *turtle* to move forward from its initial position by 450 units – this will leave a thin red trail, in accordance with the preceding *colour(red)* command. After another *pause(1000)*, the *turtle* is told to turn right by 90 degrees with the command *right(90)*, then *thickness(9)* tells it to thicken its pen from the standard 1 unit thickness to 9 units. This means that after *colour(blue)* and another *pause(1000)*, the final command *forward(300)* will draw a thick blue line of 300 units in what is now the direction of the *turtle* (i.e. horizontal, having turned right by 90 degrees from its original vertical direction). Having read through all this, run the program again (by clicking on “RUN”), taking note of the *turtle* status bar just above the Canvas, which records its current position and direction and the current thickness and colour of its pen.

For more details on programming commands and structures, see [Programming Essentials](#) [23], most of which is also summarised in the [Programming Quick Reference](#) [19]. However the best way to learn about the system is probably to look at a few more of the illustrative programs available through the [Help Menu](#) [17] to get a feel for what can be done, and then to start working through the [Exercises](#) [56] that are designed to take you through the system’s features step-by-step.

The Turtle Standalone Run-Time System

The program `TurtleRun.exe` is a standalone implementation of a “Turtle virtual machine”, enabling compiled TGC files (consisting of [PCode](#) [42] or “Turtle machine code”) to be run independently of the full Turtle Graphics system. Thus if you save a compiled TGC file using either “Save PCode file” from the [Compile Menu](#) [12], or the “Auto-save PCode on compiling” facility from the [Options Menu](#) [15], then such a file can be run by calling `TurtleRun.exe` with the relevant filename as a parameter, e.g.:

```
C:\> turtlerun.exe myprog.tgc
```

(Note that by far the most convenient way of doing this is to set up Windows in such a way that clicking on any TGC file within Windows Explorer automatically invokes `TurtleRun.exe` to run that file.)

Running TGC files within the standalone system can be aesthetically more satisfying than within the Turtle system's development environment, because it avoids the visual clutter of the full system's editor, menus, and other controls. Moreover the standalone system can be resized far more flexibly, enabling the Canvas to occupy any size up to the entire screen (if, having resized by hand, you want to ensure that the Canvas is perfectly square, just click on the little black square in the bottom-right-hand corner). TGC files have the advantage of being small and very quick to run, because they are pre-compiled. They can also be useful if for any reason you wish to hide the processing mechanisms from those who are running the program (e.g. in a teaching context, where you might ask students to work out the mechanisms for themselves).

TurtleRun.exe automatically runs the compiled TGC program as soon as it starts up, but execution can be halted by clicking the mouse on the Canvas, while double-clicking at any point restarts the program. When the program is restarted, a check is made to see if the TGC file has been modified since it was last run – if it has, then the new version will automatically be loaded and run unless this facility has been disabled (using the popup menu which will appear on right-clicking the Canvas). This automatic reloading facility can be conveniently combined with the “Auto-save PCode on compiling” facility from the Turtle system's [Options Menu](#) [15], to enable the standalone run-time system to be used interactively to view programs that are under development within the full Turtle system. This is particularly useful if you wish to take advantage of the full-width editor option provided under the [Layout Menu](#) [11], since then the full system can be used just for editing and compilation of the program, with display being handled by the run-time system.

Other options available through the popup menu include facilities for running and halting the program, loading a new PCode file, adjusting the automatic screen update setting (as in the main system's [Options Menu](#) [15]), and changing the speed at which the program executes (by adjusting the delay between successive commands).

Teaching Aims: Deep Understanding of Programming Concepts

Although its implementation of the Pascal language is far from complete (for example it makes no provision for complex data structures), this system does include some of the most important yet most commonly misunderstood features of modern programming languages, such as alternative parameter-passing mechanisms and recursion. Even university students of programming typically find these concepts hard to grasp, but in the Turtle Graphics context they can be made concrete by directly giving rise to visual patterns, often of considerable beauty. Moreover this Turtle Graphics system facilitates genuinely *deep* understanding of such concepts (as well as many others associated with compilation), by enabling students to see in detail how they can be implemented at the machine level. Published studies indicate that familiarity with these low-level mechanisms also consolidates general understanding of even high-level programming, by enriching and refining the students' mental model of how the computer is behaving (their “notional machine”).

The most distinctive feature of the system is that it contains a genuine compiler which “translates” the typed Pascal program into a sequence of numbers constituting a pseudo-machine-code program (so-called “PCode”) running on a virtual “Turtle Machine”. When the program is run, it is these numeric PCode commands (rather than the Pascal statements) that are executed, exactly as though they were being performed on a real Turtle Machine (i.e. a computer whose central processor chip is designed to respond directly to those numbers in an appropriate way). Moreover the system's “visual compiler” allows the PCode commands to be inspected directly – translated if desired from pure numbers into more comprehensible “assembler mnemonics” – and also allows them to be recorded (together with important Heap and Stack information) in an instruction-by-instruction “trace” as the program is executed. Not only do these features enable users to learn easily about the general concepts of machine code and compilation (including lexical and syntax analysis), but also, the simple Turtle Machine architecture provides an exceptionally straightforward way of learning about the complexities of Heap and Stack manipulation, and variable and parameter handling, which are central to the performance of any modern recursive programming language. For details of these more advanced features, see [Compile Menu](#) [12], [Introduction to PCode](#) [42], [Procedures and Parameters](#) [34], and [Technical Note on Variables, Procedures and Parameters](#) [44].

Current Version of the Turtle Software

The current distribution is Version 7.01 of the Turtle Graphics software, released in November 2003, a very minor upgrade to Version 7.0 (September 2003). The first version of Turtle to be published on the Web was Version 5, released in February 2001, and the details below catalogue the main changes to be made since then. Version 6 (October 2001) introduced the following improvements:

- * On startup, the initial state of the *turtle* is displayed. Likewise on any setting of the Canvas size options (in which case the Canvas is also cleared if the Canvas clear option is selected)
- * Type BOOLEAN is permitted as well as INTEGER, though without any type checking (hence BOOLEAN and INTEGER act equivalently)
- * Boolean constants TRUE and FALSE defined
- * Colour codes can be specified using the “#” or “RGB” convention as well as “\$” or “BGR”
- * RANDCOL makes *turtc* equal to a randomly chosen short code rather than the colour code equivalent, enabling it to be used as a random number generator within the 1 to 8 range (extended in Version 7.0)
- * The save program and save image dialogues ask for confirmation before overwriting an existing file
- * The File menu provides both a “Save” and a “Save As” option
- * Four additional illustrative programs available directly through the Help menu

Version 6a (released in November 2001) added the following improvements:

- * Refinement of the program loading routine, enabling it to deal intelligently with non-standard line end codes (often caused by corruption due to FTP or email)
- * The auto-format no longer converts “#” hexadecimals into the “\$” format, but it capitalises hexadecimal digits.
- * New BLANK command, enabling the entire Canvas to be blanked out with a specified colour
- * New CANVAS command, so that the Canvas dimensions and location can be set within a program to any desired values (Options menu captions also changed accordingly)

Version 6b (December 2001) added the following improvements:

- * “Exit Turtle Graphics” command added to the File menu, and the Compile menu rationalised by providing separate “Run” and “Halt” commands, and combining opposite Analysis and Trace switches into single toggled items.
- * New entry in Compile menu to allow Turtle Graphics Compiled PCode (TGC) files to be saved, for independent execution on the Turtle Graphics Runtime system (program TurtleRun.exe). Also corresponding change to Options menu to enable such PCode files to be saved automatically on compilation.
- * Other menus also reorganised, with a new Layout menu taking the Canvas- and font-related options, these being replaced within the Options menu by a range of automatic processing and configuration facilities. One of these, for example, allows programs to be run automatically on loading.
- * New full-width editor facility provided within Layout menu.
- * Automatic screen updating checks included in Options menu, to ensure that Turtle programs can’t “hang” through use of the NOUPDATE command.
- * Automatic configuration facilities include provision for Turtle Graphics Option (TGO) files, so that system configurations can be saved and reloaded; this also enables a default system configuration to be set up very easily, so that preferred settings are automatically selected on startup.
- * Keyboard shortcuts introduced for the auto-format, analysis tables and trace facilities.
- * New EXPRESSIONS tab added to compilation analysis, enabling commands and structures used in the program to be rapidly identified (particularly useful in a teaching context, but also providing a way of rapidly finding particular features within the program).

Version 6c (January 2002) incorporated the following improvements:

- * Addition of the Turtle Standalone Run-Time System
- * 100-step Undo/Redo facility added to the Edit menu.
- * “Are you sure” prompting made more selective, so that such prompts are given only if the current program has been edited since the last load or save operation.
- * Form and component scaling to enable system to run with “large fonts” even on 800x600 screen resolution.

Version 6d (June 2002) refined some of the system’s error messages to deal with particular special cases, fixed a bug in the Layout menu radio groupings, and added three sections to the Help file (namely the current section, that on the Standalone Run-Time System, and the Pascal Syntax Reference listing all the system’s reserved words or keywords).

Version 7.0 (September 2003) removed the “coursework” section from the Help file, to make the documentation independent of the Leeds University setting, and included a number of system changes that had been cumulatively developed over the previous year:

- * Turtle machine enhanced to enable it to operate with only a single stack, and corresponding options added to the Compile menu, along with another option for displaying the stack height in the trace table.
- * Improvements to the compiler’s error handling, in the lexical analyser (with ill-formed “numbers”, and to avoid multiple messages), when checking for duplicate identifiers, and to treat null “then” or “else” statements as errors only if they occur before a semicolon (since then the semicolon is almost certainly misplaced – a very common novice’s error).
- * All variables initialised to zero automatically.
- * RANDCOL no longer limited to 1 to 8 range – can be used as random number generator over any integer range.
- * Mutual recursion facilitated.
- * Various enhancements to the Standalone Run-Time System’s facilities, including file loading, screen updating, and execution speed adjustment.
- * Load/save dialogue directory management settings added to Options menu.

Version 7.01 (November 2003) made a number of minor corrections within this Help file, changed the order of items within the Compile menu, and extended the dialogue directory management to the saving of bitmap files.

Changes planned or considered for the relatively near future include type-checking for Boolean and integer variables, introduction of a “while” structure, functions, various editor improvements (including multiple files, search and replace, and possibly macros), extension of the auto-formatter to provide more assistance for error detection, and array or string variables (arrays, if implemented, could usefully be combined with an extension to the trace display enabling the literal counting of operations in, for example, sorting routines). More ambitious future possibilities include user-defined event handling, decompilation and the incorporation of alternative program syntaxes (e.g. BASIC- and C-style as well as Pascal) with mutual translation. Most ambitious of all would be a move to Java with full object-orientation and multiple turtles, ideally with the possibility of being ported to Linux at the same time.

I would like to express my gratitude to Dr Sarah Kattau for her support in an ACOM (“Computing for All”) module based on the Turtle system at Leeds University, her care and attention to detail when rigorously testing the system, and her many valuable suggestions which have led to significant improvements.

Peter Millican
University of Leeds
November 2003

Hardware Requirements and Issues

The Turtle Graphics software is designed to run under any 32-bit Windows operating system (Windows 95, Windows 98, Windows Millenium, Windows NT, Windows 2000, Windows XP etc.), though most practical testing has taken place under Windows NT and XP. It requires only a minimum screen resolution of 800x600, though a larger screen brings the advantage of more program editing space (achievable either by maximising or by dragging to enlarge the window). It works best with SMALL SCREEN FONTS (adjustable using Control Panel | Display), though it can run also with large fonts, at the cost of some inelegance in the compilation analysis tables (notably that some of the headings in the compilation tables are likely to overlap). If large fonts are used with the minimum 800x600 resolution, then the very bottom of the system window may extend below the visible area of the screen. Hence to preserve full functionality, whenever the compilation analysis tables are displayed with large fonts, the table page selection tabs will automatically appear above the Canvas rather than below.

Notification of any observed hardware problems will be particularly welcomed, given the impracticality of fully comprehensive testing under the wide variety of operating systems and screen configurations that are possible. Please send details of any such problems you may encounter to **p.j.r.millican@leeds.ac.uk**.

Menus in the Turtle Graphics System

File menu

The File menu provides facilities for clearing, loading, and saving the Program [2], and for transferring images drawn on the Canvas [2] either to the Windows Clipboard or to a disk file. Turtle Graphics program files are normally presumed to have the extension TGP, but are simple text files with each line corresponding to a line of the program (hence they can be edited in a standard text editor such as *Notepad*). Turtle Graphics images are saved as bitmaps for maximum portability to other systems. See the Options Menu [15] for details of how the system handles the initial directories chosen within the loading and saving dialogues.

File | New program

Clears the existing Program, so that you can then begin to input a totally new program.

File | Load program

Invokes a file selection dialogue, from which you can choose a TGP (Turtle Graphics Pascal) file to be loaded. The existing program statements are cleared before the new file is loaded, and the cyan (turquoise) label above the program listing is replaced with the name of the newly loaded file. Note that if you set up Windows to recognise the Turtle system as the appropriate program to run TGP files, then clicking on a TGP file within Windows Explorer will automatically start up the system with that file loaded.

File | Save program

Saves the current program under the existing name (i.e. the name of the file as originally loaded or most recently saved). If there is no existing file name, then the “Save As” option below is invoked instead.

File | Save As ...

Invokes a file selection dialogue, from which you can choose a name to be given to the existing Program, which is saved as a file under that name. By default, the file will be given the extension TGP (Turtle Graphics Pascal). The existing program statements remain unaffected, but the cyan (turquoise) label above the program listing is replaced with the name that you have given to the saved file.

File | Image to Clipboard

Transfers the current image drawn on the Canvas to the Windows Clipboard (as a bitmap), from where it can be pasted into a graphics editor or a wordprocessor document etc.

File | Image to File

Invokes a file selection dialogue, from which you can choose a name to be given to the current image drawn on the Canvas, which is saved as a file under that name. By default, the file will be given the extension BMP (because the image is saved as a bitmap).

File | Exit Turtle Graphics

Exits the system, but if any editing has taken place, first warns that the current program will be lost if this is done, and invites you to cancel.

Edit menu

The Edit menu provides “undo” and “redo” facilities, four simple operations involving the Windows Clipboard (which can therefore be used for transferring text between the Turtle system and an alternative program editor), and three operations concerned with program line indentation (to help you to lay your programs out neatly).

To use most of these editing functions, you need first to select the lines of the Program to which they are to apply (though in the case of paste Over, you can select just the first line where the pasting is to take place, while with Paste insert you need only position the cursor appropriately). Select the desired range of lines by holding down the left mouse button as you drag it over those lines, and then release the mouse button when the range you want is highlighted in blue. To select a range that extends beyond the visible area of the Program, drag the mouse above or below the Programming Area, and the Program will scroll accordingly (you can also use the PAGEUP or PAGEDOWN keys with similar effect).

Edit | Undo

Undoes the last editing operation to be performed on the current program (except for auto-formatting, which cannot be undone). The system stores up to one hundred editing operations, with any continuous editing that you perform on a single line of the program counting as one such operation. As in most other Windows systems, the shortcut CTRL-Z can be used to perform an “Undo” directly from the keyboard.

Edit | Redo

Re-performs the last editing operation to be “Undone”, as long as no other editing has been done in the meantime. Can also be used repeatedly to “Redo” a sequence of up to 100 editing operations that have been “Undone”. As in most other Windows systems, the shortcut CTRL-Y can be used to perform a “Redo” directly from the keyboard.

Edit | Cut

Removes the selected range of program lines into the Windows Clipboard, and deletes them from the Program. They can be restored immediately to any point in the Program by selecting Edit | Paste insert. As in most other Windows systems, the shortcut CTRL-X can be used to perform a “Cut” directly from the keyboard.

Edit | Copy

Copies the selected range of program lines into the Windows Clipboard. The Program itself remains unaffected. As in most other Windows systems, the shortcut CTRL-C can be used to perform a “Copy” directly from the keyboard.

Edit | Paste insert

Inserts the program statements from the Windows Clipboard into the Program, starting at the first selected line or, if no line is selected, from the line where the cursor is situated. The existing lines of the Program are retained, but pushed down to make space for the new lines from the Clipboard. As in most other Windows systems, the shortcut CTRL-V can be used to perform a “Paste” (of this kind) directly from the keyboard.

Edit | paste Over

Pastes the program statements from the Windows Clipboard over the selected range of lines (so the existing program lines are overwritten). However if the selected range is more than a single line, but does not contain the right number of lines for the Clipboard, then no pasting will take place. If you select only a single line of the Program before pasting, the pasting will be done, into that line and successive lines. The keyboard shortcut for this kind of “Paste” is CTRL-B.

Edit | Indent program lines

Indents the selected lines of the Program by one space, which is an effective way of highlighting the scope of procedures, REPEAT loops etc.

Edit | Unindent program lines

Unindents the selected lines of the Program, removing one space from the beginning of each such line (any line that does not begin with a space is unaffected).

Edit | Auto-format program

Formats neatly *the entire program* (so does not require prior selection of lines), with intelligent indentation and blank lines to highlight the program structure, and optionally – depending on your choice in the Options Menu [15] – capitalisation of keywords that signify the start and finish of major program blocks (and also of hexadecimal digits, and of the keyword VAR to highlight declarations and uses of reference parameters). Another choice in the Options Menu allows you to delete comments from the program if you wish (this can be useful for removing the clutter of explanations from the illustrative programs available through the Help Menu [17]). Note that auto-formatting will overwrite your original program and cannot be “Undone”, so it is sensible to save your program first, in case you do not like the results of the auto-format – the system will automatically display a prompt to remind you of this when necessary. Note also that auto-formatting involves full compilation of the program to establish its structure, and so cannot be performed on a program which contains syntactic errors; if any are encountered, the format will remain unchanged and an appropriate error message will be displayed, exactly as if an attempt had been made to compile the program.

Layout menu

The Layout menu provides two possible choices for the Canvas dimensions that are set when a program is run (but note that the CANVAS command can change these dimensions while the program is running). There are also four choices of font for the program text, enabling additional text to be displayed (in return for a loss of clarity).

Layout | Canvas (0,0) to (1000,1000)

This default option sets the dimensions of the Canvas [2] as 1000 x 1000, ranging from 0 to 1000 along both the x and the y axes. The initial position of the *turtle* is in the centre of the Canvas, at the point (500,500).

Layout | Canvas (-1000,-1000) to (1000,1000)

This option sets the dimensions of the Canvas [2] as 2000 x 2000, ranging from -1000 to 1000 along both the x and the y axes. The initial position of the *turtle* is in the centre of the Canvas, at the point (0,0).

Layout | Display editor and Canvas

This default option displays the program editor at the left of the screen, with the Canvas occupying the right. The program editor can be expanded by dragging the system window, but the Canvas always remains the same size. To see the program running on a resizable Canvas, use the Turtle Standalone Run-Time System [3].

Layout | Display Editor (full-width) only

This option completely hides the Canvas, so as to allow the program editor to occupy the full width of the system window. This can be helpful when editing complex programs, especially if the screen size is limited, and it can be combined with use of the Turtle Standalone Run-Time System [3] (TurtleRun.exe) for program display.

Layout | 10-point font for program

This default option sets a 10-point font (either Courier New or Ariel Narrow) for the Programming Area. You may find this easier to read than 9-point, but it restricts the number of characters you can see per line (especially if you are using a screen resolution of 800x600 or less, in which case maximising the Turtle System does not help).

Layout | 9-point font for program

This option sets a 9-point font (either Courier New or Ariel Narrow) for the Programming Area, and may be useful if you are examining a program whose lines are too long for the space available when displayed in 10-point type.

Layout | Courier New font for program

This default option sets the font type for the Programming Area as Courier New. This has the advantage of being fixed pitch (which tends to look neater with program listings) but also has the corresponding disadvantage of taking up a lot of space.

Layout | Ariel Narrow font for program

This option sets the font type for the Programming Area as Ariel Narrow, and is useful if with the Courier New font your program lines are too long to be seen completely in the space available. For maximum visibility in a limited space, use this option together with the 9-point font option.

Compile menu

The Compile menu controls the *compilation* of the Program – that is, the Program analysis and its translation into PCode which takes place just before it is run. Normally the Program will be compiled only when you click on the “RUN” button (though this menu also provides keyboard shortcuts for running and halting), and all you will see of the process will be an error message if something is wrong (e.g. if you’ve mistyped a command, or omitted a semicolon). If you are a typical user, then this will probably be all you want to see! However if you are interested in the technical details of how the system works, or are studying compiler design or machine code, then the options in the Compile menu enable you to do and see much more.

Compile | Compile to PCode

Compiles the Program without running it, which can be useful if you have made changes to the Program and wish to see how it compiles (or to check that it does indeed compile successfully), but without affecting the output on the Canvas. This also forces the Program to re-compile, unlike for example the “RUN” button or the auto-format facility, which re-compile the Program only if it has been modified since the last compilation.

Compile | Run program

When no program is running, equivalent to clicking on the “RUN”/“HALT” button, but has the advantage that it can be invoked quickly from the keyboard through the shortcut CTRL+R, and is unambiguous.

Compile | Halt program

When a program is running, equivalent to clicking on the “RUN”/“HALT” button, but has the advantage that it can be invoked quickly from the keyboard through the shortcut CTRL+H, and is unambiguous. If a program is running without continuous screen updating then this is the surest way of ensuring that it terminates, since clicking on the “RUN” button twice can have the effect of stopping and then re-running the program.

Compile | Save PCode file

Compiles the Program (unless this has already been done), and if compilation is successful, invokes a file selection dialogue, enabling you to save the compiled code as a TGC (Turtle Graphics Compiled PCode) file. This file can be run independently of the Turtle Graphics Programming System, by calling the Turtle Standalone Run-Time System [3] (TurtleRun.exe) with the new filename as a parameter.

The remaining menu items are “toggles” – that is, options which are turned “on” and “off” alternately using the same “switch” mechanism of clicking on the relevant menu item. When the option is “on” (i.e. “selected” or “checked”), a small tick will appear to the left of the menu item; if no tick appears there, then the option is “off”.

Compile | Analysis tables

Whenever compilation takes place a compilation analysis is created, though by default it is hidden (to make the interface as simple as possible for beginners). If this option is selected, then the analysis is shown in a sequence of tables that can be viewed in place of the Canvas by clicking on the appropriate tab. These tables are as follows:

1. Lexemes shows the lines of the Program divided into *lexemes* (i.e. discrete lexical items), and for each lexeme gives the number of the program line, the “index” number of the lexeme, the *type* of the lexeme (e.g. identifier, number, or the lexeme itself if it’s a *keyword* or character symbol) and the *Finite State Machine* (FSM) states that were involved in the syntactic analysis of the Program at that point. Most of the FSM states will only be of interest if you are studying the process of syntactic analysis itself, but the states that arise in the body of the Program or within procedures (labelled “PROG” and “PROC” respectively) also show where the various structures of the Program start and finish (e.g. “PROG TFB” indicates part of the main program within a BEGIN ... END section that is itself within a FOR loop following an IF ... THEN, while “PROC ER” indicates part of a procedure within a REPEAT loop which itself directly follows an ELSE).
2. Expressions gives details of which commands and structures have been used within the program, how frequently and where, and also provides totals for various categories of commands or structures.
3. Declarations is divided into two sections, showing respectively the variables and the procedures that are declared in the program.

The Variables table lists all of the variables or parameters declared in the Program, together with their scope (i.e. the procedure in which they are declared, if not the entire program), their “index” value (e.g. the second parameter or variable declared within a procedure has an index of 2), their type (either “integer” or “boolean”), whether they are standard “call by value” variables/parameters or “call by reference” parameters (see the discussion of procedure parameters [34] for an explanation of this important distinction), and the range from the BEGIN to the END that enclose each variable’s scope (this range is expressed both in terms of program lines, and PCode [42] commands). Note that the first five global variables are always *turtx*, *turty*, *turtd*, *turtt* and *turtc*, giving you direct access at all times to the *turtle*’s current x- and y-coordinates, its direction, its thickness and its colour respectively.

The Procedures table lists all of the procedures in the Program, together with their “parent” (i.e. the procedure in which they are declared, if not the entire program), the number of parameters [34] each takes, their “Heap storage allocation” (i.e. the total number of parameters plus local variables), and the range from the BEGIN to the END that enclose each procedure (just as in the Variables table, this range is expressed both in terms of program lines, and PCode commands).

4. PCode shows how the Pascal statements have been analysed into an intermediate form of code called “PCode”. This is a sort of virtual “machine code”, and provides an example of the kind of thing that is produced by a conventional compilation process. When the Turtle Graphics program is run, it is the PCode commands that are actually executed, so if you understand how these work, you can check whether the Program is operating in the way that you expected. For an explanation, see the Introduction to PCode [42].

If the “Trace on run” option has been selected from the Compile menu, then the bottom part of the PCode panel will show a “trace” of the PCode commands that are actually being executed as the Program is run. Each line of this trace display shows:

- the “cycle number” (i.e. the number of commands executed since the beginning of the program);
- the PCode instruction address (i.e. where in the compiled PCode the particular instruction being executed is located);
- the PCode instruction mnemonic (i.e. the four-letter symbol for the particular type of instruction);
- the parameters to that instruction (represented by the immediately following value(s) in the compiled PCode);
- the state of the runtime flags (i.e. whether the pen is up “Pu” or down “Pd”, and whether the display is updating “U” or not “N”);
- the Heap size on completion of the command (i.e. the amount of space currently occupied by global and local variables and parameters);

- the value of the Procedure Register at that point and the number of procedures currently in progress (the latter figure being the depth of the Procedure Register Stack);
- the top values stored on the Program Stack (up to three values will be shown, starting from the top of the Stack; note that the first of these columns is wide enough for a full 4-byte hexadecimal number to be seen if one of the hexadecimal radio buttons at the top of the PCode tab was selected when the program was running); see also the “Trace Stack height” option mentioned below.

For an explanation of how the Heap, Program Stack and Procedure Register Stack operate, see the [Technical Note on Variables, Procedures and Parameters](#) [44].

If the Program involves significant amounts of recursion or “looping” (e.g. REPEAT ... UNTIL sequences that are executed many times) then the trace display may get extremely long, and this will slow things down *considerably*. So it is advisable to activate the trace display only if you are examining it in order to see how the Program operates in detail, or to identify some error in its execution.

Compile | Trace on run

If this option is selected, then when the Program is run, a trace display will appear below the [PCode](#) table in the compilation analysis, as described above. This table will list each PCode command of the Program as it is executed, enabling you to look in detail through the execution sequence of the Program. If the Program involves significant amounts of “looping” then the trace display table can grow very long, which *considerably* slows down its execution.

Compile | Trace Stack height instead of Stack location 3

As described earlier, by default the last three columns of the trace display show the contents of the top three Program Stack locations. If this option is selected, then the last of these columns instead shows the height of the Program Stack, which can be particularly useful if the Stack grows fairly high because it is used to store procedure information (as explained in the last three options below).

Compile | Turtle machine using separate Return Stack

As explained in the [Technical Note on Variables, Procedures and Parameters](#) [44], the Turtle machine uses a Return Stack to store the PCode line numbers to which control should return when each procedure terminates. To simplify the introduction of these concepts, by default this Return Stack is kept quite separate from the main Program Stack, and hence this menu option is selected (i.e. “checked”). However if it is deselected, the Turtle machine will instead store its procedure return line numbers on the Program Stack itself, with the compiled PCode using the instruction LDRJ to load the return line number onto the Stack, and PLRJ at the end of the procedure (rather than ENDP) to pull the line number from the Stack and make the return jump.

Compile | Turtle machine using separate Heap Control Stack

Much like the previous option, if this is deselected then the Turtle machine will perform its Heap control operations using the Program Stack (with the PCode instructions LDHT, STHT, LDHB, and STHB). By default, however, the machine uses a separate Heap Control Stack (with the instructions HPCL and HPRE) as explained in the [Technical Note on Variables, Procedures and Parameters](#) [44].

Compile | Turtle machine using Procedure Register Stack

Like the previous two options this is selected by default. If it is deselected, the Turtle compiler will not generate PCode commands (PSPR and PLPR) to maintain the Procedure Register Stack, and accordingly the corresponding column of the Trace display table will show “0/0” when such compiled PCode is run.

Options menu

Most of the settings in this menu concern various automatic operations, enabling the system to be configured so that programs are processed immediately on loading (by compiling, running, and/or auto-formatting), the Canvas either cleared or preserved when a new program is run, and the auto-formatter's behaviour controlled. Later options control the interaction of directories within the various loading and saving dialogues, determining for example whether the system will expect a renamed or compiled program to be saved in the same directory from which the original program was loaded. The last two items on the menu provide commands for loading and saving option configuration files that control the settings in the Layout, Compile, and Options menus – this provides even more scope for automatic control, by enabling you to determine the settings that are initially invoked when the system is run.

Most of these items are “toggles” – that is, options which are turned “on” and “off” alternately using the same “switch” mechanism of clicking on the relevant menu item. When the option is “on”, a small tick will appear to the left of the menu item; if no tick appears there, then the option is “off”. The screen updating and directory handling options, however, involve “radio buttons”, with a bullet appearing to the left of whichever item is currently operative.

Options | Auto-Compile on loading

If this option is selected, then any program which is loaded will automatically be compiled, enabling any compilation errors in the program to be detected immediately and also generating the program analysis tables which can be viewed as described under the Compile Menu [12]. This option is particularly useful if you need to look quickly through a number of programs, checking their correctness and viewing their analyses (e.g. for marking students' work in an academic context).

Options | Auto-Run on loading

If this option is selected, then any program which is loaded will automatically be compiled and run, enabling its behaviour to be seen immediately. This is particularly useful if you are browsing through a number of programs, with an interest in the patterns they produce.

Options | Auto-Format on loading

If this option is selected, then any program which is loaded will automatically be compiled and auto-formatted (and then possibly run, depending on the previous option). This can be useful if you are browsing through a number of programs and want to ensure that they are easy to read, but also if you want to impose on yourself the discipline of ensuring that whenever you return to editing a program, you start from a neat format.

Options | Auto-save PCode on compiling

If this option is selected, then whenever compilation takes place, the compiled code will be saved as a TGC (Turtle Graphics Compiled PCode) file, with the directory being determined according to the selected option later in this menu, and the filename matching the name of the TGP Pascal source code file. The PCode file can be run independently of the Turtle Graphics Programming System, by calling the Turtle Standalone Run-Time System [3] (TurtleRun.exe) with the TGC filename as a parameter.

Options | Upper/lower case on auto-format

This option determines whether or not the auto-format facility in the [Edit Menu](#) [9] will adjust the capitalisation of your program. If it is selected (as it is by default), then most of the words in your program will appear in lower-case, with the exception of certain keywords that signify the start and finish of major program blocks (also VAR, indicating variable declarations or the occurrence of reference parameters, and hexadecimal digits), which will appear in upper-case.

Options | Strip comments on auto-format

This option determines whether the auto-format facility in the [Edit Menu](#) [9] will retain or remove any comments in your program. If comments are retained, they will be placed at the same point in the program where they were originally situated, separated from the program text by a space (sequences of comments together will appear on successive lines).

Options | Blank Canvas on RUN

Standardly, the Canvas is made blank (i.e. white all over) whenever a program is run. If this option is deselected, the Canvas will not be blanked when you click on "RUN", so you will be able to superimpose patterns on each other to create more complex effects.

Options | Update every 100,000 cycles

This option ensures that if at any point the program has run for 100,000 cycles since the last screen update, then it will be updated even if the program command NOUPDATE is currently in operation (the update status within the program is not affected, and a new cycle count starts when the forced update is made). This prevents the program from hanging, and ensures that CTRL-H ("Halt" - see [Compile Menu](#) [12]) will eventually operate when the count reaches 100,000.

Options | Update every 300,000 cycles

Similar to the previous option, except that it allows the count of non-updated cycles to reach 300,000 before forcing an update (so you may have to wait three times longer for CTRL-H to take effect). This is the default setting.

Options | Update every million cycles

Similar to the previous two options, except that it allows the count of non-updated cycles to reach 1,000,000 before forcing an update.

Options | Never force updating (may hang)

Never forces screen updating no matter how many cycles occur. If the NOUPDATE command is used in a program which fails to terminate while this option is selected, the Turtle system may hang, so this should only be used if it is important for some reason to allow a particular program to run more than a million cycles without updating.

Options | Independent load/save directories

The system's File Menu [8] and Compile Menu [12] provide facilities for loading TGP (Turtle Graphics Pascal) files, saving TGP files, and saving TGC (Turtle Graphics Compiled PCode) and BMP (bitmap graphics) files. If this option is selected then the initial directories for the relevant load/save dialogues will be managed independently, so for example the "Save As" facility will not automatically expect an edited version of a recently loaded TGP file to be saved within the same directory from which the original was loaded. (Filenames are also treated relatively independently, in that the load dialogue's memory of the last-loaded TGP file will not be affected by the use of "Save As".) This option is useful if a number of programs are being processed in turn, with the edited or compiled versions being saved into a different directory.

Options | Linked load/save directories

In contrast with the previous option, this is designed to deal with the standard situation where each individual program is being edited and developed within a specific directory, and accordingly the initial directory of the various load/save dialogues are all reset to whichever directory was most recently used for loading or saving a TGP file. So for example the "Save As" facility will expect an edited version of a recently loaded TGP file to be saved within the same directory from which the original was loaded. Linking also affects the filename interplay between the dialogues, in that after a "Save As" operation (which is typically used to change a program's name), the "Load program" dialogue will no longer present the latest filename to be loaded.

Options | Flexible load/save directory handling

If this default option is set, then the system attempts to handle the initial directories of the various load/save dialogues in a way that automatically optimises their co-ordination. When a TGP file is loaded, the system records which directory it was loaded from, and then assumes by default that the next "Save As" or "Save PCode file" operation should start from that same directory (i.e. the directories are linked much as in the previous option). However if the next "Save As" or "Save PCode" operation ends up using a different directory, then the link between the load dialogue and the corresponding save dialogue is temporarily broken, so that the two become independent (as in the independent directories option above). The link will then be restored again when a relevant save operation is performed using the same directory from which the TGP file was loaded.

Options | Load settings from options file

This command is used to load settings for the options in the Layout, Compile, and Options menus, from a Turtle Graphics Options (TGO) file selected by the user, thus enabling any chosen system configuration to be set up quickly (and different configurations to be used for different purposes). If you have an options file named "DEFAULT.TGO" in the same folder as the Turtle system, then this will be loaded automatically when the system starts up, without requiring any use of the Options menu. To create an options file, use the command below.

Options | Save settings to options file

This command is used to create a Turtle Graphics Options (TGO) file which records all of the current settings of the various options in the Layout, Compile, and Options menus. This is a plain text file which can then be edited to remove any options that you do not wish to treat as part of your set configuration.

Help menu

This menu provides direct access to various sections of this Help file, and also to a number of illustrative programs that are built into the system.

Help | Turtle Help

Displays the Help file contents, as determined by the “Turtle.cnt” file which is part of the package. Double-clicking on one of the headings will expand the sub-headings below it, and double-clicking on any of these sub-headings will then display the corresponding Help file section.

Help | Quick reference

Displays the [Programming Quick Reference](#) [19] section of this Help file, which is likely to be the one that will be referred to most often when programming within the system.

Help | Illustrative programs

Holding the mouse over this option brings up a sub-menu of illustrative programs. Clicking on one of the named programs will then load it into the system, ready to run. The programs provided are as follows:

Name within <u>Help</u> sub-menu	Name within program
Simple drawing with pauses	drawpause
FOR (counting) loop	forloops
Nested FOR loops	nestedloops
Simple procedure	simpleproc
Procedure with parameter	parameterproc
Recursion	triangles
REPEAT loop	repeatloop
Combining structures	ballsteps
Reference (VAR) parameters	refparams
Multiple bouncing balls	multibounce
Cycling colours	cyclecolours
Using Booleans	flashlights
Using POLYGON with FORGET	polygonrings
3-D effects with colour	balls3D

Help | Recursion factory

Loads the [The Recursion Factory](#) [39] illustrative program into the system, and also displays the corresponding page from this Help file, which gives advice for producing a wide variety of intricately recursive patterns using the program.

Help | Exercises

Displays the [Exercises Page](#) [56] of this Help file, which contains 12 exercises designed to introduce the main features of the system and the essentials of programming, in a carefully graded manner suitable for self-teaching.

Turtle Graphics Programming

Programming Quick Reference

For quick reference, here is a list of the commands and examples of the structures available, under the same categories that are used in [Programming Essentials](#) [23] (which should be consulted if you need more details).

Program Structure

The following example is provided as a quick memory-aid – if you don't understand any of it, and want a simple introduction to program layout, see the section on "Program Structure and Declarations" at the beginning of [Programming Essentials](#) [23].

```
e.g.  PROGRAM anyname;
      VAR global1,global2: integer;
          global3: integer;

      PROCEDURE proc1;
      VAR local1: integer;
      BEGIN
        {body of the procedure PROC1}
      END;

      PROCEDURE proc2(param1,param2: integer);
      VAR local1: integer;
      BEGIN
        {body of the procedure PROC2}
        {note that LOCAL1 in PROC2 is a quite}
        {different variable from LOCAL1 in PROC1}
      END;

      PROCEDURE proc3(param1: integer; VAR param2: integer);
      BEGIN
        {body of the procedure PROC3}
        {note that PARAM1 and PARAM2 in PROC3 are quite}
        {different variables from PARAM1 and PARAM2 in PROC2}
        {note also that PARAM2 here is a "reference" parameter}
      END;

      BEGIN
        {body of the program ANYNAME}
      END.
```

Variable Assignment, Arithmetical and Boolean Operators and Constants

v := n make variable *v* equal to value *n*.

Arithmetic operators: + - * / MOD

Comparisons: = <> < > <= >=

Boolean operators: NOT AND OR XOR

Boolean constants: TRUE FALSE (interpreted numerically as -1 and 0 respectively, so that the Boolean operators can act also as bitwise numerical operators)

Turtle Movement – Relative (i.e. relative to the *turtle's* current position)

FORWARD(n) move forward by *n* units, drawing if the pen is down (may be abbreviated to FD).

BACK(n) move backwards by *n* units, drawing if the pen is down (may be abbreviated to BK).

LEFT(n) turn to the left by *n* degrees (may be abbreviated to LT).

RIGHT(n) turn to the right by *n* degrees (may be abbreviated to RT).

(the turtle's direction may also be set by assignment to the global variable turtd)

MOVEXY(n1,n2) move a distance *n1* in the x-direction, and *n2* in the y-direction, without drawing.

DRAWXY(n1,n2) move a distance *n1* in the x-direction, and *n2* in the y-direction, drawing if the pen is down.

Turtle Movement – Absolute (i.e. to a specific location on the Canvas)

HOME return the *turtle* to its original starting point in the centre of the Canvas (without drawing), and restore its original direction (i.e. "North" or 0 degrees).

SETX(n) move parallel to the x-axis (i.e. horizontally) to a position whose x-coordinate is *n*, without drawing or changing the *turtle's* direction.

SETY(n) move parallel to the y-axis (i.e. vertically) to a position whose y-coordinate is *n*, without drawing or changing the *turtle's* direction.

SETXY(n1,n2) move to a position whose x-coordinate is *n1* and whose y-coordinate is *n2*, without drawing or changing the *turtle's* direction.

(the turtle's absolute position may also be set by assignment to the global variables turtx and turty)

Shape Drawing

<u>CIRCLE</u> (<i>n</i>)	draw the outline of a circle, radius <i>n</i> units, around the current position of the <i>turtle</i> .
<u>BLOT</u> (<i>n</i>)	draw a filled circle, radius <i>n</i> units, centred on the current position of the <i>turtle</i> .
<u>POLYLINE</u> (<i>n</i>)	draw a sequence of <i>n-1</i> straight lines connecting together the <i>n</i> points which the <i>turtle</i> has most recently visited using any of the <i>turtle</i> movement commands listed above.
<u>POLYGON</u> (<i>n</i>)	draw a filled shape bounded by straight lines connecting together the <i>n</i> points which the <i>turtle</i> has most recently visited using any of the <i>turtle</i> movement commands listed above.
<u>FORGET</u> (<i>n</i>)	“forget” about the last <i>n</i> points that the <i>turtle</i> has visited, for the purposes of POLYLINE and POLYGON.
<u>REMEMBER</u>	“remember” the point where the <i>turtle</i> is currently located (used to record points that have been moved to by direct assignment to the global variables <i>turtx</i> and <i>turty</i>).

Drawing Control

<u>THICKNESS</u> (<i>n</i>)	set the thickness of the <i>turtle</i> 's pen to <i>n</i> pixels. <i>(the turtle's thickness may also be set by assignment to the global variable turtt)</i>
<u>COLOUR</u> (<i>c</i>)	set the colour of the <i>turtle</i> 's pen to any of BLUE, GREEN, CYAN, RED, MAGENTA, YELLOW, WHITE, and BLACK (for which the numbers 1 to 8 may also be used), or to a hexadecimal code which specifies the red, green and blue components in either the “\$bbgrr” or “#rrggbb” form. COLOUR may be written as COLOR. <i>(the turtle's colour may also be set by assignment to the global variable turtc)</i>
<u>RANDCOL</u> (<i>n</i>)	randomly choose a colour for the <i>turtle</i> 's pen, making this choice within the range 1.. <i>n</i> (hence if <i>n</i> is at most 8, a choice will be made from the first <i>n</i> of the standard colours). Thus <i>randcol</i> (6), for example, makes <i>turtc</i> equal to a random number between 1 and 6 inclusive. Values of <i>n</i> greater than 8 are also usable, especially if the generated value of <i>turtc</i> is intended for use as a random number rather than a colour code.
<u>BLANK</u> (<i>c</i>)	blank out the entire Canvas with the specified colour <i>c</i> .
<u>PENUP</u>	raise the <i>turtle</i> 's pen, so no drawing is done as the <i>turtle</i> moves around the Canvas (may be abbreviated to PU).
<u>PENDOWN</u>	lower the <i>turtle</i> 's pen (may be abbreviated to PD).
<u>NOUPDATE</u>	stop the Canvas picture being updated while the <i>turtle</i> is drawing.
<u>UPDATE</u>	re-enable the updating of the Canvas.
<u>PAUSE</u> (<i>n</i>)	make the <i>turtle</i> wait for <i>n</i> milliseconds (i.e. thousandths of a second) before continuing.
<u>CANVAS</u> (<i>n1,n2,n3,n4</i>)	set the coordinate size and position of the Canvas so that its top-left corner has coordinates (<i>n1,n2</i>) and its bottom-right corner has coordinates (<i>n3,n4</i>), with the HOME position in the centre. The physical size of the Canvas is unaffected, and the <i>turtle</i> 's coordinates are automatically adjusted so that it does not move on the screen. This behaviour enables the CANVAS command to be used to draw ellipses, by moving to the desired position, temporarily changing the Canvas dimensions, and then drawing either a circle or a blot before resetting the dimensions to what they were.

Control Structures

FOR LOOPS

e.g.

```
for v:=2 to 10 do
  begin
    forward(v*10);
    right(20)
  end;
```

give the variable *v* in turn every value between 2 and 10 inclusive, and for each value given, execute the command after DO with *v* taking this value (*begin* and *end* are needed if there is more than one command in the loop). The loop will not be executed at all if the start value is higher than the end value - use DOWNTO instead of TO for counting downwards.

REPEAT LOOPS

e.g.

```
repeat
  back(10)
until turtx<100
```

execute the command(s) between REPEAT and UNTIL, and carry on doing so repeatedly until the UNTIL condition is fulfilled.

PROCEDURES

e.g.

```
PROCEDURE whitecirc(size: integer);
BEGIN
  colour(white);
  blot(size);
  colour(black);
  circle(size)
END;
```

define a new command by encapsulating a number of commands within a “procedure” (see [Programming Essentials \[23\]](#) and [Procedures and Parameters \[34\]](#) for details).

IF ... THEN ... ELSE

e.g.

```
if (turtx>-400) and (turtx<400) then
  blot(100)
else
  blot(50);
```

if the IF condition is *true*, then perform the command after THEN; otherwise, perform the command after ELSE (*begin* and *end* are needed in each case if there is more than one command).

Programming Essentials

This section provides a general introduction to Turtle Graphics programming, including a description of every command available. It is therefore long, but experience suggests that this is more useful than a segmented help system when initially learning programming, because it simplifies the process of looking for the commands you need when you don't yet know what kinds of facilities are available, or how they might be indexed. Another reason for putting all this material together is to enable you to print it out for reference as straightforwardly as possible. For a summary of the information contained here, go instead to the [Programming Quick Reference](#) [19].

To create a new Program within the system, type it into the Programming Area on the left hand side of the screen (and note that this area can be altered in size by dragging the window edges or maximising; also see the [Layout Menu](#) [11]). To run the Program once you have created it, click on the "RUN" button. Any Turtle Graphics Program consists of declarations and statements. Declarations define the name of the Program and also such things as the number and type of variables and procedures to be used. But the main bulk of most Programs consists of the statements which instruct the *turtle* how to move around the Canvas, and what to draw as it moves. These statements can be divided into two general types, *command statements*, which are direct instructions to the *turtle*, and *structural statements*, which create structures that determine the order in which the instructions are carried out. The two main sections below give a complete list of all the command statements and structural statements available within the system. But first, a brief introduction to declarations and a word on presentation.

Program Structure and Declarations

Any valid Pascal program must start with a program name declaration, and the program statements must be enclosed between "BEGIN" and "END." (with a full stop after this final "END"), and separated from each other by semicolons. So the simplest program structure is of the following form:

```
PROGRAM fancygrafix;  
BEGIN  
  statement1;  
  statement2;  
  statement3  
END.
```

If the Program uses global variables, these must also (with just five exceptions) be declared at the very beginning, immediately after the "PROGRAM" declaration. All variable names, like the program name, must satisfy the rules for being a valid "identifier" (e.g. a single word which is not a keyword, see the [Pascal Syntax Reference](#) [33] section). Only two types of variable are permitted in Turtle Graphics, either *integer* (i.e. whole number) or *Boolean* (i.e. true or false) variables, and you must declare them accordingly:

```
PROGRAM fancygrafix;  
VAR var1: integer;  
    var2: integer;  
    var3: boolean;  
BEGIN  
  statement1;  
  statement2;  
  statement3  
END.
```

The only exceptions are five special global integer variables which you do not need to declare because they are internally defined within the system, namely *turtx*, *turty*, *turtd*, *turtt* and *turtc*, which respectively provide direct access to the *turtle*'s x-coordinate, y-coordinate, direction, thickness and colour, and whose values can be both read and set from within your programs just like ordinary global variables.

If your Program uses any procedures they too, just like variables, must be declared, and their structure is very similar to that of the main program (except that the "END" with which they finish is followed by a semicolon):

```

PROGRAM fancygrafix;
VAR global1: integer;
    global2: boolean;

PROCEDURE drawone;
VAR local1: integer;
BEGIN
    statementp1;
    statementp2;
    statementp3
END;

BEGIN
    statement1;
    drawone; {calls the procedure DRAWONE}
    statement3
END.

```

Any “local” variables to be used within the procedure must be declared right at its beginning, in much the same way as the global variables are declared at the program’s beginning. All this bookkeeping can seem irritating when you first start programming, but it plays an important role in specifying exactly what the program is doing, and can be a great help when you start to deal with programs of relative complexity. If at any point you need more detail on program layout and structure, start by consulting the [Introduction to Pascal Syntax](#) [32].

Indentation, Capitalisation, and Comments

Good programming is very largely a matter of keeping clear in your mind exactly what you want your program to do, and meticulously keeping track of your progress in achieving that as the program develops. All this will be much simpler to do if you are careful to present your program in a way that makes its structure clear by using appropriate indentation and so forth, a technique which you can begin to pick up for yourself by copying the examples provided here.

Good presentation is greatly facilitated by two features of the Pascal programming language (which it shares with many others). First, surplus “white space” within any program is ignored – as long as you don’t break up words or compound symbols (e.g. by changing “forward” to “for ward”, or “>=” to “> =”, you can add any number of spaces and blank lines within a valid program, and its operation will be entirely unaffected. So do take advantage of this freedom, to insert blank lines where they help to clarify logical divisions (e.g. before a procedure declaration), and even more importantly to provide indentation of program structures in the sort of way illustrated in this Help file. The second useful feature of Pascal is that it ignores capitalisation – upper- and lower-case letters are treated exactly equivalently, so for example “BEGIN”, “begin” and even “BeGiN” are all equally valid. Again this provides an opportunity for highlighting the structure of your program, for example by capitalising the most important BEGINS and ENDS whilst leaving others lower-case. Many of the examples here use capitalisation for the same reason, but also sometimes for emphasis or to distinguish the commands under discussion. *For more detailed advice on indentation and capitalisation within your own programs, see the [Introduction to Pascal Syntax](#) [32], and note the auto-format facility available from the [Edit Menu](#) [9].*

A third useful feature of Pascal is that it makes provision for comments within the body of the Program text. Anything within the Program that is enclosed within curly brackets {like this} will be completely ignored, so you can take advantage of this to insert verbal clarifications of what is intended or being done at appropriate places. This is particularly useful if your program is long and/or contains procedures, since then comments can help a great deal in making its structure clear (e.g. you might put a comment, explaining the purpose of a procedure, directly before it). It is also often a good idea to put a comment after any variable declaration, to explain the purpose of the variable, for example:

```

VAR sides: integer; {counts the sides}

```

Note that comments can extend over a number of lines, so if you want to “cut out” a section of code from your program without actually erasing it (e.g. for testing purposes), you can put an opening curly bracket at the beginning of that section of code, and a closing curly bracket after it.

Command Statements

In what follows, where any command name is followed by a letter in brackets, that letter indicates the kind of *parameter* or *argument* which would usually be given to the command. Thus for example BACK(*n*) indicates that the BACK instruction takes a numeric parameter, e.g. *back(100)*, while COLOUR(*c*) indicates that the COLOUR instruction expects a colour parameter, e.g. *colour(blue)*. Note, however, that “numbers” and “colours” are strictly both Pascal *integers*, so for example ordinary numbers can in fact be given as parameters to COLOUR. The place of any integer parameter can be taken by an integer variable or arithmetic expression (so for example the commands *rad:=100*; *circle(rad+20)* will draw a circle of radius 120).

Variable Assignment, Arithmetical and Boolean Operators and Constants

v:=n makes variable *v* equal to value *n*. So for example *height:=50* makes the numeric variable *height* take the value 50, while *count:=count+4* makes the numeric variable *count* equal to 4 more than its previous value.

The standard arithmetical operators are represented in a very straightforward way that is common to many other computer systems. Listed below are in order: plus, minus, times, divided by, modulus, equals, is not equal to, is less than, is greater than, is less than or equal to, is greater than or equal to:

Basic arithmetic operators: + - * / MOD
Arithmetic comparisons: = <> < > <= >=

It should be noted that “/” here represents *integer* division, because the system makes no allowance for decimal or fractional numbers – this means for example that the expression “11/4”, instead of giving the result 2.75, gives the result 2 (the decimal part gets “thrown away”). MOD gives the remainder from integer division, so for example “11 mod 4” yields 3, and “x mod 2 = 1” tests whether x is an odd number (for an example where MOD is particularly useful, see under COLOUR in the section on Drawing Control below).

The standard Boolean operators are straightforwardly expressed in words:

Boolean operators: NOT AND OR XOR

Likewise the standard Boolean constants (which are interpreted numerically as -1 and 0 respectively, so that the Boolean operators can serve also as bitwise numerical operators):

Boolean constants: TRUE FALSE

Brackets can be used to build up complex expressions, but the standard arithmetical and logical rules of precedence apply, so that, for example, “2+3*4” is treated as “2+(3*4)” rather than as “(2+3)*4”.

Turtle Movement – Relative (i.e. relative to the turtle’s current position)

FORWARD(*n*) instructs the *turtle* to move forward by *n* units (where the Canvas is by default a 1000-unit square, so *forward(250)* would move forward one quarter of the size of the Canvas). If the pen is down (see PENDOWN and PENUP), then as the *turtle* moves it will draw a line on the Canvas in the current pen colour and thickness (see COLOUR and THICKNESS). FORWARD may be abbreviated to FD.

BACK(*n*) instructs the *turtle* to move backwards by *n* units (this does not affect the *direction* in which the *turtle* is pointing – it moves in reverse rather than turning). If the pen is down (see PENDOWN and PENUP), then as the *turtle* moves it will draw a line on the Canvas in the current pen colour and thickness (see COLOUR and THICKNESS). BACK may be abbreviated to BK.

LEFT(*n*) instructs the *turtle* to turn to the left by *n* degrees (where a full circle is 360 degrees, so that *left(90)* would turn left by a right angle). This changes the *turtle*’s direction, and thus affects its future movement, but does not itself draw anything on the Canvas. LEFT may be abbreviated to LT. To make the *turtle* face in a specific absolute direction (e.g. towards the left of the screen), set the global variable *turtd* using an explicit assignment statement, e.g. *turtd:=270*. Note here that “North” (i.e. upwards) is direction 0 – which is the initial direction of the *turtle* – and that the angle is counted clockwise, so “East” is direction 90, “South” is 180 and “West” is 270.

RIGHT(n) instructs the *turtle* to turn to the right by n degrees (where a full circle is 360 degrees, so that *right(90)* would turn right by a right angle). This changes the *turtle*'s direction, and thus affects its future movement, but does not itself draw anything on the Canvas. RIGHT may be abbreviated to RT. To make the *turtle* face in a specific absolute direction (e.g. towards the right of the screen), set the global variable *turtd* using an explicit assignment statement, e.g. *turtd:=90*. Note here that "North" (i.e. upwards) is direction 0 – which is the initial direction of the *turtle* – and that the angle is counted clockwise, so "East" is direction 90, "South" is 180 and "West" is 270.

MOVEXY(n1,n2) instructs the *turtle* to move a distance $n1$ in the x-direction, and $n2$ in the y-direction (so for example *movexy(100,0)* would move the *turtle* 100 units to the right of its current position). The *turtle*'s direction does not change, and nothing is drawn on the Canvas as it moves even if the pen is down.

DRAWXY(n1,n2) instructs the *turtle* to move a distance $n1$ in the x-direction, and $n2$ in the y-direction, and if the pen is down, to draw a straight line in the current pen colour and thickness as it moves (so for example *drawxy(50,-50)* would draw a diagonal line in a northeasterly direction, starting from the current position of the *turtle* and extending overall 50 units to the right and 50 upwards). The *turtle*'s direction does not change.

Turtle Movement – Absolute (i.e. to a specific location on the Canvas)

HOME returns the *turtle* to its original starting point in the centre of the Canvas, and also restores its original direction (i.e. facing "North", or 0 degrees). The *turtle* is lifted directly to its home position without drawing anything on the Canvas.

SETX(n) instructs the *turtle* to move parallel to the x-axis (i.e. horizontally) to a position whose x-coordinate is n . The *turtle*'s direction does not change, nor does its y-coordinate, and nothing is drawn on the Canvas as it moves. This command is equivalent to *turtx:=n*, except that *setx(n)* automatically records the new position as visited (see under POLYLINE, POLYGON and REMEMBER below).

SETY(n) instructs the *turtle* to move parallel to the y-axis (i.e. vertically) to a position whose y-coordinate is n . The *turtle*'s direction does not change, nor does its x-coordinate, and nothing is drawn on the Canvas as it moves. This command is equivalent to *turty:=n*, except that *sety(n)* automatically records the new position as visited (see under POLYLINE, POLYGON and REMEMBER below).

SETXY(n1,n2) instructs the *turtle* to move to a position whose x-coordinate is $n1$ and whose y-coordinate is $n2$. The *turtle*'s direction does not change, and nothing is drawn on the Canvas as it moves. This command is equivalent to *turtx:=n1; turty:=n2*, except that *setxy(n1,n2)* automatically records the new position as visited (see under POLYLINE, POLYGON and REMEMBER below).

Shape Drawing

CIRCLE(n) draws on the Canvas the outline of a circle, radius n units, around the current position of the *turtle* and in the *turtle*'s current pen colour and thickness.

BLOT(n) draws on the Canvas a filled circle, radius n units, centred on the current position of the *turtle* and in the *turtle*'s current pen colour.

POLYLINE(n) draws on the Canvas a sequence of $n-1$ straight lines, in the *turtle*'s current pen colour and thickness, connecting together the n points which the *turtle* has most recently visited using any of the *turtle* movement commands listed above (so for example if the *turtle* has recently moved through the sequence of points A-B-C-D, ending up at point D, then *polyline(3)* will connect B to C and C to D, while *polyline(4)* will connect A to B, B to C, and C to D). A complete unfilled polygon bounded by straight lines can be drawn by moving the *turtle* through the relevant sequence of points (the command *movexy* is often most convenient for this purpose), and then returning to the start position before using *polyline* (e.g. moving the *turtle* through the sequence of points A-B-C-A, and then using *polyline(4)*, will draw the triangle ABC). Although this sort of drawing can also be done using the command *drawxy*, the advantage of *polyline* is that it draws all the relevant lines in a single command, so that for example all the lines can be redrawn in a different colour without the *turtle* having to move.

POLYGON(n) draws on the Canvas a filled shape, in the *turtle's* current pen colour, bounded by straight lines connecting together the *n* points which the *turtle* has most recently visited using any of the *turtle* movement commands listed above (so for example if the *turtle* has recently moved through the sequence of points A-B-C-D, ending up at point D, then *polygon(3)* will draw the triangle BCD, while *polygon(4)* will draw the quadrilateral ABCD).

FORGET(n) is used in conjunction with POLYLINE and POLYGON – it makes the *turtle* “forget” about the last *n* points that it has visited (so for example if the *turtle* has recently moved through the sequence of points A-B-C-D-E, ending up at point E, but the command *forget(2)* was executed just after the *turtle* moved to point D, then *polygon(3)* will draw the triangle ABE, because the *turtle* will have “forgotten” the two points C and D).

REMEMBER is used in conjunction with POLYLINE and POLYGON – it makes the *turtle* “remember” the point where it is currently located. Since the standard *turtle* movement commands automatically “remember” the points they visit, the main use of REMEMBER is to record points that have been moved to by direct assignment to the global variables *turtx* and *turty*.

Drawing Control

THICKNESS(n) sets the thickness of the *turtle's* pen to *n* pixels, so that any lines or outline shapes are drawn accordingly. The command *thickness(n)* is directly equivalent to *turt:=n*.

COLOUR(c) changes the colour of the *turtle's* pen, and hence the colour of any drawing which will be done as the *turtle* moves around or draws shapes on the Canvas. Thus *colour(c)* is directly equivalent to *turtc:=c*. The specified colour *c* can be any of the following: BLUE, GREEN, CYAN, RED, MAGENTA, YELLOW, WHITE, BLACK. These 8 colours can also be expressed as the integers from 1 to 8 respectively, so that *colour(cyan)* for example may be alternatively expressed as *colour(3)*. COLOUR may be written as COLOR.

If you want to “cycle” through some of the various standard colours (say, the first six of them), you can do this by assigning an appropriate initial value to *turtc* (e.g. *turtc:=1*) and then repeatedly using the command *turtc:=(turtc mod 6)+1*, which cycles from 1 to 2 to 3 to 4 to 5 to 6 to 1 etc. (because $6 \bmod 6 = 0$).

Technical Note:

In fact *colour(cyan)* and *colour(3)* are not exactly equivalent, because *cyan* here stands not for the number 3, but rather for the number 16776960, which happens to be the standard colour code for cyan (this point is important if you intend to do any direct manipulation of *turtc*, the global variable representing the *turtle's* colour). A related point is that if any numerical value outside the range 1 to 8 is entered, it is interpreted as a number code whose digits, when it is expressed in hexadecimal notation, specify the Blue, Green and Red components of the colour to be assigned (the number should be preceded by “\$” to show that it is hexadecimal). Hexadecimal numbers are numbers expressed in base 16, with the letters “A” to “F” serving as digits beyond 9, so for example “\$0F” represents the value 15 (decimal); “\$10” the value 16, “\$23” the value 35 ($2 \times 16 + 3$), “\$FE” the value 254 ($15 \times 16 + 14$), and \$100 the value 256 (16×16). The upshot for colour codes is relatively straightforward, because each individual primary colour has an intensity value between \$00 (0) and \$FF (255), and these two-digit hexadecimal numbers are simply joined together to produce the overall colour code. For example *colour(\$40FFC0)* will produce a colour whose Blue component is \$40 (i.e. $64/255$), whose Green component is \$FF (i.e. $255/255$), and whose Red component is \$C0 (i.e. $192/256$). The code for cyan, 16776960 in decimal, is far more easily understood in hexadecimal, where it is \$FFFF00 – maximum intensity of blue and green, but zero intensity of red.

In addition to the “\$” or “BGR” convention of specifying colours explained in the previous paragraph, Turtle Graphics also allows the “#” or “RGB” convention which is standard in Web page design and some other contexts – note that this is provided purely for the convenience of those who are used to handling colours in the “Red, Green, Blue” order, and so should not be used as a means of specifying hexadecimal numbers except when they are used as colour codes. Using the RGB convention on the example above, the command *colour(#C0FF40)* is exactly equivalent to *colour(\$40FFC0)*.

RANDCOL(n) randomly chooses a colour value for the *turtle's* pen (i.e. the global variable *turtc*) making this choice within the range 1..*n* (hence if *n* is at most 8, a choice will be made from the first *n* of the standard colours, as listed under the COLOUR instruction above). Thus for example *randcol(4)* will randomly select one of the colour values 1, 2, 3 or 4 (which display as *blue*, *green*, *cyan* and *red* respectively, since these are the first 4 standard colours). If *n* is greater than 8, then *turtc* will be made equal to a random integer in the range 1..*n*, and the colour of the *turtle* will be set accordingly – either to one of the standard colours if *turtc* is 8 or less, or to a “BGR” colour otherwise (see COLOUR above). This enables RANDCOL and *turtc* to be used to generate random numbers over any chosen integer range.

BLANK(c) blanks out the entire Canvas with the specified colour *c*.

PENUP raises the *turtle's* pen, so that no drawing is done as the *turtle* moves around the Canvas (e.g. while executing FORWARD or BACK instructions; this does not affect pure drawing instructions such as CIRCLE). PENUP may be abbreviated to PU.

PENDOWN lowers the *turtle's* pen, so that drawing is done as the *turtle* moves around the Canvas (e.g. while executing FORWARD or BACK instructions). PENDOWN may be abbreviated to PD.

NOUPDATE stops the Canvas picture being updated while the *turtle* is drawing, thus speeding up its action considerably. NOUPDATE has no effect if the trace display facility (described under the Compile Menu [12]) is enabled, since display updating is needed if the trace facility is to operate. Moreover any PAUSE command automatically updates the Canvas while the specified delay takes place. An example of using NOUPDATE for the sake of speed is provided by The Recursion Factory [39]; NOUPDATE and UPDATE can also be used very effectively to give an impression of smooth motion, as in some of the built-in illustrative programs (where NOUPDATE prevents the Canvas from being updated during a sequence of blot drawing commands, leaving all the consequent changes to be enacted almost simultaneously when the UPDATE command is reached).

UPDATE re-enables the updating of the Canvas, canceling the effect of NOUPDATE.

PAUSE(n) makes the *turtle* wait for *n* milliseconds (i.e. thousandths of a second) before continuing.

CANVAS(n1,n2,n3,n4) sets the coordinate size and position of the Canvas so that its top-left corner has coordinates (n1,n2) and its bottom-right corner has coordinates (n3,n4), with the HOME position in the centre of the Canvas. If the coordinates are impossible or do not give the Canvas a minimum width and height of 2 units, then the command has no effect. The physical size of the Canvas on the screen is unaffected, and the *turtle's* coordinates are automatically adjusted so that it does not move on the screen (moreover POLYGON and POLYLINE can still work using the previously remembered physical screen positions). This behaviour enables the CANVAS command to be used to draw ellipses, by moving to the desired position, temporarily changing the Canvas dimensions, and then drawing either a circle or a blot before resetting the dimensions to what they were. For example *canvas(0,0,500,1000)* will set the unit sizes so that that 500 horizontal units are equivalent to the physical width of the Canvas, while 1000 vertical units are equivalent to its physical height; hence *blot(250)* will then display an ellipse extending the entire width of the Canvas but only half its height, after which *canvas(0,0,1000,1000)* will restore the default Canvas situation, leaving the ellipse displayed.

Structural Statements

The following structures typically involve combinations of structural statements which need to appear in appropriate combinations (e.g. REPEAT should always be followed by an appropriate UNTIL statement). In each case a simple example is given to illustrate the required structure.

Looping Structures

FOR indicates the beginning of a *for loop*, of which the following is an example:

```
for count:=25 to 75 do
  circle(count);
```

This starts by assigning the value 25 to the variable *count*, then draws a circle with a radius of that value, then increments *count* by 1, draws a circle with that new value (i.e. 26), again increments *count* by 1 and draws a circle with the new value (i.e. 27), and goes on doing this until *count* exceeds 75 (so the last circle drawn will have radius 75, but the variable *count* will end up with the value 76). To enclose a number of commands within a *for* loop, bracket them between BEGIN and END, and note also that the expressions on either side of the keyword TO can be complex expressions rather than just numbers:

```
for count:=middle-50 to middle+50 do
  begin
    randcol(6);
    circle(count)
  end;
```

Also note that if the “final value” of a *for ... to* loop is less than the “starting value” (e.g. *for count:=75 to 25 do ...*), then the command(s) within the loop will not be executed at all. To work downwards within a *for* loop, use the keyword *downto* in place of *to*, for example:

```
for count:=75 downto 25 do
  circle(count);
```

REPEAT indicates the beginning of a *repeat loop*, which is a sequence of commands that are to be performed repeatedly until some specified condition (the *until* condition) becomes true. For example the structure:

```
repeat
  circle(radius);
  radius:=radius+10
until radius>100
```

will draw a circle using the current value of the variable *radius* (whatever that may be), and will then successively increase *radius* by 10 until its value becomes greater than 100, drawing a circle each time just before the value is incremented. (Note, however, that the *until* condition is tested only at the end of the loop; so the loop terminates only at that point, even if the *until* condition becomes false somewhere in the middle of the sequence of loop commands. For the same reason, a *repeat* loop will always be performed at least once, even if the *until* condition was already false before it began.)

Many loops can equally well be implemented using either FOR or REPEAT, but the following advice might be helpful. In general, you should use a *for loop* when the loop is to be executed a set number of times in a fairly straightforward manner. By contrast, you should prefer a *repeat loop* when the repetitions are less straightforward, or when it is easier to fix the condition under which the loop should terminate than it is to fix in advance the number of times that it should execute.

Procedures

A *procedure* is a sequence of statements that is given a name, and may thereafter be used as a single command in the Program. Hence constructing a procedure effectively enables you to define your own instructions. Here is an example of a program using a procedure:

```
PROGRAM squares;
VAR count: integer;

PROCEDURE drawsquare50;
BEGIN
  forward(50);
  right(90);
  forward(50);
  right(90);
  forward(50);
  right(90);
  forward(50);
  right(90)
END;

BEGIN
  for count:=1 to 8 do
  begin
    randcol(6);
    drawsquare50;
    forward(50)
  end
END.
```

Notice that the program body occurs AFTER the procedure itself, and that the procedure has its own BEGIN and END to show how far it extends. However the END which ends the procedure has a semicolon after it, whereas the END which ends the program has a full stop. Within the main program body, the command *drawsquare50* calls the procedure.

If you have a program involving two or more procedures, where one of these is called by another one but otherwise not used at all, it is often a good idea to place the first procedure within the body of the second, to make clear their relationship. To place one procedure (the “child”) within another (the “parent”), open a gap between the declaration part of the “parent” and its body (i.e. the gap should be immediately before its first BEGIN line). Then insert the entire “child” procedure within that gap. Here for the sake of illustration is how the procedure *drawsquare50* in the program above might be reorganised along these lines (note that the variable *sidecount*, belonging to the parent procedure, is declared before the body of the child procedure):

```
PROCEDURE drawsquare50;
VAR sidecount: integer;

PROCEDURE drawside;
BEGIN
  forward(50);
  right(90)
END;

BEGIN
  for sidecount:=1 to 4 do
  drawside
END;
```

The true power of procedures only becomes apparent when you use them with *parameters* – for an explanation of this vitally important technique, see [Procedures and Parameters](#) [34].

Conditional Structures

IF ... THEN enables a command to be performed *conditionally*, so that it will be performed only if a particular condition is fulfilled. For example the structure:

```
if radius<100 then
  circle(radius);
```

will draw a circle using the current value of the variable *radius* IF that value is less than 100 – otherwise it will do nothing. The IF...THEN structure can be combined with the bracketing words BEGIN...END, or with procedures, to enable long sequences of statements to be performed conditionally, for example:

```
if length=50 then
  begin
    forward(length);
    right(90);
    forward(length);
    right(90)
  end;
```

Or:

```
if count<=5 then
  drawit;
```

(where *drawit* is a procedure which may consist of a large number of program lines – note here that “<=” means “is less than or equal to”, likewise “>=” means “is greater than or equal to”, and “<>” means “is not equal to”).

IF...THEN can involve more complicated conditions, combined with the words NOT, AND, OR, and XOR (“XOR” means “exclusive or”). But in this case brackets must be used, in the following sort of way:

```
if (x<0) or (x>1000) then ...

if (y>250) and (y<750) then ...

if not(length=10) then ...
```

Be warned that if you don't put the brackets in correctly, you will get an error message from the compiler as it tries to make sense of what you've written (probably in ways that you haven't imagined!).

IF ... THEN ... ELSE is an extension of IF...THEN, providing an alternative course of action if the original condition is not satisfied. So for example the structure:

```
if radius<100 then
  circle(radius)
else
  begin
    radius:=100;
    blot(radius)
  end;
```

will draw a circle using the current value of the variable *radius* IF that value is less than 100, but OTHERWISE will set the value of *radius* to 100 before drawing a blot. Note that the bracketing words BEGIN...END can be used after ELSE just as they can after THEN. Also note that *there is no semicolon between the THEN part and the ELSE part of a conditional structure* – if you put a semicolon before ELSE, the compiler will generate an error, because a semicolon always marks a gap between statements, and no statement can start with ELSE.

Introduction to Pascal Syntax

The syntax of a Pascal program is fairly straightforward, but follows much stricter rules than the syntax of English prose – if you get something wrong, the compiler will complain! Two things that are not strict, however, are *capitalisation* and *use of spaces*. As far as Pascal is concerned, the words “repeat” and “REPEAT” are totally equivalent, as indeed is “RePeAt”, so you are free to use upper- and lower-case letters however you prefer. One useful policy is to use lower-case letters most of the time, but to use upper-case for major structural words such as “PROGRAM”, “PROCEDURE”, and the “BEGIN” and “END” which bracket the major sections of the program (this is roughly the policy followed by the auto-format facility under the [Edit Menu](#) [9], and by most of the examples in this Help file, notably in the sections on [Programming Essentials](#) [23] and [Procedures and Parameters](#) [34]). But if you use descriptive variable names (which you should do unless your program is extremely short and simple to understand), you can also vary upper- and lower-case to highlight their meaning – for example you might have a variable named “SumOfNumbers”, where the capitalisation makes it easier to understand.

Just as Pascal allows you free use of capitalisation, so it also allows you free use of “white space” – you can insert additional spaces, and even blank lines, wherever you like without affecting the program’s meaning at all. The great advantage of this is that it enables you to set out your program in a way that clarifies its meaning, for example by “indenting” the lines that are enclosed in any form of “bracketing” such as BEGIN...END or REPEAT...UNTIL, and also indenting the lines that lie within the scope of an IF...THEN or an ELSE (again, all of the examples in this Help file conform to this sort of convention, as does the auto-formatter).

Every Pascal program must begin with a line which looks something like this:

```
PROGRAM myprog;
```

The word “PROGRAM” is a *keyword* – a word which can only be used for the purpose that Pascal determines (so you’re not allowed to have a variable named “program”, for example, but you can have a variable named “yellow”, which isn’t a keyword). This is followed by the name or “identifier” which you choose to give to your program, which can be more or less anything you like as long as it begins with a letter, consists only of letters and digits, and is not a keyword. (For more on both keywords and “identifiers”, see the [Pascal Syntax Reference](#) [33] section.) The program name has to be followed by a semicolon.

Following the program name “declaration”, you will normally have one or more *variable declarations*, which “declare” the names that you are giving to the global variables in your program, and what *type* they have (in Turtle graphics the type will always be either “integer” or “boolean”). If you are declaring only one variable, then you do so using a line such as:

```
VAR length: integer;
```

But if you are declaring more than one, you can omit the keyword VAR in subsequent lines, though it’s a good idea to insert four extra spaces to line things up neatly:

```
VAR length: integer;  
    side: integer;
```

You can even list the variables on a single line, though this can be less clear to read, and prevents you from putting a comment next to each declaration to explain its meaning (use of comments is explained in the [Programming Essentials](#) [23] section):

```
VAR length, side: integer;
```

Notice that all of these lines have to finish with a semicolon, though as we shall see not every line in the program has to do so. Having declared your global variables, the main body of your program can start, being introduced by the keyword BEGIN (this is one of the lines that does *not* need a semicolon following, though no harm will come if you insert one anyway):

```
BEGIN
```

Then follow the statements which constitute the main body of your program, usually separated from each other by semicolons. Finally comes the keyword END to finish the program, and in this case it must be followed by a full stop:

```
END.
```

Here we have only introduced the syntax of simple programs which don't involve procedures – to learn about how these are laid out, see the section on “Procedures” in the [Programming Essentials](#) [23] section. Note that procedures are included *between* the main program's variable declarations and the body of the main program, and that whereas the main program body is enclosed between “BEGIN” and “END.”, each procedure is enclosed between “BEGIN” and “END;” (i.e. with a semicolon rather than a full stop after END). For the syntax of procedure declarations and calls involving parameters, see [Procedures and Parameters](#) [34].

For examples of programs which show all this in action, take a look at the relevant “Illustrative programs” available through the [Help Menu](#) [17].

Pascal Syntax Reference

Identifiers

An *identifier* is a name given to a program, procedure, or variable. The general rule is that any identifier must start with a letter, and must consist only of letters and digits (so, for example, no spaces or punctuation characters are allowed within a variable name). However for these purposes the underscore character “_” counts as a letter, so you can if you wish name your program “my_prog”. Because Pascal is case-insensitive, you are also free to combine upper- and lower-case letters to make your identifiers more easily understood, for example by calling a variable “AddToTotal” or even “Add_To_Total”.

When choosing names for variables or procedures within a Turtle Graphics Pascal program, it is obviously sensible to avoid using words such as “true” or “yellow” which already have a meaning within the language. However only relatively few such words are actually *illegal* as identifiers, and these are called “keywords” or “reserved words” (because they are *reserved* for their predefined use and cannot be used for anything else).

Reserved Words (Keywords)

The reserved words within Turtle Graphics Pascal are as follows:

```
AND
BEGIN
DO
DOWNTO
ELSE
END
FOR
IF
MOD
NOT
OR
PROCEDURE
PROGRAM
REPEAT
THEN
TO
UNTIL
VAR
XOR
```

Procedures and Parameters

Simple procedures were introduced in the section on [Programming Essentials](#) [23], where an example was presented involving the *drawsquare50* procedure, of which this is a streamlined version using a counting loop:

```
PROCEDURE drawsquare50;
VAR sidecount: integer;
BEGIN
  for sidecount:=1 to 4 do
    begin
      forward(50);
      right(90)
    end
  END;
```

This procedure draws a square of side 50 with a corner at the initial *turtle* position, and ends leaving the *turtle* back at the same place and pointing in the same direction.

Introducing Simple Value Parameters

No doubt the procedure above might be of use if you are designing a program which includes lots of squares of side 50, but clearly it is extremely limited. If you needed a wider variety of squares, then it wouldn't be much fun having to produce dozens of similar procedures, all for different sizes – how much better to have a single procedure which could produce squares of all the sizes you might need! Fortunately it's very easy indeed to adapt the procedure to this purpose:

```
PROCEDURE drawsquare(size: integer);
VAR sidecount: integer;
BEGIN
  for sidecount:=1 to 4 do
    begin
      forward(size);
      right(90)
    end
  END;
```

This new procedure has been defined as taking a single integer *parameter* – called “size” – and within the procedure this parameter acts just like a local variable, but one whose initial value is set by the procedure call. The procedure is called in the same way as any built-in *turtle* command that takes a parameter. So for example the two commands:

```
drawsquare(100);
drawsquare(250);
```

inserted later in the program will call this procedure with parameter values 100 and 250, generating a square of side 100 and a square of side 250 respectively. We could go even further in the direction of flexibility, by changing the beginning of the procedure to:

```
PROCEDURE drawsquare(size, col: integer);
VAR sidecount: integer;
BEGIN
  colour(col);
```

This small change gives us a procedure with two integer parameters, the first of which specifies the size of the square to be drawn, and the second the colour. So now the command:

```
drawsquare(300,red);
```

for example, will draw a red square of size 300, while

```
drawsquare(200,nextcol);
```

will draw a square of size 200 in whatever colour the variable *nextcol* represents at the point when the command is issued. Our initial limited procedure has very straightforwardly become far more flexible and useful.

Recursion

One of the most powerful features of advanced programming languages such as Pascal is *recursion*, whereby a procedure is able to “call itself”. This concept often proves hard to master in the abstract, but fortunately the Turtle Graphics context makes it relatively easy to illustrate, often producing beautiful patterns in the process.

Consider first the following simple program, which calls a procedure that just draws an equilateral triangle of the specified size and (very importantly!) returns the *turtle* to the same position and direction that it had to start with:

```
PROGRAM triangles;

PROCEDURE triangle(size: integer);
BEGIN
  if size>=2 then
    begin
      forward(size);
      right(120);
      forward(size);
      right(120);
      forward(size);
      right(120)
    end
  END;

BEGIN
  triangle(256)
END.
```

It is also very important for what follows that the procedure has a limit – unless the *size* parameter is greater than or equal to 2, then it does nothing. (Note: if you try to produce a recursive program, and get some sort of memory overflow error when it runs, that’s probably because the relevant procedure does not have this sort of limiting mechanism to make it stop at some point, so your program is trying to run on forever.)

To make this program recursive, all we need to do is to put in one or more calls to the procedure *triangle* from within itself. It’s instructive to insert these recursive calls one at a time, to see what pattern results. But here is the full version with three recursive calls, which is one of the built-in illustrative programs:

```
PROGRAM triangles;

PROCEDURE triangle(size: integer);
BEGIN
  if size>=2 then
    begin
      forward(size);
      triangle(size/2); {first recursive call}
      right(120);
      forward(size);
      triangle(size/2); {second recursive call}
      right(120);
      forward(size);
      triangle(size/2); {third recursive call}
      right(120)
    end
  END;

BEGIN
  triangle(256)
END.
```

Copy this program into Turtle Graphics (or load it from “Illustrative programs | Recursion” under the [Help Menu](#) [17]) and run it – you may be surprised at what you see! When the procedure *triangle* is called, it is not only drawing a triangle but also calling three copies of itself, and importantly, these three copies draw triangles of half the size. They in turn are calling three times as many copies of a quarter-size triangle, and so on *until* (and this is where the limiting mechanism mentioned above comes in) the size gets down below 2, when no more triangles are drawn and the program can continue on its way dealing with the other triangles that are still waiting to be done. (Note – *this simplified description of what’s going on ignores the complications of the order in which the triangles are drawn; see if you can work out how this happens by watching it run, and if not, add a pause(50) command within the procedure*) When the program finally finishes, it has drawn 1 triangle of size 256, 3 of size 128, 9 of size 64, 27 of size 32, 81 of size 16, 243 of size 8, 729 of size 4, and 2187 of size 2, arrayed in a beautiful pattern. All that, from less than 20 lines of program code! As long as you keep the points emphasised above in mind (notably that you start and finish in the same place and in the same direction, and provide a limit), it’s easy to make similar types of recursive pattern using almost any other shape as a basis, and without much additional complexity you can add variety by specifying the shape’s colour as a second parameter to the procedure (with different colours depending on the size). Here Turtle Graphics really comes into its own, enabling you to produce patterns of an intricacy which would be quite impossible without programming. To enable you to experiment making elaborate colourful patterns without having to do any programming yourself, however, a special program called [The Recursion Factory](#) [39] is available through the [Help Menu](#) [17].

You might well be puzzled as to how recursion can work. The program above starts off calling the procedure *triangle*, in which the parameter *size* is given the value 256, and this procedure then almost immediately calls itself, giving *size* a new value of 128. So why doesn’t this mess up the workings of the original procedure, by changing its *size* parameter from 256 to 128 halfway through? The answer lies in the fact that every *instance* of the procedure (i.e. every calling of it) creates a new copy of all of its local variables, including any parameter variables. These different sets of local variables are all stored quite separately from each other, so each instance of the procedure can operate perfectly well using its own particular values until it has run its course (at which point the memory space given over to its local variables is “released”). It is this sophisticated handling of local variables that distinguishes powerful recursive languages such as Pascal, and enables such elegant results to be generated at so little effort. If you want to find out more about how this handling is achieved within Turtle Graphics, see the [Technical Note on Variables, Procedures and Parameters](#) [44].

Scope

Because (as just explained) each instance of any procedure stores “private” copies of its own local variables, this implies that any local variable within one procedure is quite distinct from any local variable within another, and indeed from any global variable, *even if they happen to have the same name*. Accordingly, there is nothing to prevent you from defining a global variable called “size”, together with a local variable called “size” in every single one of your procedures (though it might be confusing!). If you do this, then the meaning of the name “size” will vary from one context to another, referring to the relevant local variable within any procedure, but referring to the global variable if it is used in the main program. Here the basic rules are, first, that a variable name can only refer to a local variable if it is within that local variable’s *scope* – i.e. if the name is located somewhere within the procedure in which that local variable is defined. Secondly, that (subject only to the first rule) a variable name refers to whichever variable of that name has the smallest “scope”. Hence if a global variable and a local variable have the same name, the local always supersedes the global within the relevant procedure.

Note that where one procedure is enclosed within another, the “child” procedure can make use of the “parent” procedure’s variables (since a “child” procedure is within the “parent” procedure’s scope), but not vice-versa.

Formal and Actual Parameters

We are now moving on to a fairly advanced topic which many students of programming find extremely confusing, and which is beyond what is required to complete the Exercises set within this Help system. In the following discussion, it will be useful to distinguish between the [formal parameters](#) to a procedure, which are the variables used within the procedure to hold the values given, the [actual parameters](#), which are the expressions used as “arguments” when the procedure is called, and the [actual values](#), which are the numerical values which the actual parameters have at that time. If we assume that *nextcol* is a global variable with the value 6 when *drawsquare* is called in one of the examples given earlier:

```

PROCEDURE drawsquare(side, col: integer);
...
drawsquare(200,nextcol);

```

then *side* and *col* are the formal parameters of the procedure *drawsquare*, 200 and *nextcol* are the actual parameters within the procedure call, while the actual values are 200 and 6.

Value and Reference Parameters

When the *drawsquare(200,nextcol)* command above is performed, the number 200 and the current value of *nextcol* are stored in the *drawsquare* procedure's variables *side* and *col* respectively (i.e. the formal parameters are given the actual values which the actual parameters have when the procedure is called). So the procedure continues just as if it had begun with the two statements:

```

side:=200;
col:=nextcol;

```

Suppose, for example, that when the procedure is called the variable *nextcol* has the value 4. This value will be passed on to the procedure's variable *col*, and from that point the variable *nextcol* itself will play no further part in the procedure. So here we have an example of a *value parameter* to a procedure – in which the only relevance of the actual parameter to the procedure is to determine the actual value which is initially given to the formal parameter.

For most purposes, value parameters are perfectly adequate, and many programmers use nothing else. But the Pascal programming language also makes provision for a quite different kind of parameter, called a *reference parameter*, in which the formal parameter within the procedure, instead of just taking the actual value from its corresponding actual parameter, becomes in effect a continuing *alias* (i.e. an alternative name) for that actual parameter. This then means that changes made to the formal parameter within the procedure will directly affect the variable which was given as the actual parameter (which implies that *only* a variable can be passed as such a parameter – there is no way that a formal parameter can act as an alias for a number or a complex arithmetical expression). All this is rather abstract and far more easily understood by example, so consider the following short program (which is adapted from one of the “Illustrative programs” under the [Help Menu](#) [17] – select “Reference (VAR) parameters” to load the full version):

```

PROGRAM dotlines;

PROCEDURE dots(VAR coord: integer);
VAR count: integer;
BEGIN
  for count:=1 to 8 do
  begin
    coord:=coord+100;
    blot(40)
  end
END;

BEGIN
  setxy(50,50);
  dots(turtx);
  dots(turty)
END.

```

Look first at the header of the procedure *dots*, which includes the keyword VAR – this (when it occurs in the context of a procedure parameter specification) indicates that the following parameter *coord* is a reference rather than a value parameter. Then the body of the procedure simply executes a counting loop in which *coord* is successively incremented by 100 eight times, with a blot being drawn each time. This might seem pointless, because the procedure contains no explicit commands to move the *turtle* between drawing the eight blots (so you might expect that a blot is just going to be redrawn in the same place again and again), but the key to what is going on lies in the procedure calls within the main program:

```

dots(turtx);
dots(turty)

```

When the procedure is called the first time, the global variable *turtx* (a system variable which stores the *turtle*'s x-coordinate) is passed as the actual parameter, so that within the procedure, "*coord*" becomes in effect another name (an "*alias*") for *turtx*. Hence the successive additions performed by the *coord:=coord+100* commands actually change the value of *turtx* (i.e. "*coord:=coord+100*" is for the moment just another way of saying "*turtx:=turtx+100*"), and hence these commands move the *turtle* across the Canvas. So the first procedure call draws eight blots in a horizontal line, with their centres 100 units apart. The second call of the procedure has a similar effect, only this time it is *turty* rather than *turtx* for which "*coord*" becomes an alias. Hence now the procedure draws eight blots in a vertical line. This technique, therefore, enables a procedure to be written that will perform some specified operation on any chosen variable(s). It also enables procedures to return values to the rest of the program, because the variable passed to the procedure is directly affected by any changes made within it (just as the procedure call *dots(turtx)* changes the value of *turtx*).

A standard illustration of the value of reference parameters is provided by the following sort of "swap" routine:

```

PROCEDURE swap(VAR a, b: integer);
VAR temp: integer;
BEGIN
  temp:=a;
  a:=b;
  b:=temp
END;

```

This swaps the values of *a* and *b* (and hence of the two variables for which they are acting as aliases), making use of *temp* as a temporary buffer. But the procedure has an effect on the program which calls it only because both *a* and *b* are reference parameters – if they were value parameters, then the swapping within the procedure would have no effect whatever on the variables that serve as the actual parameters to the procedure.

As a final example of wider relevance to Turtle Graphics, suppose that you are implementing a ball-bouncing program, in which you are recording the coordinates and velocity of a number of balls, and each of these needs to be "bounced" by inverting the velocity in the x-direction when, say, the x-coordinate reaches a value less than -450 or greater than 450. You might do this very straightforwardly as follows:

```

PROCEDURE xbounce(xp: integer;VAR xv: integer);
BEGIN
  if (xp<-450) or (xp>450) then
    xv:=-xv
  END;

```

Then you can deal with the bouncing of as many balls as you like through a sequence of commands such as:

```

xbounce(xpos1,xvel1);
xbounce(xpos2,xvel2);
xbounce(xpos3,xvel3);

```

Because *xv* is a *reference* parameter to the *xbounce* procedure, any change to its value within that procedure is passed back to the relevant actual parameter, enabling all the balls to be dealt with using the same procedure and thus providing a far more elegant solution to the problem than would otherwise be possible. For a more sophisticated development of this idea, in which each ball's normal movement as well as bouncing is handled by a single procedure, see the "Multiple bouncing balls" example available through "[Illustrative programs](#)" from the [Help Menu](#) [17].

The Recursion Factory

The Recursion Factory is a program available through the [Help Menu](#) [17], which enables you to experiment with countless different recursion patterns. The program is extensively annotated, so once you are familiar with Turtle Graphics programming, you should be able to work out how it functions from a technical point of view. The explanation provided below is mainly for those who want to understand its effects, and to be able to play around with it, without necessarily caring about the technicalities.

Quick-Start Guide – Playing with Patterns

If you just want to get on with playing as quickly as possible, go straight to the bottom of the program, where you'll see something like the following:

```
{*****}

{SETTINGS - vary these to produce different patterns}

numsides:=6; {number of sides in each polygon }
initssize:=240; {initial length of polygon side }
shrink:=40; {shrink n/120ths each recursion }
rangle:=180; {angle from centre line to recurse}
polygap:=0; {polygon gap, n/120ths at RANGLE }
levels:=5; {how many levels of recursion }
slowdraw:=3; {levels to be shown slow-drawing }
mode:=1; {1=lines ; 2=blots ; 3=circles }
firstcol:=1; {colour of first level polygon }
colinc:=4; {increment for successive colours }
minthick:=1; {minimum line thickness }
addthick:=0; {extra thickness per level x 12 }
circlsize:=60; {radius n/120ths of polygon side }

{*****}
```

These values are the only parts of the program that you'll need to alter, since they determine what pattern is produced. To begin with, make sure that the Canvas is set to the default "(0,0) to (1000,1000)" mode (using the [Options Menu](#) [15]), click on "RUN" and see what happens – a pattern of recursive hexagons will appear. If you now perform in turn each of the steps that follow, you'll have a rapid tour through the various settings available, after which you can play with the system as much as you like:

- First change the value of MODE in turn to 2 and then 3, and run it again each time – with MODE:=2 the pattern is drawn with blots at the corners of each hexagon instead of lines down the edges, and with MODE:=3 circles are used instead of blots.
- Leaving MODE as 3, now change CIRCSIZE from 60 to 120 and run again – this changes the circles' radius to the same length as the hexagon sides, giving a very different effect.
- Having seen that, change MODE back to 1, run the program again (to remind yourself what it's like), and change RANGLE to 150 – you'll see that this changes the angle between each hexagon and the smaller "child" hexagons that sprout off it.
- In this case the hexagons overlap, rather spoiling the pattern, so to deal with this, change POLYGAP to 60 – which puts a gap between the hexagons – and run again.
- By now the pattern will more than fill the Canvas, but you can reduce it (without affecting its structure) by changing INITSIZE to 180.
- A very different way of changing the pattern's size is by adjusting SHRINK, which affects how much shrinkage takes place between each hexagon and its "child" hexagons. Try now putting SHRINK to 60, and see what happens – in this case it's a bit of a mess, but if you put RANGLE back to 180 and POLYGAP back to 0, this cleans it up. Even better, having made these changes set MODE to 2 and then to 3, which give rather nice effects.

- (g) Leaving MODE as 3, now change LEVELS to 3 – this means that only 3 levels of recursion take place, giving a far more “open” pattern.
- (h) This pattern can be made a bit more interesting by changing ADDTHICK to 12 – with this setting, the circles at the lowest level of recursion are drawn with thickness 1 (set by MINTHICK), and each level above that increases in thickness by 1.
- (i) ADDTHICK uses units of twelfths, so if you now set it to 3, then the thickness will increase by 1 every 4 levels. Having done this, change LEVELS back to 5 and NUMSIDES also to 5 – changing NUMSIDES means that pentagons are drawn instead of hexagons, and you’ll see clearly through the gaps in the pattern that all the lines are thickness 1 except for those at the top level, which appear in the middle.
- (j) Finally, you can experiment with the values of FIRSTCOL and COLINC, which change respectively the colour used at the top level (e.g. FIRSTCOL:=1 gives *blue* at the top level), and the way in which the colours are then “cycled” at subsequent levels. Seven colours are provided, in the standard *turtle* order: BLUE, GREEN, CYAN, RED, MAGENTA, YELLOW, and WHITE, but these are cycled so that BLUE follows on from WHITE. Try putting FIRSTCOL to 3 and COLINC to 2 – this means that *cyan* is the first colour used, followed by *magenta* then *white* then *green* then *red*. More aesthetic, perhaps, though a little bit harder to work out, is COLINC:=4.
- (k) The only setting we’ve not experimented with here is SLOWDRAW, which is probably best kept as 3. If you set it to 5, it will mean that all 5 levels of recursion will be drawn onscreen with simultaneous updating – this is very slow, and seeing 3 levels being drawn is usually quite enough to have the visual effect of recursion in action. For maximum speed, make SLOWDRAW equal to 2 (any lower setting than this is ignored, because it can be disconcerting if nothing appears onscreen for ages, and the speed gain is minimal until the low levels are reached).

Recursion Factory Reference

A simple example of recursion is provided by the *triangles* program in the [Procedures and Parameters](#) [34] section, which is also available as an illustrative program through the [Help Menu](#) [17]. The Recursion Factory is essentially a generalisation of the same technique, for regular polygons of many different sorts (not just triangles) and with a wide range of adjustable parameters. These parameters are described in what follows:

<code>numsides</code>	specifies the type of polygon to be drawn (e.g. 3 implies a triangle, 4 a square etc.)
<code>initsize</code>	specifies the length of the side of the biggest polygon – the first to be drawn

Thus the first thing that the Recursion Factory does is to draw a “regular” polygon with the specified number of sides, these sides all being the given length (all the polygon’s angles are equal too, which is why it’s called “regular”). But then the program goes on to draw more polygons of the same type, only smaller. How much smaller they are is determined by the value given to:

<code>shrink</code>	determines the “shrinkage” from one level of recursion to the next, as a fraction of 120. Thus for example a value of 40 implies that the second set of polygons to be drawn will be 40/120, or one third, the size of the initial polygon. (The denominator of 120 is chosen in preference to a percentage because this enables most common fractions to be expressed as integer values.)
---------------------	--

If your original polygon has, say, 5 sides (i.e. it is a pentagon), then at the “second level of recursion” the Recursion Factory will draw 5 smaller pentagons, each beginning from one corner of the original. However you are able to determine the angle at which they are placed, because the parameter:

<code>rangle</code>	sets the angle between the centre of the n^{th} -level polygon and the centre of the $(n+1)^{\text{th}}$ -level polygon, as measured from the corner of the n^{th} -level polygon. If, for example, <i>rangle</i> is 180, then this implies that the second-level pentagon will be directly “opposite” the first-level pentagon, whereas if <i>rangle</i> is 0, the second-level pentagon will be inside the first-level pentagon. Other angles give a less “regular” but interesting effect.
---------------------	---

Moreover your first-level and second-level polygons do not have to touch, because the parameter:

<code>polygap</code>	sets the gap between polygons as a proportion (out of 120) of the earlier polygon.
----------------------	--

These parameters not only determine the relationship between the first-level polygon and the second-level polygons – they also affect, in exactly the same way, how the third-level polygons are produced from the second-level ones. How many “levels” of recursion are produced by the program depends on the parameter:

`levels` determines how many levels of recursion are performed by the program – thus a value of 1 will result in a single polygon, 2 will produce one set of “child” polygons, and so on.

A related parameter, but one which only affects the speed of display, is:

`slowdraw` determines how many levels of recursion are performed with simultaneous onscreen updating – the default value of 3 thus performs fourth- and fifth-level recursions “offscreen” before updating the screen, which significantly speeds up the pattern production.

So far we have been assuming that the polygons on which the pattern is based are themselves being drawn, but this need not be so. The Recursion Factory can operate in three different “modes”, depending on the value of:

`mode` if *mode* has the value 1, then the sides of each polygon are drawn as lines; if 2, then blots are drawn at each corner of the polygon, but the sides themselves are not drawn; if 3, circles are drawn at each corner, but again the sides themselves are not drawn.

It is also possible to change the colours used for the drawing, by setting the following two values:

`firstcol` determines the colour used for the first-level polygon – seven are available, in the standard *turtle* order: BLUE, GREEN, CYAN, RED, MAGENTA, YELLOW, and WHITE (so *firstcol(3)*, for example, draws the first polygon in cyan).

`colinc` determines the increment used for cycling through subsequent colours (from 1 to 7 inclusive); for this purpose, the colours are cycled so that BLUE follows on from WHITE.

Likewise the thickness of lines and circles depends on the values of:

`minthick` the minimum line thickness, used to draw the polygons/circles at the very last level of recursion.

`addthick` the “added” line thickness per level, expressed as twelfths. If this is zero, then polygons/circles at all levels will be drawn with the same thickness (i.e. *minthick*). If this is 12, then polygons/circles at the n^{th} -level will be 1 thicker than polygons/circles at the $(n+1)^{\text{th}}$ -level.

And the size of circles and blots depends on:

`circsize` determines the radius of blots/circles as a proportion of the polygon side (measured in 120ths). Thus if *circsize* is 30, the radius of any blots or circles drawn will be one quarter the side of the polygon at that level.

This completes the specification of the adjustable parameters of The Recursion Factory. Now the best way to find out what it can do is to experiment with it, changing the values of the parameters listed above and seeing what happens.

The Visual Compiler and the Turtle Machine

An Introduction to PCode

PCode is a form of programming language intermediate between a high-level language such as Pascal (which contains expressions rather similar to those of English) and genuine machine code (the “native” instructions which compose “EXE” files, which are in effect just sequences of numbers). Most commercial Pascal systems (such as *Delphi*) produce genuine machine code when their Pascal programs are compiled, but this Turtle system produces instead a simplified form of PCode. This variety of PCode is very like machine code – indeed it can be thought of as machine code for a virtual “Turtle Machine” – but it is far easier to understand than real-life varieties of machine code, so studying how it operates within the Turtle system provides a useful introduction to the general concepts of program compilation.

This section gives a general introduction to PCode, and presupposes that you have read about the compilation analysis display available through the [Compile Menu](#) [12]. For more detail on PCode, see next the [Technical Note on Variables, Procedures and Parameters](#) [44] (which explains how the Turtle Machine handles variable storage and parameter passing), before consulting the [PCode Reference Guide](#) [49].

The PCode Display

Assuming that the “Analysis tables” option has been selected from the [Compile](#) menu, the PCode generated by the compilation process is displayed in a grid on the “PCode” tab of the compilation analysis. A simple Turtle Pascal statement will typically correspond to a single row or “code line”, though that code line may contain a number of PCode commands (with each PCode command consisting of an instruction possibly followed by one or more parameters). Each cell in the PCode table has a unique label given by the code line number and the column number (the latter being preceded by a decimal point) – thus cell “9.4”, for example, represents the 4th instruction or parameter within the 9th PCode line of the Program.

Strictly, PCode (like genuine machine code) is simply a sequence of numbers. To see it in numeric form, click on the radio button marked *decimal “Machine Code”* or, if you’d prefer to see the numbers in hexadecimal (i.e. base 16) form, click on *hexadecimal “Machine Code”*. Although these numbers are not strictly machine code on a real personal computer, it would be possible to create a computer chip (a “Turtle Machine”) which did operate using this sort of numeric code as machine instructions – hence it is here referred to as “machine code”.

If you now click instead on one of the “Assembler” *instructions* buttons, you’ll see that some of the numbers (but not all of them) get replaced again by instruction codes such as LDIN or STVG. This illustrates the kind of display typical of an “Assembler”, which is a programming system that enables human programmers to construct machine code directly rather than through compilation. Take for example the Machine Code sequence:

```
16 25 33 7
```

The Assembler display translates this as:

```
LDIN 25 STVG 7
```

indicating that code 16 stands for the instruction LDIN (i.e. load an integer), and code 33 for the instruction STVG (i.e. store the loaded value to a global variable). What this sequence of commands means, therefore, is “load the integer value 25, and store the loaded value to the seventh global variable”. In other words, this whole PCode sequence might be equivalent to the Pascal statement “b:=25” (assuming, say, that *a* is the first and *b* is the second declared global variable in the Program, thus making *b* the seventh global variable overall after *turtx*, *turty*, *turtd*, *turtt*, *turtc*, and *a* – this is where the number 7 following STVG comes in).

When you get used to the Assembler display, you may be able to work out how the whole of your program is really operating. Suppose for example that you had the following simple program:

```

PROGRAM spiral;
VAR length: integer;

PROCEDURE lineturn;
BEGIN
  forward(length);
  right(60)
END;

BEGIN
  length:=10;
  repeat
    randcol(4);
    lineturn;
    length:=length+10
  until length>400
END.

```

This is compiled into the following PCode, consisting of only 34 numbers (but here displayed using the 4-letter assembler “mnemonics” where appropriate). It is this sequence of PCode instructions (and not the Pascal program that you have written) which is actually being followed when the Turtle system runs:

```

(1)  PSPR 1
(2)  LDVG 6  FWRD
(3)  LDIN 60  RGHT
(4)  PLPR  ENDP
(5)  LDIN 10  STVG 6
(6)  LDIN 4   RNDC
(7)  PROC 1
(8)  LDVG 6  LDIN 10  PLUS  STVG 6
(9)  LDVG 6  LDIN 400  MORE  IFNO 6
(10) HALT

```

The first four code lines give the content of the “lineturn” procedure. PSPR 1 “pushes” the value of 1 onto the Procedure Register, indicating to the system that it has entered the first declared procedure in the program. LDVG 6 FWRD loads the value of the sixth global variable (i.e. *length*, because the first five are always *turtx*, *turdy*, *turtd*, *turtt* and *turtc*), and calls the FWRD routine, which moves the *turtle* forward by the specified amount. Then LDIN 60 RGHT loads the integer 60 and turns the *turtle* to the right by the specified number of degrees. Finally, PLPR pulls the (previously pushed) 1 off the Procedure Register, and ENDP indicates the end of the procedure.

Line (5) of the PCode (which corresponds to the beginning of the main program) assigns the value 10 to the sixth variable (*length* again), and line (6) loads the integer 4 before calling the RNDC routine to assign a random colour from the first 4 standard colours. Line (7), PROC 1, then calls the procedure which begins on line 1 of the program. Line (8), which is executed after the procedure call has returned, is more complicated:

```

(8)  LDVG 6  LDIN 10  PLUS  STVG 6

```

To understand this properly it is helpful to know about “reverse Polish notation”, but here is a brief explanation. LDVG 6 loads the value of the sixth global variable (i.e. *length*), and LDIN 10 then loads the integer 10 – note, however, that when one thing is loaded after another, it does not displace the other, but simply “covers it up” (so the original value will reappear when the second is removed). So here, the integer 10 is “stacked on top of” the value of *length*. The PLUS instruction has the effect of removing the top two items from the Program Stack, adding them together, and pushing the sum back onto the Stack. So in this case, it adds 10 to the value of *length*, and leaves the result on the Program Stack (i.e. as though that value had just been “loaded” with LDIN). STVG 6 then stores this result to the sixth variable (i.e. *length*). Thus the entire sequence of PCode instructions on line (8) is exactly equivalent to the Pascal statement “length:=length+10”.

The first four codes of line (9) should now be reasonably easy to understand:

```

(9)  LDVG 6  LDIN 400  MORE  IFNO 6

```

These load *length* and the integer value 400 onto the Program Stack. MORE then compares the two values on top of the Stack, leaving a *true* result in this case if *length* is greater than 400 (and *false* otherwise – note that *true* is represented by the number -1, and *false* by 0). IFNO 6 then looks at this result, and jumps back to line (6) if the result is *false* (i.e. 0) – this has the effect of implementing the REPEAT ... UNTIL loop of the original Pascal program. When *length* eventually achieves a value greater than 400, line (10) is finally reached, and this terminates the program.

Technical Note on Variables, Procedures and Parameters

This section is addressed to those who wish to know more about the detailed mechanisms of dynamic variable storage within the Turtle Machine, having already mastered the concepts outlined in the discussion of [Procedures and Parameters](#) [34] and in the [Introduction to PCode](#) [42]. It will make reference at various points to PCode instructions beyond those covered in that introduction, and for these you should consult the [PCode Reference Guide](#) [49] as the need arises. Note that the overall form of mechanism described here – involving a heap, procedural pointers and so forth – is the standard method of implementing dynamic variables in recursive languages, so the concepts learned by examining how the Turtle Machine operates will be of general benefit in facilitating understanding of the important (but notoriously difficult) topic of compilation. Here at least these concepts should prove relatively easy to acquire, because the Turtle Machine's PCode has been designed precisely to make its operation as straightforward and transparent as possible.

The Heap and its Maintenance Mechanisms

Obviously any programming system that supports recursion must allow for more than one “instance” of a procedure to be in progress at the same time. A clear example of this is given by the *triangles* program through which the concept of recursion was introduced (in the section on [Procedures and Parameters](#) [34]), where the first procedure instance *triangle(256)* itself calls *triangle(128)*, which in turn calls *triangle(64)*, and so on. All these different instances of the procedure must have their own “private” copies of their variables, with the different private instances of the variable *size* here taking the values 256, 128 and 64 respectively. But this raises two major practical problems: first, where are these private variables to be stored? And second, how can each instance of the procedure keep track of its own private variables and ensure that the right ones are used?

All Turtle Graphics variables are stored in a structure called the Heap, which is essentially a long sequence of memory locations each of which is capable of holding a single integer. At the “bottom” of the Heap (i.e. at index position 1) is stored the first global variable, which is always *turtx*, the x-coordinate of the *turtle*. This is followed at index positions 2, 3, 4 and 5 by *turty*, *turtd*, *turtt* and *turtc* respectively (the *turtle*'s y-coordinate, direction, thickness, and colour), and then, from index positions 6 onwards, by any global variables declared in the Program, in order of declaration – these index positions are shown in the “Variables” table on the “Declarations” tab of the compilation analysis (the same table also shows the relative index positions of the local variables for each procedure, which will be referred to below).

The global variables are the only ones to be stored in a fixed location on the Heap in the way just described. The local variables for each procedure, by contrast, are created only when that procedure is invoked, and are destroyed when the procedure terminates. Or more precisely, as explained above, a new set of local variables is created whenever a new *instance* of the procedure is invoked, and that set is destroyed when the corresponding *instance* of the procedure terminates. Hence each set of variables is intimately tied to its own particular instance of the procedure. (The reason why the local variables are destroyed when the instance of the procedure terminates is simply to avoid wastage of memory space on the Heap which could otherwise be used up very quickly by a highly recursive program.)

To manage the memory space on the Heap, a “pointer” is maintained which keeps track of the highest point on the Heap that is currently in use; thus if at some stage the Heap contains 57 variables (global and local combined), then this “Heap Top Pointer” will have a value of 57. Now let us look at the PCode analysis produced when a simple recursive program is compiled, to see how new local variables are created. Here is the program, which you will see produces an interesting visual effect if you copy it into the Programming Area and run it:

(Note that the global variables *g1*, *g2* and *g3*, and the local variable *v1*, play no role whatever in the operation of this program, but are included purely to illustrate variable storage mechanisms.)

```
PROGRAM recurse;
VAR g1,g2,g3: integer;

PROCEDURE drawblot(size,col: integer);
VAR v1: integer;
BEGIN
  if size>1 then
    begin
      forward(1);
      colour(col);
      blot(size);
      drawblot(size-2,col+1)
    end
  END;
BEGIN
  right(90);
  drawblot(300,blue)
END.
```

Now here is the PCode produced when this program is compiled:

```
(1)  PSPR 1  HPCL 1 3
(2)  ZERO 1 3  STVV 1 2  STVV 1 1
(3)  LDVV 1 1  LDIN 1  MORE  IFNO 8
(4)  LDIN 1  FWRD
(5)  LDVV 1 2  COLR
(6)  LDVV 1 1  BLOT
(7)  LDVV 1 1  LDIN 2  SUBT  LDVV 1 2  LDIN 1  PLUS  PROC 1
(8)  HPRE 1  PLPR  ENDP
(9)  LDIN 90  RGHT
(10) LDIN 300  LDIN 16711680  PROC 1
(11) HALT
```

The code for procedure *drawblot* starts at line (1) with PSPR 1 which “pushes” the value of 1 onto the Turtle Machine’s Procedure Register Stack – this stack provides information for the trace display, with its top value (i.e. the Procedure Register) recording which number procedure is currently running, and the stack’s height indicating how many procedures are active. HPCL 1 3 (the “1” indicates the index number of the procedure) then “claims” space on the Heap for 3 local variables, namely the two parameters *size* and *col*, and the local variable *v1*. The effect of this on the Heap Top Pointer can be seen very clearly by inspecting the trace display which will be revealed on the PCode tab if you select “Trace on run” from the [Compile Menu](#) [12] and then run the Program. You will see there that the Heap Top Pointer starts out with the value 8 – for the global variables *turtx*, *turty*, *turtd*, *turtt*, *turtc*, *g1*, *g2* and *g3* – and that every subsequent operation of HPCL 1 3 then increases the pointer value by 3. Because the procedure is called recursively a total of 151 times before any instance of it terminates, the Heap Top Pointer reaches a maximum of 461 (8 + 151×3) before quickly declining. At this maximum, there are thus 151 instances of the local variable sets stored on the Heap!

Claiming space on the Heap prevents the relevant memory locations being claimed by any other procedure (or any other instance of the current procedure). But before the procedure itself gets down to business these locations must be filled with the appropriate initial values, which is achieved using the commands ZERO 1 3 (which sets to 0 the location corresponding to the local variable *v1* having index value 3), STVV 1 2 (which removes the top value from the Program Stack into the parameter *col* having index value 2), and STVV 1 1 (which removes what is now the top value from the Program Stack into the parameter *size* having index value 1 – again, in all of these cases the first “1” parameter is the procedure index number). This illustrates a general pattern, whereby at the beginning of any procedure all the local variables are set to 0 and the parameters are filled with values from the Stack (this being done in reverse order of index value, for a reason which will soon become clear).

Lines (3) to (6) of the PCode are fairly straightforward – LDVV 1 1 loads the value of the local variable in procedure 1 (procedure *drawblot*) that has index value 1 (i.e. the parameter *size*) onto the Program Stack, while the other commands that follow should be unproblematic if you have worked through the [Introduction to PCode](#) [42]. Line (7) has the effect of putting the values *size-2* and *co+1* onto the Stack and then calling (with PROC 1) the procedure that begins at line (1) – this, of course, is the recursive call of procedure *drawblot*. Note that the reverse order “unpacking” of the formal parameters within the procedure is a direct implication of the correct order “packing” of the actual parameters before the procedure is called, because the Program Stack operates on a “last-in-first-out” basis. Finally, the procedure ends with the commands HPRE 1, PLPR, and ENDP; the first of these releases the Heap memory previously claimed by the procedure with index 1 (so the Heap Top Pointer is reduced to the value it had before the current procedure call), the second pulls the top value from the Procedure Register Stack, and the third terminates the procedure and returns execution to the position in the PCode from which this particular procedure instance was called (so if the procedure instance was called recursively from line (7), then execution resumes from line (8); whereas if the procedure instance was called from the main program at line (10), execution instead jumps to line (11) and hence immediately halts). The Turtle Machine maintains a Return Stack which keeps track of the code line from which each procedure was called. When the procedure is called with PROC, the current line number is pushed onto the Return Stack; when the procedure terminates, the line number is pulled from the Return Stack and the jump made to the end of that line (which effectively means that execution continues from the immediately following line).

The main program itself consists of code lines (9) and (10). LDIN 90 RGHT turns the *turtle* to the right by 90 degrees, while LDIN 300 LDIN 16711680 loads the two values 300 and 16711680 onto the Program Stack before calling procedure *drawblot*. Note here that the decimal value 16711680 is equivalent to the hexadecimal value \$FF0000, as can be seen by clicking on one of the hexadecimal radio buttons on the PCode tab (you will see that a narrower font is used for this hexadecimal display – this is precisely to enable a six-digit hexadecimal colour code to be seen in its entirety). Conversion to hexadecimal makes the colour codes more comprehensible, because then the blue, green and red components of any colour are represented by different digits. Thus “FF0000” represents pure blue because the first two hexadecimal digits concern the intensity of blue (from “00” to “FF”, i.e. 0 to 255), the next two the intensity of green, and the final two the intensity of red.

That explains the interpretation of the PCodes, but now we need to examine some of the internal workings of the Turtle Machine to understand how each instance of a procedure manages to use the correct set of variable storage locations. We saw above that a Heap Top Pointer is maintained to keep track of the currently highest used point on the Heap, but the mechanism used to accomplish this is more complicated than a single pointer value, involving the use of a Heap Control Stack which – like the main Program Stack – operates on a “last-in-first-out” basis. Whenever new memory is claimed from the Heap during execution (i.e. whenever a new instance of a procedure commences), the old value of the Heap Top Pointer is not destroyed; instead it is “covered up” by the new value, building up a stack of values as the program runs. It is the top value on this Heap Control Stack which acts as the Heap Top Pointer (just as the Procedure Register is the top value of the Procedure Register Stack). Finally, each procedure also has its own individual pointer called a Procedure Heap Pointer (PHP), which keeps track of the *value that the Heap Top Pointer had when the current instance of the procedure was called* (and *before* the current instance claimed space on the Heap). This Procedure Heap Pointer gives the base address from which the location of that instance’s local variables can be calculated. This can all seem extremely complicated in the abstract, and is far better understood by illustration.

Suppose, for example, that we have a program with three global variables (*g1*, *g2* and *g3*) and two procedures, *proca* and *procb*, each of which has two local variables (*a1*, *a2*, *b1* and *b2* – these could be parameters or just ordinary variables). Suppose also that procedure *proca* is recursive, and that procedure *procb* calls procedure *proca*. When the program starts, the Heap Top Pointer will have a value of 8 (just as in the case of the program *recurse* above), so the Heap Control Stack will contain just this one value, and the order of the variables stored on the Heap will be as listed below. Also shown here are the two Procedure Heap Pointers (which soon play a role), and the Procedure Register Stack (which services the trace display, but has no effect on execution).

HC Stack:	8
Heap:	<i>turtx</i> , <i>turty</i> , <i>turtd</i> , <i>turtt</i> , <i>turtc</i> , <i>g1</i> , <i>g2</i> , <i>g3</i>
PHP1:	0
PHP2:	0
PR Stack:	0

When *procb* is called, the first command will be PSPR 2, pushing the value 2 onto the Procedure Register Stack (because *procb* is the second procedure declared in the program). Then will come the command HPCL 2 2, which reserves space on the Heap for *procb*'s two local variables by first exchanging the current Heap Top Pointer with the relevant Procedure Heap Pointer (here PHP2), and then pushing onto the HC Stack the newly calculated value of the Heap Top Pointer (i.e. the old value of 8 plus 2). The two local variables themselves are stored in order of their declaration (which determines their index order: *b1* is given index 1, and *b2* index 2):

```

HC Stack:    0  10  (top value is shown last)
Heap:        turtx, turty, turtd, turtt, turtc, g1, g2, g3, b1, b2
PHP1:        0
PHP2:        8
PR Stack:    0  2

```

Suppose now that *procb* calls *proca*. Exactly the same mechanism just described, but now with the commands PSPR 1 and HPCL 1 2 and hence involving *proca*'s variables and PHP1, results in the following situation:

```

HC Stack:    0  0  12
Heap:        turtx, turty, turtd, turtt, turtc, g1, g2, g3, b1, b2, a1, a2
PHP1:        10
PHP2:        8
PR Stack:    0  2  1

```

A recursive call of *proca* to itself results in:

```

HC Stack:    0  0  10  14
Heap:        turtx, turty, turtd, turtt, turtc, g1, g2, g3, b1, b2, a1, a2, a1, a2
PHP1:        12
PHP2:        8
PR Stack:    0  2  1  1

```

And one more recursive call:

```

HC Stack:    0  0  10  12  16
Heap:        turtx, turty, turtd, turtt, turtc, g1, g2, g3, b1, b2, a1, a2, a1, a2, a1, a2
PHP1:        14
PHP2:        8
PR Stack:    0  2  1  1  1

```

Focus now on what is going on while this third instance of *proca* is executing, and suppose that the PCode command:

```
LDVV 1 2
```

is encountered, meaning that the second local variable of the first procedure should be loaded onto the Program Stack. The Turtle Machine identifies the appropriate location on the Heap by taking the Procedure Heap Pointer for the relevant procedure (here PHP1, with a current value of 14) and adding the variable index (here 2) to yield a result of 16. Counting through the Heap listing above, you will see that this gives the location of the third instance of *a2*, as indeed it should (because that is the second local variable within the current instance of procedure *proca*). **This is the general mechanism of local variable reference**, which is in itself quite straightforward. But what now needs to be understood is how the various stack operations succeed in maintaining that mechanism's integrity through procedure termination.

Suppose, then, that the current instance of *proca* terminates, which will involve the commands HPRE 1, PLPR, and ENDP. When the HPRE 1 command is executed, this releases the previously claimed memory on the Heap by removing the top value from the Heap Control Stack, and exchanging the new top value with the

Procedure Heap Pointer for the procedure with index 1 (i.e. PHP1). PLPR then removes the top value from the Procedure Register Stack, and ENDP returns control to the point within the previous instance of *proca* from which the terminating instance was called. All this results in the following situation:

```
HC Stack:    0  0  10  14
Heap:        turtx, turty, turtd, turtt, turtc, g1, g2, g3, b1, b2, a1, a2, a1, a2, (a1), (a2)
PHP1:        12
PHP2:        8
PR Stack:    0  2  1  1
```

The crucial point to recognise is that (thanks to the relevant stack operations) this is exactly the same situation as obtained before the third instance of *proca* was called, except that the now redundant local variables of that instance are still sitting uselessly on the Heap. (Those instances of the local variables are now inaccessible, because their locations lie beyond the new value of the Heap Top Pointer – i.e. 14 – and these locations are thus free to be claimed by any future procedure call and overwritten by a new set of local variables.) This return to the pre-call situation shows that integrity of the local variable reference mechanism has indeed been maintained, and you should now find it relatively easy to check for yourself that this will continue to operate correctly as the remaining outstanding procedure instances terminate in a similar manner.

Dealing with Reference Parameters

All of the above discussion has taken for granted that we are dealing with ordinary value parameters and variables. Reference parameters need to be handled differently, because they are *indirected* – in other words, any command to load or store a reference parameter must be interpreted as referring indirectly to some other variable which is not local to the procedure in question, with the local reference parameter effectively acting as an alias for this other variable. However the Heap control mechanisms are identical, because the same issues involving multiple instances of procedures and the need to maintain referential integrity arise whatever the nature of the parameters involved.

The essential difference between value and reference parameters is that with the former, the corresponding location on the Heap stores the parameter's current value (just as with a local variable); whereas with the latter, the corresponding location on the Heap stores the *Heap address* of the variable for which the parameter is standing as an alias (i.e. the numerical location on the Heap where that aliased variable is itself stored). It is these Heap addresses that are loaded and stored by the special instructions LDAG, LDAV, LDAR and STAR, and interpreted by the instructions LDVR and STVR (which load from and store to the indirected Heap address).

PCode Reference Guide

There follows a complete list, in numerical order, of all the PCode instructions and their effects. The PCode instructions are grouped functionally, according to a pattern reflected in their hexadecimal codes. Each instruction mnemonic – we here take **LDIN** as an example – is followed by:

1. The instruction's numeric value in decimal and hexadecimal notation, with the latter being enclosed in brackets and preceded by "\$". In the case of LDIN, this gives: **16 (\$10)**
2. The number of PCode parameters taken by the instruction, and the overall effect of the instruction on the Program Stack. Thus in the case of LDIN, **{1,+1}** signifies that one PCode number after the instruction itself is taken to be a parameter to LDIN, and the overall effect is to add one integer (namely, that very PCode) to the Program Stack. Where either value within the curly brackets is zero, it is shown as a dot. When the value added or taken from the Program Stack is the Heap address of a parameter or variable, this is shown as **+a** or **-a** instead of **+1** or **-1** respectively.
3. An informal description of the instruction's effect. For any instruction that takes one or more parameters, an example command is given, highlighted in bold for easy reference.

All of this reference guide presupposes the concepts outlined in the [Introduction to PCode](#) [42], notably the Program Stack. However some of the more sophisticated instructions require also an understanding of the Heap, the Heap Control Stack, the Procedure Heap Pointers, the Procedure Register Stack, and the Return Stack. All of these are covered in the [Technical Note on Variables, Procedures and Parameters](#) [44].

Null Command

NULL 0 (\$0) {.,.} The NULL instruction has no effect whatever, being completely ignored by the Turtle Machine.

Loading and Storage of Variables

LDIN 16 (\$10) {1,+1} Load (or "push") onto the Program Stack the integer value of the following PCode. Thus **LDIN 50** will load the value 50.

LDVG 17 (\$11) {1,+1} Load (or "push") onto the Program Stack the current value of the global variable indexed by the following PCode. Thus **LDVG 6** will load the current value of the sixth global variable (note that the first five global variables are always *turtx*, *turty*, *turtd*, *turtt*, and *turtc*).

LDVV 18 (\$12) {2,+1} Load (or "push") onto the Program Stack the current value of the local variable (or value parameter) indexed by the following two PCodes. Thus **LDVV 3 2** will load the current value of the second local parameter or variable defined within the third procedure.

LDVR 19 (\$13) {2,+1} Load (or "push") onto the Program Stack the current value of the local reference parameter indexed by the following two PCodes. Thus **LDVR 3 2** will load the current value of the second local parameter or variable defined within the third procedure. (LDVR is used in place of LDVV when the parameter concerned is a *reference* rather than a *value* parameter – thus the value loaded onto the Stack is retrieved from an indirected Heap address.)

LDAG 20 (\$14) {1,+a} Load (or "push") onto the Program Stack the Heap address of the global variable indexed by the following PCode. Thus **LDAG 6** will load the Heap address of the sixth global variable (this address is, in fact, simply the number 6).

LDAV	21	(\$15)	{2,+a}	Load (or “push”) onto the Program Stack the Heap address of the local variable (or value parameter) indexed by the following two PCodes. Thus LDAV 2 5 will load the Heap address of the fifth local parameter or variable defined within the second procedure. This Heap address indicates the actual location on the Heap where the relevant instance of that parameter or variable is stored.
LDAR	22	(\$16)	{2,+a}	Load (or “push”) onto the Program Stack the indirected Heap address of the local reference parameter indexed by the following two PCodes. Thus LDAR 2 5 will load the indirected Heap address of the fifth local parameter or variable defined within the second procedure (unlike in the case of LDAV , this address will depend on which variable has been given as the actual parameter to the procedure – if, for example, the actual parameter is in fact the sixth global variable, then the address will be the number 6).
STVG	33	(\$21)	{1,-1}	Remove (or “pull”) the top value from the Program Stack, and store this value in the global variable indexed by the following PCode. Thus STVG 7 will store the value taken from the Program Stack into the seventh global variable (note that the first five global variables are always <i>turtx</i> , <i>turty</i> , <i>turtd</i> , <i>turtt</i> , and <i>turtc</i>).
STVV	34	(\$22)	{2,-1}	Remove (or “pull”) the top value from the Program Stack, and store this value in the local variable (or value parameter) indexed by the following two PCodes. Thus STVV 3 2 will store the value taken from the Program Stack into the second local parameter or variable defined within the third procedure.
STVR	35	(\$23)	{2,-1}	Remove (or “pull”) the top value from the Program Stack, and store this value in the local reference parameter indexed by the following two PCodes. Thus STVR 3 2 will store the value taken from the Program Stack into the second local parameter or variable defined within the third procedure. (STVR is used in place of STVV when the parameter concerned is a <i>reference</i> rather than a <i>value</i> parameter – thus the top value from the Stack is saved to an indirected Heap address.)
STAR	38	(\$26)	{2,-a}	Remove (or “pull”) the top value from the Program Stack, and store this as the indirected Heap address of the local reference parameter indexed by the following two PCodes. Thus STAR 4 3 will store the value taken from the Program Stack as the indirected Heap address of the third local parameter or variable defined within the fourth procedure (note that this instruction only ever occurs at the beginning of a procedure, assigning the relevant indirected addresses to that procedure’s reference parameters).
ZERO	39	(\$27)	{2,.}	Store the value 0 in the local variable indexed by the following two PCodes. Thus ZERO 1 4 will store 0 into the fourth local parameter or variable defined within the first procedure (note that this instruction only ever occurs at the beginning of a procedure, initialising that procedure’s local variables to zero).

Flow Control and Procedure Handling

JUMP	48	(\$30)	{1,.}	Jump to the PCode line indexed by the following PCode. Thus after a JUMP 5 command, execution will continue from line 5 of the compiled PCode.
IFNO	49	(\$31)	{1,-1}	Remove (or “pull”) the top value from the Program Stack, and test whether it is zero or non-zero. If it is zero, then jump to the PCode line indexed by the following PCode. Thus after an IFNO 5 command, if the value previously on the Program Stack was 0, then execution will jump to line 5 of the compiled PCode; otherwise, execution will continue in the usual way without any jump. (Note that the value of 0 is treated as equivalent to <i>false</i> , so is left on the Program Stack as the result of any binary comparison which is false; likewise the value -1 is equivalent to <i>true</i> .)

PROC	50	(\$32)	{1,..}	Jump to the procedure which starts on the PCode line indexed by the following PCode. So after a PROC 5 command, execution will continue from line 5 of the compiled PCode (just as with JUMP). However PROC differs from JUMP in having an additional effect: it pushes the current PCode line number onto the Return Stack, so that when the procedure finishes with ENDP , execution can continue from the point from which the procedure was called.
ENDP	51	(\$33)	{,..}	Remove (or “pull”) the top value from the Return Stack, representing the PCode line number of the PROC command which called the current procedure. Then continue execution from that point, with the immediately following PCode (i.e. the first instruction of the code line after the relevant PROC call).
HPCL	52	(\$34)	{2,..}	Claim from the Heap sufficient space for a procedure's local parameters and variables, as specified by the following two PCodes. Thus HPCL 2 5 claims space for five local integer variables, as required by the second procedure of the program. Technically, this is achieved as follows: HPCL exchanges the value currently on top of the Heap Control Stack (this value represents the Heap size prior to the HPCL command) with the Procedure Heap Pointer for the relevant procedure (in the case of HPCL 2 5 , the procedure with index 2), and then pushes on to the Heap Control Stack the new Heap size (which, in the case of HPCL 2 5 , will be the previous Heap size plus five). Overall, this means that: (a) the relevant procedure's Heap Pointer will point to the bottom of the space which has been set aside for the new variables; (b) the old value of that Heap Pointer will be stored on the Heap Control Stack, ready for restoration when the procedure (instance) returns; (c) the new Heap size will be on top of the Heap Control Stack ready for the next procedure call.
HPRE	53	(\$35)	{1,..}	Release the space previously allocated on the Heap for the local parameters and variables of the procedure indexed by the following PCode. Technically, this is achieved by removing the value currently on top of the Heap Control Stack (this value represents the Heap size prior to the HPRE command) and then exchanging the new value on top of the Heap Control Stack with the Procedure Heap Pointer for the relevant procedure (e.g. the third procedure in the program in the case of HPRE 3). As can be seen by reference to HPCL above, this restores the Heap situation prior to that procedure's being called.
PSPR	54	(\$36)	{1,..}	Load (or “push”) onto the Procedure Register Stack the value of the following PCode. Thus PSPR 2 sets the Procedure Register (i.e. the current top value on the Procedure Register Stack) to 2, indicating that the second procedure in the program has now been entered, and enabling this to be shown in the trace display.
PLPR	55	(\$37)	{,..}	Remove (or “pull”) the top value from the Procedure Register Stack.
HALT	56	(\$38)	{,..}	HALT brings the entire program to a halt. It can occur only as the last instruction of the compiled PCode.

Turtle Commands Adjusting Run-Time Flags

All of the instructions in the next two groups are exact PCode equivalents of Turtle Graphics commands. For details of these and other Turtle Graphics equivalents, see [Programming Essentials](#) [23].

PNUP	64	(\$40)	{,..}	PNUP is an exact PCode equivalent of the Turtle Graphics command PENUP .
PNDN	65	(\$41)	{,..}	PNDN is an exact PCode equivalent of the Turtle Graphics command PENDOWN .
UDAT	66	(\$42)	{,..}	UDAT is an exact PCode equivalent of the Turtle Graphics command UPDATE .

NDAT 67 (\$43) {...} NDAT is an exact PCode equivalent of the Turtle Graphics command NOUPDATE.

Other Turtle Commands Having No Stack Effect

HOME 80 (\$50) {...} HOME is an exact PCode equivalent of the Turtle Graphics command HOME.

RMBR 81 (\$51) {...} RMBR is an exact PCode equivalent of the Turtle Graphics command REMEMBER.

Turtle Movement and Colour Commands, Taking 1 Value from Stack

*All of the instructions in the next two groups are PCode equivalents of Turtle Graphics commands, except that the PCode instructions take their parameter (or “argument”) from the top of the Program Stack. Thus for example if the value 50 is at the top of the Program Stack, then the **FWRD** instruction performs the equivalent of FORWARD(50) and removes the 50 from the Stack. For details of these and other Turtle Graphics equivalents, see [Programming Essentials](#) [23].*

FWRD 96 (\$60) {.,-1} FWRD is a PCode equivalent of the Turtle Graphics command FORWARD, taking its parameter from the Program Stack.

BACK 97 (\$61) {.,-1} BACK is a PCode equivalent of the Turtle Graphics command BACK, taking its parameter from the Program Stack.

LEFT 98 (\$62) {.,-1} LEFT is a PCode equivalent of the Turtle Graphics command LEFT, taking its parameter from the Program Stack.

RGHT 99 (\$63) {.,-1} RGHT is a PCode equivalent of the Turtle Graphics command RIGHT, taking its parameter from the Program Stack.

SETX 100 (\$64) {.,-1} SETX is a PCode equivalent of the Turtle Graphics command SETX, taking its parameter from the Program Stack.

SETY 101 (\$65) {.,-1} SETY is a PCode equivalent of the Turtle Graphics command SETY, taking its parameter from the Program Stack.

THIK 102 (\$66) {.,-1} THIK is a PCode equivalent of the Turtle Graphics command THICKNESS, taking its parameter from the Program Stack.

COLR 103 (\$67) {.,-1} COLR is a PCode equivalent of the Turtle Graphics command COLOUR, taking its parameter from the Program Stack.

RNDC 104 (\$68) {.,-1} RNDC is a PCode equivalent of the Turtle Graphics command RANDCOL, taking its parameter from the Program Stack.

BLNK 105 (\$69) {.,-1} BLNK is a PCode equivalent of the Turtle Graphics command BLANK, taking its parameter from the Program Stack.

WAIT 106 (\$6A) {.,-1} WAIT is a PCode equivalent of the Turtle Graphics command PAUSE, taking its parameter from the Program Stack.

Turtle Shape-Drawing Commands, Taking 1 Value from Stack

CIRC 112 (\$70) {.,-1} CIRC is a PCode equivalent of the Turtle Graphics command CIRCLE, taking its parameter from the Program Stack.

BLOT 113 (\$71) {.,-1} BLOT is a PCode equivalent of the Turtle Graphics command BLOT, taking its parameter from the Program Stack.

POLY 114 (\$72) {.,-1} POLY is a PCode equivalent of the Turtle Graphics command POLYLINE, taking its parameter from the Program Stack.

- FILL** 115 (\$73) {.,-1} **FILL** is a PCode equivalent of the Turtle Graphics command POLYGON, taking its parameter from the Program Stack.
- FRGT** 116 (\$74) {.,-1} **FRGT** is a PCode equivalent of the Turtle Graphics command FORGET, taking its parameter from the Program Stack.

Turtle Commands Taking 2 or More Values from Stack

All of the following instructions are PCode equivalents of Turtle Graphics commands, except that the PCode instructions take their parameters – in reverse order – from the top of the Program Stack. Thus for example if the value 50 is at the top of the Program Stack, with 100 as the next value down, then the **MVXY** instruction performs the equivalent of **MOVEXY(100,50)** and removes both the 50 and the 100 from the Stack. For details of these and other Turtle Graphics equivalents, see [Programming Essentials](#) [23].

- MVXY** 128 (\$80) {.,-2} **MVXY** is a PCode equivalent of the Turtle Graphics command MOVEXY, taking both its parameters from the Program Stack.
- DRXY** 129 (\$81) {.,-2} **DRXY** is a PCode equivalent of the Turtle Graphics command DRAWXY, taking both its parameters from the Program Stack.
- TOXY** 130 (\$82) {.,-2} **TOXY** is a PCode equivalent of the Turtle Graphics command SETXY, taking both its parameters from the Program Stack.
- CANV** 136 (\$88) {.,-4} **CANV** is a PCode equivalent of the Turtle Graphics command CANVAS, taking all four of its parameters from the Program Stack.

Unary Numeric Operators, Replacing Top Value on Stack

- NEG** 144 (\$90) {.,.} **NEG** replaces the value currently on top of the Program Stack with its arithmetic negation. Thus for example if the value 50 is at the top of the Stack, **NEG** will replace it with -50.

Unary Boolean Operators, Replacing Top Value on Stack

- NOT** 160 (\$A0) {.,.} **NOT** replaces the value currently on top of the Program Stack with its Boolean negation. Thus for example if the value 0 (representing the Boolean value *false*) is at the top of the Stack, then **NOT** will replace it with -1 (representing the Boolean value *true*), and vice-versa. The effect of Boolean negation on other numbers is generated by bitwise inversion, but it can seem surprising (e.g. **NOT**(8) gives -9); this is a consequence of the *twos-complement* system of number representation, which is beyond the scope of this Help file.

Binary Numeric Operators, Overall Reducing Stack by 1

These five instructions are PCode equivalents of the standard arithmetic operators, which take their two operands – in reverse order – from the top of the Program Stack. Thus if the value 50 is at the top of the Program Stack, with 100 as the next value down, then the **DIV** instruction performs the integer division 100 div 50, removing both the 50 and the 100 from the Stack but putting back the result 2 on top of the Stack. Note that this is **INTEGER** division, since Turtle Graphics does not allow fractional numbers, so for example 20 div 3 will yield the result 6.

- PLUS** 176 (\$B0) {.,-1} **PLUS** adds the top value on the Program Stack to the second value on the Stack, removing both values from the Stack but then placing back on the Stack the result of the addition.
- SUBT** 177 (\$B1) {.,-1} **SUBT** subtracts the top value on the Program Stack from the second value on the Stack, removing both values from the Stack but then placing back on the Stack the result of the subtraction.

- MULT 178 (\$B2) {.,-1}** **MULT** multiplies the second value on the Program Stack by the top value on the Stack, removing both values from the Stack but then placing back on the Stack the result of the multiplication.
- DIV 179 (\$B3) {.,-1}** **DIV** divides the second value on the Program Stack by the top value on the Stack, removing both values from the Stack but then placing back on the Stack the result of the division. As noted above, **DIV** implements *integer* division (commonly signified in programming languages by *div*), since fractional results are not permitted (thus 17 div 3 = 5, because 3 goes 5 times into 17, with a remainder of 2).
- MOD 180 (\$B4) {.,-1}** **MOD** divides the second value on the Program Stack by the top value on the Stack, removing both values from the Stack but then placing back on the Stack the remainder of the division (e.g. 17 mod 3 = 2, because 3 goes 5 times into 17, with a remainder of 2).

Binary Boolean Operators, Overall Reducing Stack by 1

These three instructions are PCode equivalents of the standard binary Boolean operators, which take their two operands from the top of the Program Stack. Their effect is described below only in terms of the standard numeric representations of the two Boolean values TRUE and FALSE (i.e. -1 and 0 respectively). Applied to positive integers, the result is generated by bitwise ANDing, ORing and XORing respectively. The effect of these Boolean operators on negative numbers can seem surprising, as a consequence of the twos-complement system of number representation which is beyond the scope of this Help file.

- AND 192 (\$C0) {.,-1}** **AND** performs a Boolean AND operation (“conjunction”) on the top two values on the Program Stack, removing those two values but then placing back onto the Stack the result of the conjunction. With the usual Boolean numeric equivalents (0 for *false* and -1 for *true*) this satisfies the standard *truth-table* for AND (whereby P AND Q yields *true* if and only if *both P and Q* are *true* individually).
- OR 193 (\$C1) {.,-1}** **OR** performs a Boolean OR operation (“disjunction”) on the top two values on the Program Stack, removing those two values but then placing back onto the Stack the result of the disjunction. With the usual Boolean numeric equivalents (0 for *false* and -1 for *true*) this satisfies the standard *truth-table* for OR (whereby P OR Q yields *false* if and only if *both P and Q* are *false* individually).
- XOR 194 (\$C2) {.,-1}** **XOR** performs a Boolean XOR (“exclusive disjunction”) operation on the top two values on the Program Stack, removing those two values but then placing back onto the Stack the result of the operation. With the usual Boolean numeric equivalents (0 for *false* and -1 for *true*) this satisfies the standard *truth-table* for XOR (whereby P XOR Q yields *true* if and only if P and Q differ in truth-value).

Binary Comparison Operators, Overall Reducing Stack by 1

*These six instructions are PCode equivalents of the familiar basic numerical comparison operators, which take their two operands – in reverse order – from the top of the Program Stack. Thus if the value 50 is at the top of the Program Stack, with 100 as the next value down, then the **EQAL** instruction removes both the 50 and the 100 from the Stack and puts back the result 0 (representing falsehood), because 50 and 100 are not equal. The **MORE** instruction, however, would put back the result -1 (representing truth), because 100 is indeed greater than 50.*

- EQAL 208 (\$D0) {.,-1}** **EQAL** performs a numerical comparison of the top two values on the Program Stack, yielding the result -1 (representing *true*) if the second value is equal to the top value, and 0 (representing *false*) otherwise. The two compared values are removed from the Stack, and the result is placed back onto it.

NOEQ	209	(\$D1)	{.,-1}	NOEQ performs a numerical comparison of the top two values on the Program Stack, yielding the result -1 (representing <i>true</i>) if the second value is different from the top value, and 0 (representing <i>false</i>) if they are the same. The two compared values are removed from the Stack, and the result is placed back onto it.
LESS	210	(\$D2)	{.,-1}	LESS performs a numerical comparison of the top two values on the Program Stack, yielding the result -1 (representing <i>true</i>) if the second value is less than the top value, and 0 (representing <i>false</i>) otherwise. The two compared values are removed from the Stack, and the result is placed back onto it.
MORE	211	(\$D3)	{.,-1}	MORE performs a numerical comparison of the top two values on the Program Stack, yielding the result -1 (representing <i>true</i>) if the second value is greater than the top value, and 0 (representing <i>false</i>) otherwise. The two compared values are removed from the Stack, and the result is placed back onto it.
LSEQ	212	(\$D4)	{.,-1}	LSEQ performs a numerical comparison of the top two values on the Program Stack, yielding the result -1 (representing <i>true</i>) if the second value is less than or equal to the top value, and 0 (representing <i>false</i>) otherwise. The two compared values are removed from the Stack, and the result is placed back onto it.
MREQ	213	(\$D5)	{.,-1}	MREQ performs a numerical comparison of the top two values on the Program Stack, yielding the result -1 (representing <i>true</i>) if the second value is greater than or equal to the top value, and 0 (representing <i>false</i>) otherwise. The two compared values are removed from the Stack, and the result is placed back onto it.

Instructions to Provide Stack Variations on the Turtle Machine

These six instructions enable the Return Stack and Heap Control Stack to be dispensed with in the Turtle Machine (subject to the choice of options made through the Compile menu), so that it can run as a single-stack machine with complex "stack frames" maintaining the relevant return line numbers and Heap parameters.

LDRJ	224	(\$E0)	{.,+1}	LDRJ pushes onto the Program Stack the PCode line number of the instruction itself, as a preliminary to making a procedure call.
PLRJ	225	(\$E1)	{.,-1}	PLRJ pulls the top value from the Program Stack, interpreting it as a PCode line number and jumping accordingly, to the end of the code line concerned – enabling execution to continue from the following code line. PLRJ thus provides an alternative to ENDP as a method for ending a procedure, in which the return line number is taken from the Program Stack rather than the usual Return Stack. (But to avoid overflow, PLRJ also pulls the top line number from the Return Stack, so that stack is still maintained though not used.)
LDHT	226	(\$E2)	{.,+1}	LDHT pushes onto the Program Stack the current value of the Heap Top Pointer, to enable Heap maintenance to be carried out using the Program Stack and standard arithmetical operators rather than with HPCL and HPRE.
STHT	227	(\$E3)	{.,-1}	STHT pulls the top value from the Program Stack, and stores it as the new value of the Heap Top Pointer.
LDHB	228	(\$E4)	{.,+1}	LDHB pushes onto the Program Stack the current value of the Procedure Heap Pointer for the procedure indexed by the following PCode. Thus LDHB 2 loads onto the Stack the current Heap Pointer for the program's second procedure.
STHB	229	(\$E5)	{.,-1}	STHB pulls the top value from the Program Stack, and stores it as the new value of the Procedure Heap Pointer for the procedure indexed by the following PCode. Thus STHB 3 stores the pulled value as the Procedure Heap Pointer for the third procedure in the program.

Learning with the System

Exercises

The exercises below are intended to get you started using Turtle Graphics programming, and to take you fairly systematically through the various facilities that are provided. If you are following a taught course based on this system, then you should work through these exercises in turn, taking advantage of any demonstration sessions that are provided (where supervision may be available to provide extra assistance if necessary). If you are using the system independently, be particularly sure to familiarise yourself with the structure of this Help system, so you know how to find assistance for yourself if you have problems.

If you have difficulty getting to grips with any of the concepts as they are introduced, you might find it helpful to load and run some of the illustrative programs that are available through the [Help Menu](#) [17]. Note, however, that a few of these illustrative programs are quite complex, so stick to the ones that are at your current level.

Before You Start the Exercises

Go to the [Help Menu](#) [17], select the first of the Illustrative programs (called “Simple drawing with pauses”) and see this appear in the Programming Area at the left of the screen. Click on the “RUN” button and watch what happens. Having done this, read through the section on [The Program](#) [2] so that you understand what’s going on, and then return back here.

Exercise 1

If a program is currently loaded into the system, select “New program” from the Edit menu to clear it. Make sure the flashing cursor is in the Programming Area at the left of the screen (click there if necessary), and type in the following program:

Note that program examples from this Help file can also be “cut and pasted” into the system. To try this, drag the mouse over the example below so that all six lines (within this Help file) are highlighted. Then either select “Copy” from the Help system’s Edit menu, or press CTRL-C (i.e. hold down the “Ctrl” key and press “C”) to copy the selection into the Clipboard. Now go into the Turtle system, click in the part of the Programming Area where you want the selected lines to be pasted (line 1 if no program exists yet), and then either select “Paste insert” from the Edit menu, or press CTRL-V. It’s worth getting used to using CTRL-V and CTRL-C for these operations, because they’re standard in nearly all Windows applications and are quicker than using the menus.

```
PROGRAM tgp1;  
BEGIN  
  forward(100);  
  right(120);  
  forward(100)  
END.
```

Then press on the RUN button to run it. You should see two sides of an equilateral triangle appear on the [Canvas](#) [2] (the square area on the right of the screen where the drawing is done).

Now add “statements” (i.e. Turtle Graphics Pascal commands) to the program to complete the equilateral triangle (take a look at the beginning of [Programming Essentials](#) [23] if you find the program structure at all confusing). RUN the new program to check that it works.

Next, edit the program (i.e. add, delete, or modify statements) so that the same triangle is drawn, but this time with a horizontal base (RUN it to check, as usual). Then edit it again so that, *in addition to the triangle*, it draws a red horizontal line exactly below the base, at a distance of 50 units (note – here you will need the COLOUR command, and also the PENUP/PENDOWN commands, and you may want to use RIGHT and/or BACK, but do not at this stage use MOVEXY, DRAWXY or SETXY etc.). If you are unsure about any of the commands that you need, consult [Programming Essentials](#) [23] (or [Programming Quick Reference](#) [19] to jog your memory).

Save your program, calling it **TGPX1.TGP**, within an appropriately named directory on your computer or the network that you are using (e.g. you could use Windows Explorer to create a “Turtle” directory if you don’t have one already). Then, unless you’ve already laid it out very neatly, you might like to try selecting the “auto-format” option from the [Edit Menu](#) [9]. If this gives a better result, save the neatened version in place of the old one. *Note – it is a good idea to use the “auto-format” option whenever you’ve done a major edit on the structure of your program, because this keeps all the indenting in order, but **always** save the program first in case you don’t like the result for any reason.*

Then choose “New program” from the [File menu](#) before starting the next exercise. (If you like, you could then try reloading the program you’ve saved, just to check out the loading and saving operations). If you need help on any of these operations, or want to find out more about them, see the page on the [File Menu](#) [8].

Exercise 2

Write from scratch a program to draw a face, enclosed in a red circle (radius 100), with small green blots (i.e. filled circles) for eyes, a thick blue triangle for a nose, and a red smiling mouth. The simplest way of making the mouth is to draw a red blot, and then to draw a white blot slightly higher, leaving a crescent of red (but if you do it this way, you may need to think carefully about the order in which you draw the features). You will need to make use of the CIRCLE, BLOT, and THICKNESS commands, in addition to those mentioned earlier, and you can if you wish also use MOVEXY and/or DRAWXY (described in [Programming Essentials](#) [23]). Save your program as **TGPX2.TGP**.

Exercise 3

Now edit the program from the previous exercise, adding a “loop” so that it draws a row of five or more faces across the Canvas. As explained in the [Programming Essentials](#) [23] section, you can do this in either of two ways, using a FOR loop ...

```
PROGRAM tgp3f;  
VAR facesdrawn: integer;  
BEGIN  
  {put statements to position the first face here};  
  for facesdrawn:=1 to 5 do  
    begin  
      {put statements to draw a face here};  
      {put statements to move to the next face position here}  
    end  
  END.
```

or using a REPEAT loop ...

```
PROGRAM tgp3r;  
VAR facesdrawn: integer;  
BEGIN  
  {put statements to position the first face here};  
  facesdrawn:=0;  
  repeat  
    {put statements to draw a face here};  
    {put statements to move to the next face position here};  
    facesdrawn:=facesdrawn+1  
  until facesdrawn=5  
  END.
```

Note that both of these require the “declaration” of a *variable* which is used to count the number of faces that have been drawn – here this variable is called *facesdrawn*, and you should note what form the declaration takes (i.e. the “VAR” line, occurring between the “PROGRAM” line and the “BEGIN” which starts the program proper). For further explanation of variable declaration, see the section on “Program Structure and Declarations” in [Programming Essentials](#) [23] and also the [Introduction to Pascal Syntax](#) [32].

In the case of the FOR loop, the variable *facesdrawn* is given in turn each value between 1 and 5, and for each of these values a face is drawn (so 5 are drawn altogether). In the case of the REPEAT loop, the variable *facesdrawn* is initially given the value 0, and is then incremented by 1 each time a face is drawn (with the command *facesdrawn:=facesdrawn+1*), until it reaches the value 5 when the loop stops (in accordance with the *until facesdrawn=5* statement). Now try editing your original face-drawing program in one of these ways, and when you’ve done it in one way, edit it further so that it works in the other way instead. Save the version with a FOR loop as **TGPX3F.TGP**, and the version with a REPEAT loop as **TGPX3R.TGP**. Having produced these, can you edit one of them so that it produces four rows, each of five faces? (Hint: use a variable called ROWS, which counts the number of rows just as FACESDRAWN counts the number of faces in each row – take a look at the illustrative program called “nestedloops” to see the type of structure that results in the case of FOR loops) Save whatever you manage to produce as **TGPX3.TGP**.

Exercise 4

Write a program using at least one FOR loop and one REPEAT ... UNTIL loop, to create an abstract design of your own choice. Use the command RANDCOL to make it colourful. Save your program as **TGPX4.TGP**.

Exercise 5

In the section on [Programming Essentials](#) [23], the following example program is used to illustrate the use of procedures:

```
PROGRAM squares;
VAR count: integer;

PROCEDURE drawsquare50;
BEGIN
  forward(50);
  right(90);
  forward(50);
  right(90);
  forward(50);
  right(90);
  forward(50);
  right(90);
END;

BEGIN
  for count:=1 to 8 do
  begin
    randcol(6);
    drawsquare50;
    forward(50)
  end
END.
```

Start a new program and type this example in or paste it using the Clipboard (see the [Edit Menu](#) [9]), noting as you do the following points about the program structure, several of which should by now be fairly familiar:

- (a) The program begins with PROGRAM followed by its name and a semicolon;
- (b) A variable *count* is then declared as an INTEGER, and this declaration is followed by a semicolon;
- (c) The program then contains a PROCEDURE called *drawsquare50*;
- (d) Both the procedure, and the main body of the program, are enclosed within BEGIN ... END statements;

- (e) The FOR loop is also enclosed with begin ...end statements, but only because it involves more than one command;
- (f) All of the commands within the program, and within the procedure, are separated by semicolons;
- (g) There is a full stop after the END which finishes the program, but only a semicolon after the END which finishes the procedure.

Having noted all these points, read through the section on “Program Structure and Declarations” in [Programming Essentials](#) [23] to consolidate what you have just learned. Then run the program you’ve entered and check that you understand how it works. Save it under the filename `SQUARES.TGP`.

Now read through the section on “Introducing Simple Value Parameters” in [Procedures and Parameters](#) [34], and adapt the `drawsquare50` procedure to produce a more versatile `drawsquare` procedure which can draw squares of different sizes and in different colours. Adapt the program so that it produces an interesting design of your choice, preferably involving at least one loop (and maybe, if you’re feeling ambitious, *recursion* as explained later on in the Procedures and Parameters section). Save the result as `TGPX5.TGP`.

Exercise 6

Reload the final program you did for exercise 3, and reorganise it so that the statements which draw the face are enclosed within a procedure called FACE. Make sure that it works correctly, and that you fully understand what is going on. Save your revised program as `TGPX6.TGP`. Can you adapt the procedure so that it is able to draw faces of different sizes and/or colours? If you manage this, save the resulting program as `TGPX6P.TGP`.

Exercise 7

Create a program which draws a row of three houses, and which includes BOTH a procedure called HOUSE, which draws one house, and another procedure called WINDOW, which draws a window of a house (and is therefore called by the HOUSE procedure – this means that it must be placed in the program text *before* the HOUSE procedure). The structure of your program should be something like this:

```
PROGRAM tgp7;
VAR counthouses: integer;

PROCEDURE window;
BEGIN
  {put statements to draw a window here}
END;

PROCEDURE house;
VAR countwindows: integer;
BEGIN
  {put statements to draw a house here, calling window}
END;

BEGIN
  {put main program statements here, calling house}
END.
```

(Note that `countwindows` here is a “local” variable inside the `house` procedure, because it is only used within that procedure. For a note on variable *scope*, see the [Procedures and Parameters](#) [34] section, and for a comment on an alternative method of laying out this sort of program (treating `window` as a “child” procedure of `house`), which you are welcome to adopt if you prefer, see the section on procedures in [Programming Essentials](#) [23].)

You may find the POLYGON command useful in this exercise, since it will enable you to display a filled shape rather than just a line drawing.

If you haven’t already done so, now adapt the `window` and `house` procedures so that each of them takes at least one parameter, enabling your program – simply by editing these parameters – to produce different designs of windows and houses (e.g. different sizes or colours, or possibly even more elaborate variations). Save your program as `TGPX7.TGP`.

Exercise 8

Write a program using at least two PROCEDURES, to create an abstract design of your own choice. If you wish, make use of the new commands SETX, SETY, NOUPDATE, and UPDATE. Save your program as **TGPX8.TGP**.

Exercise 9

This exercise asks you to produce a program containing a MOVING ball, which you can do using the following steps:

- (a) Define a variable X to signify the x-coordinate of the ball.
- (b) Define a variable XVEL to signify the velocity of the ball in the x-direction (i.e. the amount that the ball moves in the x-direction for each “cycle” of the program – if XVEL is positive then the ball will move to the right; if XVEL is negative, then the ball will move to the left).
- (c) Set X and XVEL to appropriate initial values.
- (d) Repeatedly:
 - (d1) Draw a white blot;
 - (d2) Move the *turtle* to the position signified by X (use the SETX command for this);
 - (d3) Draw a red blot;
 - (d4) To make the motion appear smooth, add the command sequence *update; noupdate;* at this point.
 - (d5) Add XVEL to X, so as to “move” to a new position.

(Note that if your program only uses a single ball, and if the turtle makes no movements except to draw that ball, you can take advantage of the global variable TURT_X to record the turtle’s position. But for this exercise you should use a variable X as explained above, because this method is more versatile.)

If you need further help, simply copy the following program, run it, and then work out what’s going on – can you see how it produces the effect of a moving ball?

```
PROGRAM tgp9;  
VAR x: integer;  
    xvel: integer;  
BEGIN  
  x:=0;  
  xvel:=2;  
  repeat  
    colour(white);  
    blot(25);  
    setx(x);  
    colour(red);  
    blot(25); {  
      update;  
      noupdate; {for smooth movement}  
    }  
    x:=x+xvel  
  until x>1000  
END.
```

You’ll notice that when this program is run the “ball” appears to flash a lot, because the effect of movement depends on an alternating sequence of red and white blots, with each white blot “rubbing out” the previous red blot in preparation for a new red blot to be drawn in the next position. To eliminate this flashing and give an effect of smooth movement, simply delete the opening curly bracket after the second *blot(25)* statement, which will add both the *update* and the *noupdate* instructions to the program. Now run the program again – this time, each white blot is updated to the Canvas only when the subsequent red blot command has also been executed: hence the blots appear simultaneously, and an impression of relatively smooth movement is given. For more examples of this technique, see the built-in illustrative programs that involve movement.

Having understood all this, see if you can produce more elaborate ball-moving effects for yourself, and save your finished program as **TGPX9.TGP**.

Exercise 10

Now adapt the program which you produced in Exercise 9, to make the ball move in TWO dimensions (hint: you will need variables called “Y” and “YVEL” as well as “X” and “XVEL”, and you might find the new command SETXY helpful). When you have done this, save your program as **TGPX10.TGP**.

Exercise 11

Take the program which you produced in Exercise 10 (or if you were unable to complete that, the program you produced in Exercise 9), and modify it so that the ball “bounces” when it reaches the edge of the Canvas. To do this, you will need to use the IF instruction, which is described in the section on [Programming Essentials](#) [23]. The trick is to change the ball’s velocity at an appropriate point, so that instead of moving right (or whatever), it starts moving left instead. You can do this with a statement such as:

```
if (x<25) or (x>975) then
  xvel:=-xvel
```

which changes the direction of horizontal motion whenever the ball gets within 25 units of the edge of the Canvas (assuming you’re using the standard size Canvas). When you have finished your bouncing ball program, save it as **TGPX11.TGP**.

Exercise 12

Write a program using at least one PROCEDURE, at least one loop, and at least one IF ... THEN ... ELSE structure, to create an abstract design of your own choice, incorporating at least some movement (and you can use the PAUSE command to control the speed of the movement). Try also to make some use of the commands POLYLINE and/or POLYGON (and possibly FORGET and REMEMBER), which as explained in [Programming Essentials](#) [23] are useful for drawing complex shapes, and maybe CANVAS to modify the effect of CIRCLE and BLOT so they draw ellipses rather than circles. If you intend to include a number of moving objects, then you might find it useful to try out the techniques involving reference parameters in the section on [Procedures and Parameters](#) [34]. Save your program as **TGPX12.TGP**.

Having Completed the Exercises

When you’ve worked through all of these exercises, it’s a good idea to read right through the section on [Programming Essentials](#) [23] to check up on anything that you’ve missed or misunderstood. Also look through any of the illustrative programs (available through the [Help Menu](#) [17]) that you may not have seen yet. Finally, having mastered Turtle Graphics Pascal programming, why not move on to the more theoretical aspects of the system, looking “under the bonnet” to see how computers work, starting with the [Introduction to PCode](#) [42]?