Turtle: Innovative Software for the Learning of Computing Concepts

Peter J.R. Millican

The University of Leeds

School of Computing

Submitted in accordance with the requirements for the degree of MSc by Research, February 2004. The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

Contents

Figures and Illustrative Programs

Figures

References to Turtle’s Built-In Illustrative Programs *(excluding Appendix A)*

References to Other Illustrative Programs

Acknowledgements

I am grateful to the School of Computing at the University of Leeds for encouraging me in this project, and in particular, to Professors Graham Birtwistle (now retired) and Roger Boyle for acting as my supervisor and providing useful feedback on the draft of this thesis, as a result of which I believe it is substantially improved. I am also very grateful to Dr Nick Efford, for helpful discussions on the teaching of compiler design, and would like to express my special appreciation of Dr Willi Riha (now retired), who taught me so much about programming at both a high and low level, and inspired my interest in learning the fundamental computer science that informs this thesis.

Many people have also given helpful comments on the system as it evolved, including students and staff at the University of Leeds, but most especially Dr Sarah Kattau, who produced additional teaching materials to accompany the course based on this software, and in 2003 took over its teaching entirely. Her carefulness and attention to detail picked up a number of issues both in the software and its documentation, resulting in significant improvements.

I would like to end by expressing my deepest gratitude to my family for their moral support, especially my wife Pauline who has generously tolerated, when necessary, my spending far more time at the computer than is compatible with giving a fair contribution to family life. Our children David, Katie and Jonathan have also helped by playing with the system, bringing to light various ways of improving its appropriateness for self-learning by young users, independently of any formal tuition.

Abstract

The Turtle Graphics Programming System developed for this MSc ("Turtle" for short) is designed to enable non-specialist students to learn a range of fundamental concepts of programming and Computer Science in a straightforward but fairly rigorous manner, within a user-friendly environment that makes the first steps as easy as possible while providing scope for advanced experimentation. Chapter 1 explains the overall form of the system – why it is based on Turtle Graphics, why it uses Pascal source code, and why it incorporates a virtual machine and a self-contained compiler to translate that source code into Turtle Machine "PCode". The chapter ends with an outline of the remainder of the thesis, and a list of what I take to be the project's main achievements and novel contributions.

Chapter 2 provides a quick tour around the finished system, including its "Visual Compiler" displays that aim to make the Turtle Machine's inner workings transparent. Chapter 3 then explains in detail why its Pascal source language has the commands and structures that it does, bearing in mind the system's primary role as a learning system for introductory programming. Chapter 4 reviews its actual performance in this role, drawing on student feedback and assessments to establish its effectiveness.

Chapters 5 to 7 cover the system's secondary role, of providing a vehicle for the learning of fundamental concepts such as machine code, compilation, dynamic memory management, automata, and stacks. Chapter 5 introduces the specially designed virtual Turtle Machine and its basic operation. Chapter 6 takes this further into procedure calls and dynamic memory management. Then Chapter 7 explains the detailed operation of the Turtle compiler, designed in a modular fashion to provide a relatively accessible way in to compiling and associated techniques (FSM, PDA, recursive descent). In the absence of specific classroom experience to validate the effectiveness of these aspects of the system, Chapters 5 to 7 have all been written in an expository style intended to demonstrate, by example, the Turtle Machine's simplicity of operation and its appropriateness as a teaching tool. Finally Chapter 8 reflects on the system's overall value, suggesting surprisingly strong links between its two main roles, before ending with a retrospective summary of the project's achievements.

Chapter 1 Background, Aims, and Achievements

1.1 The Electives "Market" and its Needs

Beginning students find it notoriously hard to learn basic programming concepts, as testified both by the Computing Education literature (see §4.4 below) and by the many intense and long-running discussions of the issue in schools of Computing. But if this is a major problem with students who are already committed to specialist Computing degrees – and who can therefore be expected to put in serious effort for considerable amounts of time if necessary – then it is obviously even more of a difficulty (albeit less grave in its implications) when the students concerned are on "elective" modules, with no guaranteed formal background, limited time at their disposal, and no need to pass the module in order to proceed with their degree programme.

One possible reaction to this difficulty would be to give up the attempt to teach programming to elective students, but such a reaction would be very regrettable:

- Most non-Computing students will not have encountered programming before entering university, and so may be interested in a "taster" course to see if it appeals sufficiently to merit a change of degree programme (e.g. to include "minor" Computing), or consideration as a possible future career.

- A basic mastery of programming concepts is genuinely useful even for those who have no intention of pursuing a computing career, since the powerful modern software systems that they are likely to encounter in the workplace (notably the ubiquitous "Office" packages) provide considerable scope for customisation and automation using "macros", which can be made vastly more powerful if the user understands, for example, the concept of a loop. The 1999 report from the National Academy of Sciences *Being Fluent with Information Technology* [30] goes even further, arguing on this sort of basis that programming is now "critical to FITness" (section 3.1), though this conclusion is of course controversial (e.g. Urban-Lurain and Weinshank [132] [133]).

- For students of many academic disciplines, there is value in acquiring an understanding of algorithmic thinking quite independent of its potential practical usefulness to them. Students of Philosophy, History and Philosophy of Science, or Cultural Studies, for example, can thus gain an appreciation of a pervasive modern mode of thinking that now profoundly influences our culture in radically novel ways (cf. Dijkstra [39]). This influence is also significant in a vast range of disciplines where algorithmic models are widely used, from social sciences such as Economics, Political Theory and Psychology, to "hard" sciences such as Biology and even fundamental Physics.[1]

- In some disciplines, algorithmic models are sufficiently accessible to provide ways of learning about the domain by active development and experimentation (Shafto [117], Harel and Papert [52], Lippert [76], cf. Guzdial [49]). A simple Turtle Graphics example would be to give a Physics student the task of writing a program to simulate the flight of cannonballs under gravity

So there are strong reasons for wishing to make basic programming concepts available to non-specialist students. The main aim of the software described in this thesis was to provide a way of doing so which could be engaging and "friendly" enough to interest a high proportion of such students, whilst at the same time providing scope for those few who wish to dig deeper to learn some more advanced concepts of computer science (such as recursion, machine code, compilation, and memory models) in a straightforward and approachable, but nevertheless suitably rigorous, context.

1.2 Why Turtle Graphics?

For students like those described above – whose other subjects demand the lion's share of their commitment and time, and with little academic motivation for programming except in so far as they find the work interesting and enjoyable – it would be hard to find a better starting-point than Seymour Papert's idea of "Turtle Graphics". Papert's own

[1] Of course the extent to which such models are *appropriate* is debatable in some of these cases, perhaps most strikingly in Stephen Wolfram's recent attempt to provoke an algorithmic revolution within Physics through his book *A New Type of Science* [139].

famous and seminal discussion [97] provides a detailed and powerful case for this conclusion on general educational grounds, based in part on Jean Piaget's insights into the theory of learning; here I shall be much briefer, and will not endorse Papert's more ambitious claims about the value of programming for the general development of transferable problem-solving skills (cf. Pea [100], Mayer et al. [80], Kurland et al. [70], Palumbo [94], Soloway [124], Urban-Lurain and Weinshank [132], [133]).[2] But I will add some additional considerations that bear particularly on the context of teaching programming concepts to elective students in a university.

Piaget's theory of human intellectual development, as presented in works such as *The Origin of Intelligence in the Child* [103] and *The Child's Construction of Reality* [104], has had a huge impact on modern educational practice, and also carries a number of potentially significant implications for the learning of programming in particular. His work has given rise to the educational theory of "constructivism", whose central theme is that knowledge must be constructed by the learner rather than taught by authorities or read passively from the world.[3] This theme has sometimes been taken to epistemological extremes, as in idealist or postmodern rejections of the very possibility of objective knowledge,[4] but its dominant educational manifestation has been an emphasis on *student-centred* learning. Boyle [15] usefully summarises the implications under five headings, in the context of a discussion of learning environments for computing. The first three headings are particularly relevant here:[5]

---

[2] But as remarked earlier, programming can have wider educational value quite independently of these concerns, where it is used as a vehicle for facilitating learning about other domains.

[3] For a wide range of essays on constructivism and education assembled by the Maryland Collaborative for Teacher Preparation, see www.towson.edu/csme/mctp/Essays.html.

[4] Ben-Ari [12] outlines how an educational paradigm can impact on attitudes to ontology and epistemology, as well as methodology and pedagogy. He then goes on to suggest how a more sober constructivism can bring positive benefits to computer science education.

[5] The last two are "Experience *WITH* the knowledge construction process" (i.e. learning how to learn), and "Metacognition" (i.e. reflecting on one's learning to generate more effective learning strategies). Both of these involve higher-level educational goals which do not apply in the early stages.

(a) Authentic Learning Tasks: "… learning tasks should be embedded in problem solving tasks that are relevant [to the learner]".

(b) Interaction: "interaction is … the primary source material for the cognitive constructions that people build to make sense of the world". One particularly influential approach, ultimately deriving from Vygotsky [134], is Cognitive Apprenticeship theory, which "emphasises the active role of the teacher in supporting the learner. … The teacher first provides a model of expert performance in the task [and] actively coaches the learner in acquiring the target skills and knowledge … then gradually removes this support forcing the learner to become increasingly independent." (Such a removable support is commonly referred to as *scaffolding* within the educational literature.)

(c) Encourage Voice and Ownership in the Learning Process: "students should be allowed to choose the problems they will work on [and] the teacher should serve as a consultant to help students to generate problems which are relevant and interesting to them".

Turtle Graphics is an excellent exemplar of this paradigm, on all three criteria.[6] First, it replaces traditional number- or text-based programming exercises with creative graphical design, something which is interesting and motivating to a far higher proportion of students. Secondly, it provides a good vehicle for interactive learning, precisely because the tasks it involves are ones that can easily be appreciated and discussed; moreover a Turtle Graphics system – with a simplified environment and built-in primitives designed to facilitate the creation of graphics – can itself serve as a "scaffold" to develop the skills needed for more advanced systems. Finally, the creativity implied by graphical design positively invites students to generate their own ideas: our human visual imagination ensures that there is no difficulty whatever in having an entire class all working on their own individual problems.

---

[6] Along with Turtle Graphics, perhaps the best-known "microworld" approach to introductory programming is that of *Karel the Robot* (Pattis [99]), which in particular gives an excellent basis for learning about procedural abstraction. But Turtle Graphics scores better on these three criteria, in part because it delivers an obvious graphical *product* rather than being focused primarily on a *task*.

In addition to these general constructivist considerations, there were a number of more specific reasons for favouring Turtle Graphics in the context of this project:

- Turtle Graphics is simple and intuitive and hence easy to grasp, even for young children, because it is based on a familiar type of metaphor (cf. Mayer [79], Brusilovsky et al. [19], Pane and Myers [95] section 5.1). A "turtle" (which need not be represented visually) moves around the screen, drawing as it goes, and following straightforward English instructions such as "forward", "left", "circle", and "colour". This metaphor is so straightforward as virtually to guarantee that students will have no difficulty in understanding it.
- No doubt for obvious evolutionary reasons, humans find visual comprehension far easier and more natural than (for example) mathematical, as testified by the frequent use of visual language to describe our grasp even of abstract ideas. With Turtle Graphics one can *literally* "see" the program working through its stages, especially if it is possible to pause the processing at intermediate steps.[7]
- If a Turtle Graphics system makes provision for loops, then it is fairly easy to learn how to use it to produce attractive patterns far more intricate than would be feasible by hand. Hence such a system can appear "useful" even to students who have no interest in computation or information processing as such.
- Although very straightforward, Turtle Graphics is not intrinsically limited. Both novices and more advanced students can find a challenge within such a system, because patterns of arbitrary complexity can be generated with more or less the same simple tools as are used by beginners.
- When a Turtle Graphics program goes wrong, it is relatively easy to see *where* it has gone wrong and *in what way*, thus greatly easing the pain of debugging.
- Likewise in Turtle Graphics it is quick and straightforward to test "what if" scenarios, changing the program and identifying how the output has changed as a result (McConnell [82] pp. 52-3 stresses the value of such "active learning").

---

[7] The virtues of using graphics to mediate the learning of programming are also stressed, with supporting references and a brief survey, by Cooper, Dan et al. [31], [32], though Naps et al. [88] argue that such benefits do not necessarily extend to visualisation of more abstract algorithms.

- Turtle Graphics gives ample scope for personal creativity, one of the greatest satisfactions to be derived from programming, though largely closed off from beginners in typical "training" situations (e.g. where they are set a fixed information processing task). Not only is this enjoyable, but there is evidence that "Arts" students in particular benefit educationally from the opportunity of learning in a creative, "divergent" manner, rather than in the "linear" manner which tends to characterise traditional data- or mathematically-oriented teaching of programming (Hartley and Greggs [53], Nulty and Barrett [91]).
- Partly because Turtle Graphics lends itself so well to divergent, creative development, it is relatively easy to structure a course around it that enables learners to work at their own speed (cf. Liffick and Aiken [74], Moser [85]) and which stresses "discovery" learning (cf. Baldwin [9]).
- Assessment of Turtle Graphics programs can take advantage of the marker's intuitive familiarity with visual patterns, and consequent ability for easy recognition of the program's behaviour and results. One consequence of this is that creative graphics coursework not only greatly reduces the frequency of plagiarism; it can also make plagiarism relatively simple to detect.

For all these reasons, I decided that my intended introductory programming module should start with Turtle Graphics. But this still left open the difficult (and in other contexts notoriously controversial) choice of programming language and "target" development environment.

1.3 Why Turtle Graphics *Pascal*?

Turtle Graphics was developed by Papert in tandem with LOGO, his programming language for children. This is in many ways an excellent and versatile language, belonging to the LISP family, but due in part to its childish associations (and the fact that in schools it is seldom exploited to anything like its potential), it suffers from low esteem and "credibility". This lack of credibility has presumably both contributed to, and been exacerbated by, the language's virtual invisibility in the commercial world.

Although the choice of language in a programming module designed for elective students perhaps need not be so influenced by these sorts of concerns as in a module for prospective computing professionals, there is little doubt that an introduction to programming based on LOGO – whatever its educational merits – would have difficulty attracting sufficient recruits to be viable. An elective programming module must balance a number of factors, some of which tend to pull in opposite directions:

(a) Ease of learning by students who cannot be presumed to have a strong commitment to mastering programming for career or academic progression.

(b) Quick results – early positive feedback so that right from the start a student can get the satisfaction of visible progress (again a need which is greatly amplified by the lack of presumed career commitment from these students).

(c) Minimisation of "debug frustration", the common experience of beginning students struggling with multiple syntax errors or manifestly incorrect output.

(d) Suitability for appropriate learning of both fundamental concepts and software disciplines (to pave the way for any students who ultimately decide to continue with Computing through, say, a major/minor programme).

(e) Perceived "usefulness", so that even those students who do no more programming beyond this introductory module will feel that they have learned something of practical value to them.

Obviously it is (very) arguable which programming language and environment best satisfy these sorts of criteria (see for example Jarc [60], Brilliant and Wiseman [17], Cantù [25], Trott [130], Hadjerroult [50], Warford [135], Callear [22], Smyth [123], Burton and Bruhn [21], and Jenkins [63]), but at the time of developing the module I took the view that the optimal combination was to teach Pascal with a view to Borland's *Delphi* development environment for Windows.[8]

---

[8] There would now be good reason to consider an alternative solution based on Java syntax, given that language's commercial value and general prominence in an Internet-aware culture, its capacity to provide immediate perceived "usefulness" through Web applets, and the (free) availability of highly-regarded environments such as *BlueJ* which themselves exploit graphics and visualisation to facilitate further learning. See Appendix B for some discussion of possible future developments in this direction.

Briefly, the reasons for this decision were as follows. First, it is widely accepted that Pascal's syntax is significantly easier for beginners to read and understand than that of C++ or Java (probably in part because its control structures are more verbose, cf. Sime et al. [118]); moreover I had nearly a decade's experience of teaching Turbo/Borland Pascal, which gave me a high opinion of the system's learnability and robustness, notably error messages that are relatively unconfusing (criteria a and c). Secondly, as regards rigour and appropriateness for software engineering (criterion d), Pascal seemed preferable to Visual Basic, which was another obvious contender on grounds of initial simplicity and "credibility" (moreover Green et al. [46] found that syntactically, Pascal is easier than Basic for beginners to parse in what they term the "parsing-gnisrap" cycle where new "chunks" are inserted into existing code). Visual Basic would also have implied commitment to a specific proprietary platform (notably Microsoft Windows and its variants), and although Windows was indeed the obvious default platform to use given the target audience of the module (on grounds of familiarity, availability, ease, and perceived "usefulness" to them – criteria a and e), it was a significant virtue of *Delphi* that plans were afoot to port it to Unix, soon to result in the *Kylix* system (since 2001).

Choice of *Delphi*, however, left the problem of initial learning and the need for quick results (criterion b). *Delphi* is a complex environment, and I remember myself finding the initial sight of it somewhat daunting, despite all my prior programming experience. Even provision of a "Turtle Graphics" unit within *Delphi* would not help much to overcome the problem that attempting to initiate novice students *both* into the language *and* the environment at the same time might well prove overwhelming.[9] So the ideal approach seemed to be to separate their initial learning of the Pascal language from their learning of the *Delphi* environment. Hence the need for a self-standing environment that could support an introduction to Pascal with an emphasis on Turtle Graphics, but with sufficient sophistication that it could prepare the students for a

[9] I was confident, however, that using Turtle Graphics as a bridge into the Pascal language itself would work smoothly, since I had personal experience of using a simple custom-written "unit" in this way when teaching Turbo Pascal to elective students in 1994/95.

relatively easy initiation into *Delphi*, and also preferably enable the more proficient students to "stretch themselves" even prior to this initiation.

1.4 Why an Integrated Visual Compiler?

The idea of using Turtle Graphics to introduce algorithmic thinking to novice programmers has proved extremely popular over recent years, and there are any number of such systems to be found on the Web and referenced in the Computer Science teaching literature. Dedicated environments, however, tend to be almost exclusively based around the language LOGO,[10] because Turtle Graphics systems focused on the learning of other languages (such as C++, *Delphi*, Java, or Visual Basic) are usually provided as mere "units" or "classes" within a standard programming environment.[11] The closest approximation I have found to the sort of system envisaged in this project, which became available recently (albeit only in a beta version that lacks a user manual and most of the planned "lessons"), is Otherwise Software's *Jurtle* for Java.[12] Though a commercial system, this is evidently targeted for the educational market, being user-friendly, reasonably priced, and also provided in an unlicenced version that allows temporary trial of the product.

---

[10] See for example the *LOGO Foundation* website at http://el.media.mit.edu/logo-foundation/logo/.

[11] A Web search can very quickly find, for example the *GrWin Graphics Library* for Fortran and C/C++; the turtle classes in Donovan [40] for C++; and the many Turtle Graphics Windows components on various *Delphi* and Visual Basic freeware and shareware sites. Turtle Graphics is now enjoying something of a resurgence in the teaching of introductory Java, as for example in Martin [77], Caspersen and Christensen [26], Schaub [116], Slack [120], and Ariga and Tsuiki [7]. This emphasis on early graphics is presumably fostered by Java's association with Web applets, but unfortunately its standard stateless "redraw every time" mechanism imposes limitations on a turtle-style system. Roberts and Picard [109] and Bruce et al. [18] have therefore developed alternative approaches which, though somewhat inspired by Turtle Graphics, are based instead around the idea of persisting graphical objects with their own internal state.

[12] See www.otherwise.com/Jurtle.html. Another system which may be of a similar type (since it is described as "a graphics environment" but requires an external compiler) is Sparkling Light Software's

Jurtle requires prior installation of the standard Sun JDK (Java Development Kit), and integrates nicely with it: once the location of the Java compiler has been set within *Jurtle*, it hands over control seamlessly when necessary, and displays the appropriate compiler messages within the programming environment. The problem is that these messages are not designed to be suitable for a novice, so as soon as things start to go wrong, such a user is likely to feel overwhelmed and intimidated. Here, as a very mild example, is the message given when the word "forward" is misspelt within "BoxTurtle", one of the simplest illustrative programs provided:

```
C:\Program Files\Jurtle\Examples\BoxTurtle.java:19: cannot resolve symbol
symbol  : method forwad (int)
location: class BoxTurtle
        forwad( size.height - 40 );
        ^
1 error
```

Far worse, however, is what happens if the user happens to omit the "{" which follows the line "public void runTurtle()" within this same program. Not only is there no correct diagnosis of the error (the first error message given refers to a missing semicolon rather than a missing brace), but also, the messages run to well over 100 lines, with no fewer than 38 errors identified in what is a very short program!

This particular type of problem would not have been quite so serious in a system based on the *Delphi* compiler, which generally gives very helpful error messages, but even if it had been possible to overcome the accompanying complexity of the *Delphi* context (cf. §1.3 above), the same fundamental issue would remain, that messages appropriate to the expert programmer – and therefore typically generated by any "industrial strength" standalone compiler – will often be unsuitable or unhelpful for the novice. It was clear, therefore, that the envisaged Turtle Graphics Programming System would have to provide its own syntactic error handling.

Turtlebox system for C++, though given the characteristics of that language and its compilers, this seems less likely to be suitable for novice programmers.

The very first version of the *Turtle* system, which was piloted in a self-standing three-week "option" course that made no claim to lead on to higher things, did not incorporate a compiler, but made do instead with a relatively superficial interpreter. This proved sufficiently successful (e.g. in respect of student results and feedback) to give confidence in the overall approach, but the lack of full syntax analysis somewhat restricted the complexity of programming constructs that were permissible, and made it extremely difficult to generate *appropriate* error messages for all syntactic failures. Moreover the system made no provision for procedures (an absolute essential if it was to act as a pathway into *Delphi*) or, consequently, for recursion, greatly limiting both its scope for genuinely stimulating experimentation and also its potential as an introduction to any programming concepts beyond the most basic.

Having established that a fundamental redesign of the pilot system's internal operation would be required for further progress, it seemed an obvious next step to make a virtue of necessity, and develop the system not only as an introduction to the craft of programming, but also to the concepts of programming language systems such as compilation, machine code, and dynamic memory management. Having come through the route of assembler programming myself, I find it regrettable (albeit very understandable) that as the realm of applications has expanded, this fundamental aspect of the discipline has tended increasingly to be pushed out of modern syllabuses, so that even many specialist programming students today remain virtually ignorant of the underlying mechanisms (with consequent detriment to their understanding of other areas, for example data structures and complexity theory). Hence my decision to develop this system around a virtual "Turtle Machine" with its own "machine code" into which the user's Pascal programs would be compiled, and giving visual access to the results of that compilation, both static and dynamic.

This decision made Pascal a particularly good choice of language in yet another respect, since Pascal was designed in part with ease of compilation in mind (e.g. Jensen and Wirth [65], p. 8), making it feasible to develop a compiler whose operations would be accessible even to non-specialist students. Unfortunately, however, the original type of Pascal "P-code" developed by Niklaus Wirth, the designer of the Pascal language, was too complex to provide a plausible basis for such teaching (see for example Wichmann [136], Bell and Wichmann [11], and Pemberton and Daniels [101] ch. 10).

So it was necessary to design and develop a new type of Turtle Machine, with as simple a design as possible, yet adequate to support a wide enough range of programming constructs to meet the various educational aims outlined above.

1.5 An Introduction to *Computing* Concepts

The upshot of all this was that the *Turtle* system ended up as something rather broader than originally envisaged. Having begun life as a vehicle for introducing only basic *programming* concepts, it became also a potential aid to exploration of the broader context of programming languages within *Computer Science*, and a means of developing a practical understanding of a wide range of relevant concepts such as:

- high- and low-level languages, machine and assembler code;
- the concept of a stack, and its value in computer architecture;
- subroutine calling, return addresses, "stack frames" etc.;
- heap structures and dynamic variable management;
- parameter passing mechanisms, "call by reference" and "call by value";
- finite state machines and pushdown automata, and their value in parsing;
- recursion, and its application in recursive descent parsing.

Many of these are particular hard to grasp in the abstract, but perhaps their concrete and transparent use within the Turtle Machine may prove of value in helping students to learn them.[13] The system has not yet been used for teaching at this level, but I hope before long to develop materials to support such use.

[13] It is a common observation, reinforced by Piaget, that concrete understanding develops far earlier in life than abstract, and is typically far easier to acquire at any stage of development. Hazzan [55], for example, drawing on parallels with research in mathematics education, gives a range of examples where students are assisted in learning computer science concepts by "reducing abstraction". Hence it is not unrealistic to expect that studying a concrete realisation of such concepts would make them easier to learn, even taking into account the overhead of understanding the concrete model.

1.6 Outline of the Thesis

The remainder of the thesis is structured as follows. Chapter 2 provides a very quick tour of the *Turtle* system from the user's point of view. Then Chapters 3 and 4 together cover use of the system for introducing programming to novices, with the former focusing on the choice of programming language structures and commands, and the latter on practical experience of three years' teaching using the system (or its earlier prototype). Chapter 4's conclusion effectively sums up the first half of the thesis, reflecting on the overall value of the *Turtle* system in the learning and teaching of introductory programming.

Chapters 5 to 7 then turn from introductory programming to the Turtle virtual machine, its "PCode" (virtual machine code), and its compiler. Chapter 5 explains the general design of the Turtle Machine, and the way in which Turtle Graphics commands and simple Pascal control structures can be represented within PCode. Chapter 6 builds on this to tackle the far more complex subject of procedures, digging deeper into the Turtle Machine to show how Pascal procedures can be handled by it, including recursion and parameter calling methods. Finally, Chapter 7 explains the workings of the *Turtle* compiler, which translates the user's Pascal source code into equivalently functioning *Turtle* PCode, in conformity with the design explained in the previous two chapters.

It is important to note that Chapters 5 to 7 are deliberately written in a more discursive manner than might be expected in a typical thesis, because their role is not only to explain the system's design, but also, crucially, to *illustrate* how that design can be put across in a way that would be relatively accessible to a non-expert in Computer Science. These aspects of the *Turtle* system have not yet been used in practice to teach the concepts of machine code or compilation, but the system is intended to provide a basis for doing so. In the absence of such road-testing, the best way of demonstrating the system's potential for the task is to manifest by example how it can indeed serve as a vehicle for explanation at roughly the appropriate level.[14]

---

[14] However if used for teaching the system would be supplemented by additional materials such as definitions of the relevant abstract machines and data structures, which are clearly not necessary here.

Chapter 8 concludes the thesis, starting by drawing together the threads of the previous chapters to reflect on the system's value in the learning and teaching of Computer Science (complementing the exclusive emphasis on novice programming which was the theme of Chapter 4's conclusion). It ends by summing up the project's achievements, with reference to §1.7 below.

Appendix A contains the lecture plans and coursework used when teaching the *Turtle* system in Leeds University, Appendix B collects together some ideas for future development, and Appendix C contains the entire contents of the system's online Help file, with independent page numbering (and cross-references to those page numbers replacing the online hyperlinks to facilitate use in printed form).

The *Turtle* system itself, which includes both the software and the online Help file, is available from www.leeds.ac.uk/jcom/turtle/.

1.7 Achievements and Novel Contributions

My initial aim in this project was to produce a tool for a job, namely, to enhance the teaching of introductory programming, and I had no particular focus on creating something novel for its own sake. However both my studies at the time and subsequent reviews of the literature have confirmed that the *Turtle* system does indeed represent a genuine innovation, not so much in the technologies that it builds on, but rather, in the way that these technologies are combined and presented. The main achievements and novel contributions represented by this work are, I believe, as follows:

(A1) To provide a strong case for the development of an integrated programming environment for novices (and non-Computing students in particular), combining standard programming structures, simple graphics primitives, comprehensive help facilities, and a standalone user-friendly compiler. *(Case presented in §1.1 – §1.4, supplemented in §4.4)*

(A2) To select a suitable subset of the Pascal language, combining overall simplicity with considerable graphical and algorithmic power, suitable to pave the way for graduation to *Delphi*. *(Discussion in Chapter 3)*

(A3) To design, program and progressively enhance a powerful, robust, and attractive programming environment, conforming to the requirements implied by (A1) and (A2), a system which has proved in practice its suitability for the innovative and effective teaching of novices. *(System illustrated in Chapter 2, and evidence of its practical value given in Chapter 4)*

(A4) To design a virtual Turtle Machine capable of supporting the language commands and structures implied by (A2) – including dynamic memory management, dual parameter call methods, and recursion – and yet sufficiently simple to be potentially comprehensible to a non-specialist student after a relatively modest investment of effort. *(Virtual machine described in Chapters 5 and 6, which also aim to demonstrate by example how its behaviour can be explained at a relatively simple level)*

(A5) To design a compiler from the Pascal source code (A2) to the Turtle Machine's PCode (A4), operating by methods that are sufficiently simple to be explained without presupposing any significant prior knowledge of formal language or machine theory, and sufficiently general that they can provide a practical introduction into the wider uses of abstract machines and compilation techniques. *(Compiler described in Chapter 7, its operation divided into three levels each of which is sufficiently manageable that its overall behaviour can be summarised in a single one-page diagram, and which serve respectively to introduce finite state machines, pushdown automata, and recursive descent.)*

(A6) To incorporate into the programming environment a visual interface to the operations of the virtual machine and compiler, to enhance the system's value as an introduction to Computing concepts. *(Visual Compiler displays introduced in §2.3, and also illustrated in §§7.3-5; overall value of the system in this capacity discussed in Chapter 8)*

Chapter 2 External Design

The external design of the Turtle Graphics Programming System can most efficiently be conveyed by means of a quick "guided tour" through its main facilities, so this chapter will accordingly be devoted to such a tour. For simplicity, it is written as addressed to a user who has a computer running with the *Turtle* system installed.

2.1 Getting Started

To start up *Turtle*, navigate to the appropriate directory (here presumed to be `C:\Turtle`) and run the program `turtle.exe`. To load an illustrative program, go to the Help menu and move the mouse down to the "Illustrative programs" item; then click on the first program in the list, named "Simple drawing with pauses". When the program appears, click on the "RUN" button just to the right of the menu:

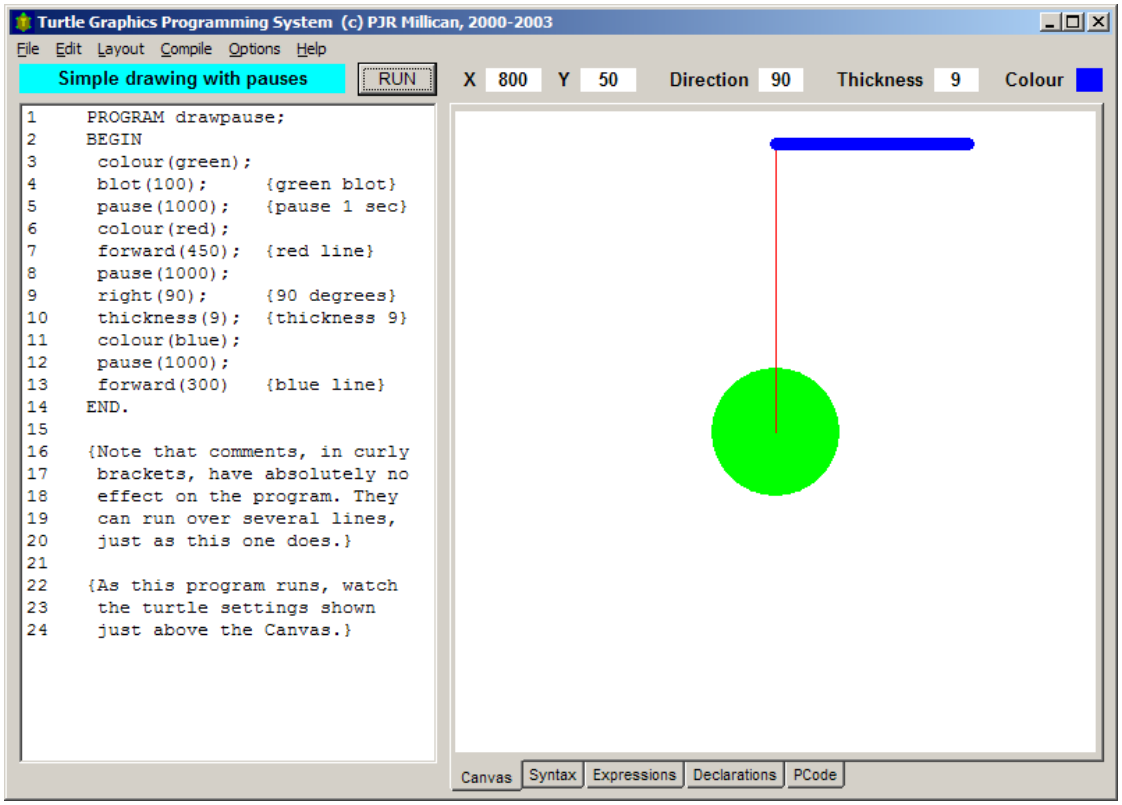


Figure 1: *Turtle* running the "drawpause" illustrative program

As shown in the picture above, the left-hand section of the screen is the Programming Area which incorporates a simple editor, while the right-hand section is the "Canvas" on

which graphics are created by the drawing "statements" in the program.[15] The syntax of the program is essentially that of Pascal, except that the system includes built-in drawing procedures, some of which are common to virtually all Turtle Graphics implementations (e.g. "forward", "right"), though others are specially provided to take advantage of this system's particular facilities (e.g. "blot", "pause").[16] The Help file contains a detailed walk-through of this particular program in its third section, entitled "The Program"; this can be consulted if further details of the syntax or the relevant instructions are required.

2.2 The System Menus

Again the Help file contains a full description of the various system menus (in the six sections entitled "File menu", "Edit menu", "Layout menu", "Compile menu", "Options menu", and "Help menu"), and there is no need to discuss them in detail here. However the following summary may prove helpful for drawing attention to some of the system's more distinctive aspects:

- The **File menu** provides typical Windows-style facilities for clearing, loading, and saving the program (standardly as a text file with extension `tgp`), and for transferring images drawn on the Canvas (as bitmaps) either to the Windows Clipboard or to a disk file.

---

[15] Pascal commands are formally called "statements", a convention mostly followed here. The words "instruction" and "command" will generally be reserved for referring to PCode; where the distinction matters, "instruction" is used to mean a PCode (or Turtle Pascal) primitive, and "command" a particular instance of an instruction together with its arguments.

[16] The original Turtle Graphics philosophy, as exemplified by Papert's discussion [97] mentioned earlier in §1.2, is to include as few primitives as possible, encouraging students to build up for themselves more complex instructions (e.g. "circle") as procedures based on the simple primitives such as "forward" and "right" (this approach is also shared by *Karel the Robot* [99]). The current system departs from this philosophy, in order to provide students with more immediately satisfying feedback from their programming efforts (e.g. by simplifying the construction of complex patterns, and increasing the efficiency of circle and "blot" drawing to facilitate reasonably fast real-time movement). See §3.4.1 below for more on this.

- The **Edit menu** again largely follows the typical Windows pattern, providing an "undo"/"redo" facility (which remembers up to 100 steps) and four simple operations involving the Windows Clipboard (which can therefore be used for transferring text into and out of the program editor, e.g. from the Help system or into a text file). Program line indentation controls are also provided, including most notably an "auto-formatter", which can if desired impose standard capitalisation and appropriate indentation on any legal program (and is thus particularly useful for a teaching context, to encourage students to keep their program code neatly structured).
- The **Layout menu** provides two possible choices for the Canvas dimensions (especially useful for beginning students who find their program drawing beyond the standard boundaries, but have not yet encountered the "canvas" instruction). It also gives various options for screen layout and font choice, which can be particularly helpful if the system is being run on the minimum 800x600 resolution screen.
- The **Compile menu** gives access to the various "machine code" and memory analysis facilities described below in §2.3, and also provides a method of running or halting the program independently of the "RUN" button.
- Most of the settings in the **Options 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. The various configuration settings (which prove particularly valuable when processing large quantities of student assessments) can also be saved for future use, or set as defaults.
- The **Help menu** provides direct access to various sections of the Help file, and also to a number of illustrative programs that are built into the system.

2.3 The Visual Compiler Displays

To see the Visual Compiler at work, first use the Help menu to select the built-in illustrative "triangles" program (labelled "Recursion" in the menu), and click on "RUN". The program will generate the recursive pattern shown here (with eight levels of triangles of successively halved dimensions), but note that while it executes the "RUN" button changes to "HALT": you can click on this at any time to terminate it manually.

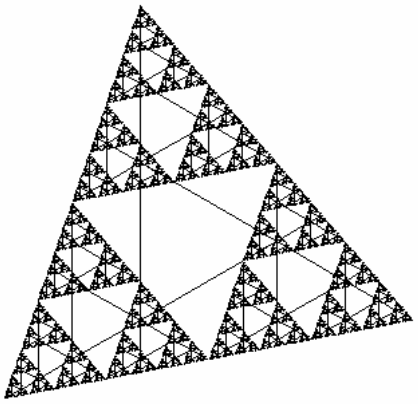


Figure 2: Recursive triangles

Manual termination is particularly important if the trace facility is operative; to enable this, select "Trace on run" from the Compile menu, which automatically selects also the "Analysis tables" setting, making visible a set of "tab controls" along the bottom of the Canvas. Now again click on "RUN", then "HALT" the program after a few seconds, and click on the tab control labelled "PCode":

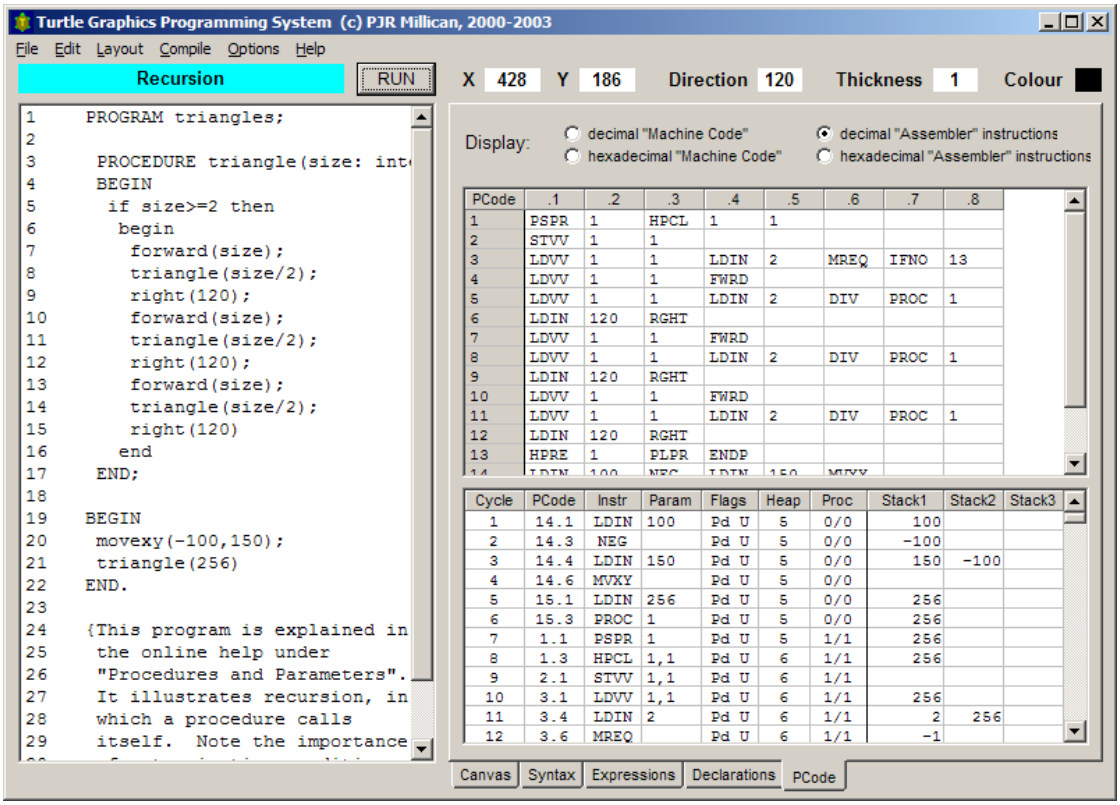
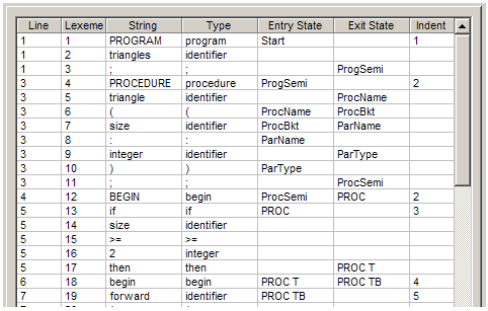


Figure 3: PCode and Trace displays as the "triangles" program executes

The "PCode" table here shows the "assembler code" into which the triangles program has been compiled; the instructions can also be shown as purely numeric "machine code" by clicking on one of the left-hand radio buttons above the table. The lower table is the trace display, which will appear only if the trace facility is operative. This shows the assembler commands that have actually been executed while the program was running, together with their parameters. It also shows the momentary state, just prior to each command's execution, of the various program "flags", the Heap Top Pointer (showing the extent of variable storage) and Procedure Register (showing which procedure is in progress, and how many procedures are active), and also the resulting state – immediately *after* the command has been performed – of the top three locations on the Program Stack (which stores the parameters of the various instructions and operators as these are executed, cf. §5.1 and note 47 in §6.7).

Three other displays are also provided through the bottom tab controls:

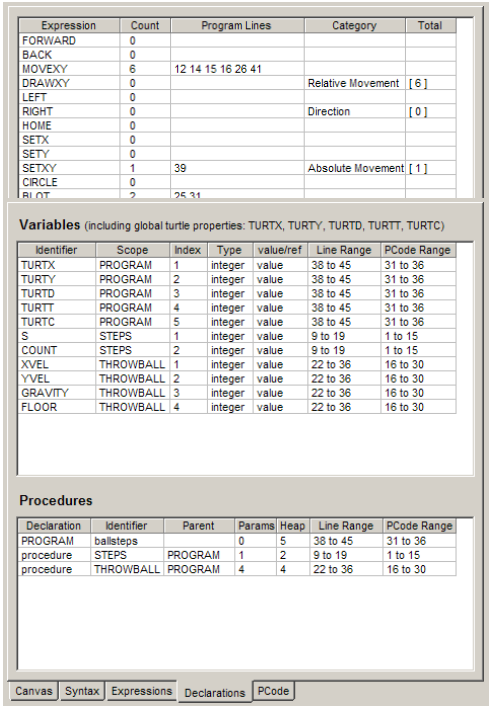
- A syntax analysis, including details of lexical divisions and also the finite state machine and pushdown automaton used to parse the Pascal program (cf. §§7.3-5), together with the corresponding indentation calculations for the Edit menu's auto-indent facility;



Line	Lexeme	String	Type	Entry State	Exit State	Indent
1	1	PROGRAM	program	Start		1
1	2	triangles	identifier			
1	3	;	;		ProgSemi	
3	4	PROCEDURE	procedure	ProgSemi		2
3	5	triangle	identifier		ProcName	
3	6	(	(	ProcName	ProcBkt	
3	7	size	identifier	ProcBkt	ParName	
3	8	:	:	ParName		
3	9	integer	identifier		ParType	
3	10	)	)	ParType		
3	11	;	;		ProcSemi	
4	12	BEGIN	begin	ProcSemi	PROC	2
5	13	if	if	PROC		3
5	14	size	identifier			
5	15	>=	>=			
5	16	2	integer			
5	17	then	then		PROC T	
6	18	begin	begin	PROC T	PROC TB	4
7	19	forward	identifier	PROC TB		5

Figure 4: Syntax Analysis Display

- An "expressions" analysis, tallying the various types of statement that have been used, grouped together by category and referenced by line number – this is particularly useful for marking student assignments that may require them to demonstrate competence with an appropriate range of commands (as with the coursework in Appendix A);
- A list of "declarations", showing both



Expression	Count	Program Lines	Category	Total
FORWARD	0			
BACK	0			
MOVEXY	6	12 14 15 16 26 41		
DRAWXY	0		Relative Movement	[6]
LEFT	0			
RIGHT	0		Direction	[0]
HOME	0			
SETX	0			
SETY	0			
SETXY	1	39	Absolute Movement	[1]
CIRCLE	0			
PLOT	2	25 31		

Variables (including global turtle properties: TURTX, TURTY, TURTD, TURTT, TURTC)

Identifier	Scope	Index	Type	value/ref	Line Range	PCode Range
TURTX	PROGRAM	1	integer	value	38 to 45	31 to 36
TURTY	PROGRAM	2	integer	value	38 to 45	31 to 36
TURTD	PROGRAM	3	integer	value	38 to 45	31 to 36
TURTT	PROGRAM	4	integer	value	38 to 45	31 to 36
TURTC	PROGRAM	5	integer	value	38 to 45	31 to 36
S	STEPS	1	integer	value	9 to 19	1 to 15
COUNT	STEPS	2	integer	value	9 to 19	1 to 15
XVEL	THROWBALL	1	integer	value	22 to 36	16 to 30
YVEL	THROWBALL	2	integer	value	22 to 36	16 to 30
GRAVITY	THROWBALL	3	integer	value	22 to 36	16 to 30
FLOOR	THROWBALL	4	integer	value	22 to 36	16 to 30

Procedures

Declaration	Identifier	Parent	Params	Heap	Line Range	PCode Range
PROGRAM	ballsteps		0	5	38 to 45	31 to 36
procedure	STEPS	PROGRAM	1	2	9 to 19	1 to 15
procedure	THROWBALL	PROGRAM	4	4	22 to 36	16 to 30

Canvas | Syntax | Expressions | Declarations | PCode

Figure 6: Declarations Display

> the variables and the procedures that have been defined within the program, together with their scope and the corresponding ranges of Pascal and PCode instructions. For each variable, the type and calling method are also shown, and for each procedure, the parent routine, parameter count and heap storage requirements.[17]

For a discussion of the design of the virtual "Turtle Machine" and its "PCode", see Chapters 5 and 6 below. For details on the operation of the compiler, see Chapter 7.

We now proceed to explore the system's use as a vehicle for the learning and teaching of introductory programming, focusing first (in Chapter 3) on its Turtle Graphics Pascal source language, and then moving on (in Chapter 4) to see how it has performed in practice within taught modules at the University of Leeds.

---

[17] In Figure 6, like Figure 5, the "ballsteps" illustrative program is used, since this gives a more interesting display than the "triangles" program used in Figure 4.

Chapter 3 The Turtle Graphics Pascal Source Language

The idea of starting students on a simplified subset of a programming language is very familiar from the educational literature.[18] Sometimes the emphasis here is on self-contained "mini-languages" such as that used by *Karel the Robot* [99] (see also the discussion by Brusilovsky et al. [19]), and sometimes on progressive introduction of more structures within a standard language, extending the relevant subset as the students develop (e.g. Brusilovsky et al. [20], DePasquale [36]). However given *Turtle*'s first intended role as an introduction to programming concepts for total beginners, designed to lead on to the far more sophisticated *Delphi* environment after only four lectures, it was clearly sensible to choose a subset of Pascal (rather than any specially invented language), and to fix that subset with initial learning in mind rather than to attempt to cater for the students' later development. Far better that they should achieve genuine competence (and confidence) with a small number of types, structures, commands etc., than that they should run any risk of being confused at this early stage by the niceties of distinctions in meaning and usage between a variety of syntactic options.[19]

The same point applies to some extent to the system's role as a vehicle for introducing more advanced concepts of compilation. There would be little point, from this perspective, in providing a variety of control structures that are compiled in much the same way (e.g. "repeat" *and* "while"), since this would add complexity without widening the conceptual repertoire. More potential benefit would come from adding a greater range of types and data structures (e.g. reals, strings, enumerations, sets, arrays, records, objects), methods of access (e.g. pointers), and subroutines (e.g. functions, methods, units), since these would involve significant extensions to the machine

---

[18] In addition to the references in the text, see for example du Boulay [41], du Boulay et al. [42], and Motil and Epstein [86].

[19] Lewis and Olson [72] identify abundance of low-level primitives as a major cognitive barrier to programming, while Eisenberg et al. [43] found confusions arising particularly where novices were presented with a variety of syntactic structures to achieve a similar effect.

processes and compiler techniques that the system could be used to illustrate. However such extensions would have taken more time than was available in this project, and their implementation is anyway perhaps best postponed until practical experience has been gained of the system's use in teaching computing concepts and relatively basic compilation, to enable more reliable judgements to be made as to whether additional system complexity is likely to carry too great a cost in comprehensibility and accessibility.

3.1 Data Types

3.1.1 Numbers

Any Turtle Graphics system must make use of numbers, most obviously to specify lengths or angles, and here integers are indispensable. They are simpler to handle than real numbers, more efficient both in respect of storage and processing speed, but also they are *precise*, enabling conditional tests for equality to be performed on them without having to worry about rounding inaccuracies. Related to this is the fact that integers can be divided precisely into bytes, and thus conveniently used for representing a wide range of colours in the standard "24-bit" manner, as explained in §3.4.2 below.[20] The question still arises, however, whether any other numeric types should be permitted within the system. A case could be made both for real numbers, notably to enable greater accuracy in recording fractional coordinates within intricate recursive patterns, and also for bytes, to reflect the limited range of the virtual "machine code" instructions. The latter case is, however, rather weak, because the *arguments* to these instructions (e.g. representing lengths and angles) anyway have to be integers, and the overall aim of simplicity therefore implies a preference for treating the basic "machine code" unit as being an integer in general, even if the instructions themselves are limited to a single byte. The case for real numbers is somewhat stronger, but by no means decisive, because limitations of fractional co-ordinate accuracy are unlikely to be much of an issue within

[20] As implemented, *Turtle* uses four-byte integers, following the *Delphi* standard. Moreover the PCode display allows integers to be shown in hexadecimal, so that byte values can be distinguished.

the programs produced by beginners, while more advanced programmers can straightforwardly mitigate them if necessary by adjusting the Canvas dimensions, and by saving and restoring coordinates at appropriate places (as illustrated by the built-in "Recursion factory" program). Given also the decision to use integers as the "machine code" unit, the complications of introducing real numbers seem to outweigh any benefits. Hence only integer numbers are accepted within the system.

3.1.2 Other Data Types

Turtle implicitly uses Boolean values in conditional and iterative structures (e.g. after "if" and "until"), but in common with many other systems, represents these as integers, thus avoiding the need for type distinctions within the Program Stack. FALSE is represented as 0, and TRUE as −1,[21] though any non-zero value will be treated as TRUE in conditional processing. *Turtle* also explicitly recognises the "boolean" type (e.g. in procedure or variable declarations – as shown in the "flashlights" illustrative program), which can be helpful for explaining and motivating type declarations in a system where otherwise these might seem to be pointless.[22]

No other data types are recognised, because integers and Booleans suffice for the fundamental concepts necessary to the system's practical teaching role (as a preparation for *Delphi*), though there is something to be said for introducing arrays in particular, not only to provide more conceptual range and power for users, but also to develop the possibility that the Visual Compiler provides for the practical illustration of complexity considerations (e.g. by enabling "machine code" cycles to be counted within sorting algorithms). Another natural suggestion would be to move in the direction of object-

---

[21] −1 is a more appropriate choice than 1 because the four-byte "twos-complement" hexadecimal representation of −1 is FFFFFFFF, which equals the bitwise NOT(0), cf. §3.2 below.

[22] *Turtle* could do with further development here, since there is currently no type compatibility checking between Booleans and integers, and hence little benefit to be gained from declaring variables or procedure parameters as "boolean" rather than "integer". However in practice, students taught using the system have moved on to *Delphi* before detailed discussion of types has become appropriate, so it

orientation, with “turtles” as objects and therefore potentially multiple (cf. Appendix B below).

3.2 Arithmetical and Boolean Operators

The four basic arithmetical operators (“+”, “–”, “*”, “/”) are accommodated, though “/” is interpreted as *integer* division (i.e. “div”) given that reals are not permitted. The inclusion of the complementary operator “mod” is obviously desirable too, playing a particularly valuable role in the creation of alternating patterns (as exemplified in the illustrative programs “Combining structures” and “Cycling colours”).

The arithmetical comparison operators (“=”, “<>”, “<”, “>”, “<=”, “>=”) are likewise included, yielding the “boolean” values −1 (true) or 0 (false) as in §3.1.2 above. Use of these values enables the Boolean operators “not”, “and”, “or” and “xor” to be incorporated as *bitwise* arithmetical operators also, without ambiguity, since thus interpreted the Boolean and arithmetical results correspond (the point here being that the “twos-complement” binary representation of −1 has all bits set).

3.3 Program Structures

Clearly the basic conditional structure “if … then … else” is essential, and is sufficiently general to make “case” unnecessary. More delicate is the choice of conditional iterative structure between Pascal’s “repeat … until” and “while”: both are similarly expressive (in combination with “if … then … else”), so only one of them is required within a system whose primary aim is simplicity. In some respects “while” is preferable, since in technical programming situations, initial non-satisfaction of a loop condition typically implies total non-performance of the loop. However in Turtle Graphics “null” loops are relatively rare, and personal experience indicates that beginners tend to find the idea of a loop that is always performed *at least once* rather more intuitive, perhaps because it is easier to think of repeating an operation already concretely performed than to think of a

---

has sufficed to use the mere possibility of Booleans to illustrate how variable types can differ, and hence to explain why more advanced systems such as *Delphi* require type declarations.

loop that is entirely conditional (see Soloway et al. [125], Rogalski and Samurçay [112]). There is also a more particular confusion about the Pascal "while" statement deriving from its natural language meaning, leading some novices to expect that the loop condition will be tested continuously rather than once per iteration (Sleeman et al. [121], Bonar and Soloway [13]). Finally, when sequences of operations are involved (as is usually the case when drawing repeated patterns) the "repeat … until" construct in Pascal is syntactically far easier than "while … do begin … end", since "repeat" and "until" already provide the necessary bracketing. For all these reasons, "repeat … until" has been preferred to "while".

Given the provision of "if … then … else" and "repeat … until", there is no strict need for the counting iterative structure "for … do". However this is so useful in situations where an exact number of iterations is required, and so helpful and relatively natural for introducing the concept of a counting variable, that there is very good reason to include it.

Finally, any system that aspires to teach good programming practice must make provision for modular divisions of the program, and given the intention of using *Turtle* as a bridge to *Delphi*, procedures are absolutely essential (since *Delphi* methods take this form). Procedures if properly implemented should also permit recursion (for which a graphics system can provide a wonderful introduction) and – in combination with Pascal's distinction between "call by value" and "call by reference" parameters – enable results to be returned and hence the effective definition of functions. Such quasi-functions lack the syntactic convenience of literal functions,[23] but contexts in which this is a major issue seem unlikely to arise within introductory Turtle Graphics. For the sake of simplicity, therefore, and to motivate the use (and hence learning) of these parameter distinctions, *Turtle* does not incorporate literal "functions" as such.[24]

---

[23] For example a function "dist" that calculates the distance between coordinates permits the elegant syntax "if dist(x1,y1,x2,y2) > 100 then …", whereas the same condition requires two statements if the distance is instead returned as the "VAR" parameter to a procedure.

[24] Arguably the importance of functions is such that they ought to be included nonetheless, an argument that is likely to become decisive if *Turtle* is extended to other source languages.

3.4 Special Turtle Graphics Facilities

3.4.1 Turtle Graphics Instructions

In accordance with the purposes of the system, *Turtle*'s graphics primitives (all of which are explained in the online Help file) have been selected with the aim of making it easy to produce interesting and satisfying patterns, while at the same time avoiding unnecessary complexity. "Complexity" here is partly a psychological matter – for example the provision of both "forward" and "back" makes it *psychologically* simpler to produce movement in two opposite directions, though *logically* it would be more economical to provide only the single instruction "forward", which can be used with either a positive or a negative parameter (a similar point applies to "left" and "right"). The same principle of psychological ease requires inclusion of the "circle" and "blot" (i.e. filled circle) commands, even though in Seymour Papert's original conception of Turtle Graphics, "circle" is available only as a defined rather than as a primitive instruction (see note 16 in §2.1 above). Again, "thickness" is presumably theoretically dispensable (since one could draw multiple adjacent lines, each of width 1), but its provision obviously makes life much simpler.

Most other Turtle Graphics primitives involve no such potential redundancy, and are designed straightforwardly to extend its graphics capabilities. The "colour" instruction is obviously required; "randcol" less so, though it greatly facilitates the creation of visually striking patterns by novice programmers. The four instructions "home", "setx", "sety", and "setxy" provide absolute movement (as do operations involving the global coordinate variables "turtx" and "turty", provided together with the direction, colour, and thickness variables "turtd", "turtc" and "turtt"); note that "home" here is not redundant, because the home position is dependent on the Canvas dimensions and so does not involve specific coordinate settings.[25] (A similar point applies to the

[25] However "setx", "sety", and "setxy" do involve some redundancy, e.g. "setx(100)" is equivalent to (though psychologically simpler than) "setxy(100,turty)". Educationally it is convenient to introduce simple x- and y-coordinate setting commands before students have been taught about the relevant global variables; then "setxy" serves as a useful shorthand when both x- and y-coordinates are to be set.

"blank" instruction, which is not equivalent to any specific rectangle filling "polygon" command.) None of these four absolute movement commands involves drawing, whereas "forward" and "back" are sensitive in this respect to the state of the pen, controlled by "penup" and "pendown". Such sensitivity also applies to "drawxy", but not to "movexy", both of which provide relative movement.[26]

The remaining *Turtle* primitives are "canvas" (to set the Canvas dimensions dynamically), "pause" (invaluable for simple debugging as well as visual effects), "update" and "noupdate" (which are needed only to mitigate the display speed limitations of computer hardware), and the related group of "polyline", "polygon", "remember", and "forget" – these last four give a great deal of power and obviate the need for more specific polygon-drawing or -filling routines. In contrast with the case of "circle", note that there is no compelling argument here for dedicated "triangle", "square", or "rectangle" procedures (etc.), because the versatile "polyline" and "polygon" commands do not involve any increased psychological complexity: identifying a square by visiting its corners in turn is no less natural, but is significantly more generalisable, than specifying its coordinates within a dedicated routine.

3.4.2 Colour Handling

Turtle allows "24-bit" colours to be specified, following the *Delphi* convention of giving the three components in "BGR" order within a hexadecimal number (prefixed by "$"). Thus "colour($0080FF)" will create an orange, with zero intensity of blue, 128 of green, and 255 of red. However for compatibility with the widespread HTML convention, *Turtle* also allows these colours to be specified in "RGB" order, prefixed by "#", e.g. "colour(#FF8000)". 24-bit colouration allows very fine variation, which can be used to produce "3D" shadowing effects (as in the built-in "balls3D" illustrative program).

For introductory teaching purposes it is important to be able to access the basic colours more simply, and therefore eight predefined colour constants have also been

---

[26] Again without redundancy, since if "movexy" were to be defined as "penup" followed by "drawxy", then the state of the pen would be affected, whereas "movexy" leaves it unchanged.

provided, namely *blue*, *green*, *cyan*, *red*, *magenta*, *yellow*, *white*, and *black*. Each of these constants is precisely equivalent to the corresponding BGR integer – hence "colour(red)", for example, has the same effect as "colour($FF)" or "colour(255)".

Further special provision is needed to facilitate such things as multi-colour counting loops (used to cycle colours in the "cyclecolours" illustrative program). For this purpose, the numbers 1 to 8 (which following the BGR convention would all appear more or less jet black, with no blue or green and only a minimal intensity of red) are specially distinguished as not following that convention; instead, they represent the eight standard colours, in the order listed above (this order being derived from the traditional Turtle Graphics ordering). It is these special codes that are randomly chosen by the "randcol" command, and this also has the virtue of enabling "randcol" to be used as a simple random number generator (as shown in two of the illustrative programs: "Cycling colours" and "Using Booleans")

Having now examined the Pascal language structures that are supported by the Turtle system, we shall see in Chapter 4 how it has performed as a vehicle for introducing this limited language within elective modules at the University of Leeds (where it has served as a prelude to Delphi Pascal).

Chapter 4 Using *Turtle* to Teach Introductory Programming

The *Turtle* system is designed primarily as a learning rather than a teaching system, in that it aims to encourage students to experiment and thus discover for themselves the creative pleasure of programming, in particular by working through the self-teaching exercises. But it has been used since 2000 to introduce programming within formal taught modules for elective students, and since 2001 as the basis for the "Programming Concepts" core of a full 10-credit module which then led on to *Delphi* programming. So it is appropriate here to review this experience with the system, before going on to a wider discussion of its place within the educational curriculum

4.1 Brief Outline of the Course

The "Programming Concepts" component consists of four compulsory lectures followed by a final optional lecture which gives a tour of the Visual Compiler. Full lecture plans for the compulsory lectures, together with an example coursework, are provided in Appendix A. The lectures are built almost entirely around the exercises provided in the *Turtle* Help file, and in that sense resemble example classes rather than typical formal lectures.

1. The first lecture introduces the *Turtle* system and the idea of an algorithm, running through simple drawing commands and using the opportunities afforded by the first two exercises to familiarise students with relevant illustrative programs, help resources, and other *Turtle* facilities that they might find useful.

2. The second lecture builds on the first, quickly reviewing what it covered and then moving on to introduce the idea of a variable and of a counting loop, combining these with a discussion of program layout, systematic program development, and debugging techniques (e.g. using pauses and "blot" markers).

3. The third lecture is focused on procedures, emphasising the importance of modularity to enhance comprehensibility and going on to show the amazing power of recursion though a simple "triangle" example which is explained using the metaphor of the "Cat in the Hat",

with recursive calls being introduced one at a time until the picture shown in Figure 2 (p. 20) is produced. The built-in "Recursion Factory" program invites the students to experiment for themselves with recursive patterns of real complexity and aesthetic interest.

4. The fourth lecture introduces repeat loops in the context of producing animated graphics, such as the "bouncing balls" effect in two of the illustrative programs ("ballsteps" and "multibounce"). The remaining illustrative programs are also briefly reviewed, to give the students ideas that they can follow up for themselves in their coursework.

The coursework, assigned at the end of the fourth lecture, is designed primarily as a learning rather than an assessment exercise, to give confidence in elementary programming through pleasurable exploration and creativity. Accordingly, students are given free rein to write a graphical program of their choice, and are encouraged to make it entertaining. To ensure appropriate coverage they are required to use a wide range of *Turtle* commands and program constructs, but the work is not difficult, and even timid students can safely get through it with support – though the main focus of such support is enabling them to help themselves by making use of the *Turtle* system's built-in resources (notably the illustrative programs and the Help file). In addition to their main program, students are also asked to produce two short illustrative programs of their own, each designed to highlight the features of a different language construct. The intention of this exercise is to give them a relatively concrete task to start on while working out their ideas, to encourage them to reflect on the use of program structures, and to extend their coursework without requiring them to put all their eggs in the one basket of their main program. It also brings the benefit of giving insight into students' perception of the structures in question, and can provide useful additional illustrative resources to benefit future cohorts.

The next four pages show examples of the students' work (labelled and arranged to correspond with their organisation on the *Turtle* website), though virtually every one of these programs involved animation, and the pictures show only a snapshot – or in some cases two – of what were in many cases amusing programs. It is very clear that the *Turtle* system both entertained these students and inspired them creatively.

4.2 Students' Perceptions of the Course

As discussed in §1.2 above, the primary aim of the course is to introduce novice students to programming in an engaging and non-threatening manner, preparing the ground for more "serious" programming in follow-on courses. For this reason, student perception is the most important single criterion by which the course must be judged, and accordingly, anonymous feedback was sought systematically through a specially written program which invited students to express their views immediately after having submitted their coursework. (Similar feedback was also solicited for all other components in the ACOM teaching programme, enabling comparisons to be drawn.) To maximise the chance that all students would indeed express their views as fully as possible, the feedback interface was divided in two, the first simply asking them to click on "radio" buttons to give an overall appraisal, and the second inviting more detailed opinions. As a result, the feedback response rate for the first part was extremely high (typically 75% to nearly 100%, depending on the question), but only around one in six of the students – apparently disproportionately those with the stronger opinions – also

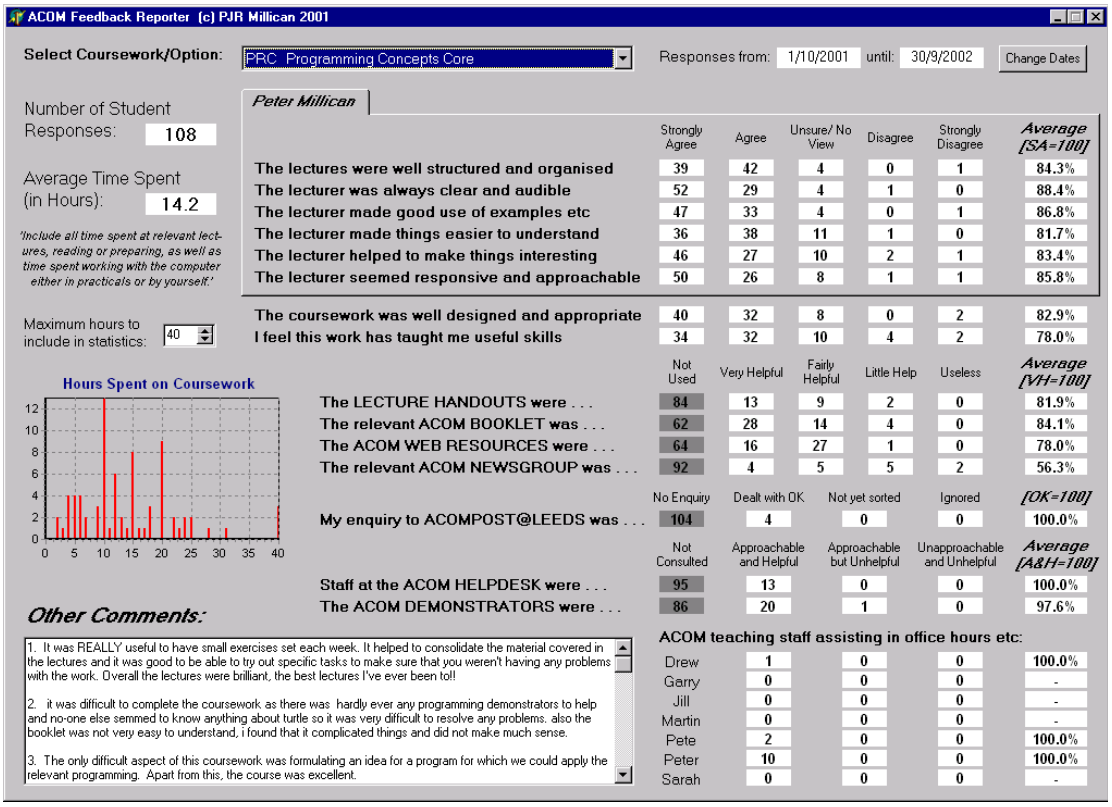


Figure 7: The automated student feedback analysis program

took advantage of the second part to emphasise specific points. The feedback was analysed using a Feedback Reporter program, which broadly mirrored the structure of the students' feedback interface (see Figure 7).

In assessing this student feedback, most weight must be put on the overall appraisal "radio button" responses, because they are so much more representative than the written comments. Over the three sessions in question, the results were as follows, averaging each question's responses on a scale that ranges from 100% (for the most positive available answer) to 0% (for the most negative):

	2000-2001	2001-2002	2002-2003	*(ACOM Norm)*
The lectures were well structured and organised	87.0%	84.3%	85.7%	*(70%)*
The lecturer was always clear and audible	90.0%	88.4%	89.3%	*(70%)*
The lecturer made good use of examples etc.	–	86.8%	89.3%	*(70%)*
The lecturer made things easier to understand	–	81.7%	79.8%	*(66%)*
The lecturer helped to make things interesting	–	83.4%	86.9%	*(62%)*
The lecturer seemed responsive and approachable	89.0%	85.8%	90.5%	*(67%)*
The coursework was well designed and appropriate	83.8%	82.9%	76.8%	*(67%)*
I feel this work has taught me useful skills	81.0%	78.0%	75.6%	*(73%)*
The lecture handouts (or booklet) were very helpful	90.0%	84.1%	83.9%	*(72%)*

It is apparent that these are extremely positive results, and this impression is confirmed when they are compared against the overall responses for the 27 highly varied components that have been delivered within the ACOM "IT Skills" programme over the period, which have averaged around the approximate figures shown in the last column. Even though the ACOM programme as a whole is seen as extremely successful with students, "Programming Concepts" has consistently come at the top of its feedback league. Only one of the other 26 components has ever exceeded it over this time, namely the small option "IT, Politics and Society", and that only in 2002-03.

Although the students' written feedback must be viewed with considerable caution, because of its unsystematic and unrepresentative nature, it gratifyingly conforms very

well with the positive impression given by the overall statistics above, with many comments expressing positive enjoyment of the learning experience:

- I found this by far the most enjoyable thing I have done in ACOM so far!
- I found this to be a very tough, yet very enjoyable module. With practice, all of the concepts become clear and the results are very satisfying. I thought Peter Millican's teaching was excellent.
- The online Turtle Graphics help was very helpful.
- Handouts weren't really necessary as all of the necessary information was contained within the program itself. I found myself quite keen on doing the exercises and coursework but this may have been because it gave me a break from my other homework. A good programming starter program.
- I found this core both stimulating and enjoyable. Peter Millican's approach to teaching programming makes the subject less mind-boggling.
- Very well structured and well taught – Peter made it very easy to understand the concepts of the programming language being taught.
- A superb introduction to computer programming; I'd recommend it to anyone from the very young to the very old. Peter Millican's performance of "The Cat in the Hat", to elucidate recursion, just has to be seen to be believed; it ought to be filmed and distributed to all programming students everywhere.
- Being able to download "Turtle Graphics" to use on my computer was very useful.
- It was REALLY useful to have small exercises set each week. It helped to consolidate the material covered in the lectures and it was good to be able to try out specific tasks to make sure that you weren't having any problems with the work. Overall the lectures were brilliant, the best lectures I've ever been to!!
- The only difficult aspect of this coursework was formulating an idea for a program for which we could apply the relevant programming. Apart from this, the course was excellent.
- It was a pleasant and not very daunting introduction to programming.
- I found this part of the module very enjoyable and very useful.
- I really enjoyed doing this, but I sat down and worked it all out for myself rather than reading the notes. The lecturer was very helpful and seemed really enthusiastic about the work himself.
- Enjoyed it. Turtle Graphics could be easier if it had a grid on the page so you could always see where you are
- I thought the course was very interesting and quite challenging. I thought the turtle graphics program was very helpful as the introduction to programming as there was visible results so you felt like you were actually achieving something.
- I enjoyed the 'cat in the hat' analogy to recursion. Though it did seem somewhat silly, it did help to clarify the idea of recursion.
- The Programming Core was a fun and interesting one
- very interesting
- It was a very good module, thanks.
- Programming was a new experience for me but one I have enjoyed despite it being so time consuming, I look forward to pursuing more programming options later in my university life.
- Turtle graphics' in-built help and examples were extremely useful. I found I learnt a lot just by playing around with the illustrative programs. The lectures were also very useful, well explained and relevant. Happy customer!
- Would have been interesting to study the applications of recursion in greater detail, as this was a fascinating area of the course. Many thanks.
- I thought this was a great set of lectures. I found them useful, interesting, and thoroughly enjoyable.

Of the total of 38 comments provided over 2000-2002, 30 were clearly positive and only 3 negative, with one of these expressing uncertainty over what the coursework was really looking for, and the others concerning the availability of demonstrators. Other comments included suggestions for the course, namely that more time should be spent on basic mathematics, repeat loops, or procedures and parameters, and three recommendations for improvements to the *Turtle* system itself, namely, inclusion of a visual grid, making the "Are you sure?" quitting prompt more selective, and adding keyboard shortcuts. The last two of these suggestions have since been acted on.

4.3 How Well Did the Students Learn?

As explained above in §4.1, the coursework required students to produce three programs, the longest of which was expected to use each of a fair range of commands and structures. (The required range, together with a tally for the last-compiled program, can be seen in the "Expressions" table in the Visual Compiler display, shown on p. 21 – this was available to students to check their work, and proved invaluable for markers.) To do even moderately well, students had to exhibit reasonable competence with a number of programming constructs, and to put these together within a coherent program of their own design. Moreover because their programs were all so visibly distinctive, and were often evidently objects of pride and enjoyment (as manifest from the coloured plates), there was little temptation or opportunity for students to copy from others, except in the unlikely event that one of them effectively did the coursework twice over. Hence one can be fairly confident that those students who passed the module had at least mastered some of the basic skills of programming, such as linking sequences of commands together, dividing code into procedures, iterating with counting and repeat loops, and using simple conditionals.

Given this background, it is gratifying how few students failed to reach a pass standard in their coursework – over the sessions 2000-2003, only 5.9% were graded less than 40. To put this in perspective, the nearest comparator available within Leeds University is the module COMP1150, "Introduction to Programming on the PC", which ran as an elective for a similar range of students between 1993 and 1999, also teaching Pascal. Obviously over the last decade there has been considerable change, not only in computer systems and staff teaching allocations, but perhaps even more so in the typical

profile of a Leeds student (more IT literate but arguably less "elite", mathematically weaker, and more used to "spoon-feeding"), so such comparisons must be viewed with some caution. But if we focus on the last three years of COMP1150, over which period it was taught by two different lecturers with a single package – namely Borland Pascal for Windows – the contrast with the Turtle Graphics course (also taught by two different lecturers over the relevant period) is very striking:

Figure 8: Grade profile for the 1996-99 course teaching Borland Pascal

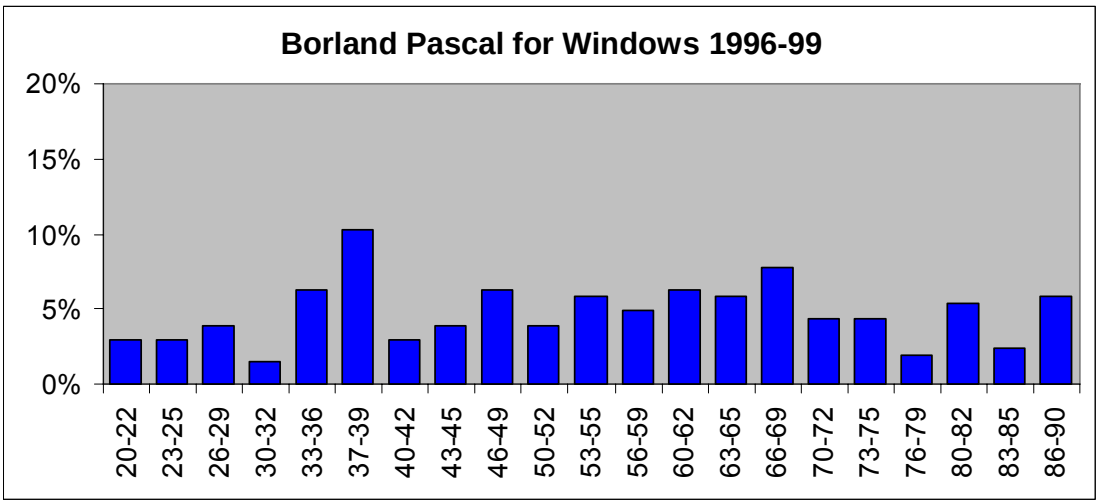
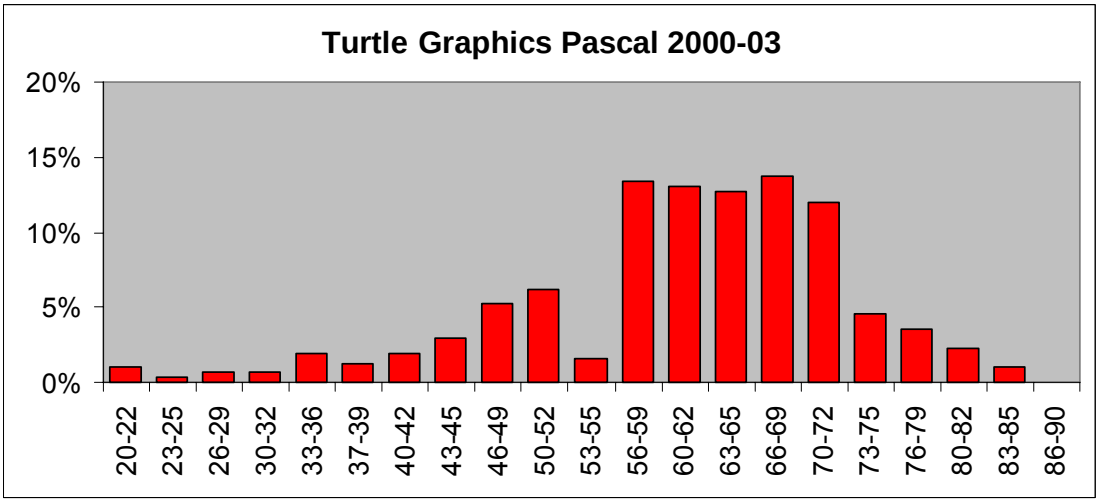


Figure 9: Grade profile for the 2000-03 course teaching Turtle Graphics Pascal



The peak in the Borland Pascal "37-39" column actually represents *pass* grades, because prior to 2000 the Leeds pass threshold was 37.[27] However the peak itself occurs because students are limited to a bare pass on resitting, so the fact remains that many of these students initially failed.

The detailed figures are shown in the table below. Particularly notable is that on the earlier course, 27.8% of students scored below 40, and of these, 17.6% below 37, whereas on the Turtle Graphics course, these figures reduced to 6.0% and 4.7% respectively. But any suspicion that the contrast between the grade profiles might have resulted from a crude "dumbing down" is easily dispelled by looking at the other end of the scale, where on the Borland Pascal course 13.7% scored over 80, while the corresponding figure for Turtle Graphics was only 3.3%. As this indicates, the later course was in some ways rather *more* demanding than its predecessor, involving additional "advanced" concepts such as recursion. The crucial difference between the two was apparently that *Turtle* made it far easier for students *to get started*.

Borland Pascal for Windows 1996-1999		**Turtle Graphics Programming 2000-2003**	
Number ***205***	*Average* ***55.05***	*Number* ***307***	*Average* ***61.05***
20-22	2.9%	20-22	1.0%
23-25	2.9%	23-25	0.3%
26-29	3.9%	26-29	0.7%
30-32	1.5%	30-32	0.7%
33-36	6.3%	33-36	2.0%
37-39	10.2%	37-39	1.3%
40-42	2.9%	40-42	2.0%
43-45	3.9%	43-45	2.9%
46-49	6.3%	46-49	5.2%
50-52	3.9%	50-52	6.2%
53-55	5.9%	53-55	1.6%
56-59	4.9%	56-59	13.4%
60-62	6.3%	60-62	13.0%
63-65	5.9%	63-65	12.7%
66-69	7.8%	66-69	13.7%
70-72	4.4%	70-72	12.1%
73-75	4.4%	73-75	4.6%
76-79	2.0%	76-79	3.6%
80-82	5.4%	80-82	2.3%
83-85	2.4%	83-85	1.0%
86-90	5.9%	86-90	0.0%

Again, it must be emphasised that this teaching did not take place in the context of a controlled study or systematic comparisons with other courses, so conclusions regarding

[27] This is why the grouping in the 30's takes the form "30-32", "33-36" and "37-39". Within the 40's (and likewise for the other "decades") it more appropriate to use the grouping "40-42", "43-45" and "46-49", because of all these grades, 49 is by far the least used.

the effectiveness of the *Turtle* system must be to that extent tentative. Potential complicating factors include the following:

- Those taking the introductory programming component were self-selecting elective students, free to drop out or switch options had they so chosen, and this might be expected to reduce the "tail" of failures characteristic of compulsory programming courses. However this does nothing to impugn the favourable comparison with the 1996-99 Borland Pascal course, which was also entirely optional.[28]

- *Turtle* covered only the first four weeks of programming, enabling students to get by without having to handle seriously complex programs. However this point should be taken in the context of §4.4 below, which highlights evidence that fundamental student difficulties occur at a very early stage of understanding, so an introductory component arguably faces a task which is harder, rather than easier, compared to later components. Moreover the programs produced on the *Turtle* component were not in any case particularly simple – many extended to several hundred lines and numerous procedures (though admittedly they were simple in terms of data structures and techniques).

- *Turtle* was taught by two members of staff who were individually very committed to the system, including its developer (the other was Dr Sarah Kattau, who has contributed teaching materials for *Turtle* and been its most assiduous tester). Moreover both teachers have scored well in other teaching, implying that the good results and feedback in the programming component might have been due at least in part to their general teaching abilities and their specific enthusiasm for *Turtle*.[29] This concern probably has an element of truth, but it risks putting the cart before the horse, because part of what makes an effective teacher is the ability to identify and develop effective tools and methods. If *Turtle* is seen by teachers elsewhere as a useful tool, there is no evident reason why it should not inspire them and their students with comparable enthusiasm to that which in Leeds has led to such promising results.

---

[28] Indeed if anything this makes the comparison more favourable, since the earlier course was an integrated module, making early dropouts far more likely than in the *Turtle* component where students were not anyway committed to continuing with programming beyond the first four weeks.

[29] A sort of "Hawthorne Effect" (Mayo [81], ch. 3) might be suspected here, whereby the mere act of studying students' performance can serve to improve it, but in fact this supposed phenomenon is considered rather dubious in the light of later studies (e.g. Parsons [98], Adair [2]), and besides, the *Turtle* component – with its self-learning materials – put very little emphasis on student monitoring.

4.4 Conclusion: The Value of *Turtle* as a Vehicle for Introducing Programming

Clearly the *Turtle* system's ambitions in respect of introducing novices to programming are very limited – it aims to provide a good basis for the first month or so of learning, and makes no attempt to convey software engineering principles of any sophistication. *Turtle*'s value must accordingly be measured against this limited ambition, and by that standard the evidence given above strongly suggests that it is a success. Not only do novice students cope with it (and many choose to go on to more "serious" programming in *Delphi*,[30] generally with success), but they are excited and enthused at discovering for themselves the pleasure of intellectual creativity. Here I would suggest a strong similarity between the potential appeal of programming and that of playing chess, as famously expressed by the great Siegbert Tarrasch in the preface to his last book, *The Game of Chess*:

> Chess is a form of intellectual productiveness, therein lies its peculiar charm. Intellectual productiveness is one of the greatest joys – if not the greatest one – of human existence. It is not everyone who can write a play, or build a bridge, or even make a good joke. But in chess everyone can, everyone must, be intellectually productive, and so can share in this select delight. I have always a slight feeling of pity for the man who has no knowledge of chess, just as I would pity the man who has remained ignorant of love. Chess, like love, like music, has the power to make men happy.

If playing with *Turtle* can harness some of this power, then the benefits may be considerable.

Scepticism about the value of this approach is most likely to be prompted by the thought that *Turtle* is just a "toy" system, far *too* simple to be of significant use for introducing *real* programming. But such scepticism is hard to maintain when faced with

---

[30] I have not included statistics from the two *Delphi* courses that followed on from the *Turtle*-based "Programming Concepts" core, partly for reasons of space but also because they were optional, making comparison with previous modules problematic. In practice, the vast majority of students have passed, with an overall average around 56-60, and feedback has been very positive (mostly 75%-85%).

evidence of how badly many students fare when taught programming in a traditional manner. Thus McCracken et al. [83] highlight "concerns expressed by many computer science educators about their students' lack of programming skills … often … focused on basic mastery of fundamental skills", and give supporting references together with research indicating that such concerns are fully warranted. In the abstracts to two recent papers, Chalk et al. [28] and Jenkins [61] both give considerable prominence to a depressing assessment of the current situation: "There is a national crisis in the teaching and learning of programming" and "The graduating student who professes a complete inability to write the simplest program is commonplace". Likewise according to Jenkins and Davy [64], many students "appear to be totally unable to grasp the basic concepts", leading them later in their studies to "insist that they want to avoid programming at all costs".[31]

But what is particularly striking about all this is the level of expertise at which the students concerned are faltering, as shown by a wide variety of studies.[32] Variable assignment and initialisation are often misunderstood (du Boulay [41], Samurçay [114]), while particularly common bugs are those associated with loops (du Boulay [41], Spohrer et al. [128]) and conditionals (Sleeman et al. [121], Hoc [56], Putnam et al. [107]), and general difficulties with tracing flow of control (Perkins et al. [102], Rogalski and Samurçay [112]) – precisely the sorts of problems that might be alleviated if students spent time experimenting with systems of this type.[33] In the context of such problems with flow of control, it is hardly surprising that recursion is widely viewed as

---

[31] Many more references could easily be given, for example Shackelford and Badre [115], and those in Deek [34] and in the first paragraph of Sims-Knight [119]. For consideration of *why* programming is so difficult, see e.g. Guzdial [49], Jenkins [62], Moser [85] and the references they cite.

[32] Two useful reviews of such studies are provided by Winslow [138] and Robins et al. [111].

[33] Here I focus on *Turtle*'s role as a vehicle for introductory program development and execution, but we shall see later (in §8.1.1) that it has another role which can help in dealing with these problems. There is considerable evidence that students' problems with control structures derive in large part from an inability to conceive an appropriate model of the computer's behaviour (see for example the reviews by Pane and Myers [95], pp. 30, 32; Robins et al. [111], pp. 149-53). *Turtle* is distinctive amongst introductory programming systems in incorporating an explicit virtual machine which is designed to enable even non-specialist users (though admittedly not total novices) to establish such a mental model.

an extremely difficult topic (e.g. Dicheva and Close [38], Kahney [66], Kessler and Anderson [68], Levy [71]). Yet the *Turtle* system, for all its relative simplicity, can very effectively convey an understanding of recursion, sufficient at least to persuade most of a novice class that they grasp it after only three lectures.[34]

Even if the level of understanding conveyed by experimenting with *Turtle* were relatively modest, the very fact that it can give students confidence and enjoyment is itself of potentially major significance. Robins et al. ([111], p. 158) summarise research by Linn and Dalby [75] and Kurland et al. [70] in words that could serve as a manifesto for the type of graphics-based self-learning system that *Turtle* exemplifies:

> The reinforcement and encouragement derived from creating a working program can be very powerful. In this context students can work and learn on their own and at their own pace, and programming can be a rich source of problem-solving experience. Working on easily accessible tasks, especially programs with graphical and animated output, can be stimulating and motivating for students.

They also report a survey by Rountree et al. [113] as indicating that for novice students, "the most reliable indicator of success was the grade that the student expected to achieve" (p. 155). Obviously it can be debated here which factor was cause and which effect, but Jenkins [61] argues that a student's expectation of success is crucial to its achievement, since such expectation impacts so directly on student motivation – drawing on Keller [67], he suggests that *motivation* can be thought of as the multiplicative product of *expectancy* and the perceived *value* of success. Such a view would strongly support the use of any system, however modest, that can increase students' enjoyment and confidence and introduce programming in a positive manner.

A related virtue of the *Turtle* style of learning is that students, as their confidence grows, are strongly motivated to enhance their own designs, adding features to graphically interesting programs that evince evident pride and satisfaction. Hence from

---

[34] At the end of each year's third *Turtle* lecture, most students have raised their hands when asked to indicate whether they understand the "triangles" example (§2.3). Obviously this does not prove that they have a deep and secure understanding, but it is surely significant that this much can be achieved with novices less than an hour after they have first acquired the concept of a procedure.

simple beginnings, students characteristically end up developing programs of significant complexity, and thus encounter at an early stage the typical problems of design and plan composition that seem to underlie so many of the difficulties that beset inexperienced programmers (including many "bugs" that are standardly, though often perhaps mistakenly, attributed to syntactic difficulties, cf. Spohrer and Soloway [127], Winslow[138], Pane and Myers [95] p. 30). This gives an excellent context for novice students to encounter and overcome such problems, where they have willingly taken them on, have a strong personal motivation, also a fairly clear conception of what they are trying to achieve, and are working within a system that gives straightforward, visible, and immediate feedback. Having developed the strategies to become "effective" (Robins et al. [111], pp. 165-6) in this context, the hope is that they can then take these strategies forward to other, more advanced, systems.

The value of developing this sort of initial competence (and confidence) on simple systems is emphasised by the research of Hagan and Markham [51], who found that prior programming experience brings a major continuing benefit to those embarking on Computing degree programmes (cf. also Cooper et al. [31]). This opens another possible role for *Turtle*, as a pre-university "taster", taking advantage of the system's engaging character and built-in independent learning resources to provide advance preparation prior to formal courses. In a somewhat similar vein, *Turtle* could be used to assess student's varied programming backgrounds and aptitudes, playing the same sort of role that Poulton [105] suggests for LOGO, but with the advantage of introducing a more standard syntax that might well be useful to them in the future, and giving students the possibility of exploration into the Turtle Machine, which may be particularly appreciated by those who could otherwise find the exercise trivial.

This concludes our detailed discussion of Turtle as a vehicle for introductory programming. In Chapter 5 we move on to the second half of the thesis, concerned with the virtual Turtle Machine and its operations, and thus having a teaching orientation towards Computer Science rather than introductory programming.

Chapter 5 The Virtual Turtle Machine and its "PCode" Object Language

By far the most difficult aspect of the system design was the development of a virtual "Turtle Machine" capable of sustaining full recursive and versatile parameter handling behaviour whilst remaining simple enough to be broadly comprehensible (after only a moderate amount of study) to a non-specialist. Perhaps the acid test here is whether the Help file sections on "An Introduction to PCode", "Technical Note on Variables, Procedures and Parameters", and "PCode Reference Guide" are in fact potentially accessible to interested students who have not studied the relevant computer science in detail. Experience here is positive, both by reference to the students themselves and to a peer observer (Dr. John Davy), no doubt in large part because this aspect of the system has been confined to purely optional final lectures, designed to appeal to self-selected enthusiasts within the student body. (However even these enthusiasts would have had little chance of understanding standard textbook presentations of the relevant theory, so this success is gratifying.) In these next two chapters, I shall approach the virtual Turtle Machine from the opposite direction to the Help file, focusing more on explaining *why* it has the structure it does, rather than on the detail of *how* that structure operates in practice. At the same time, I shall endeavour to explain these things in a way that presupposes very little technical background, so that the chapters can serve at the same time as an illustration of how the *Turtle* system can provide an appropriate vehicle for Computer Science education at the non-specialist level.

5.1 Turtle Graphics Commands, the Program Stack, and Arithmetical/Boolean Operators

Since the educational aim of the Turtle Machine is to give an insight into the general behaviour of a compiler and dynamic memory management, rather than the specifics of graphical processing, it is clearly appropriate to keep the handling of Turtle Graphics commands as simple as possible, which is done by straightforwardly defining primitive "machine code" instructions corresponding to all of the Turtle Graphics primitives. These are shown in the following table, where each instruction is followed by the

relevant "assembler" mnemonic and "machine code" value in decimal and hexadecimal (using the *Delphi*, and hence *Turtle*, prefix convention of "$"). Here they are ordered according to their logical category and stack effects, an organisation that should be evident from the Help file's "PCode Reference Guide":[35]

***Turtle* instruction**	**Assembler mnemonic**	**Decimal code**	**Hex code**
penup	PNUP	64	$40
pendown	PNDN	65	$41
update	UDAT	66	$42
noupdate	NDAT	67	$43
home	HOME	80	$50
remember	RMBR	81	$51
forward	FWRD	96	$60
back	BACK	97	$61
left	LEFT	98	$62
right	RGHT	99	$63
setx	SETX	100	$64
sety	SETY	101	$65
thickness	THIK	102	$66

***Turtle* instruction**	**Assembler mnemonic**	**Decimal code**	**Hex code**
Colour	COLR	103	$67
randcol	RNDC	104	$68
blank	BLNK	105	$69
pause	WAIT	106	$6A
circle	CIRC	112	$70
blot	BLOT	113	$71
polyline	POLY	114	$72
polygon	FILL	115	$73
forget	FRGT	116	$74
movexy	MVXY	128	$80
drawxy	DRXY	129	$81
setxy	TOXY	130	$82
canvas	CANV	136	$88

Some of these instructions (e.g. "home") require no parameters, some require just one (e.g. "forward"), and some require two (e.g. "movexy") or more (i.e. "canvas") – given this variety, the simplest way of delivering these parameters in a uniform manner within the compiled machine code is by means of a Program Stack (or just the capitalised "Stack" for short). Such a stack also provides the simplest way of handling arithmetical calculations in reverse Polish fashion, and this integrates easily with its use for parameter

[35] In this and the next chapter, all of the various PCode commands will be discussed (with the sole exception of the "NULL" command, code 0, which does nothing whatever), but owing to constraints of space, in most cases their precise behaviour is not specified. For details of all PCode commands, please consult the Help file section just referred to.

passing. Thus for example the command:

```
movexy(3+4*5,15-7)
```

should involve the following sequence of operations:

Push 3 onto the Stack
Push 4 onto the Stack
Push 5 onto the Stack
Multiply the top two Stack values, leaving the result (i.e. 20) in their place
Add the top two Stack values, leaving the result (i.e. 23) in their place
Push 15 onto the Stack
Push 7 onto the Stack
Subtract the top Stack value from the next, leaving the result (i.e. 8) in their place
Execute MVXY, pulling its two parameters (i.e. 23 and 8) from the Stack

Since any student who aspires to understand this sort of thing obviously needs to have grasped the concept of a stack,[36] the simplest possible structure for the desired virtual Turtle Machine is one based on an explicit "hardware stack" architecture, taking advantage of this to eliminate as far as possible the need for any other parameter storage registers. As we shall see, such a simplification is indeed entirely feasible.

The use of a stack-based architecture avoids any need to consider "hardware" operations at any lower level (i.e. we can ignore the issue of how the Stack is to be implemented), and this enables us to determine very straightforwardly the required behaviour both of the Turtle Graphics primitive "machine code" commands, and also the various arithmetic and Boolean operators. These must operate on the top Stack value or the top two (and in the latter case, must treat these in an appropriate order, so that SUBT, for example, subtracts the top value from the next, as in the italicised example above);[37] the calculated result of each operation must then replace the used values on top of the Stack. Working out the specific behaviour of each particular instruction from

---

[36] This is a virtue rather than a drawback, since it means that the *Turtle* system provides a useful vehicle for introducing (and illustrating the tremendous practical value of) this vital data structure.

[37] All this applies equally to the relevant Turtle Graphics "machine code" commands from the previous table, as illustrated by "MVXY" in the italicised example.

this general pattern is then very simple – they are merely listed here, but see the "PCode Reference Guide" (in the online Help file) for details if desired:

Turtle operator	Assembler mnemonic	Decimal code	Hex code
– (unary)	NEG	144	$90
Not	NOT	160	$A0
+	PLUS	176	$B0
– (binary)	SUBT	177	$B1
*	MULT	178	$B2
/	DIV	179	$B3
mod	MOD	180	$B4
and	AND	192	$C0

Turtle operator	Assembler mnemonic	Decimal code	Hex code
or	OR	193	$C1
xor	XOR	194	$C2
=	EQAL	208	$D0
<>	NOEQ	209	$D1
<	LESS	210	$D2
>	MORE	211	$D3
<=	LSEQ	212	$D4
>=	MREQ	213	$D5

5.2 Command Parameters, Global Variables, and the Heap

An issue not covered in the previous section is how the various command parameters, or numbers to be operated upon, are to be loaded, or "pushed", onto the Program Stack in the first place. Where they are straightforward integers (rather than variables), this is accomplished using the "LDIN" machine instruction:

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
load integer (onto Stack)	LDIN	16	$10

So we can now derive the appropriate compilation of the various simple statements in the first built-in illustrative program (seen in §2.1 above), for example:

```
forward(450);                LDIN 450  FWRD
pause(1000);                 LDIN 1000  WAIT
right(90);                   LDIN 90  RGHT
thickness(9);                LDIN 9  THIK
```

Suppose, however, that we wish to compile a statement that operates on a global variable, for example:

```
forward(turty);
```

which moves the turtle forward by a distance corresponding to its current y-coordinate. (Here we take advantage of the fact that "turty" is a predefined global variable, and so can be discussed without worrying about the details of variable declarations.) This command is obviously similar to the "forward(450)" compiled above, except that here the parameter to FWRD has to be loaded onto the Program Stack from the storage address of the global variable "turty", rather than being loaded as a fixed integer.

Global variables are stored sequentially at the bottom of a structure called the Heap, which begins with the five predefined globals "turtx", "turty", "turtd", "turtt", and "turtc" (representing the current turtle x-coordinate, y-coordinate, direction, pen thickness, and colour) and then continues with any global variables that may be defined within the running program. The remainder of the Heap is used for the storage of local variables, which we'll come to later (see §§6.2-4):

Heap structure:

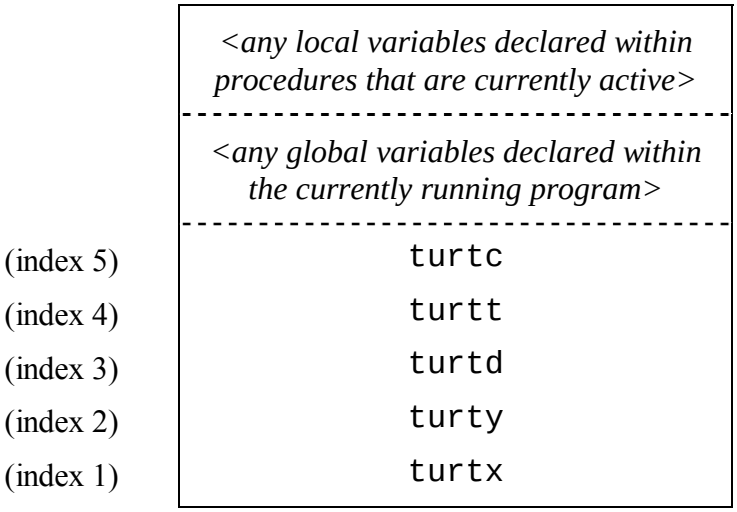


Figure 10: Overall heap structure, showing predefined global variables

Since "turty" has an index number of 2, to compile the statement "forward(turty)" we need an appropriate PCode command for loading the global variable with index 2 onto the Program Stack. This command is "LDVG 2", giving the compiled result:

```
forward(turty);                LDVG 2  FWRD
```

The inverse PCode instruction to “LDVG” is “STVG”, which pulls (or “pops”) the top value from the Program Stack and stores it in the appropriately indexed global variable. The codes of these two instructions are as follows:

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
load global variable	LDVG	17	$11
store global variable	STVG	33	$21

Together with the PCode commands already introduced, these enable the compilation of assignments and other operations involving global variables. The following fragment, for example, shows the start of one of the built-in illustrative programs:

```
PROGRAM cyclecolours;
VAR length: integer;
    cinc: integer;
    nextc: integer;
BEGIN
 randcol(5);                   LDIN 5  RNDC
 cinc:=turtc-1;                LDVG 5  LDIN 1  SUBT  STVG 7
 randcol(6);                   LDIN 6  RNDC
 length:=0;                    LDIN 0  STVG 6
```

When this is compiled, the three newly declared global variables “length”, “cinc” and “nextc” are assigned addresses 6, 7, and 8 on the Heap, directly above the five predefined globals. Hence the last compiled line above, which has the effect of loading the integer 0 onto the Stack and then storing this in global variable index 6, is indeed equivalent to “length:=0”. The second compiled line is more complex, involving a calculation prior to the global assignment. It begins by loading variable 5 (i.e. “turtc”) onto the Stack, then the integer 1 is also loaded and SUBT is performed, which leaves the result of the subtraction “turtc–1” on the Stack. Finally this result is stored in global variable index 7 (i.e. “cinc”), yielding the desired “cinc:=turtc–1”.

5.3 PCode Sequential Structure and Flow Control

5.3.1 PCode Line Structure, Storage, and Sequencing

Given the educational purpose of the *Turtle* virtual machine, easy comprehensibility of its PCode is far more important than economic storage and processing. Hence it makes sense to divide it, for both logical and display purposes, into small sections (henceforward "code lines") corresponding to individual Turtle Graphics commands or other relatively self-contained units, as illustrated by the four code lines ("LDIN 5 RNDC" etc.) shown in the short compilation table on the previous page. This implies a storage overhead, because to preserve the *line structure* of the code, it is not enough just to store the relevant PCode integers as a continuous undifferentiated sequence. An additional code is needed within each line, either at the beginning (to specify the length of each line), or at the end (to act as a terminator). The former method is used for the actual storage of "Turtle Graphics Compiled PCode" files (extension ".tgc") when they are saved from the Compile menu, and this enables the line structure to be restored very easily when it is loaded into the Turtle Run-Time Standalone System, which is designed to run compiled PCode directly. In addition to these individual line length codes, three other numeric codes are stored within each PCode file at the very beginning, representing in turn the total number of compiled code lines, the index of the code line from which execution is to start, and the number of global variables – these too are needed to facilitate the standalone execution of the compiled PCode.

The organisation of PCode into code lines naturally suggests their use for determining the sequencing and flow of control when the compiled code is executed, though this involves some departure from the behaviour of a typical real-life processor (whose code locations, jumps and branches are all likely to be specified by absolute or relative memory addresses). It also brings some inefficiency, because the compiled PCode has to be set up in a two-dimensional array, with consequent processing and space overheads. Conceptually, however, the departure is not huge, because it is obvious that the code line organisation can easily be translated into a linear form, and the sequencing of control flow along each code line and then from one code line to the next is closely analogous to the familiar process of following the text of a book.

5.3.2 The Conditional "if … then"

To compile "if … then" structures within this context, we need one PCode instruction that will jump to the beginning of a specified code line *unconditionally*, and another that will branch *conditionally*, depending on the evaluation of some condition. For simplicity, this condition is always that the value on top of the Program Stack is 0, i.e. "false", so the branch is followed if, and only if, the previously tested condition is *not* satisfied: (The "HALT" instruction is also included here for completeness.)

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
jump to code line	JUMP	48	$30
if not, branch to code line	IFNO	49	$31
halt execution	HALT	56	$38

To see how all this fits together, consider the following simple program and its compiled PCode equivalent (in which the code lines are numbered):

```
PROGRAM randblots;
BEGIN
 randcol(2);              1.  LDIN 2  RNDC
 if turtc=1 then          2.  LDVG 5  LDIN 1  EQUL  IFNO 5
  blot(300)               3.  LDIN 300  BLOT
 else                     4.  JUMP 7
  begin
   circle(300);           5.  LDIN 300  CIRC
   blot(200)              6.  LDIN 200  BLOT
  end;
 colour(red);             7.  LDIN 255  COLR
 blot(100)                8.  LDIN 100  BLOT
END.                      9.  HALT
```

The crucial code line here is the second, where the conditional test is performed. First the current value of "turtc" is pushed onto the Stack ("LDVG 5"), then the integer 1 ("LDIN 1"). The command "EQUL" then tests the top two Stack values for equality, leaving a "true" result (i.e. −1) in their place if they are indeed equal, and a "false" result (i.e. 0) if they are different. If the result was "true", then the following command

"IFNO 5" is simply ignored, and processing continues through code line 3, before meeting the unconditional "JUMP 7" command at line 4, which jumps over code lines 5 and 6 directly to 7. If on the other hand the result was "false", then "IFNO 5" branches the processing to code line 5 and continues from there. In either case, the program halts at code line 9.

5.3.3 The Iterative "repeat … until"

The "repeat … until" structure requires only a single conditional backward branch, and so can be straightforwardly compiled using the IFNO instruction; for example:

```
PROGRAM concentric;
VAR radius: integer;
BEGIN
 radius:=500;            1.  LDIN 500  STVG 6
 repeat
  randcol(8);            2.  LDIN 8  RNDC
  blot(radius)           3.  LDVG 6  BLOT
  radius:=radius-12      4.  LDVG 6  LDIN 12  SUBT  STVG
                             6
 until radius<20         5.  LDVG 6  LDIN 20  LESS  IFNO
                             2
END.                     6.  HALT
```

Here code line 5 makes the test for "radius<20" – if this is false, then processing branches back to code line 2, repeating the loop again.

5.3.4 The Counting Loop "for … do"

The "for … do" loop (or "for … downto" if the counting is downwards), like "if … then", involves one jump and one conditional branch, though it is significantly more complex because of the handling of the loop variable. Here is the complete "forloops" illustrative program together with its compilation:

```
PROGRAM forloops;
VAR count: integer;
BEGIN                          1.  LDIN 1
 for count:=1 to 200 do        2.  STVG 6  LDIN 200
  begin                        3.  LDVG 6  MREQ  IFNO 11
   forward(count/3);           4.  LDVG 6  LDIN 3  DIV  FWRD
   right(5);                   5.  LDIN 5  RGHT
   colour(red);                6.  LDIN 255  COLR
   blot(200);                  7.  LDIN 200  BLOT
   colour(black);              8.  LDIN 0  COLR
   circle(200)                 9.  LDIN 200  CIRC
  end                          10. LDVG 6  LDIN 1  PLUS  JUMP 2
END.                           11. HALT
```

The program starts by loading 1 onto the Stack, and at code line 2 this value is moved into the variable "count" and replaced on the Stack by the loop termination value of 200. Now the variable is reloaded onto the Stack in order to compare it against that 200,[38] with the operator "MREQ" leaving the result "true" (i.e. −1) on the Stack if and only if 200 is more than, or equal to, the variable's current value. Hence if the result is "false" (i.e. 0), this implies that "count" has reached a value greater than 200, and hence that the loop should terminate, which is achieved by the "IFNO 11" command in code line 3 which branches to the "HALT" at code line 11. (This structure ensures that counting loops within *Turtle* conform to the Pascal standard in not operating at all if the initial value of the loop variable is already beyond the termination value.) The body of the loop consists of code lines 4 to 9; then line 10 pushes the loop variable "count" back onto the Stack, increments it by 1 (leaving the incremented value on the Stack), and unconditionally jumps back to code line 2, where the new value gets stored and then tested, continuing on to the next iteration of the loop.

[38] It may seem wasteful to load the "count" value onto the Stack, then save it to the variable, only to reload it again after the termination value. The explanation for this will become clear shortly, when we see future iterations of the loop (via the "JUMP" command in code line 10) making use of the very same commands, with the important exception of the preliminary loading of the initial value.

Chapter 6 Turtle Machine Procedures

6.1 Procedure Calls, and the Return Stack

One of the most fundamental characteristics of procedures, which helps to make them so valuable in modular programming, is that they can be "called" from many different locations within the program code, *without* disrupting the local flow of control. A procedure acts as a self-contained unit, running its course and then immediately returning control to the location from which it was called, which can vary from occasion to occasion. Hence the implementation of procedures within PCode requires a form of flow management more flexible than the fixed jumps and branches discussed in the previous chapter. To cope with the potential flexibility of calling locations, the PCode must be able to save the appropriate return location when each procedure is called, and to jump back to it when the procedure terminates. A further complication is that procedures can be nested or recursive, with one procedure calling another, and that yet another, and so on without theoretical limit; thus numerous procedures, or instances of a single procedure, can all be active simultaneously. This in turn implies that the sequence of pending return locations – corresponding to the hierarchy of procedures currently in progress – cannot be stored in fixed addresses, but must be stored in a stack structure (last-in-first-out, since the last procedure to start is always the first to finish). The implementation of procedures therefore requires that the Turtle Machine should incorporate a Procedure Return Stack (or "Return Stack" for short).

Operation of the Procedure Return Stack clearly requires (at least) two PCode commands – one when each procedure (or, more accurately, procedure instance) is called, and one when it terminates. The command that calls a procedure must perform two distinct functions: transferring control to the code line corresponding to the start of the procedure's body, and pushing the appropriate return location (i.e. the code line from which the procedure was called) onto the Procedure Return Stack. The command that terminates the procedure should then pop this same return location from the Return Stack, and transfer control accordingly back to the (end of the) code line from which the procedure was called. These two commands are as follows:

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
call procedure	PROC	50	$32
end procedure	ENDP	51	$33

6.2 Local Variable Storage, the Heap Top Pointer, and Procedure Heap Pointers

§5.2 above explained how global variables are stored at the bottom of the Heap, and accessed (through the instructions "LDVG" and "STVG") by means of an index number that gives their "address" within the Heap. Local variables are also stored in the Heap, above the globals, but what makes their processing more complicated is that they do not reside there permanently, but are created and destroyed in turn as the procedure instance to which they belong first starts and then terminates.

The basic principle for referencing local variables is quite similar to that for globals, in that each variable has an index number which indicates its relative address within the appropriate area of the Heap. The major difference is that this area is not fixed, but determined dynamically when the procedure concerned begins. To achieve this, the system continuously maintains a "Heap Top Pointer" which keeps track of the top of the Heap – i.e. the top limit of that part of the Heap being used for active variable storage. Then whenever a procedure is called, the system takes note of the current value of this Heap Top Pointer, and stores it in association with that procedure; this storage address is called the "Procedure Heap Pointer" (or "Procedure Heap (Base) Pointer") for the procedure concerned.[39] The local variables of the new procedure instance are then stored immediately above the "Heap Base" address indicated by its Procedure Heap Pointer, in the order of their index codes (which correspond to the order of their

[39] Hence a program that contains three procedures will involve three Procedure Heap Pointers. It is not possible to make do with a single Procedure Heap Pointer, dynamically adjusted to whichever is the "current" procedure, because two (or more) procedures can be nested, in which case when the "inner" procedure is running, the local variables of both must be *simultaneously* accessible.

declaration within the procedure). The two instructions used to load these local variables onto the Program Stack, and to store them from the Program Stack, are as follows:[40]

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
load variable *(or value parameter)*	LDVV	18	$12
store variable *(or value parameter)*	STVV	34	$22

Suppose, for example, that a program has the five procedures "proc1" to "proc5", of which the first is called by the third, which is in turn called by the fifth.[41] Suppose also that the first procedure "proc1", currently running, has three local variables named "one", "two", and "three". Then assuming that the only other currently active procedures are "proc5" and "proc3", the context in which "LDVV" and "STVV" reference these variables can be pictured as shown in Figure 11 below. To load the second local variable of proc1 (i.e. "two") onto the Program Stack, the appropriate command is "LDVV 1 2" (where "1" indicates the first procedure in the program, and "2" the second local variable of that procedure). To identify the required address on the Heap, we simply take the Procedure Heap Pointer for the procedure in question (i.e. Procedure Heap Pointer 1), and add the index of the particular variable (i.e. 2). As Figure 11 shows, this indeed generates the correct address, and the same applies to the corresponding inverse command "STVV 1 2".[42]

---

[40] In these mnemonics, the first "V" stands for "variable", and the second for "value", because these are the commands pertaining to local variables and *value* parameters of procedures. Reference parameters need different commands, whose mnemonics are "LDVR" and "STVR" – see §6.4.2 below.

[41] In Turtle Graphics Pascal, the called procedure must precede the calling procedure, except when two procedures are nested, in which case mutual references are possible, cf. note 44 in §6.3 below.

[42] This handling of local variables can be seen as a natural extension of the method for global variables explained in §5.2. In effect, all global variables are indexed relative to a single Program Heap Pointer which points just below the entire Heap, whereas local variables are indexed relative to a Procedure Heap Pointer which is specific to the particular procedure within which they occur.

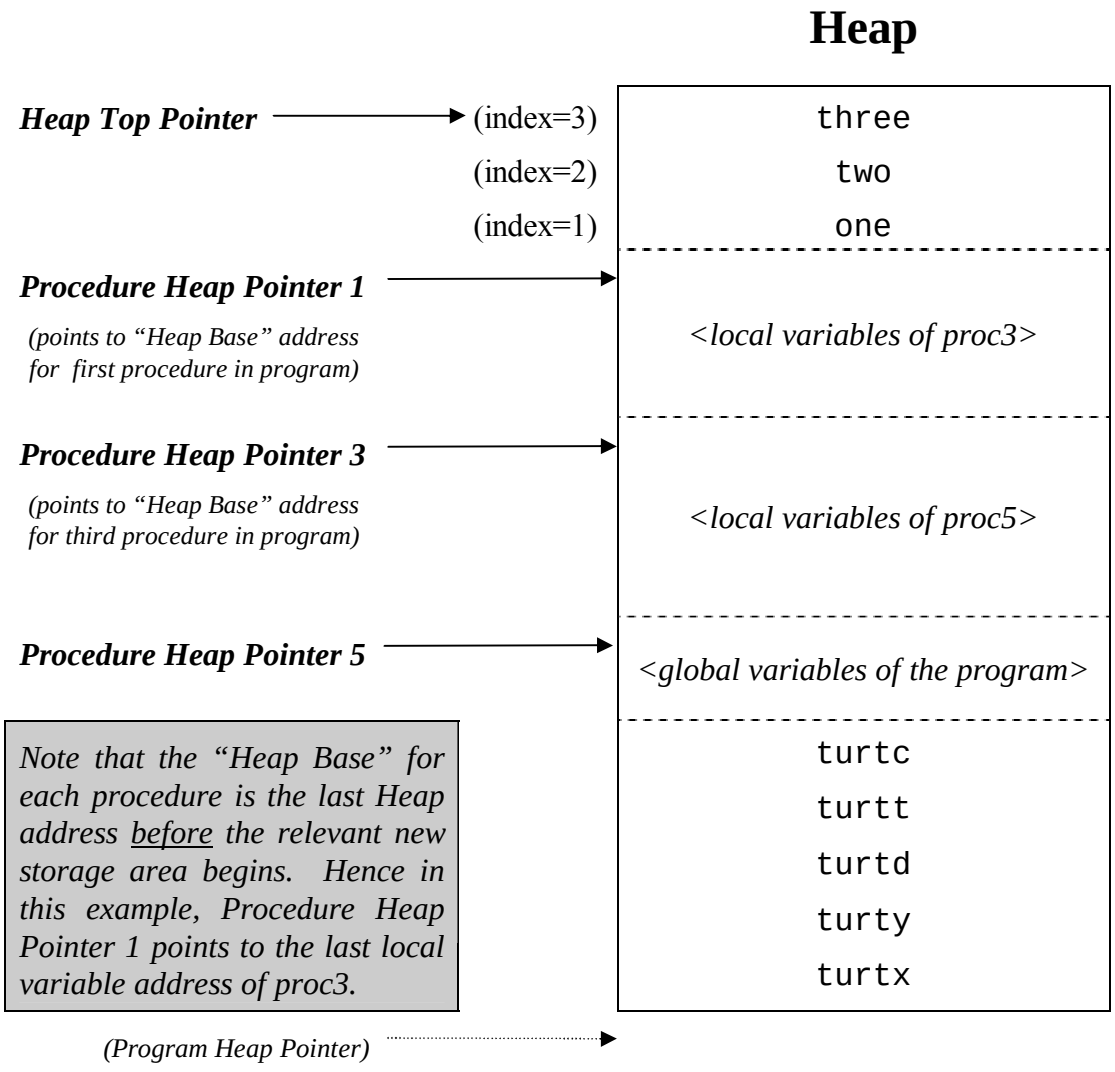


Figure 11: Illustrative Heap structure while three procedures are active

6.3 Claiming and Releasing Heap Space; the Heap Control Stack

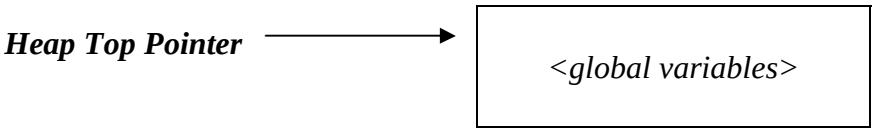
We must now deal with the details of how the Heap reference mechanism outlined above is to be maintained while a program is running. Clearly at least two more PCode instructions are needed, one ("HPCL") to "claim" space on the Heap for the local

variables of each new procedure,[43] and another ("HPRE") to "release" that space once the procedure has terminated:

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
claim space on Heap	HPCL	52	$34
release space on Heap	HPRE	53	$35

The intended behaviour of "HPCL" and "HPRE" can be deduced if we compare the situation before, and after, the first procedure call made within the hypothetical five-procedure program described in §6.2 above:

Situation prior to initial call of proc5:



Situation following call of proc5:

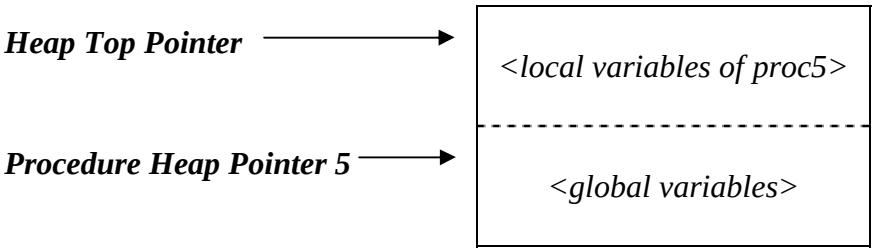


Figure 12: Top of Heap before, and after, a procedure call

Moving from the first situation to the second (i.e. claiming Heap space for local variables) is straightforward – when the procedure starts, its Procedure Heap (Base) Pointer should be given the current value of the Heap Top Pointer, then the Heap Top Pointer itself should be incremented by the number of local variables in the procedure.

[43] Strictly, procedure *instance*, because in a recursive program the same procedure can have many instances, each with its own local variables. For simplicity I shall here usually refer simply to "procedures", but take for granted that procedure *instances* are understood where appropriate.

The inverse operation, releasing the claimed Heap space when the procedure terminates, might seem to be equally straightforward, simply requiring that the Heap Top Pointer be reduced by the same amount that it was previously incremented. However this crude method will fail to restore the Procedure Heap Pointer also to its own previous value, which though unproblematic if the just-terminated procedure is no longer operative, is disastrous if *another instance* of that procedure is still active (a common situation if the procedure is recursive). The obvious refinement of this crude method would be to decrement the Procedure Heap (Base) Pointer also, by the same amount as the Heap Top Pointer, and this will indeed work if the only recursion in the program involves single procedures that call themselves. However a *Turtle* program can contain two procedures that are *mutually* recursive (as long as one is a sub-procedure of the other),[44] and in such a case, maintenance of the Heap structure through the possibly alternating recursive calls requires a far more robust approach.

The upshot of all this is that when a procedure is called and its Procedure Heap Pointer is updated accordingly, the *previous* value of this pointer must be saved, to be restored when the current instance of the procedure terminates. Since there is no limit in principle to the depth of recursion, and hence to how many such previous values may be involved, the most efficient way of storing them is in a stack structure. Rather than have a separate stack for each procedure, the most elegant way of achieving this is to maintain a single "Heap Control Stack", of which the Heap Top Pointer is the topmost value. Then the required result can be achieved by specifying the behaviour of "HPCL" and "HPRE" as follows:

HPCL: First, exchange the value currently on top of the Heap Control Stack – i.e. the Heap Top Pointer – with the Procedure Heap Pointer for the

---

[44] In standard Pascal, two procedures can be defined as mutually recursive using the *forward* directive, but this is not provided here because "forward" has a very different traditional meaning within Turtle Graphics. It would be possible to define an alternative directive (the word "deferred" might be appropriate), but since simple mutual recursion can already be achieved in *Turtle* by nesting the procedures, this complication (and the inevitable inconsistency with Pascal as implemented within *Delphi* and other systems) seems best avoided.

procedure concerned (this *both* sets the Procedure Heap Pointer appropriately, and *also* saves its previous value on the Heap Control Stack). Then push onto the Heap Control Stack the calculated new value for the Heap Top Pointer (which is equal to the old value plus the number of local variables in the procedure).

HPRE: Remove the top element of the Heap Control Stack (i.e. the Heap Top Pointer). Then exchange the new top element with the Procedure Heap Pointer for the procedure concerned (this restores the previously saved values of *both* the Heap Top Pointer *and* the Procedure Heap Pointer).

It should be clear from these descriptions that they are indeed inverse operations – "HPRE", thus defined, restores exactly the Heap situation prior to the corresponding "HPCL", and because all the relevant values are saved on the Heap Control Stack, this will remain true no matter how many recursive or other procedure calls are executed.

6.4 Dealing with Parameters

Pascal procedures can have two different types of parameter: those that are "called by value", and those that are "called by reference" (or "value parameters" and "reference parameters" for short). The syntactic difference between the two is minimal, in that the latter are simply prefixed by the keyword "VAR" (hence the informal term "VAR parameters"). But the two need to be handled by quite different mechanisms.

6.4.1 Value Parameters

Value parameters behave essentially as ordinary local variables, except that their initial value on entry to the procedure is set to the value of the corresponding actual parameter. (*Turtle* initialises other local variables, and all global variables, to zero – see note 46 in §6.6 below) So the only further issue that needs to be discussed here is how those initial values get passed into the procedure.

Given the structure of the virtual Turtle Machine, the obvious and natural way to pass parameters is via the Program Stack (this also avoids any need to limit artificially the number of parameters that can be passed, which would apply if a fixed number of registers were to be used). So the PCode sequence prior to calling a procedure with one parameter will be exactly as when calling a Turtle Graphics command with one parameter (cf. §5.2 above), namely, pushing the actual parameter value onto the Program Stack; then the procedure is called by transferring control to the appropriate codeline with "PROC". The same mechanism applies if there are more parameters, with these being pushed onto the Stack in their natural syntactic order (i.e. the first parameter gets pushed onto the Stack first). To match up with this, the PCode within the body of the procedure will start with one "STVV" command for each value parameter, except that now they must be dealt with in reverse order (because when the "PROC" command is performed, the last parameter will be at the top of the Stack – see §6.6 below for a detailed example of this last-in-first-out unpacking).

6.4.2 Reference Parameters

Reference parameters are significantly different in behaviour from value parameters, in that instead of specifying a *value* to be given to the corresponding formal parameter within the procedure, they specify a *variable* for which the formal parameter is to act as an "alias" within the procedure. Thus the actual parameter to the procedure must be a variable, and it is in effect the *address* of this variable that gets passed into the procedure, rather than its value.

It follows that handling reference parameters requires a completely different set of PCode instructions from those that apply to value parameters. The two most basic are:

Informal description of command behaviour	**Assembler mnemonic**	**Decimal code**	**Hex code**
load reference parameter	LDVR	19	$13
store reference parameter	STVR	35	$23

which within the procedure do more or less the same job for reference parameters that "LDVV" and "STVV" do for value parameters: that is, they provide access to the relevant formal parameter within the procedure's code lines, enabling that parameter to be loaded onto the Program Stack or saved from the Program Stack much like any normal variable. Obviously the method by which "LDVR" and "STVR" operate, however, is very different, because in order to identify which address on the Heap should be loaded or stored (respectively), a process of *indirection* needs to take place, whereby the actual parameter's storage address is looked up in a "register", where it will be indexed under the formal parameter which is serving as its alias. This register cannot be "hard wired", because one and the same procedure could be called from many different places in the program, with a wide variety of different variables providing the actual input parameter (implying that the same formal parameter would be acting as an "alias" for different actual variables on these different occasions). So the obvious way of dealing with this is to assign a local variable storage address for each reference parameter, just as for value parameters, except that when these addresses are accessed using "LDVR" and "STVR" (as opposed to "LDVV" and "STVV"), they will be interpreted as registers containing *indirection addresses*, rather than as storage locations for variables themselves. So if, in the situation we were considering earlier (at page 57), parameters *one* and *three* of the first procedure were reference rather than value parameters, then the top of the Heap would be as follows:

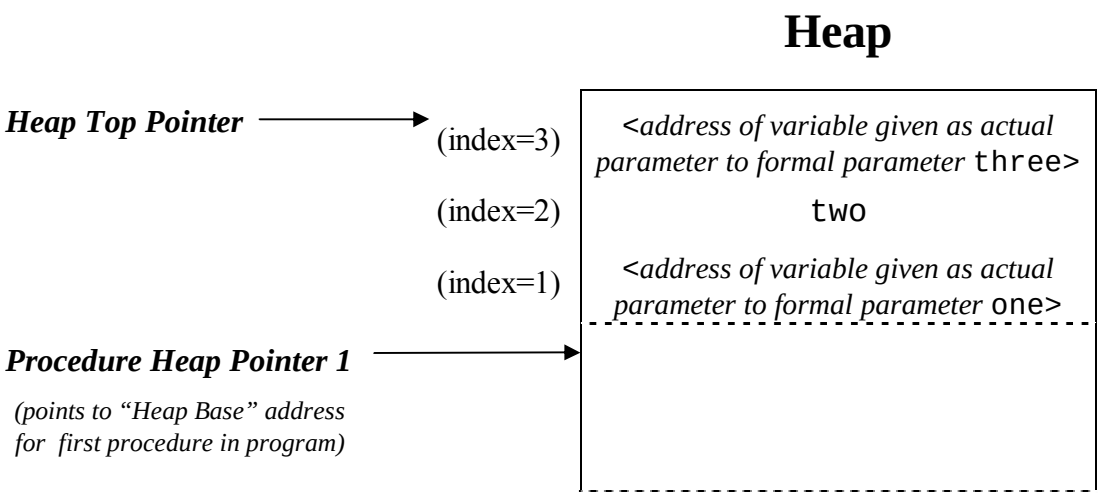


Figure 13: Top of Heap where procedure involves reference parameters

Then "LDVR 1 3", for example, will work by first identifying the required indirection register on the Heap, calculating this by taking the Procedure Heap Pointer for the procedure in question (i.e. Procedure Heap Pointer 1), and adding the index of the particular variable (i.e. 3). But then instead of simply loading onto the Program Stack whatever value – 5, say – it finds in that address (as would happen in the case of "LDVV 1 3"), it instead proceeds to interpret this value as itself *an indirected Heap address*, and accordingly loads the value which is to be found in the corresponding Heap location – in this case, at address 5 (which is the address of the *turtc* value, so this is the situation that will arise where *turtc* is the actual parameter corresponding to the formal parameter "three"). "STVR" acts very much like "LDVR", except that it *stores into* the indirected Heap address rather than *loading from* it.

The other PCode instructions that handle reference parameters are used for setting up the appropriate aliases when a relevant procedure is called:

Informal description of command behaviour	**Assembler mnemonic**	**Decimal code**	**Hex code**
load address of global variable	LDAG	20	$14
load address of local variable *(or value param)*	LDAV	21	$15
load indirected address of reference parameter	LDAR	22	$16
store indirected address of reference parameter	STAR	38	$26

The first three of these are used to load the Heap address of the relevant actual parameter variable onto the Program Stack, for transfer into the procedure body. Three instructions are needed because a procedure call can occur not only from the main program but also from another procedure, one of whose own variables or parameters might therefore be "aliased" by the reference parameter within the called procedure. Hence the actual input parameter could be any one of a global variable ("LDAG"),[45] a local variable or value parameter of the calling procedure ("LDAV"), or a reference parameter of the calling procedure ("LDAR"). Whichever of these is involved,

[45] "LDAG" is in fact equivalent to "LDIN", since the Heap address of the n^{th} global variable is just n. But for the sake of clarity, the two uses are differentiated by using separate instructions.

however, the same instruction ("STAR") is used within the procedure code to store the actual parameter's Heap address in the procedure's corresponding formal parameter "indirection register".

6.5 The Procedure Register Stack

We saw earlier (in §2.3) that the *Turtle* system incorporates a "trace" facility which displays each PCode command as it is executed, together with relevant contextual information including the numeric index of whichever procedure (if any) is running at any particular time, and how many procedures are active. However the instructions introduced so far are not designed to keep track of this information, and it cannot be deduced by simply counting how many procedure calls have been made and recording the last such call. For as the middle line of the following sequence illustrates, the current procedure is not always the last to have started, nor the next to finish:

proc2 starts
    proc1 starts and finishes
(proc2 running)
    proc1 starts and finishes
proc2 finishes

The simplest solution to this bookkeeping problem is to maintain yet another stack, the "Procedure Register Stack", whose topmost value (the "Procedure Register") is set according to the procedure which is currently running, and whose height indicates how many procedures are active. The operation of this stack is very straightforward: when a procedure begins, its number is pushed onto the stack, and whenever a procedure ends, the top value is pulled from the stack, using the following two instructions:

Informal description of command behaviour	**Assembler mnemonic**	**Decimal code**	**Hex code**
push procedure index onto Procedure Register Stack	PSPR	54	$36
pull procedure index from Procedure Register Stack	PLPR	55	$37

6.6 Putting Procedures Together

We can now see how all this fits together in compiling a program with a fairly complex procedure interface. This example produces the effect of a ball bouncing back and forth horizontally, while performing a slow "random walk" vertically:

```
PROGRAM randomdrift;
VAR bgcol,xpos,xvel,ypos: integer;

 PROCEDURE doball(size,bgc,col: integer;
                  VAR xp,xv,yp: integer);   1. PSPR 1  HPCL 1 7
 VAR ymove: integer;                        2. ZERO 1 7  STAR 1 6  STAR 1 5  STAR 1 4  ...
 BEGIN                                         ...  STVV 1 3  STVV 1 2  STVV 1 1
  randcol(7);                               3. LDIN 7  RNDC
  ymove := turtc-4;                         4. LDVG 5  LDIN 4  SUBT  STVV 1 7
  setxy(xp,yp);                             5. LDVR 1 4  LDVR 1 6  TOXY
  colour(bgc);                              6. LDVV 1 2  COLR
  blot(size);                               7. LDVV 1 1  BLOT
  xp := xp+xv;                              8. LDVR 1 4  LDVR 1 5  PLUS  STVR 1 4
  yp := yp+ymove;                           9. LDVR 1 6  LDVV 1 7  PLUS  STVR 1 6
  setxy(xp,yp);                            10. LDVR 1 4  LDVR 1 6  TOXY
  colour(col);                             11. LDVV 1 3  COLR
  blot(size);                              12. LDVV 1 1  BLOT
  update;                                  13. UDAT
  noupdate;                                14. NDAT
  if (xp<size) or (xp>1000-size) then      15. LDVR 1 4  LDVV 1 1  LESS  LDVR 1 4  LDIN 1000  ...
                                               ... LDVV 1 1  SUBT  MORE  OR  IFNO 17
    xv := -xv                              16. LDVR 1 5  NEG  STVR 1 5
 END;  {procedure doball}                  17. HPRE 1  PLPR  ENDP

BEGIN  {main program}
 bgcol := $5000;                           18. LDIN 20480  STVG 6
 blank(bgcol);                             19. LDVG 6  BLNK
 xpos := 30;                               20. LDIN 30  STVG 7
 xvel := 1;                                21. LDIN 1  STVG 8
 ypos := 500;                              22. LDIN 500  STVG 9
 repeat
   doball(30,bgcol,yellow,xpos,xvel,ypos)  23. LDIN 30  LDVG 6  LDIN 65535  LDAG 7  ...
                                               ...  LDAG 8  LDAG 9  PROC 1
 until (ypos<0) or (ypos>1000)             24. LDVG 9  LDIN 0  LESS  LDVG 9 ...
                                               ...  LDIN 1000  MORE  OR  IFNO 23
END.  {main program}                       25. HALT
```

This program is adapted from one of the built-in illustrative programs, which uses reference parameters to enable a single "doball" procedure to handle the motion of two (or more) balls that are bouncing around the screen, each with its own stored set of position and velocity x- and y-components. Only one ball is included here, but because the procedure is designed in this way, it would be easy to extend the program to include more balls, perhaps to make a simple betting game ("Which ball will reach the top or bottom of the Canvas first?"). Note also the "ymove" local variable, which is set randomly (via "randcol") to a value between plus and minus 3 to change the ball's y-coordinate accordingly, thus generating the ball's vertical random walk.

When run, the PCode starts executing at code line 18, which sets the global variable "bgcol" to an appropriate background colour – here the hexadecimal colour code $005000 (20480 in decimal) specifies $50/$FF (80/255 in decimal) intensity of green, with zero of blue and red (see §3.4.2). Then code line 19 uses this global variable to "blank" out the Canvas accordingly, after which lines 20 to 22 set up the starting values of the other three global variables (§5.2).

The body of the "repeat" loop is just code line 23, which executes the procedure call to "doball". Code line 24 then performs a conditional branch back to line 23, unless the condition "(ypos<0) or (ypos>1000)" evaluates as true (the mechanism by which a repeat loop achieves the required conditional branching is explained in §5.3.3). Most of the PCode in line 24 is devoted to evaluating the relevant condition, operating in standard reverse Polish fashion using the Program Stack (§§5.1-2).

Coming back now to code line 23, this simply loads the various actual procedure parameters, in order, onto the Program Stack, before performing "PROC 1" to call the procedure which starts at code line 1. The first and third actual parameters are numerical constants (yellow being $FFFF in hexadecimal, 65535 in decimal); so these are loaded onto the Stack using "LDIN". The second is a global variable serving as a *value* parameter to the procedure, so its current value is also simply loaded onto the Stack using "LDVG". But the last three actual parameters, by contrast, are global variables corresponding to *reference* ("VAR") parameters within the procedure, so it is not their *values* that get loaded onto the Stack, but rather, their *Heap addresses*: this requires use of the instruction "LDAG" rather than "LDVG". Once all these Stack

operations have been completed, the "PROC 1" command at the end of code line 23 transfers control to line 1, but before doing so (as explained in §6.1), it first saves the appropriate return location (in this case, 23) on the Procedure Return Stack, so that when the procedure terminates with "ENDP", the Turtle Machine will have a record of where execution should continue (in this case, from the end of code line 23, hence code line 24 will be the next to be executed after the procedure terminates).

By the time procedure "doball" is called, the condition of the Stack is as follows:

Stack position (top = 1)	**Item on the Program Stack**	**Corresponding formal parameter within procedure**
1.	Absolute Heap address of global variable "ypos" (= 9)	yp (reference parameter)
2.	Absolute Heap address of global variable "xvel" (= 8)	xv (reference parameter)
3.	Absolute Heap address of global variable "xpos" (= 7)	xp (reference parameter)
4.	Integer 65535 (or $FFFF) – the colour code of yellow	col (value parameter)
5.	Current value of global variable "bgcol" (= 20480 or $5000)	bgc (value parameter)
6.	Integer 30	size (value parameter)

Note that *all six* items on the Stack are integers, but the top three are destined to be used as Heap addresses rather than as integer values in their own right. To see how this happens, we must now turn our attention to the code lines of the procedure itself.

The first thing that happens after procedure "doball" is entered (at code line 1) is that "PSPR 1" pushes the value of 1 onto the Procedure Register Stack, the 1 here signifying that the *first* procedure in the program is now "active". Then "HPCL 1 7" claims space on the Heap for the seven local "variables" of this first procedure – these are the six formal parameters in the table above, plus the one genuine local variable "ymove". "HPCL 1 7" does this by adjusting the various Heap pointers as described in

§6.3 above; this ensures in particular that Procedure Heap Pointer 1 is set up to point just below the seven Heap locations that "doball" needs in order to hold (or, in the case of reference parameters, to indirect) its seven local variables.

With the necessary Heap locations now claimed, code line 2 performs the task of "unpacking" the Program Stack, so that all seven local variables are appropriately set up. As already mentioned in §6.4.1, this unpacking must take place in reverse order because of the last-in-first-out stack structure. For neatness the same reverse ordering is extended to genuine local variables, although these do not require any use of the Stack – hence the first operation in code line 2 is the one that deals with "ymove", the seventh local "variable", whose initialisation to 0 involves a new PCode instruction:[46]

Informal description of command behaviour	**Assembler mnemonic**	**Decimal code**	**Hex code**
zero local variable	ZERO	39	$27

Thus "ZERO 1 7" sets the seventh local variable of the first procedure to 0, without requiring any values to be passed via the Program Stack. This command is not strictly necessary for the operation of the Turtle Machine, since a sequence such as "LDIN 0 STVV 1 7" would have a similar result, but use of "ZERO" significantly shortens the initialisation sequence for procedures that have genuine local variables.

The rest of code line 2 involves "unpacking" from the Stack either variable addresses (in the case of the three reference parameters) or integer values (in the case of the three value parameters). Variable addresses are unpacked using the PCode instruction "STAR" (cf. §6.4.2), while integer values are unpacked using "STVV". By this process the six items from the Program Stack shown in the table above are all stored appropriately for the procedure to make use of them.

---

[46] The Turtle Machine initialises all variables to 0, except for the five predefined globals. Here it differs from *Delphi*, which initialises global variables to 0 but leaves local variables initially undefined. Although students should be taught that it is good practice to initialise all variables explicitly, the confusion that can result from undefined variables (and resulting inconsistent program behaviour) seems to make automatic initialisation most appropriate in a system designed for novice programmers.

The main body of the procedure can now get under way, working through the following sequence (the numbers at the left refer to code lines of the compiled PCode):

3-4: "ymove" is set to a random value between −3 and +3

5-7 a background-coloured blot is drawn using the x- and y-coordinates "xp" and "yp" (which are aliases for the globals "xpos" and "ypos")

8-9 "xp" and "yp" are adjusted by the required horizontal ("xv") and vertical ("ymove") movement, but since "xp" and "yp" are aliases for "xpos" and "ypos", it is really these two global variables that are changed.

10-12 a yellow blot is drawn using the new x- and y-coordinates "xp" and "yp" (which, yet again, are aliases for "xpos" and "ypos")

13-14 screen updating takes place only here in the procedure, aiming to give an illusion of smooth motion rather than flashing as the blots are drawn

15-16 a test is performed to see whether "xp" (alias for "xpos") is within a ball's radius of the side of the Canvas; if so, then "xv" (alias for "xvel") is inverted so that the ball will move back in the opposite direction

Code line 17, which ends the procedure, contains three commands. First, "HPRE 1" releases the Heap space previously claimed by "HPCL 1 7" (as explained in §6.3); then "PLPR" pulls the procedure index from the Procedure Register Stack (§6.5); finally "ENDP" returns control to the main program, pulling the code line from which the procedure was called (here, 23) from the Procedure Return Stack. Hence execution then continues from code line 24, where the test is made to determine whether the "repeat" loop should continue. Eventually "ypos" will randomly drift to be less than 0 or more than 1000, at which point the "IFNO" branch back to line 23 will cease to operate, and the program will terminate when it reaches the "HALT" instruction at code line 25.

6.6.1 Note on the Trace Facility

In closing this discussion, recall (from §2.3 and §6.5) that *Turtle* contains a trace facility which can display in detail the execution steps of any program, including all transfer of data via the Stack. This trace can be used with the program above to inspect the mechanisms described, but before doing so, it is helpful to modify the "until" condition to "until turtx>=33", to restrict the repeat loop to three procedure calls. Each procedure call involves around 50 program "cycles" – i.e. individual PCode commands – so allowing the unmodified program to run will quickly build the trace display to an unmanageable length (with very evident effects on program speed).

6.7 Stack Variations on the Turtle Machine

As described so far in this chapter and the last, the Turtle Machine involves no fewer than four stack structures:

- The main Program Stack (§5.1), used for holding command parameters, calculating intermediate values, and parameter passing;
- The Procedure Return Stack (§6.1), used for storing return code line locations for program continuation after each procedure terminates;
- The Heap Control Stack (§6.3), used for managing Heap pointers that indicate where each procedure's dynamic storage is to be found;
- The Procedure Register Stack (§6.5), used to keep track of the active procedures, in order to service the trace display facility.

These stacks are made independent in the default version of the software, to enable each mechanism to be introduced separately in an educational context, but only the first of them is genuinely necessary, and the Compile menu provides three options which respectively enable each of the others to be dispensed with (involving some new PCode instructions). These options are intended to facilitate teaching of the concept of a stack frame, since they illustrate how a variety of data relevant to a procedure call (including not only parameters, but also return location and heap pointers) can all be combined into a "frame" on the Stack, and viewed as such through the trace display facility, which

shows the contents of the top three Program Stack locations at every "cycle" of the program.[47]

To explain the differences between the options, we can see how they affect the compilation of a short program with one (recursive) procedure. The procedure draws a coloured blot, then moves forward (hence the name "bf") and turns before calling itself with an incremented parameter. The visual effect is of nine "pillars":

```
PROGRAM pillars;

 PROCEDURE bf(r: integer);      1.  PSPR 1  HPCL 1 1
 BEGIN                          2.  STVV 1 1
  if r<350 then                 3.  LDVV 1 1  LDIN 350  LESS  IFNO 9
    begin
     randcol(6);                4.  LDIN 6  RNDC
     blot(r-5);                 5.  LDVV 1 1  LDIN 5  SUBT  BLOT
     forward(r*2);              6.  LDVV 1 1  LDIN 2  MULT  FWRD
     right(40);                 7.  LDIN 40  RGHT
     bf(r+1)                    8.  LDVV 1 1  LDIN 1  PLUS  PROC 1
    end
 END;                           9.  HPRE 1  PLPR  ENDP

BEGIN
 blank(black);                  10. LDIN 0  BLNK
 penup;                         11. PNUP
 movexy(-140,40);               12. LDIN 140  NEG  LDIN 40  MVXY
 bf(50)                         13. LDIN 50  PROC 1
END.                            14. HALT
```

In the default situation, shown here, the procedure starts with PSPR + HPCL and ends with HPRE + PLPR + ENDP; the parameter is passed through the Program Stack (at code lines 8 and 13), but no other information is passed by that route.

---

[47] Note also the option provided through the Compile menu, to display the height of the Stack (rather than the contents of the third Stack location) in the last trace column.

6.7.1 Doing Without the Procedure Return Stack

Now suppose that we opt to compile the program without use of the Procedure Return Stack; this means that the procedure "return jump" location must be passed via the Program Stack, for which we use the "LDRJ" and "PLRJ" instructions respectively to "load" and "pull" it:[48]

Informal description of command behaviour	Assembler mnemonic	Decimal code	Hex code
load procedure return jump	LDRJ	224	$E0
pull (and make) return jump	PLRJ	225	$E1

Since however the address is intended to remain on the Stack after the parameters have been "unpacked", it is best to load it before the parameters. This happens twice, once when the procedure is called recursively (at code line 8), and once when it is called from the main program (at code line 13):

```
PROCEDURE bf(r: integer);     1.  PSPR 1  HPCL 1 1
BEGIN                         2.  STVV 1 1
  . . .                       ...
    bf(r+1)                   8.  LDRJ  LDVV 1 1  LDIN 1  PLUS  PROC
                                  1
  end
END;                          9.  HPRE 1  PLPR  PLRJ

BEGIN
 . . .                        ...
 bf(50)                       13. LDRJ  LDIN 50  PROC 1
END.                          14. HALT
```

[48] In the same spirit as note 45 above, it is perhaps worth noting that the "PLRJ" instruction could be used to replace "JUMP" if the aim were to minimise the instruction set, since "JUMP 10", for example (which jumps to *the beginning* of code line 10), is equivalent to "LDIN 9 PLRJ" (which jumps to *the end* of code line 9).

The only other change is that the procedure here ends with "PLRJ" rather than "ENDP" – procedure termination has to involve a different command if the machine is to continue execution from the return location stored on the Program Stack (the effect of "PLRJ") rather than looking for the return location on the Procedure Return Stack.[49]

6.7.2 Doing Without the Heap Control Stack

Managing Heap space without the Heap Control Stack is a relatively complex business, requiring that the relevant Heap pointers be passed and calculated on the Program Stack rather than being controlled through a specially tailored mechanism. The Heap Top Pointer is retained as a single register (rather than as the top of a stack), and each procedure still has its own Procedure Heap Pointer, keeping track of the relevant Heap Base address. But now these must be maintained explicitly, using the instructions "LDHT" and "STHT", which respectively load and store the current Heap Top Pointer, and "LDHB" and "STHB", which load and store the Procedure Heap (Base) Pointer for a specified procedure:

Informal description of command behaviour	**Assembler mnemonic**	**Decimal code**	**Hex code**
load Heap Top Pointer	LDHT	226	$E2
store Heap Top Pointer	STHT	227	$E3
load Heap Base Pointer	LDHB	228	$E4
store Heap Base Pointer	STHB	229	$E5

This Heap maintenance is done as follows: (Some code line numbers are changed, but for convenient comparison, the *previous* numbering is shown, in brackets.)

[49] For the sake of simplicity the procedure call still involves "PROC", which adjusts the Procedure Return Stack, so that stack is in fact maintained. However it is not *used* – "PLRJ" removes the return location from this stack (to prevent overflow) but just discards it, using the Program Stack value instead.

```
PROCEDURE bf(r: integer);     1.  PSPR 1  LDHT  STHB 1
                                  LDHT  LDIN 1  PLUS STHT
BEGIN                         (2) STVV 1 1
  . . .                       ...
   bf(r+1)                    (8) LDHB 1 LDHT
                                  LDVV 1 1  LDIN 1  PLUS  PROC 1
                                  STHT  STHB 1
  end
END;                          (9) PLPR  ENDP

BEGIN
. . .                         ...
bf(50)                        (13) LDHT
                                   LDIN 50  PROC 1
                                   STHT
END.                          (14) HALT
```

At the start of the procedure, "HPCL 1 1" is replaced by a command sequence which explicitly sets Procedure Heap Pointer 1 to the current value of the Heap Top Pointer ("LDHT STHB 1"), and then adds 1 to the value of the Heap Top Pointer itself ("LDHT LDIN 1 PLUS STHT") – this "claims" space on the Heap for the one local variable/parameter "r". The process of releasing this space is more complex and must be done outside the procedure itself, because it depends on whether the procedure is called from a procedure (as at code line "8") or from the main program (as at code line "13"). In the latter case, the only thing needed to release the space is to reset the Heap Top Pointer to the value it had prior to the procedure call; this is easily done by pushing that value onto the Program Stack before the procedure is called (the "LDHT" in code line "13"), and then pulling it from the Stack to reset the Heap Top Pointer after the procedure has terminated (the "STHT" in code line "13"). In the former case, however, where the procedure call is itself within a procedure, it is essential to save and restore *both* the Heap Top Pointer and *also* the Procedure Heap (Base) Pointer of the calling procedure; hence the sequence "LDHB 1 LDHT ... STHT STHB 1" in code line "8", bracketing the procedure call.

The options discussed in these last two sections can be combined, so that *both* the procedure return location *and* the Heap pointers are handled through the Program Stack rather than using Procedure Return and Heap Control Stacks. In this case the relevant lines from the previous example would be modified as follows:

```
    bf(r+1)              (8)  LDHB 1 LDHT
                              LDRJ  LDVV 1 1  LDIN 1  PLUS  PROC 1
                              STHT  STHB 1
  end
END;                     (9)  PLPR  PLRJ
...                      ...
bf(50)                   (13) LDHT
                              LDRJ  LDIN 50  PROC 1
                              STHT
```

The only complication here is that the Heap pointers must be loaded onto the Program Stack (using "LDHB" and "LDHT") *before* the procedure "return jump" location is loaded (with "LDRJ"), enabling this location to be "pulled" at the end of the procedure (by "PLRJ") without affecting the Heap pointers.

6.7.3 Doing Without the Procedure Register Stack

Since the Procedure Register, and the stack that maintains it, are used by the Turtle Machine only to enable procedures to be "traced", the "PSPR 1" and "PLPR" commands can simply be removed from the compiled PCode without significantly affecting it. When it is then run, the only apparent difference (apart from a very small increase in speed) will be within the trace display, where the "Proc" column will continuously show "0/0" instead of, for example, "1/1", "1/2", "1/3" etc. depending on how many copies of the procedure are active.

In Chapter 7, we move on to the Turtle system's compiler, examining the mechanisms that it uses to translate the Pascal source language presented in Chapter 3 into the Turtle Machine code structures explained in Chapters 5 and 6.

Chapter 7 The *Turtle* Compiler

Having examined the virtual Turtle Machine, we must now turn to the *Turtle* compiler, whose job is to turn source Pascal code (as described in Chapter 3) into *Turtle* "machine code" or PCode (as described in Chapters 5 and 6). The compiler involves nearly 2,000 lines of *Delphi* Pascal,[50] so obviously it would be inappropriate here to attempt to consider it all in detail. Hence this chapter will focus only on the compiler's overall structure and processes, and on points of sufficient generality to be of educational interest independently of the specific environment and programming language used. It aims to provide not only an *explanation* of the compiler's processes, but also an *illustration* of how the system can provide a vehicle for presenting compiling techniques to students in a relatively accessible manner, without any ascent to the level of abstraction typically demanded by textbooks on compilation theory.

7.1 Standard Compilation Subtasks

Standard accounts of compilation typically divide the process conceptually into some combination of the following set of subtasks:[51]

(a) **Lexical analysis**, handled by a "scanner" which decomposes the source text into individual lexical units (e.g. treating ">=", and whole words, as single such units).

(b) **Screening**, which refines the output of the lexical analysis by processes such as the elimination of units that are not part of the execution code (e.g. separators and comments), and distinguishing between source words that are mere *identifiers*, and those that are *keywords* or *reserved words* of the language (e.g. "begin").

---

[50] Out of a system total of around 6,500 lines of *Delphi* Pascal, plus another 1,000 lines or so of interface (i.e. Windows "form") specifications.

[51] See for example Aho and Ullman [3], ch. 1; Bornat [14], ch. 1; Wilhelm and Maurer [137], ch. 6. The description here is based mainly on the last of these, which is the most detailed and up-to-date, incorporating various terminological distinctions (e.g. between a scanner and a screener, deriving from DeRemer [37]) that have become widespread since the earlier books were written.

(c) **Syntax Analysis**, which parses the output of the screener to produce a syntax tree for the source program, determining its syntactic structure. This tree may be stored explicitly within the compiler, or it may be an implicit abstraction from the compiler's behaviour as it traverses the source code, processing as it goes.

(d) **Semantic Analysis**, "decorating" the syntax tree with semantic information such as type assignments, and performing appropriate checks for consistency.

(e) **Machine-Independent Optimisation**, aiming to perform transformations on the decorated syntax tree to increase the efficiency of the resulting program.

(f) **Address Assignment**, the first of the "synthesis" (as opposed to "analysis") stages that take account of the specific machine architecture, here involving such considerations as word length and the potential for data packing.

(g) **Code generation**, which finally creates the object code for the target machine, selecting appropriate command sequences and use of internal registers etc.

(h) **Machine-Dependent Code Improvement**, focused mainly on local optimisation rather than the sort of global considerations typically involved at stage (e).

For reasons of efficiency and convenience, however, these processes are not usually separated into independent sequential modules, and traditionally a common approach has been to aim for a *two-pass* compiler (e.g. Bornat [14], p. 17), in which the first "pass" through the program performs the bulk of the "analysis" – phases (a) to (d) if not (e) also – and then the second "pass" performs the "synthesis" that remains to be done. However what mainly drives the separation of the passes here is not so much the conceptual distinction between analysis and synthesis, but rather the separation of machine-independent from machine-dependent considerations, to avoid the runaway complexity that would result from attempting to deal with both of these together where they interact, notably in the optimisation stages (e) and (h). Where no code optimisation is required, it is often simplest to perform the synthetic code generation together with the analysis, resulting in a single-pass compiler.[52]

---

[52] See for example Wilhelm and Maurer [137], p. 232. Even a single-pass compiler is unlikely to process the program purely sequentially, since there is often a need for *backpatching* (e.g. Aho and Ullman [3], p. 9). Typical examples of such backpatching occur within the *Turtle* compiler when control structures are terminated (described in §7.5 below), and also when a compiling "sub-procedure"

7.2 The Structure of the *Turtle* Compiler

The *Turtle* compiler, like the *Turtle* virtual machine, aims as its first priority for easy comprehensibility rather than efficiency, so for example no attempt is made to optimise the code at any stage, and all source language constructs are consistently "translated" in the same manner (as outlined in Chapters 5 and 6). Moreover since it handles only integer data, there is no need for type checking (cf. §3.1.2 above) or any other kind of semantic processing intermediate between syntactic analysis and code synthesis. All this enables code generation to be integrated very easily into the syntax analysis, so that a one-pass compiler would be fairly straightforward to implement. However in order to facilitate the creation of lexeme-indexed analysis tables that are transparently accessible (see §§7.3-4 below), the *Turtle* compiler operates in two passes, with the first combining lexical analysis and screening (subtasks (a) and (b)), after which the initial analysis tables can be created, while the second pass integrates syntax analysis, address assignment, and code generation (subtasks (c), (f) and (g)) in a manner that avoids any need for an explicitly stored syntax tree. Again partly for educational reasons, this second pass is itself modularised into three fairly independent levels, each being handled by a different mechanism which can be learned and understood separately as complementary compiling techniques:

	Compilation subtask	*Mechanism involved in subtask*
PASS 1	**Lexical Analysis**	Simple finite state machine (FSM)
§7.3	**Screening**	Tokenisation
PASS 2	*Parsing and Code Generation for:*	
§7.4	**Program Block Structure**	Complex FSM plus counter
§7.5	**Control Structures**	Pushdown automaton (PDA)
§7.6	**Commands and Expressions**	Recursive descent

---

makes reference to a (parent) procedure that has not yet itself been compiled. Grune et al. [47] point out that increased memory capacity has recently led to the development of "broad" compilers, which read the entire program and then transform it, removing any need for multiple passes (pp. 26-7).

7.3 Lexical Analysis and Screening

The first pass of the compiler involves the lexical analyser (function *lexanalyse*), which works through each line of the source code identifying the lexical items ("lexemes") it contains, and assigning an appropriate lexical type to each such item. Categorising the lexemes in this way greatly facilitates the later processes, by removing numerous complications. For example at this stage the Pascal "keywords" or "reserved words" are picked out (since the lexical analyser is integrated with a "screener" cf. §7.1 above), so that from now on the compiler, when checking for, say, the keyword "else" to match with a previous "if … then", does not need to look for the four characters "e" "l" "s" "e" (with no other contiguous letters or digits etc.), but need only check for a single lexeme of type *ltElse*. Because the keywords are recognised at the beginning of compilation, and henceforth treated as "tokenised" lexemes rather than as strings, they cannot be used in any other capacity (e.g. "from" cannot be an "identifier" and hence cannot be the name of a variable); indeed it is this fixedness of interpretation that characterises a Pascal "reserved word".[53]

Each keyword has its own lexical type; thus "and" is of type *ltAnd*, "begin" is of type *ltBegin* and so on through "do", "downto", "else", "end", "for", "if", "mod", "not", "or", "procedure", "program", "repeat", "then", "to", "until", "var", "while", and "xor". The other lexical types are listed in the following table:

---

[53] Although the terms "keyword" and "reserved word" are often (as here) used interchangeably, there is a subtle difference between them in the case of a language which prevents certain words from being used as identifiers (i.e. they are reserved) but which does not use them itself (so they are not strictly "keywords"). For simplicity the *Turtle* system reserves only its own keywords, though there would be an argument for reserving also such words as "case", "function" or "while" that are both reserved words and keywords within *Delphi* and other full implementations of Pascal.

ltSemicolon	;
ltComma	,
ltLbkt	(
ltRbkt	)
ltColon	:
ltAsgn	:=
ltPlus	+
ltMult	*
ltDivide	/

ltMinus	− (ambiguous)
ltSubt	− (binary)
ltNeg	− (unary)
ltEqual	=
ltNotEq	<>
ltLess	<
ltMore	>
ltLessEq	<=
ltMoreEq	>=

ltDot	. (after final "end")
ltInt	decimal integer
ltHex	hexadecimal integer
ltIdK	identifier or keyword
ltId	identifier
ltNull	no type yet assigned
ltError	illegal character

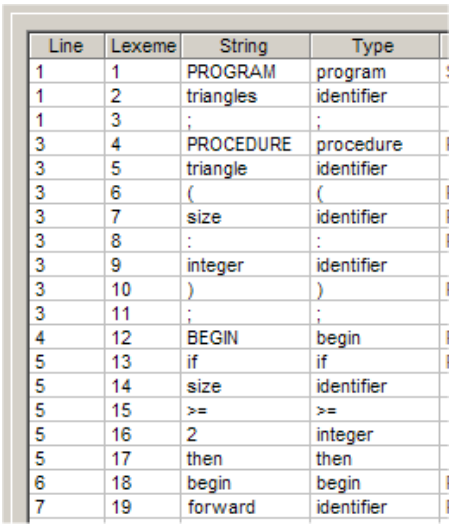
{comments} are completely ignored; illegal characters generate an error

The process that assigns these lexical types is very straightforward, reading through the source text one character at a time, and sending each character through a modified finite state machine structure which either makes an immediate decision about the lexical type (e.g. in the case of semicolons and brackets), or else moves to an appropriate "pending" state to await the next character(s) and process accordingly. When a "<" is encountered, for example, the FSM moves to state *stLess*, in which:

- if the next character is either ">" or "=", it is linked together with the "<" as a single lexeme of type *ltNotEq* ("<>") or *ltLessEq* ("<=") respectively;
- otherwise, the "<" is recorded as of type *ltLess*.

In the former case, the next character to be sent through the FSM will be the one following ">" or "="; in the latter case, the character just tested (as not being either ">" or "=") will itself be resent through the FSM. But either way, the FSM reverts back to the default initial state (*stNew*) before continuing.

As shown on the right here, the lexical analysis results can be inspected, after compilation has taken place, in the "String" and "Type" columns of the "Syntax" table which is one of the Visual Compiler displays (introduced in §2.3 above). In this table the types are usually shown as the literal expressions themselves (e.g. ">=" rather than "ltMoreEq"), but where different expressions can be of the same type, that type is indicated using a standard word such as "identifier" or "integer".



Line	Lexeme	String	Type
1	1	PROGRAM	program
1	2	triangles	identifier
1	3	;	;
3	4	PROCEDURE	procedure
3	5	triangle	identifier
3	6	(	(
3	7	size	identifier
3	8	:	:
3	9	integer	identifier
3	10	)	)
3	11	;	;
4	12	BEGIN	begin
5	13	if	if
5	14	size	identifier
5	15	>=	>=
5	16	2	integer
5	17	then	then
6	18	begin	begin
7	19	forward	identifier

Figure 14: Part of the "Syntax" table

Some lexical types involve various complications, for example *ltInt* and *ltHex* must be sequences of *valid* decimal or hexadecimal digits, and generate error messages if illegal characters appear in the sequence (most other error messages produced by the lexical analyser concern characters that are illegal wherever they appear, such as "&"). But the most complicated type to deal with is *ltIdK*, a temporary place-holder (in effect filling the gap between pure syntax analysis and screening) which indicates a sequence of alphanumeric characters starting with a letter; when the character sequence is complete, it is looked up in the keyword list to establish its nature. Another place-holder type is *ltMinus*, signifying a "–" character which could be either a unary negative (as in "x := −1") or a binary subtraction (as in "x := y − z"). This ambiguity, however, is resolved not by the lexical analysis, but by the recursive descent phase of the later syntax analysis (cf. §7.6, especially note 61).

7.4 Parsing the Block Structure of the Program

The second pass of the compiler integrates syntax analysis and code generation, controlled by the function *syntax* which is called repeatedly while working through the sequence of lexemes created by the first pass. As outlined in §7.2, this task is broken down into three levels, the first of which involves parsing the program's large-scale structure, identifying the "blocks" that constitute its procedures, sub-procedures, and main body, and dealing with their associated declarations (including scope issues). This top level of processing invokes, as necessary, the middle (§7.5) and bottom (§7.6)

levels, which together analyse and translate the source code lying between the "begin" and "end" that bracket each block's body. But this interplay between the levels is almost entirely one way, enabling the block structure parsing to be treated as an independent task which is sufficiently straightforward to be handled almost completely by a finite state machine (FSM). Moreover where the task goes beyond the capacities of an FSM, this can be remedied by the addition of a single counter (to track the depth of nested procedures), resulting in the simplest possible form of "counter machine" (e.g. Krishnamurthy [69], p. 81). Handling the parsing in this way illustrates nicely how abstract machines can reduce a potentially confusing problem to a transparent relative simplicity, and thus provides an excellent opportunity to motivate students' interest in, and appreciation of, such machines (just as the use of stacks within the Turtle Machine can motivate interest in and appreciation of them, cf. note 36 in §5.1).

Turtle's FSM for analysing the block structure of Pascal source code is represented by Figure 16 on page 83, labelled with the same state names that are used in the "Syntax" table of the Visual Compiler display, pictured below. (In the *Delphi* code that implements the compiler these "syntax state" names are prefixed with "ss" to ensure uniqueness and indicate their type, but such prefixes are ignored here except when explicitly quoting code.) Most of the transitions between states in this FSM involve more than a single lexeme; moving from *ProcVName* to *ProcVSemi*, for example, requires a colon, followed by an identifier, followed by a semicolon. But it would be wasteful and confusing to add states to the FSM corresponding to each of the intermediate stages of such a sequence, because once the colon has been identified, the only legal continuation is an identifier followed by a semicolon. Hence this part of the compiler is implemented in a way that directly mirrors the structure of the FSM, as illustrated by that part of the *Delphi* Pascal code (itself part of a larger *case* statement) which handles the situation where the syntax state is *ProcVName*:

String	Type	Entry State	Exit State
PROGRAM	program	Start	
triangles	identifier		
;	;		ProgSemi
PROCEDURE	procedure	ProgSemi	
triangle	identifier		ProcName
(	(	ProcName	ProcBkt
size	identifier	ProcBkt	ParName
:	:	ParName	
integer	identifier		ParType
)	)	ParType	
;	;		ProcSemi
BEGIN	begin	ProcSemi	PROC

Figure 15: FSM states in the "Syntax" table

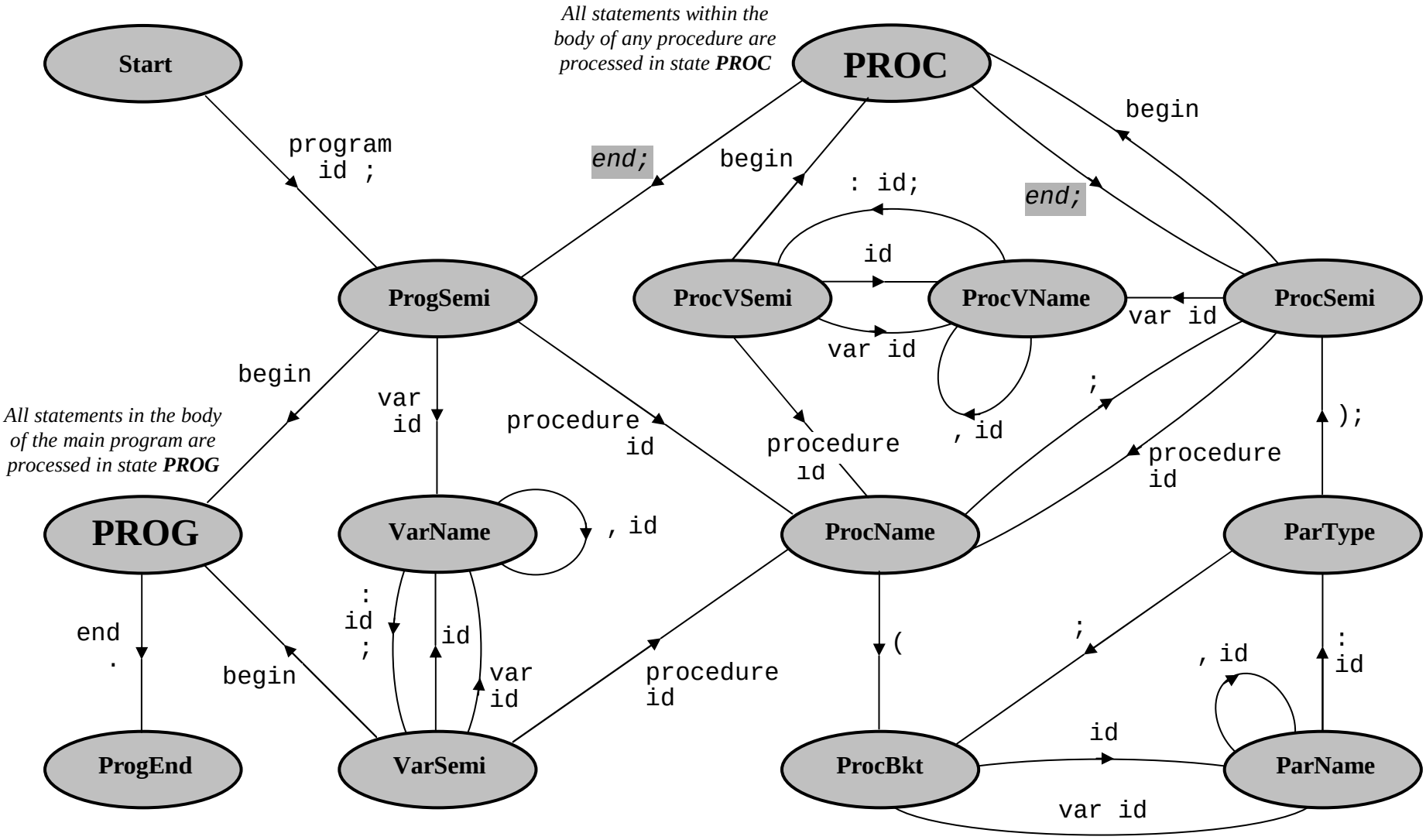


Figure 16: Finite State Machine to analyse the block structure of Pascal source code

```
case lexitems[lexnum].lextype of
 ltComma: nextlex(ltId,ssProcVName,
                  'Comma must be followed by a variable name');
 ltColon: begin
           if nextlex(ltId,ssProcVName,
                      '":" must be followed by a type name') then
            if nextlex(ltSemicolon,ssProcVSemi,
                       'VAR declaration must be followed by ";"') then
              addvtype(lexnum-1)
          end;
 else
          begin
           result:='Variable must be followed by a type specification';
           dec(lexnum)
          end
end {case}
```

Here *lexnum* is the index of the next lexical item (lexeme) to be processed, within the array *lexitems*, and hence *lexitems[lexnum].lextype* gives its identified lexical type (as discussed in §7.3 above). If that type is *ltColon*, then the *nextlex* function is called twice, first checking that the following lexeme is of type *ltId* – i.e. an identifier – and then that the next after that is of type *ltSemicolon*; to facilitate such iteration, *nextlex* automatically increments *lexnum* as it goes. If both checks succeed, then a transition is made to syntax state *ProcVSemi*, and the *addvtype(lexnum: integer)* routine is called – this checks that the specified variable type (i.e. the identifier in question) is either "integer" or "boolean", and if so, assigns it to the variable (or list of variables) concerned. If any of these various checks fails, then an appropriate error message is given to report precisely where the failure occurred. The error location is usually given by the value of *lexnum* at the point where it is located, but sometimes this needs to be adjusted, as in the final *else* clause where "dec(lexnum)" ensures that the error message for a missing type specification will be linked to the source line containing the variable name itself rather than whatever lexeme generated the error.

Simple mechanisms of the sort just discussed, tied into the FSM state transitions, deal with most of the parsing of block headers, but various processes need to go on behind the scenes, "bookkeeping" and keeping track of the overall program structure. For example the *addvtype* routine mentioned above presupposes the operation of a prior *addvar(lexnum: integer; isparam: boolean)* routine, called whenever a new parameter

or variable name is declared (i.e. any transition to state *ParName*, *ProcVName* or *VarName*), and which adds that name to a table into which the corresponding type can later be inserted by *addvtype*. Recording of variable identifiers also brings the complication of checking for non-duplication, which requires that issues of scope be taken into account. This is done by means of a "procedure stack" which monitors the nesting of procedures, with each procedure's index being pushed when it starts (through a transition to state *ProcName*) and popped when it finishes (through either of the "end;" transitions from state *PROC*). The condition of the procedure stack also determines the choice between the two possible "end;" transitions from state *PROC* (which are shaded in Figure 16 on page 83 to indicate the deviation here from a pure finite state machine to what is in effect a "1-counter machine", the simplest kind of pushdown automaton, with a single stack and only one symbol in its stack alphabet). If the procedure stack is empty then a transition is made to *ProgSemi*, but otherwise to *ProcSemi*, since the latter indicates a move into a parent procedure rather than to the main program.

7.4.1 Generation of Procedure Code

Although code generation plays relatively little role in the processing of block structure, it does feature when procedures start and finish. Just as any parameter or variable name declaration triggers the *addvar* routine, so any procedure name declaration will call the routine *addproc(lexnum: integer)* at the point of transition to state *ProcName*. This routine checks the name for non-duplication and initialises a new procedure record, primed so that parameters and variables can be bound to it by any immediately subsequent calls to *addvar* and *addvtype*. These bindings then influence how code is generated when the procedure body is reached with the transition to state *PROC*, which calls the routine *procbegin(lexnum: integer)*; this code deals with the Turtle Machine's activation of the procedure as described in §§6.3 and 6.5 (claiming Heap space and pushing the Procedure Register), and the initialisation of local variables and unpacking of parameters as illustrated in §6.6. Finally, when the procedure finishes with one of the "end;" transitions mentioned above, the *procend(lexnum: integer)* routine is called to insert the PCode required to terminate the procedure (releasing Heap space, popping the Procedure Register, and performing an appropriate return jump etc.).

7.5 Parsing and Code Generation for Program Control Structures

The middle level of parsing and code generation is concerned with the statement flow-control and grouping structures that lie within the main program body, or within the body of individual procedures; these mainly involve the keyword combinations "if … then … else", "repeat … until", "for … do", and "begin … end". Unlike the top level, this parsing task goes well beyond what can be dealt with by a simple FSM or counter machine, in that the various keyword structures can be intermixed, and nested to an arbitrary degree. Hence we resort instead to a pushdown automaton (PDA) that operates a stack of pending operations, yet another abstract machine whose practical behaviour and usefulness can be nicely illustrated by this form of compiler. The overall behaviour of this automaton is represented by Figure 17 on page 87.[54]

---

[54] Figure 17 is slightly simplified, in omitting a special test for null statements following "then", "else" and "do". In structures such as "`if x=1 then <action1> else <action2>`" or "`for x:=1 to 10 do <action3>`", it is standardly permissible for any of the `<action>` statements to be absent (i.e. null), and these structures can occur embedded within other structures, for example: "`repeat for x:=1 to 10 do {nothing}; if y<100 then y:=y*2 else {nothing} until y>=100`". The *Turtle* compiler differs from standard Pascal in treating null statements as illegal if they are immediately followed by a semicolon, since in novice programs such semicolons are almost invariably misplaced. But other null statements within these structures need to be compiled appropriately, and for this purpose the *Turtle* compiler makes the following test as soon as each new lexeme is read. If *either* the new lexeme is in [`until`, `end`] and the top pending operation on the stack is in [*poTo*, *poDownTo*], *or* the new lexeme is in [`until`, `end`, `else`] and the top pending operation is in [*poIfThen*, *poIfElse*], *then* the lexeme pointer is decremented by 1 (so the new lexeme will be read again next time round the loop) and processing continues within the pushdown automaton from immediately below the "Process Command" node, as though a command had just been dealt with.

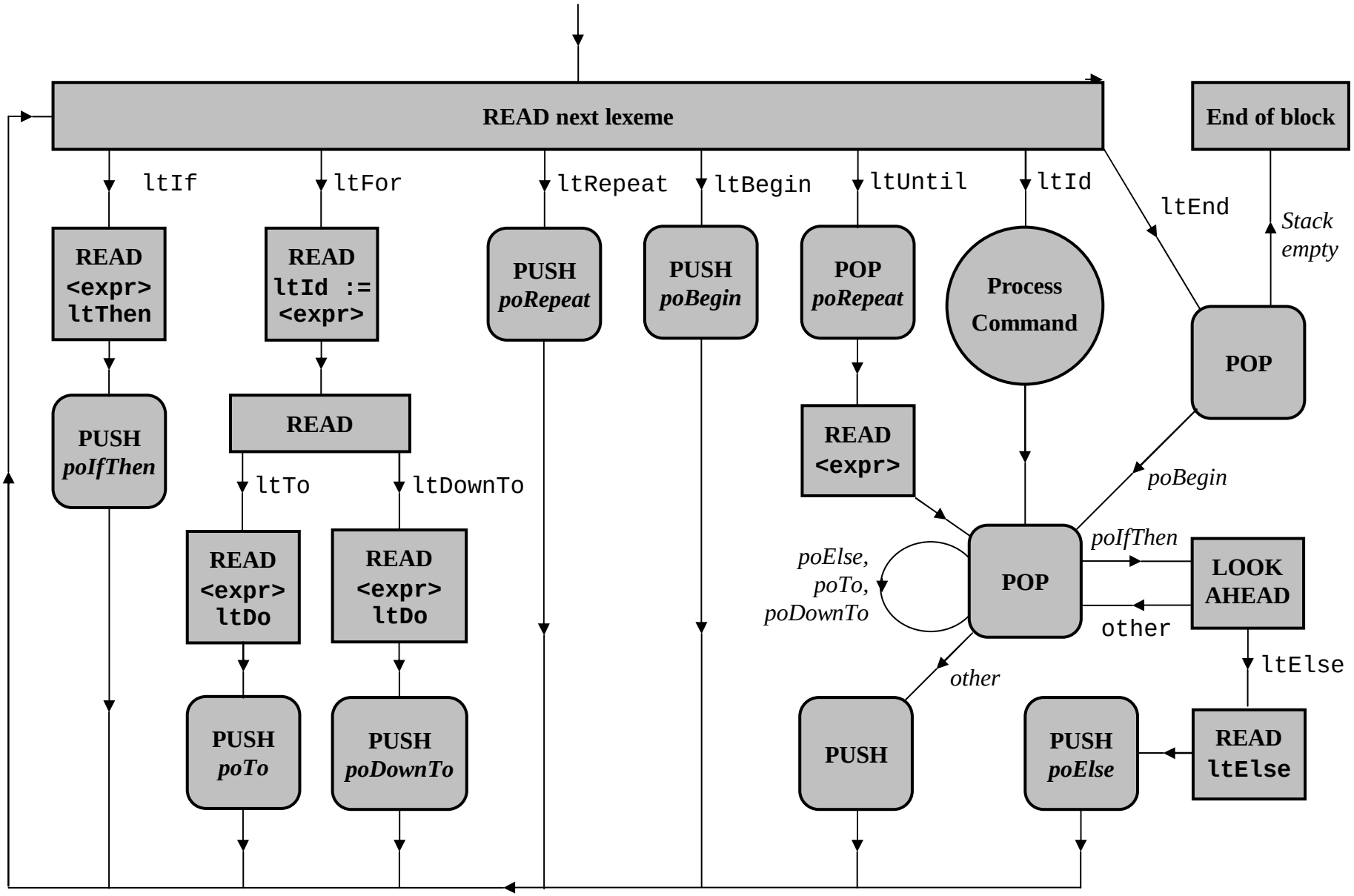


Figure 17: Pushdown Automaton to analyse program control structures

The main loop of the automaton begins with the reading of a tokenised lexeme which – if the program is syntactically correct – must be either an identifier or one of the keywords “if”, “for”, “repeat”, “begin”, “until”, or “end”.[55]

- “if” must be followed immediately by an *expression* (cf. §7.6) and the keyword “then”;[56] assuming these are found, the PDA loops back to the beginning, but first pushes *poIfThen* (one of the constants of enumerated type *TPendOp*) onto the stack of pending operations. Immediately following the processing of the expression, a PCode IFNO command is generated, to jump around the “then” part of the conditional structure (cf. §5.3.2).[57]
- “for” must be followed immediately by an identifier (which must be a declared integer variable name), then the assignment operator “`:=`”, then an expression, and then one of the keywords “to” or “downto” followed by another expression and the keyword “do”. Again assuming all these are found, the PDA loops back having pushed *poTo* or *poDownTo* respectively onto the stack. Meanwhile, PCode commands are generated to store the first expression’s value in the counting variable, and then (after the processing of the second expression), to test the counting variable against the second expression’s value and branch with an IFNO command accordingly (cf. §5.3.4).[58]
- “repeat” and “begin” each simply pushes the corresponding pending operation (*poRepeat* and *poBegin* respectively) onto the stack before looping back.

---

[55] Or a semicolon, which just loops back without any further action and so is effectively ignored. The treatment of semicolon statement separators is not discussed here because they have little relevance to the PDA operation, but the compiler of course checks that they occur appropriately. (It also combines with the auto-formatter to remove redundant semicolons.)

[56] This is the meaning of the “READ `<expr>` ltThen” box in the PDA diagram. Here and elsewhere, the diagram shows only the syntactically legal options; if the syntactic constraints are not respected, the compiler will generate an appropriate error message and terminate.

[57] Since the extent of the “then” part is not yet known, however, this IFNO instruction is given a temporary dummy argument which will in due course be backpatched as in (c) below.

[58] This IFNO instruction is given a dummy argument which will be backpatched as in (b) below.

- "until" pops the top pending operation from the stack, and checks that this is *poRepeat*, before going on to read an expression (which specifies the until condition); then a PCode IFNO command is generated to branch back to the pcode line where the "repeat" occurred, depending on the expression's value (cf. §5.3.3).[59] "end" likewise pops the stack, checking that either it is empty (in which case the "end" signifies the finish of the current program block) or else that the popped pending operation is *poBegin*. These checks ensure that the relevant syntactic structures have been properly nested, so that each "until" matches with a previous "repeat", and each "end" with a previous "begin".
- An identifier indicates an assignment statement, a Turtle Graphics command, or a procedure call – the main processing of these is dealt with in §7.6.1 below.

As the diagram shows, the last two cases here (encompassing "until", "end", assignment statements, Turtle Graphics commands, and procedure calls) do not immediately loop back to read the next tokenised lexeme, because before doing this a check needs to be made to see whether the current statement (which may be a statement sequence bracketed by "repeat … until" or "begin … end") is within the immediate scope of a conditional or a "for" loop. This check is carried out by popping the top pending operation from the stack, and what happens next depends on the nature of this pending operation.

(a) If the pending operation is *poElse*, then the conditional structure is terminated by backpatching the argument to the PCode JUMP instruction which jumps around the "else" part (cf. §5.3.2).

(b) If the pending operation is *poTo* (or *poDownTo*), then the counting loop structure is terminated by generating PCode commands to load and increment (or respectively decrement) the counting variable before the final JUMP; the argument to the IFNO instruction at the beginning of the structure is also backpatched (cf. §5.3.4).

---

[59] This implies that the relevant code line number was recorded within the *poRepeat* pending operation when it was pushed onto the stack. Every pending operation is in fact stored as a record structure which includes both an item of type *TPendOp* and also a code line reference.

(c) If the pending operation is *poThen*, the complier looks ahead to see whether the next lexeme is "else"; if so, that "else" is read, *poElse* is pushed onto the pending operations stack, and a PCode JUMP command is generated to jump around the "else" part of the conditional structure.[60] Finally, whether or not the next lexeme was "else", the argument to the IFNO instruction at the beginning of the conditional structure is backpatched to branch around the "then" part (cf. §5.3.2).

(d) If the pending operation was anything other than *poElse*, *poTo*, *poDownTo*, or *poIfThen*, it is pushed back onto the stack of pending operations.

In cases (a) and (b), and also case (c) where there is no "else", the process of popping the top pending operation from the stack continues, as shown in Figure 17. Thus at this point in the pushdown automaton, it is possible for several pending operations to be dealt with in turn, reflecting the fact that where conditional and counting loop structures are nested, a number of them can terminate together.

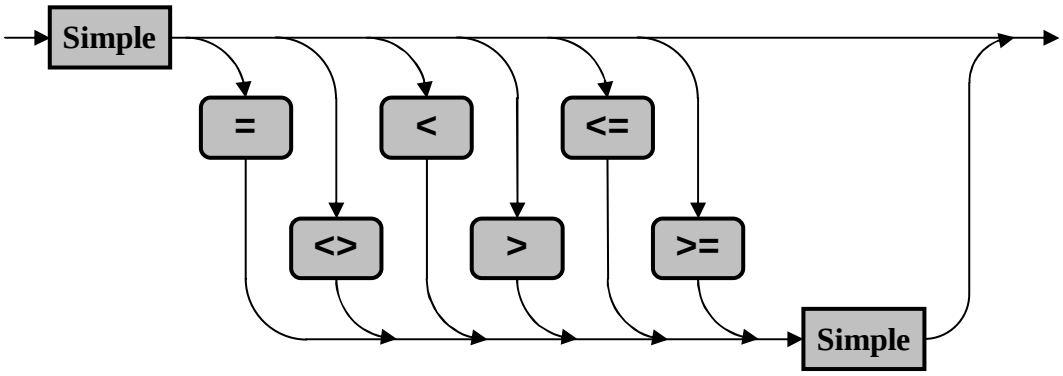
As the program is parsed, the PDA stack is shown in the "Syntax" table of the Visual Compiler, following "PROC" or "PROG" depending on the FSM state. This table also shows the calculated indents which are to be used if the program is auto-formatted.

String	Type	Entry State	Exit State	Indent
to	to			
s	identifier			
do	do		PROC F	
if	if	PROC F		4
ok	identifier			
then	then		PROC FT	
right	identifier	PROC FT		5
(	(			
45	integer			
)	)			
else	else		PROC FE	4
home	identifier	PROC FE	PROC	5
;	;	PROC	PROC	
movexy	identifier	PROC		3

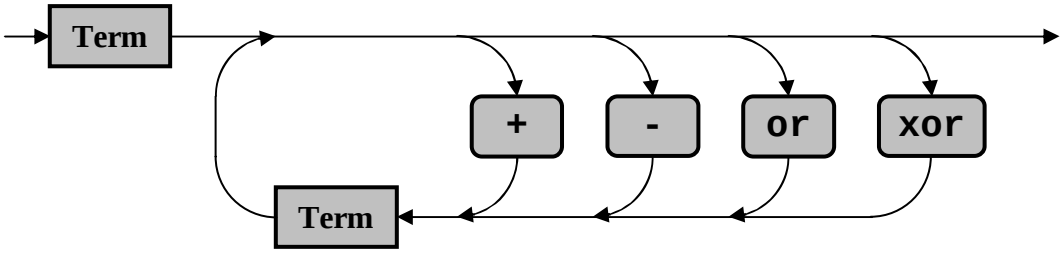
Figure 18: PDA stack and indents in the "Syntax" table

[60] Though yet again, since the extent of the "else" part is not yet known, this JUMP instruction is given a dummy argument which will in due course be backpatched as in (a) above. Note that this treatment of conditionals provides a resolution of the notorious "dangling else" problem which besets grammar-based parsers, since any "else" will automatically be paired with the nearest previous eligible "if", in accordance with both standard and *Delphi* Pascal (e.g. "`if <a> then if <b> then <c> else <d>`" will be disambiguated as "`if <a> then begin if <b> then <c> else <d> end`").

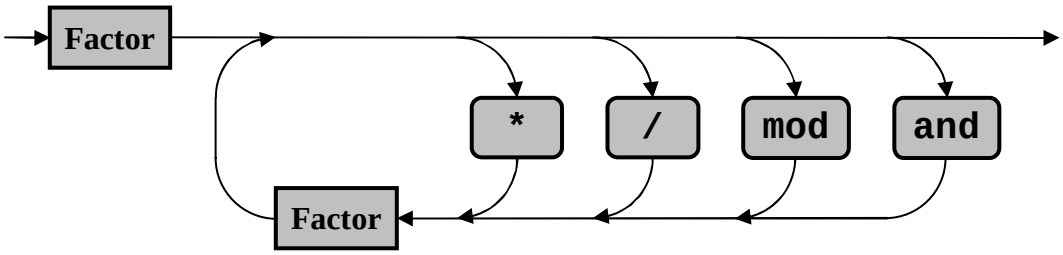
Expression



Simple Expression



Term



Factor

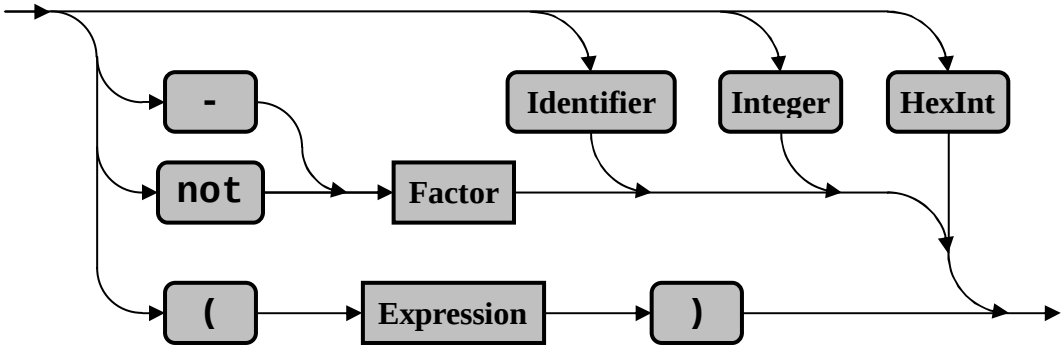


Figure 19: Syntax diagrams for parsing by recursive descent

7.6 Parsing and Code Generation for Individual Commands and Expressions

The lowest of the three levels of parsing and code generation deals with individual Turtle Graphics or assignment commands and procedure calls, and also the processing of arithmetical or boolean expressions. We shall deal with the latter first, for as we have seen, such expressions not only occur within individual commands, but also play a crucial role within the conditional and looping structures discussed in §7.5.

To minimise complexity, this level of the compiler uses a fairly straightforward implementation of the method of *recursive descent*, combining code generation with parsing in a manner that directly reflects the relevant syntactical structures. These structures are standardly represented in syntax diagrams such as those on the previous page – note that they are mutually recursive, since an *expression* always includes a *simple expression*, which in turn always includes a *term*, which in turn always includes a *factor*, which in turn can include either an *expression* or a *factor*. However a *factor* can also be an individual identifier (i.e. a variable name), an integer, or a hexadecimal integer (all of which have previously been identified and tokenised as described in §7.3 above), and this is where the recursion in any particular case ultimately stops.

To illustrate how all this is implemented, here is the code that compiles a *factor*, enclosed within a function of the same name:

```
function factor(start: integer): integer;
begin
 case lexitems[start].lextype of
  ltId,ltInt,ltHex: begin
                     addloadstack(numcommands,start);
                     result:=start+1
                    end;
```

continued on next page

```
   ltMinus: begin
             lexitems[start].lextype:=ltNeg; {disambiguate}
             result:=factor(start+1);
             addpcode(numcommands,pcNeg)
            end;
   ltNot:   begin
             result:=factor(start+1);
             addpcode(numcommands,pcNot)
            end;
   ltLbkt:  begin
             result:=expression(start+1);
             if (result>0) then
              begin
               if (lexitems[result].lextype=ltRbkt) then
                inc(result)
               else
                result:=0
              end
            end;
   else     result:=0
  end
 end; {function factor}
```

When the function is called, *start* specifies the index of the next lexeme to be processed, i.e. the lexeme which is expected to be the beginning of a *factor*. As the syntax diagram indicates, a *factor* can be simply an identifier, integer, or hexadecimal integer – in any of these cases, the procedure *addloadstack(numcommands,start)* is called to make appropriate checks (e.g. that an identifier is a defined variable), and if these checks are passed, to generate the necessary PCode for loading the relevant item onto the Program Stack. Then *start+1* is returned as the value of the *factor* function, to enable syntactic analysis to continue from the next lexeme.

If the next lexeme is of lexical type *ltMinus* or *ltNot*, then as the syntax diagram indicates, this should be followed by a *factor*. Hence in this case the function calls itself from the following lexeme with *result:=factor(start+1)*, and when this returns, generates the single PCode instruction (*pcNeg* or *pcNot*) corresponding to the operator lexeme.[61] The point here is that the recursive call, if successful, will not only parse the

[61] Note that the code dealing with *ltMinus* disambiguates this to *ltNeg* (unary negation), because

factor starting at lexeme *start+1*, but will also generate PCode designed to leave that *factor's* result on the Program Stack (as in the previous paragraph). Hence that result can be negated, as implied in the source code, by applying an appropriate PCode negation operator (i.e. *pcNot* or *pcNeg*) that acts on the top Program Stack value.

Finally, if the lexeme with which the *factor* starts is a left-bracket, the system expects this to be followed by an *expression* and then a right-bracket. So the function *expression(start+1)* is called in order to identify, and generate PCode for, the *expression* in question. If this fails, 0 will be returned and passed on also as the return value of the *factor* function, 0 being generally used to signify a syntactic error. But if parsing of the *expression* succeeds, the value returned by *expression(start+1)* will be the index of the next lexeme after the end of the identified expression; accordingly a check is then made to ensure that this lexeme is indeed a right-bracket, in which case *inc(result)* increments the function's return value to point to the lexeme after that bracket. Note that brackets in themselves (unlike identifiers, integers, negation operators etc.) do not yield any specific PCode instructions. They simply ensure that the PCode generated by the *expressions* that they enclose corresponds appropriately to the sequence of processing indicated by the bracketing hierarchy.

7.6.1 Code Generation for Commands and Procedure Calls

The processing of individual Turtle Graphics commands, assignments, and procedure calls comes into play whenever the pushdown automaton illustrated on page 87 encounters an identifier at the beginning of its main loop. Assuming the program is syntactically correct, there are then three possibilities which are checked in turn:[62]

---

the other possible interpretation of *ltMinus* (namely *ltSubt* for binary subtraction) cannot occur as the first lexeme of a *factor*. See §7.3 for a brief discussion of this ambiguity.

[62] The order of processing shown below allows for the possibility that a variable or procedure will be declared with a name which is also a Turtle Graphics instruction, and if so, the identifier will be interpreted accordingly when that name is used within the scope of the declaration (i.e. Turtle Graphics instructions are *not* reserved words).

- The identifier is a recognised (and in scope) variable name, in which case it should be followed by the assignment operator "`:=`" (lexical type *ltAsgn*) and an *expression*.
- The identifier is a declared (and in scope) procedure name, in which case it should be followed by an appropriate number of parameters in brackets.
- The identifier is a Turtle Graphics command name, in which case it should be followed by an appropriate number of parameters in brackets.

In the first case, the *expression* is processed by the method described in §7.6, which generates the necessary PCode for leaving the result of the *expression* on the Program Stack. Then depending on the nature of the variable, an appropriate store command ("STVG", "STVV" or "STVR") is generated to perform the assignment.

The second and third cases are initially treated in the same way – the opening bracket is checked, and then the parameter *expressions*, separated by commas, are repeatedly processed as in §7.6 until the final bracket is encountered (any deviation from this pattern producing an error message). The PCode thus generated will leave one item on the Program Stack for each *expression* encountered, and these will be passed as the parameters for the procedure (cf. §6.6) or command (cf. §§5.1-2). Hence a test is made to ensure that the number of parameters is correct, and if so, the necessary PCode is added either to call the recognised procedure (as in §6.1) or to execute the relevant Turtle Graphics command (see §5.1).

This concludes our detailed discussion of the Turtle Machine and its compiler, which has been intended to show by example how relatively straightforward the system is, enabling it to be used effectively as a vehicle for understanding, at a relatively early stage, such fundamental concepts of Computer Science as machine code, stack operations, compilation, recursive descent, and various automata. In Chapter 8 we conclude by bringing the two main themes of the thesis together, and briefly examining whether they may be more complementary than contrasting.

Chapter 8 Conclusion

8.1 The Value of *Turtle* as a Vehicle for Introducing Computing Concepts

We have now seen two apparently distinct aspects of the *Turtle* system. On the one hand, as discussed in Chapters 3 and 4 and summed up in §4.4, it can be used as a teaching environment for novice programmers. On the other hand, as illustrated in Chapters 5 to 7, it can be used as a vehicle for explaining the concepts of machine code and compilation, presumably to more advanced students. So far, it might seem that these two teaching tasks are largely unrelated, but I shall now conclude by suggesting that in fact they are complementary, and that there may be unexpected benefits in having a system that combines them.

8.1.1 Notional Machines

As discussed in §4.4, many students of programming – and not by any means only novices – experience considerable difficulty in understanding flow of control within a program. But Robins et al. point out that this can be overcome when they develop an adequate mental model: "Many studies have noted the central role played by a model of (an abstraction of) the computer, often called a 'notional machine' … to provide a foundation for understanding the behaviour of running programs" ([111], p. 149, cf. p. 158).[63] Du Boulay [41] accordingly identifies the absence of an adequate notional

[63] Studies listed by Robins et al. in this connection include du Boulay [41], du Boulay et al. [42], Mayer [79], Hoc and Nguyen-Xuan [58], Mendelsohn et al. [84], and Cañas et al. [24]. The paper by Ben-Ari [12] is also particularly interesting, because he links the philosophy of constructivism (cf. §1.2 above) with the need to teach a model, based on the unfamiliarity of algorithmic mechanisms within the student's pre-existing conceptual framework. "You have to *construct* a viable model that will enable you to *predict* the outcome of any operation on the model … The relevance for CSE is that courses … must *explicitly* address the construction of a model … If the student does not bring a preconceived model to class, then we must ensure that a viable hierarchy of models is constructed and refined as learning progresses." (p. 260).

machine as one of the five main sources of difficulty in programming (pp. 283-4). But if so, then the two teaching aspects of the *Turtle* system are quite intimately connected after all. For one obvious way of helping students to develop an adequate mental model of computer execution is to provide an easily accessible virtual machine to study, one that can handle the problematic looping and recursive control-flow structures, but which in other respects is as straightforward to program as possible.

The natural objection to this suggestion is that learning about virtual machines and low-level code, so far from reducing the problem, may prove even more confusing and unpopular with students than learning about high-level programming. Certainly low-level code is in some obvious respects more complex than the high-level equivalent, but on the other hand it is less *abstract* and more obviously *mechanical*, which may tip the balance if the student's main problem lies in acquiring an adequate notional model of execution. At any rate, there is some evidence that even novice students can cope with machine code if due care is taken and appropriate learning tools provided. For example the "breadth-first" curriculum described by Tucker et al. [131], developed on the basis of the ACM/IEEE *Computing Curricula 1991* report [1], successfully combined the teaching of programming with an introduction to computer organisation using a simple machine and assembly language simulator called "Marina".[64] This simulator was seen as playing a key role in facilitating the computer organisation section (p. 54), helping to make it one of the most popular in the curriculum (pp. 36-7, 39). Moreover the authors stress that this was not at the cost of reducing the course to a shallow overview: "The key factor is to organize the presentation well, introducing as much of it as is necessary to give students a firm understanding of its principles … Avoid introducing a topic in a survey fashion." (p. 54). *Turtle* is clearly a potential vehicle for such a presentation.

---

[64] Another interesting example is the PIPPIN machine described by Decker and Hirschfield [33], which is explicitly designed for use with novice students, though its associated compiler, "Rosetta", is limited to the compilation of an assignment statement involving simple expression evaluation.

8.1.2 Deep Understanding

Leaving aside the possible value of the Turtle Machine to novices, it has more obvious potential in helping relatively advanced students to knit together the various aspects of the discipline, which according to many writers is the key to robust "deep" understanding (e.g. Marton and Säljö [78], Gibbs [44]). And again this seems to be closely related to having an adequate notional machine, for it is striking how both lecturers and students identify deep understanding with "having a mental picture" of what is going on (Newton et al. [90], pp. 48, 51, cf. Robins et al. [111], pp. 140, 151). Such a mental picture is far easier to acquire within a system whose workings are based on a systematic and intuitive metaphor (cf. §1.2 above), and which – like *Turtle*'s visual compiler – faithfully displays its inner workings consistently with that metaphor (cf. Pane and Myers [95] section 5.2).

8.1.3 Automata Early

Although formal automata theory is generally considered far too abstract to be given a place in the first year studies of Computing students, Chua and Winton [29] argue that automata are best introduced early, in a way that emphasises their practical applicability, in order to prepare the ground for later work by developing maturity in thinking about them. Again this approach would give *Turtle* an obvious role, enabling the explanation of machine execution to be combined with a discussion of some of the stages of compilation, aided by the Visual Compiler display showing the ongoing state of the FSM and PDA that it uses. Neither the FSM nor the PDA are overly complex, and both have a very clear practical role in the compilation process, so this can be used to motivate an appreciation of their more general practical value. It is interesting to speculate, for example, what effect such a background might have had on students faced with the kind of arithmetic expression evaluation tasks that led to such disappointing results in McCracken et al. [83], where the lack of a confident understanding of stack structures was seriously prejudicial.

8.1.4 Familiar Compilation

Courses on compilers are notoriously difficult and time-consuming, so that typically very few students take them, and those that do are given the task of compiling an unfamiliar "toy" language rather than the real thing (such languages include Haynes et al's *ORACLE* [54], Appelbe's *MINIPASCAL* [6], Aiken's *Cool* [4], Baldwin's *MinimL* [10], and numerous others with and without names). In this context there may be a real virtue in *both* using the *Turtle* system to introduce programming to novices, and also later returning to it when they come to consider the study of compilation. At least one hurdle of unfamiliarity is thus removed, probably making the subject seem both more approachable and more relevant.

8.2 Conclusion: The Achievements of this Work

The last few sections have been to some extent speculative, and the potential value of the *Turtle* system in teaching about machine code, automata, and compilation is largely independent of whether this can indeed also be closely related to the teaching of introductory programming. My more fundamental claims about the Turtle Machine and its workings are:

(A4) That it is simple enough for more advanced students to understand and to learn from (and is indeed simpler in relevant respects than any machine of comparable power that I know of).

(A5) Likewise its compiler, though in a sense relatively complex (because it mixes parsing techniques rather than starting from a comprehensive language grammar), presents the parsing process in a way that is easier to grasp, and from which to learn widely applicable practical lessons, than the more abstract treatments which dominate the teaching literature.

(A4) and (A5) here refer to the intended achievements listed in §1.7, the others of which have already been presented earlier in the thesis. To recapitulate briefly, Chapter 1 argued the case for developing an integrated Turtle Graphics environment for novice programming (A1), and this case was later strongly corroborated by the teaching

outcomes reported in Chapter 4. Chapter 2 gave a preview of the finished product, while Chapter 3 justified the choice of source language (A2). The program itself, and the feedback from its extensive practical use as presented in Chapter 4, testify to its power, robustness, and attractiveness to students (A3). Chapters 5 and 6 then discussed the *Turtle* virtual machine (A4), and Chapter 7 the compiler (A5). The visual interface to these, illustrated both in §2.3 and §§7.3-5, significantly adds to their teaching value (A6).

Overall, therefore, I am very satisfied with the outcome of this project, and feel that the *Turtle* system is a worthwhile contribution to the teaching of Computing. I hope, moreover, that it has gone at least some way to meeting the challenge laid down by Holmes and Smith [59], to present both introductory and more advanced topics of Computer Science in a way that is accessible to relative novices, "in a manner that will capture the imagination of the learners" and convey the subject's "scope, beauty, genius, and fun" (p. 204).

References

Each item is followed by a bracketed list of the pages where reference is made to it within the thesis.

[1] ACM/IEEE Joint Curriculum Task Force (1991), *Computing Curricula 1991*, New York: ACM Press. (97)

[2] Adair, G. (1984), "The Hawthorne effect: a reconsideration of the methodological artifact", *Journal of Applied Psychology* **69**, pp. 334-45. (39)

[3] Aho, Alfred V. and Ullman, Jeffrey D. (1979), *Principles of Compiler Design*, Addison-Wesley (originally published in 1977 by Bell Telephone Laboratories). (76, 77)

[4] Aiken, Alexander (1996), "Cool: a portable project for teaching compiler construction", *ACM SIGPLAN Notices* **31/7**, pp. 19-24. (99)

[5] Alfonseca, Manual and Ortega, Alfonso, "Representation of fractal curves by means of L systems" (1996), *ACM SIGAPL APL Quote Quod* **26/4**, pp. 13-21. (119)

[6] Applebe, Bill (1979), "Teaching compiler development", *ACM SIGCSE Bulletin* **11/1**, pp. 23-7. (99)

[7] Ariga, Taeko and Tsuiki, Hideki (2001), "Programming for students of information design", *ACM SIGCSE Bulletin* **33/4**, pp. 59-63. (9)

[8] Atwood, J. W., Jr., M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang, "Steering programs via time travel" (1996), *IEEE Symposium on Visual Languages* (Boulder, Colorado), pp. 4-11. (118)

[9] Baldwin, Doug (1996), "Discovery learning in Computer Science", *ACM SIGCSE Bulletin* **28/1**, pp. 222-6. (6)

[10] Baldwin, Doug (2003), "A compiler for teaching about compilers", *ACM SIGCSE Bulletin* **35/1**, pp. 220-3. (99)

[11] Bell, D. A. and Wichmann, B. A. (1971), "An ALGOL-like assembly language for a small computer", *Software Practice and Experience* **1**, pp. 61-72. (12)

[12] Ben-Ari, Mordechai (1998), "Constructivism in computer science education", *ACM SIGCSE Bulletin* **30/1**, pp. 257-61. (3, 96)

[13] Bonar, J. and Soloway, E. (1989), "Preprogramming knowledge: a major source of misconceptions in novice programmers", in Soloway and Spohrer [126], pp. 325-53. (27)

[14] Bornat, Richard (1979), *Understanding and Writing Compilers*, Macmillan. (76, 77)

[15] Boyle, Tom (2000), "Constructivism: a suitable pedagogy for information and computer science?", *1st Annual LTSN-ICS Conference* (Edinburgh). (3-4)

[16] Bridges, Susan M. (1993), "Graphics assignments in discrete mathematics", *ACM SIGCSE Bulletin* **25/1**, pp. 83-6. (119)

[17] Brilliant, Susan S. and Wiseman, Timothy R. (1996), "The first programming paradigm and language dilemma", *ACM SIGCSE Bulletin* **28/1**, pp. 338-42. (7)

[18] Bruce, Kim B., Andrea Danyluk, and Thomas Murtagh (2001), "A library to support a graphics-based object-first approach to CS1", *ACM SIGCSE Bulletin* **33/1**, pp. 6-10. (9)

[19] Brusilovsky, P., E. Calabrese, J. Hvorecky, A. Kouchnirenko, and P. Miller (1994), "Mini-languages: a way to learn programming principles", *Education and Information Technologies* **2**, pp. 65-83. (5, 23)

[20] Brusilovsky, P., A. Kouchnirenko, P. Miller, and I. Tomek (1994), "Teaching programming to novices: a review of approaches and tools", in T. Ottmann and I. Tomek (eds), *Educational Multimedia and Hypermedia: Proceedings of ED-MEDIA 94*, Vancouver: Association for the Advancement of Computing in Education, pp. 103-10. (23)

[21] Burton, Philip J. and Bruhn, Russel E. (2003), "Teaching programming in the OOP era", *ACM SIGCSE Bulletin* **35/2**, pp. 111-14. (7)

[22] Callear, David (2000), "Teaching programming: some lessons from Prolog", *1st LTSN-ICS Conference* (Edinburgh). (7)

[23] Calloni, B. and Bagert, D. (1994), "ICONIC programming in BACCII vs. textual programming: which is a better environment?", *ACM SIGCSE Bulletin* **26/1**, pp. 188-92. (117)

[24] Cañas, J. J., T. Bajo, and P. Gonzalvo (1994), "Mental models and computer programming", *International Journal of Human-Computer Studies*" **40**, pp. 795-811. (96)

[25] Cantù, Marco (1997), "Comparing OOP languages: Java, C++, Object Pascal", published on the Web at www.marcocantu.com/papers/ooplang.htm. (7)

[26] Caspersen, Michael E. and Christensen, Henrik Baerbak (2000), "Here, there and everywhere – on the recurring use of Turtle Graphics in CS1", *Proceedings of the 4th Australian Computing Education Conference, ACE2000* (Melbourne), pp. 34-49. (9)

[27] Chalk, Peter (2000), "Webworlds – Web-based modeling environments for learning software engineering", *Computer Science Education* **10**, pp. 39-56. (118)

[28] Chalk, Peter, Tom Boyle, Poppy Pickard, Claire Bradley, Ray Jones, and Ken Fisher (2003), "Improving pass rates in introductory programming", *4th Annual LTSN-ICS Conference* (Galway), pp. 6-10. (41)

[29] Chua, Y. S. and Winton, C. N. (1987), "Teaching theory of computation at the junior level", *ACM SIGAPL APL Quote Quad* **17/4**, pp. 69-78. (98)

[30] Committee on Information Technology Literacy (1999), *Being Fluent with Information Technology*, Washington DC: National Academy Press. (1)

[31] Cooper, Stephen, Wanda Dann, and Randy Pausch (2003), "Using animated 3D graphics to prepare novices for CS1", *Computer Science Education* **13**, pp. 3-30. (5, 43)

[32] Dann, Wanda, Stephen Cooper, and Randy Pausch (2000), "Making the connection: programming with animated small world", *ACM SIGCSE Bulletin* **32/3**, pp. 41-4. (5)

[33] Decker, Rick and Hirshfield, Stuart (2001), "The PIPPIN machine: simulations of language processing", *Journal of Educational Resources in Computing* **1**, pp. 4-17. (97)

[34] Deek, Fadi P. (1999), "The software process: a parallel approach through problem solving and program development", *Computer Science Education* **9**, pp. 43-70. (41)

[35] Deek, Fadi P. and McHugh, James A. (1998), "A survey and critical analysis of tools for learning programming", *Computer Science Education* **8**, pp. 130-78. (117)

[36] DePasquale, Peter J. (2002), "Subsetting language elements in novice programming environments", *Proceedings of RESOLVE 2002* (Columbus, Ohio), pp. 108-11. (23)

[37] DeRemer, Franklin L. (1976), "Lexical analysis", in Friedrich L. Bauer and Jürgen Eickel (eds.), *Compiler Construction, An Advanced Course*, second edition, Springer Verlag (first edition published in 1974), pp. 109-120. (76)

[38] Dicheva, D. and Close, J. (1996), "Mental models of recursion", in *Journal of Educational Computing Research* **14**, pp. 1-23. (42)

[39] Dijkstra, Edsger W. (1989), "On the cruelty of really teaching Computing Science", *Communications of the ACM* **32/12**, pp. 1398-404. (2)

[40] Donovan, Steve (2001), *C++ By Example*, Que. (9)

[41] du Boulay, B. (1989), "Some difficulties of learning to program", in Soloway and Spohrer [126], pp. 283-99. (23, 41, 96-7)

[42] du Boulay, B., T. O'Shea, and J. Monk (1989), "The black box inside the glass box: presenting computing concepts to novices", in Soloway and Spohrer [126], pp. 431-46. (23, 96)

[43] Eisenberg, M., M. Resnick, and F. Turbak (1987), "Understanding procedures as objects", in Olson, Sheppard, and Soloway [93], pp. 14-32. (23)

[44] Gibbs, G. (1994), *Improving Student Learning: Theory and Practice*, Oxford: Oxford Centre for Staff Development. (98)

[45] Glinert E. and Tanimoto, S. (1984), "Pict: an interactive graphical programming environment", *IEEE Computer* **17**, pp. 7-25. (117)

[46] Green, T. R. G., R. K. E. Bellamy, and J. M. Parker (1987), "Parsing and gnisrap: a model of device use", in Olson, Sheppard, and Soloway [93], pp. 132-46. (8)

[47] Grune, Dick, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen (2000), *Modern Compiler Design*, Wiley. (78)

[48] Guimaraes, M., C. de Lucena, and M. Cavalcanti (1994), "Experience using the ASA algorithm teaching system", *ACM SIGCSE Bulletin* **26/4**, pp. 45-50. (117)

[49] Guzdial, Mark (1993), "Software-realized scaffolding to facilitate programming for science learning", *Interactive Learning Environments* **4**, pp. 1-44. (2, 41)

[50] Hadjerroult, Said (1998), "Java as first programming language: a critical evaluation", *ACM SIGCSE Bulletin* **30/2**, pp. 43-7. (7)

[51] Hagan, Dianne and Markham, Selby (2000), "Does it help to have some programming experience before beginning a Computing degree program?", *ACM SIGCSE Bulletin* **32/3**, pp. 25-8. (43)

[52] Harel, I. and Papert, S. (1990), "Software design as a learning environment", *Interactive Learning Environments* **1**, pp. 1-32. (2)

[53] Hartley, James and Greggs, Michael A. (1997), "Divergent thinking in Arts and Science students: *Contrary Imaginations* at Keele revisited", *Studies in Higher Education* **22**, pp. 93-7. (6)

[54] Haynes, William R. Jr., Charles E. Hughes and Charles P. Pfleeger (1977), "Oracle: a tool for learning compiler writing", *ACM SIGCSE Bulletin* **9/7**, pp. 37-51. (99)

[55] Hazzan, Orit (2003), "How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science", *Computer Science Education* **13**, pp. 95-122. (12)

[56] Hoc, J. M. (1989), "Do we really have conditional statements in our brains?", in Soloway and Spohrer [126], pp. 179-90. (41)

[57] Hoc, J. M., T. R. G. Green, R. Samurçay, and D. J. Gillmore, eds. (1990), *Psychology of Programming*, Academic Press. (contains [58], [84], [112])

[58] Hoc, J. M. and Nguyen-Xuan, A. (1990), "Language semantics, mental models and analogy", in Hoc et al. [57], pp. 139-56. (96)

[59] Holmes, Geoffrey and Smith, Tony C. (1997) "Adding Some Spice to CS1 Curricula", *ACM SIGCSE Bulletin* **29/1**, pp. 204-8. (100)

[60] Jarc, Duane J. (1992), "Ada, Pascal's replacement for introductory courses in computer science", *Proceedings of the ACM Conference on TRI-Ada* (Orlando, Florida), pp. 126-34. (7)

[61] Jenkins, Tony (2001), "Teaching programming – a journey from teacher to motivator", *2nd Annual LTSN-ICS Conference* (London). (41, 42)

[62] Jenkins, Tony (2002), "On the difficulty of learning to program", *3rd Annual LTSN-ICS Conference* (Loughborough). (41)

[63] Jenkins, Tony (2003), "The first language – a case for Python?", *4th Annual LTSN-ICS Conference* (Galway). (7)

[64] Jenkins, Tony and Davy, John (2001), "Diversity and motivation in introductory programming", *Italics* **1**. (41)

[65] Jensen, Kathleen and Wirth, Niklaus (1991), *Pascal User Manual and Report: ISO Pascal Standard*, fourth edition revised by Andrew B. Mickel and James F. Miner, Springer Verlag (originally published as the *Pascal User Manual and Report* by Jensen and Wirth, 1974). (11)

[66] Kahney, H. (1989), "What do novice programmers know about recursion?", in Soloway and Spohrer [126], pp. 209-28. (42)

[67] Keller, John M. (1983), "Motivational design of instruction", in Charles M. Reigeluth (ed.), *Instructional Design Theories and Models – An Overview of the Current Status*, Lawrence Erlbaum Associates, pp. 383-434. (42)

[68] Kessler, C. M. and Anderson, J. R. (1989), "Learning flow of control: recursive and iterative procedures", in Soloway and Spohrer [126], pp. 229-60. (42)

[69] Krishnamurthy, E. V. (1983), *Introductory Theory of Computer Science*, Macmillan. (82)

[70] Kurland, D. M., R. D. Pea, C. Clement, and R. Mawby (1989), "A study of the development of programming ability and thinking skills in high school students", in Soloway and Spohrer [126], pp. 83-112. (3, 42)

[71] Levy, Dalit (2001), "Insights and conflicts in discussing recursion: a case study", *Computer Science Education* **11**, pp. 305-22. (42)

[72] Lewis, C. and Olson, G. M. (1987), "Can principles of cognition lower the barriers to programming?", in in Olson, Sheppard, and Soloway [93], pp. 248-63. (23)

[73] Lewis, Stuart and Mulley, Gaius (1998), "A comparison between novice and experienced compiler users in a learning environment", *ACM SIGCSE Bulletin* **30/3**, pp. 157-61. (119)

[74] Liffick, Blaise W. and Aiken, Robert (1996), "A novice programmer's support environment", *ACM SIGCSE Bulletin* **28/SI**, pp. 49-51. (6)

[75] Linn, M. C. and Dalbey, J. (1989), "Cognitive consequences of programming instruction", in Soloway and Spohrer [126], pp. 57-81. (42)

[76] Lippert, R. C. (1980), "Teaching problem solving in mathematics and science with expert systems", *Journal of Artificial Intelligence in Education* **1**, pp. 27-40. (2)

[77] Martin, Peter (1998), "Java, the good the bad and the ugly", *ACM SIGPLAN Notices* **33/4**, pp. 34-9. (9)

[78] Marton, F. and Säljö, R. (1976), "On qualitative differences in learning", *British Journal of Educational Psychology* **46**, pp. 4-11. (98)

[79] Mayer, R. E. (1989), "The psychology of how novices learn computer programming", in Soloway and Spohrer [126], pp. 129-59. (5, 96)

[80] Mayer, R. E., J. L. Dyck, and W. Vilberg (1986), "Learning to program and learning to think: what's the connection?", *Communications of the ACM* **29/7**, pp. 605-10. (3)

[81] Mayo, E. (1933), *The Human Problems of an Industrial Civilization*, Macmillan. (39)

[82] McConnell, Jeffrey J. (1996), "Active learning and its use in Computer Science", *ACM SIGCSE Bulletin* **28/SI**, pp. 52-4. (5)

[83] McCracken, Michael, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz (2001), "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students", *ACM SIGCSE Bulletin* **33/4**, pp. 125-80. (41, 98)

[84] Mendelsohn, P., T. R. G. Green, and P. Brna (1990), "Programming languages in education: the search for an easy start", in Hoc et al. [57], pp. 175-99. (96)

[85] Moser, Robert (1997), "A fantasy adventure game as a learning environment: Why learning to program is so difficult and what can be done about it", *ACM SIGCSE Bulletin* **29/3**, pp. 114-6. (6, 41)

[86] Motil, John and Epstein, David, "JJ: a language designed for beginners (less is more)", available from www.publicstaticvoidmain.com. (23)

[87] Myers, B. A., R. Chandhok, and A. Sareen (1988), "Automatic data visualisation for novice Pascal programmers", in *Proceedings of IEEE Workshop on Visual Languages* (Pittsburgh), pp. 192-8. (117)

[88] Naps, Thomas L., Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide (2003), "Exploring the role of visualisation and engagement in Computer Science education", *ACM SIGCSE Bulletin* **35/2**, pp. 131-52. (5)

[89] Neal, L. R. (1989), "A system for example-based learning", in *Proceedings of CHI Conference on Human Factors in Computer Systems* (Boston), pp. 63-8. (118)

[90] Newton, D. P., Newton, L. D. and Oberski, I. (1998), "Learning and conceptions of understanding in History and Science: lecturers and new graduates compared", *Studies in Higher Education* **23**, pp. 43-58. (98)

[91] Nulty, Duncan D. and Barrett, Mary A. (1996), "Transitions in students' learning styles", *Studies in Higher Education* **21**, pp. 333-45. (6)

[92] Olsen, K. A. (1988), "The DSP system: a visual system to support teaching of programming", in *Proceedings of IEEE Workshop on Visual Languages* (Pittsburgh), pp. 199-206. (117)

[93] Olson, G. M., S. Sheppard, and E. Soloway (eds), *Empirical Studies of Programmers: Second Workshop*, Norwood, New Jersey: Ablex. (contains [43], [46], [72])

[94] Palumbo, D. B. (1990), "Programming language/problem-solving research: a review of relevant issues", *Review of Educational Research* **60**, pp. 65-89. (3)

[95] Pane, John F. and Myers, Brad A., (1996) "Usability issues in the design of novice programming systems", Technical Report CMU-CS-96-132, Carnegie Mellon University. (5, 43, 98)

[96] Pane, John F., Brad A. Myers, and Leah B. Miller (2002), "Using HCI techniques to design a more usable programming system", *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (Arlington, Virginia), pp. 198-206. (41, 118)

[97] Papert, Seymour (1993), *Mindstorms*, second edition, Basic Books (first edition published in 1980). (3, 18, 28)

[98] Parsons, H. M. (1974), "What happened at Hawthorne?", *Science* **183**, pp. 922-32. (39)

[99] Pattis, Richard E. (1981), *Karel the Robot: A Gentle Introduction to the Art of Programming*, Wiley. (4, 18, 23)

[100] Pea, R. D. (1987), "LOGO programming and problem solving", in Eileen Scanlon and Tim O'Shea (eds.), *Educational Computing*, Wiley, pp. 155-60. (3)

[101] Pemberton, Steven and Daniels, Martin (1982), *Pascal Implementation: The P4 Compiler and Interpreter*, Ellis Horwood. (12)

[102] Perkins, D. N., C. Hancock, R. Hobbs, F. Martin, and R. Simmons (1989), "Conditions of learning in novice programmers", in Soloway and Spohrer [126], pp. 261-79. (41)

[103] Piaget, Jean, *The Origin of Intelligence in the Child* (1936), published in English translation by Routledge & Kegan Paul, 1953. (3)

[104] Piaget, Jean, *The Child's Construction of Reality* (1937), published in English translation by Routledge & Kegan Paul, 1955. (3)

[105] Poulton, John (2003), "Managing diversity in introductory programming classes: Using Logo as a diagnostic tool", *3rd LTSN-ICS one day conference on the Teaching of Programming*. (43)

[106] Proulx, Viera K. (1997), "Recursion and grammars for CS2", *ACM SIGCSE Bulletin* **29/3**, pp. 74-6. (119)

[107] Putnam, R. T., D. Sleeman, J. A. Baxter, and L. K. Kuspa (1989), "A summary of misconceptions of high school Basic programmers", in Soloway and Spohrer [126], pp. 301-14. (41)

[108] Resnick, Mitchel (1994), *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press. (119)

[109] Roberts, Eric and Picard, Antoine (1998), "Designing a Java graphics library for CS1", *ACM SIGCSE Bulletin* **30/3**, pp. 213-18. (9)

[110] Robillard, P. N. (1986), "Schematic pseudocode for program constructs and its computer automation by SCHEMACODE", *Communications of the ACM* **29/11**, pp. 1072-89. (117)

[111] Robins, Anthony, Janet Rountree, and Nathan Rountree (2003), "Learning and teaching programming: a review and discussion", *Computer Science Education* **13**, pp. 137-72. (41, 42, 43, 88, 96)

[112] Rogalski, J. and Samurçay, R. (1990), "Acquisition of programming knowledge and skills", in Hoc et al. [57], pp. 157-74. (27, 41)

[113] Rountree, N., Rountree, J., and Robins, A. (2002), "Identifying the danger zones: predictors of success and failure in a CS1 course", *Technical Report OUCS-2001-07* (Otago University, New Zealand), available from www.cs.otago.ac.nz/trseries. (42)

[114] Samurçay, R. (1989), "The concept of a variable in programming: its meaning and use in problem solving by novice programmers", in Soloway and Spohrer [126], pp. 161-78. (41)

[115] Shackelford, R. and Badre, A. (1993), "Why can't smart students solve simple programming problems?", *International Journal of Man-Machine Studies* **38**, pp. 985-97. (41)

[116] Schaub, Stephen (2000), "Teaching Java with graphics in CS1", *SIGCSE Bulletin* **32/2**, pp. 71-3. (9)

[117] Shafto, Sylvia A. S. (1986), "Programming for learning in mathematics and science", *ACM SIGCSE Bulletin* **18/1**, pp. 296-302. (2)

[118] Sime, M.E., T. R. G. Green, and D. J. Guest (1977), "Scope marking in computer conditionals: a psychological evaluation", *International Journal of Man-Machine Studies* **9**, pp. 107-18. (8)

[119] Sims-Knight, Judith E. and Upchurch, Richard L. (1993), "Teaching software design: a new approach to high school computer science", *American Educational Research Association* (Atlanta, Georgia). (41)

[120] Slack, James (2000), *Introduction to Programming and Problem Solving in Java*, Brooks/Cole. (9)

[121] Sleeman, D., R. T. Putnam, J. Baxter, and L. Kuspa (1988), "An introductory Pascal class: a case study of students' errors", in R. E. Mayer (ed), *Teaching and learning computer programming: multiple research perspectives*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, pp. 237-57. (27, 41)

[122] Sloman, Aaron (1984), "Beginners need powerful systems", in M. Yazdani (ed.), *New Horizons in Educational Computing*, Ellis Horwood, pp. 220-35. (118)

[123] Smyth, Graham (2001), "Survey on programming language preferences", *ACSEnt* (Association of Computer Studies Educators) **2**, pp. 1-3. (7)

[124] Soloway, E. (1993), "Should we teach students to program?", *Communications of the ACM* **36/10**, pp. 21-4. (3)

[125] Soloway, E., J. Bonar and K. Ehrlich (1989), "Cognitive strategies and looping constructs", in Soloway and Spohrer [126], pp. 191-207. (27, 118)

[126] Soloway, E. and Spohrer, J. C., eds. (1989), *Studying the novice programmer*, Hillsdale, New Jersey: Lawrence Erlbaum Associates. (contains [13], [41], [42], [56], [66], [68], [70], [75], [79], [102], [107], [114], [125], [127], [128])

[127] Spohrer, J. C. and Soloway, E. (1989), "Novice mistakes: are the folk wisdoms correct?", in Soloway and Spohrer [126], pp. 401-16. (43)

[128] Spohrer, J. C., E. Soloway, and E. Pope (1989), "A goal/plan analysis of buggy Pascal programs", in Soloway and Spohrer [126], pp. 355-99. (41)

[129] Stein, Lynn Andrea (1998), "What we swept under the carpet: radically rethinking CS1", *Computer Science Education* **8**, pp. 118-29. (119)

[130] Trott, Peter (1997), "Programming languages: past, present, and future", *ACM SIGPLAN Notices* **32/1**, pp. 14-57. (7)

[131] Tucker, Allen B., Keith Barker, Andrew P. Bernat, Robert D. Cupper, Charles F. Kelemen, and Ruth Ungar (1998), "Developing the breadth-first curriculum: results of a three-year experiment", *Computer Science Education* **8**, pp. 27-55. (97)

[132] Urban-Lurain, Mark and Weinshank, Donald J. (2000), "Is there a role for programming in non-major Computer Science courses?", *Frontiers in Education Conference* (Kansas City). (1, 3)

[133] Urban-Lurain, Mark and Weinshank, Donald J. (2001), "Do non-computer science students need to program?", *Journal of Engineering Education* **90**, pp. 535-44. (1, 3)

[134] Vygotsky, L. S. (1934), *Thought and Language*, published in English translation by MIT Press, 1962. (4)

[135] Warford, J. Stanley (1999), "BlackBox: a new object-oriented framework for CS1/CS2", *ACM SIGCSE Bulletin* **31/1**, pp. 271-5. (7)

[136] Wichmann, B. A. (1970), "PL516, An Algol-like assembly language for the DDP-516", UK National Physical Laboratory Report CCU9, at www.series16.adrianwise.co.uk/software/expl516/pl516/pl516.pdf. (12)

[137] Wilhelm, Reinhard and Maurer, Dieter (1995), *Compiler Design*, Addison-Wesley. (76, 77)

[138] Winslow, L. E. (1996), "Programming pedagogy – A psychological overview", *ACM SIGCSE Bulletin* **28/3**, pp. 17-22. (41, 43)

[139] Wolfram, Stephen (2002), *A New Type of Science*, Wolfram Media. (2)

[140] Zelkovitz, M. V., B. Kowalchack, D. Itkin, and L. Herman (1989), "A SUPPORT tool for teaching computer programming", in R. Fairley and P. Freeman (eds), *Issues in Software Engineering Education*, Springer-Verlag, pp. 139-67. (117)

Appendix A – Lecture Plans and Example Coursework

The following sections spell out the plan for each of the four compulsory lectures in the "Programming Concepts" component, designed to cover enough programming concepts and syntax to enable students to meet the *Delphi* system for the first time in Lecture 5. The lecture plans are in note form addressed to the lecturer, mostly copied *verbatim* from those produced for the session 2002-03, and are followed by an example coursework.

Plan for First Lecture on Turtle

(a) In *every* lecture apart from the last, emphasise that they shouldn't yet be worrying about the coursework – their concern should be simply to master what's been covered in the lecture concerned, and to have a go at the exercises that take them further. Also, they shouldn't need to make lots of notes because all the details are in the Help file – however do emphasise when you make a general "engineering" point that they should be noting it down (e.g. advice for debugging, tips for systematic working etc. that aren't in the Help).

(b) Start with the module handout, which provides syllabus and administrative details (but will need modification each semester to reflect lecture times etc.). When commenting on the structure, emphasise that the Pascal language taught in the core is the same language that is required for *Delphi*, so it's not just a toy.

(c) The first lecture's objective is to introduce the *Turtle* system to a level where students can explore it for themselves, and have a go at the first 4 exercises. Don't tell them that they're expected to complete the first 4 exercises before the next lecture; rather, say this sort of thing: "You'll benefit far more from the next lecture if you've at least had a go at exercises 1 to 4. But don't worry if you get stuck, and *don't* spend ages trying to solve problems if you encounter them – this is likely to do more harm than good. Next week I'll be going over what you need to do for these exercises, but what I say will mean more to you if you've at least made an initial try at them, even if you get virtually nowhere with them." Also advise them that the exercises provide a valuable fail-safe in case they mess up the coursework – they will be invited to submit their exercises along with the coursework, and in this case the exercises might enable them to pass even if their coursework fails (however this will not compensate for failure to make a serious attempt at the coursework).

(d) Show them how to access and start up the system, and emphasise that it can be downloaded for home use.

(e) Show them the Help system, especially the exercises section. Read some of this through, and show them how it advises to start with Illustrative program 1 (pointing out that there are other illustrative programs too). *Using this program as an illustration, informally cover all the points in the section on "The Program" which is linked from the exercises Help*, and afterwards draw their attention to the fact that everything you've said is there in the Help section, so they can read it through for themselves at leisure. Particular points to note when

running the program include the various areas of the screen (especially the program window, Canvas, "RUN" button, status bar); examples of program syntax (semicolons etc. – tell them not to worry too much about the details of this yet, just to note); and of course how the turtle's activity is determined by the commands. When showing them the Help section on "The Program", draw attention to the links it makes to the reference sections (give them a quick glimpse of these), and mention again the illustrative programs and exercises.

(f) Work through Exercises 1 and 2 in the lecture, illustrating as you do so how you are making use of the Help system to find details on new commands etc. Don't bother to do the face particularly carefully – but it's a good idea to show them how to do the mouth, illustrating how the order of commands can be important (e.g. so a white masking blot doesn't obliterate a nose you've already drawn).

(g) Exercises 1 and 2 give the opportunity to cover some of the menu features (especially in the File and Edit menus) – take advantage of this by drawing attention to them, for example by using the Clipboard commands to take stuff from the Help file.

(h) End by inviting them to do Exercises 1 and 2 for themselves before the next lecture, and to have a go at Exercises 3 and 4. Draw their attention to the illustrative programs on FOR loops, and take the opportunity to show them the Layout menu (because the second illustrative program requires the larger Canvas). Advise them to use the smaller Canvas in general, because then, *if* they find that their program moves beyond it, they can try running it instead on the larger Canvas (the other virtue of the large Canvas is that it's centred around the origin, and this can be nice if they're doing clever symmetrical patterns).

Plan for Second Lecture on Turtle

(a) Start by making sure that everyone has the module handout, and repeating the most important administrative and "don't worry" stuff.

(b) Again run quickly through the material covered in Lecture 1, adjusting your pace to take account of how many people said they didn't have the handout (i.e. had missed Lecture 1). If very few missed, then this can be significantly quicker than if lots did. But the repetition will be useful to build up confidence for everyone, so it's not wasted time.

(c) Work through Exercises 1 and 2 quite briskly.

(d) Work through Exercise 3 more slowly, showing them how to do the faces with a FOR loop (but probably leave the repeat loop for those who are keen to try for themselves). Make reference to the two relevant illustrative programs and point out how they work.

(e) Introduce variables as labelled boxes, and point out that although all the variables we'll be dealing with in *Turtle* are integers, declaration of variable types is essential once they get on to *Delphi*. Even in *Turtle*, booleans are available for keenies if they want to explore that for themselves.

(f) Draw attention to the importance of indenting, and its relation to program syntax. Emphasise use of the auto-format, and how this can reveal problems (e.g. if they put a semicolon immediately after "do").

(g) Advice when working with loops: (i) Start with the loop working once only (e.g. "for count := 1 to 1 do"); (ii) Draw a black blot at the end of the loop, so you can see clearly where the turtle's ending up; (iii) When that's OK, move on to the loop working twice only ("for count:=1 to 2 do"); (iv) Only when that's OK, move on to more. It helps to motivate this advice if you fix it that the first time you run Exercise 3, you get five faces in different orientations and on top of each other! Then you can illustrate how to fix the problem systematically in this way.

(h) Don't spend long on the nested loops, since it's important not to confuse. Show it briefly, and make reference to the relevant illustrative program, drawing attention again to the value of indentation.

(g) If there's time, briefly introduce the notion of a procedure, putting all the face commands into a procedure and showing how this makes the loop particularly easy to understand – point out that this is in Exercise 6.

(h) Finally, ask them to complete Exercises 1 to 4 before the next lecture, and invite them to try Exercises 5 to 8, but again emphasise that they shouldn't worry if they get stuck.

Plan for Third Lecture on Turtle

(a) Remind students about the multiple faces program, and introduce the notion of a procedure using this (as covered in Exercise 6), pointing out how it makes the loop(s) easier to understand (especially when there are nested loops). Again show the FOR loop illustrative programs, and remind them to look at these.

(b) Work through Exercise 5 systematically, drawing attention to all the points mentioned in the Help file in relation to it.

(c) Move on, as in the Help for Exercise 5, to the material in the section on "Procedures and Parameters" (up to and including "Introducing Simple Value Parameters"), again drawing attention to the fact that everything you are covering can be read there too, and also to the two relevant illustrative programs – the second one of these introduces DOWNTO, which is worth mentioning for the sake of completeness (but don't waste time doing any more on it).

(d) Now move on to recursion, again following the material in the "Procedures and Parameters" section (and again pointing this out) – take the opportunity to emphasise at appropriate points how a recursive procedure has to have its own local copy of the relevant variable(s), as explained in the section on "Scope".

(e) Start by introducing the Cat in the Hat (from the eponymous children's book by Dr Seuss), a mischievous cat who tries to be helpful but invariably causes trouble. A boy and a girl, left home by their mother with the job of clearing snow from the path, are visited by the Cat in the Hat who offers to help. However so far from helping, he simply manages to turn most of the snow red, so now the children have the extra job of cleaning the snow! It's at this point that the Cat in the Hat introduces his helpers, Little Cats A to Z, each of which lives inside the hat of the previous cat. The Cat in the Hat Powerpoint show gives four double-page spreads from the book, covering the unveiling of Little Cats A to G – read these through, in an appropriate rhythm to bring out the humour.

(f) Now copy and run the simple non-recursive triangles program in the "Recursion" sub-section, and explain that the procedure gives the instructions that a cat needs to follow for drawing a triangle.

(g) What's special about recursion, though, is that at any point in drawing its own triangle, a cat can "pop" another cat out of its hat, and instruct it to go and do something and then jump back into the hat, before the initial cat continues on its way. Illustrate this first by having the triangle procedure include just one recursive call, then all three (as in the illustrative program).

(h) When working through all this, it helps a lot if you walk out the steps of the various cats, and mime the hat operations etc.

(i) Show the students "The Recursion Factory" from the Help menu, and invite them to play with it to produce their own recursive designs. Explain how such designs can be saved to the Clipboard or to a file using the File menu.

(j) Finally, ask the students to complete Exercises 5 to 8 before the next lecture, and invite them to have a go at Exercises 9 to 12.

Plan for Fourth Lecture on Turtle

(a) Start by reminding students briefly about what was covered last time, making reference to the relevant illustrative programs and (especially) Help sections.

(b) Run the "REPEAT loop" and "Combining structures" illustrative programs, to show the kind of thing that's going to be covered in this lecture.

(c) Work through Exercises 9 to 11, following the order of presentation given in the Help file (which includes reference to the "Programming Essentials" section for the syntax and meaning of "if" statements).

(d) Draw attention to the commands UPDATE and NOUPDATE, which are mentioned in connection with Exercise 8 but are particularly important when simulating smooth motion.

(e) Having shown how to produce smooth motion of a single object, mention and demonstrate (but do not discuss) the illustrative programs on "Reference (VAR) parameters" and "Multiple bouncing balls", which ambitious students might wish to follow up themselves (with the subsection on "Value and Reference Parameters" from the Help section on "Procedures and Parameters").

(f) Briefly draw attention to the material covered in the remaining illustrative programs, and how this can be followed up using the Help file:

1. Cycling colours: RANDCOL as a random number generator, and use of MOD to cycle (note that MOD is also used in the "Combining structures" illustrative program to produce the steps, so it's worth pointing out how useful it can be for any oscillating pattern).

2. Using booleans: more randomness, and boolean variables.

3. Using POLYGON with FORGET: mention POLYGON and POLYLINE (both of which are relatively likely to be useful for *all* students), and explain how FORGET can be used to extend their power.

4. 3D effects with colour: just use this as an opportunity to draw attention to *Turtle*'s ability to handle colours in more powerful ways than have been shown before, making reference to the Help system for anyone who wants to follow this up.

(g) Finish off by discussing the coursework and associated administrative arrangements, and by reminding students about the final lecture on the *Turtle* compiler (to which all are invited, but for which attendance is not compulsory – advertise it as of particular interest for those who want to know what goes on "under the bonnet" of a real computer).

Illustrative "Programming Concepts" Coursework

This illustrates the typical content of a recent Turtle Graphics Programming coursework Web page, omitting cross-references, links, and general advice on submission method, deadlines, plagiarism, help sessions, newsgroups, example submissions, and other standard learning resources etc.

Outline

The coursework for this component falls into two parts:

Part I

Choose two from the following six features of Turtle Graphics Pascal:

(a) POLYLINE or POLYGON

(b) FOR – DO loops

(c) REPEAT – UNTIL loops

(d) PROCEDUREs with parameters (non-recursive)

(e) Recursion

(f) Colour codes and colour handling

Then illustrate each of your two chosen features with a short example program, preferably of less than 20 lines each, and aiming to satisfy the criteria given below (e.g. correctly named, clarity, interest, relevance). Note that these short example programs should be quite distinct from your main Part II program - you will not be given credit for merely extracting parts of that program and submitting them for Part I.

Part II

Construct a reasonably sized program (at least 50-100 lines altogether) which creates an interesting design of your choice, using every one of the types of command and structure mentioned in the Exercises page of the *Turtle Graphics* Help file (details of exactly what this requirement amounts to are given below, together with the criteria by which your program as a whole will be judged). Your design might be abstract, or it could be a picture. And it might incorporate movement, but it need not - however it should have a reasonable degree of complexity, produced in a clearly comprehensible way by making use of loops and procedural structures.

The program that you produce should contain at least two procedures, each of which should have a clear purpose. The specification of each procedure's purpose, and a brief description of the program's intended behaviour and of any issues you have encountered in its development, should be written in a text file and submitted with your programs.

Y*ou are also invited (but not expected) to submit your work on the built-in Turtle Graphics exercises, which may act as a "safety net" to help you pass if for some reason you have a disaster with your main program (e.g. if having done a lot of work on that program, you can't get it to run - you will NOT pass if you haven't at least made a substantial effort).*

Detailed Requirements and Assessment Criteria

Part I

Having chosen your two features of Turtle Graphics Pascal (from the six options of POLYLINE or POLYGON, FOR – DO loops, REPEAT – UNTIL loops, PROCEDUREs with parameters, Recursion, and Colour codes and colour handling), you should try to produce a short illustrative program for each of them, satisfying the following criteria:

- Programs named appropriately (see details of *items to be submitted* below).
- At most 20 lines in each illustrative program.
- Program *clearly* written and formatted (e.g. using the auto-format facility!), and easy to understand.
- Producing a visually *interesting* result.
- A *relevant* visual result, in that someone looking at it together with the program could see fairly clearly what use is being made of the feature being illustrated, and/or how that feature assists in creating the visual effect.

Part II

The main component of the coursework is to construct a relatively large program (at least 50-100 lines altogether) which creates an interesting design of your choice, using every one of the types of command and structure mentioned in the Exercises page of the *Turtle Graphics* Help file - this means that it should contain at least one instance from each of the following groups (and you will be given credit for showing yourself able to use a variety of commands, so it's a good idea to aim to use all of these commands and structures at least once, but don't feel you have to do this if doing so wouldn't fit smoothly into your program):

relative movement:	FORWARD, BACK, MOVEXY, DRAWXY
direction:	LEFT, RIGHT
absolute movement:	HOME, SETX, SETY, SETXY
shapes:	CIRCLE, BLOT, POLYLINE, POLYGON
pen settings:	COLOUR, RANDCOL, THICKNESS
drawing control:	PENUP, PENDOWN, NOUPDATE, UPDATE, PAUSE
procedures:	PROCEDURE (with parameters)
for loops:	FOR – TO, FOR – DOWNTO
repeat loops:	REPEAT – UNTIL
conditionals:	IF – THEN – ELSE
variable assignment:	:=

You may also want to consider using the following:

BLANK, CANVAS:	to control the appearance of the canvas;
REMEMBER, FORGET:	to control the behaviour of POLYLINE and/or POLYGON;
Arithmetical, relational and boolean operators.	

Your program is also required to contain at least two procedures, preferably both with parameters, each of which should have a clearly identifiable purpose. Ideally, these two procedures should each do a single "job" which is fairly easy to describe and understand but which is at least moderately complex (so each procedure should contain at least 5-10 commands). A procedure of this kind is particularly valuable, because not only does it make the program easier to understand by dividing it up into easily understandable tasks, but also, it can save you work in the long run because the very same procedure, once written and checked, can be copied into any other *Turtle* program where you need to do the same job (and if you got it right first time, won't need to be re-written or re-checked).

The specification of each procedure's purpose, and a brief description of the program's intended behaviour and of any issues you have encountered in its development, should be written in a text file and submitted with your programs. The point of this requirement is first, to show that you have thought about the design of your program and what it is intended to do; secondly, to show that you understand what your procedures do and how they contribute to the program's overall structure; and thirdly, to give you an opportunity to mention any aspects of your work that you are particularly pleased with, or any difficulties that you have spent time on and that you would like the markers to take into account.

Apart from the various points already mentioned above, the following criteria will be taken into account in marking your work:

- Program MUST run correctly, without syntax errors.
- Program MUST be titled with your own username – so if your username is "DEP1ABC", then the first line of your program should read: “PROGRAM dep1abc;”, and it should be saved under the filename DEP1ABC.TGP.
- Text file MUST be saved under the filename PROGDESC.TXT.

- Program *clearly* written and formatted (e.g. using the auto-format facility), and relatively easy to understand (taking into account the complexity of what it does).
- Program should be annotated with comments to help indicate what is going on in different parts of it.
- Producing a visually *interesting* (and maybe even *entertaining*) result.
- Appropriate use of commands and structures to produce this result.

Items to be Submitted

- TWO illustrative programs for Part I, satisfying the criteria specified above and with names each reflecting their purpose, so that the program title (in the first line of the program) is either EGPOLYLINE, EGPOLYGON, EGFOR, EGREPEAT, EGPROCEDURE, EGRECURSION, or EGCOLOUR as appropriate, while the filename of the program file is either EGPOLYLINE.TGP, EGPOLYGON.TGP, EGFOR.TGP, EGREPEAT.TGP, EGPROCEDURE.TGP, EGRECURSION.TGP, or EGCOLOUR.TGP.
- ONE main program for Part II, satisfying the criteria above and with a program and filename corresponding to your own username (as explained in detail above).
- ONE plain text file called PROGDESC.TXT describing your main program, as explained in detail above.
- *If you wish, the programs that you have done for the self-teach exercises in the Turtle Graphics Help file. Note (a) that to be taken into account, these exercise programs must be named correctly (e.g. "TGPX10.TGP"); and (b) that they will not usually count towards your mark unless they are needed to provide a "safety net" (e.g. where your main submitted program goes seriously wrong in some way, in which case evidence of having done the exercises competently could enable you to pass nevertheless).*

Advice on Approaching Your Work

You should *plan* what you are going to do in Part II of the coursework before you create your program. Get plenty of scrap paper to hand and think about what kind of design or effect you want to produce. Think about what you want your procedures to do, what global and local variables you will need and so on. As you begin to write your code, annotate it with comments so that you can keep track of where you are up to. You will find the coursework much easier if you have a clear idea of what you are trying to do in your program. This will also make it easier for you to write up your text file. You should also bear in mind that when your coursework is marked, the sections of your code will be examined in the context of how they contribute to the overall workings of the program. If you just throw in a random jumble of drawing instructions, you will end up with a program that looks a mess and is unlikely to produce a visually interesting result.

Always allow plenty of time for debugging and tidying up your program. This often takes much longer than working out the general ideas behind the program!

Appendix B – Possible Future Developments

The development and refinement of a system such as *Turtle* is potentially endless, because there are so many useful features that could be added to the editor, the environment, the Visual Compiler, and the help resources; indeed it would be relatively straightforward to fill an entire thesis with a discussion of the possibilities. Since space does not permit any such extended discussion, I shall confine myself here to merely listing a few of these, together with references to some relevant literature that suggests possible alternative approaches. Deek and McHugh [35] provide a particularly useful survey of many varied systems for supporting the learning of programming, making reference to most of those mentioned below.

The Editor

Currently the system's editor is relatively basic, and it could usefully be enhanced with such features as search and replace, syntax highlighting, and multiple file processing, to mention only the most obvious possibilities. More radical suggestions might include a syntax-tree-based code generator such as that incorporated within the SUPPORT system (Zelkowitz et al. [140]) which also uses Pascal, or automated translation from pseudocode as provided by SCHEMACODE (Robillard [110]), which supports Pascal as well as other syntaxes. However such options would require integration with the standard text editor if the system were to continue to provide familiarisation with conventional code editing, as required for it to serve as an appropriate bridge to *Delphi* or *Kylix*.[65]

---

[65] For the same reason, I do not consider here the use of graphics-based programming tools along the lines of Pict (Glinert E. and Tanimoto [45]), Amethyst (Myers et al. [87]), DSP (Olsen [92]), ASA (Guimaraes et al. [48]), or BACCII (Calloni and Bagert [23]).

The Environment

Another worthwhile addition might be a library of templates and examples, as provided by the Example-Based Programming System (EBPS) of Neal [89]. Sloman [122] suggests a large number of other features worthy of consideration within a programming environment for novices, including interactive stepping through a program, incremental compilation and various "intelligent" aids. Similar themes are also explored in Atwood et al. [8], while Pane et al. [96], though primarily focused on the design of a visual environment for children, draw attention to a number of usability criteria that would also be worth bearing in mind.

The Language

For the teaching of loops, it is useful to have a "break" command like that of *Delphi*, likewise "exit" to quit from a procedure (enabling a "middle exit" strategy which Soloway et al. [125] identify as being helpful for novices). Mutual recursion would be less restricted if the system had a "forward" or "deferred" directive (cf. note 44). Other additions that might be worth considering, if only as selectable options (to avoid potential overload for novices), would be functions, "while" and "case".

Interactivity is currently not possible while a program is running except to halt it, and output is restricted to geometrical shapes. Other possibilities include fixed or program-definable input boxes and buttons (together with events), and display of number strings. Arbitrary text strings would require more radical revisions to the Turtle Machine, and introduction of a string type would imply extension of the range of basic concepts conveyable by the system, since type checking would then be essential. Even if integers and booleans remain the only primitive types, such checking might be a useful addition. Arrays would also be valuable, especially if visual output of numbers is facilitated, since this would enable the system to be used to illustrate ideas of sorting and complexity. If the Visual Compiler is extended accordingly (e.g. with counting of primitive operations), then some of the ideas from the systems reviewed by Chalk [27] might be worth considering.

More fundamentally, it would be desirable to incorporate additional language syntaxes, notably that of Java, which could be combined with moving to an object-oriented approach. The notion of a turtle as an object is very natural, and would open the door to multiple turtles which can interact (as explored by Resnick [108] with his development of StarLogo). Modern systems are increasingly composed of interacting objects with independent threads of control, and object-orientation would enable programming to be taught in the spirit of this new paradigm, as strongly advocated by Stein [129] under the motto "*computation as interaction*". A less fundamental but more easily manageable change would be to allow Java syntax but remain procedural, taking advantage of the restriction to introduce intertranslatability between Pascal and Java syntax (which might in itself have a significant educational value for novices).

The Compiler

Leaving aside changes to the compiler implied by the suggestions above, it might be useful to extend the range of error messages and warnings, also adding corrective hints of the type described by Lewis and Mulley [73], for example to warn students of apparent identifier/keyword conflicts, duplicate identifiers in nested scopes, variables not used or not initialised etc.

The Illustrative Programs and Tutorial Material

If new features are added such as those listed under "The Language" above, then it will obviously be appropriate to update the illustrative programs (and corresponding tutorial Help file sections) accordingly. Apart from this, an interesting possibility is to develop the discussion of recursion into the area of fractal patterns, perhaps linking this with the theory of Lindenmayer systems ("L-systems"), for the display of which Turtle Graphics is already known to be a useful vehicle (Bridges [16], Alfonseca and Ortega [5], Proulx [106]).