Introduction to Programming Commons Portfolio

Dimitar Kazakov
Department of Computer Science
University of York
Heslington, York YO10 5DD, UK
kazakov@cs.york.ac.uk

20 June 2006

Abstract: This portfolio contains reflections on the teaching of the first introductory programming module, Principles of Programming (POP) at York. The document is a result of a joint effort, the High Education Academy project *Disciplinary Commons in Computing Education* (Oct '05 – June '06) headed by Sally Fincher (Kent), which brought together UK academics teaching introductory programming with the aim of creating a shared resource sheding light on the way the subject is approached in the participants' home institutions. Creating this material has helped the author's process of self-reflection and analysis; it would be of interest to anyone (re-)designing a similar module, and it may also help familiarise new members of staff at York with the course content, type of students, and teaching style. Finally, the existence of this resource should facilitate further coordination of the curriculum across all relevant subjects of the Department's Bachelor's and Master's degrees.

1. About the Lecturer

As for any other subject, the way the basics of Computer Science and programming are taught is influenced by the past experience of the lecturer, and the way he was introduced to the material. This should justify a snapshot of the author's professional history:

- Secondary school of Mathematics (1981-1985):
    - incl. programming: IBM 360/Fortran 4, PL/1, later Apple-style PC/Basic
    - First term: algorithms only, flow charts but no coding.

- 5-year degree in Electrical Engineering, Dept. of Control Engineering, CTU, Prague (1988-1993).
    - Diploma (MSc) project in Natural Language Processing.

- PhD in Artificial Intelligence and Bioinformatics (CTU, Prague, 1993-1999):
    - thesis on Natural Language Processing applications of Machine Learning.

- RA jobs at York, then lecturer since 1999:
    - York Certificate of Academic Practice (YCAP) (ILT certified) as part of the contract requirements.
    - Previously taught Implementation of Programming Languages, Lexical and Syntax Analysis, Logic Programming and Artificial Intelligence, Symbolic Learning of Language, Adaptive and Learning Agents.

2. The Context of Teaching Introductory Programming

At present, students in the department study a 10-credit Autumn term Principles of Programming (PoP) module in their first year. This is followed by Algorithms and Data Structures (ADS) in the Spring and Summer terms of Year 1. Some of the other related modules are Theory of Computation (TOC) (10 credits, Autumn/Y2), Lexical and Syntax Analysis (LSA) (10 credits, Spring & Summer/Y2), Logic Programming and Artificial Intelligence (LPA) (Spring & Summer/Y2) and Code Generation and Optimisation (CGO) (10 credits, Autumn/Y3). There is also an optional Crash Course on C (CCC), (0 credits, end of Summer term), which is taught in the first week following summer examinations, and is replayed on video the week after for anyone who has missed some of the lectures. This module is not assessed, and is offered to all students.[1]

Aut/Y1	*Spr/Y1*	*Sum/Y1*	*Aut/Y2*	*Spr/Y2*	*Sum/Y2*	*Aut/Y3*	*Spr/Y3*	*Sum/Y3*
				LPA	LPA			
		CCC			CCC rep.			
POP	ADS	ADS	TOC	LSA	LSA	CGO		

Fig.1: Timing of POP and related modules

Programming Languages

The department is traditionally strong in real-time systems and safety-critical engineering, so it comes as no surprise that Ada has played an important role in the teaching of programming. The introductory programming module used to be based on Ada. Around 1995, a strong case was made for the use of a language with simpler syntax and regular semantics that would give the students the chance to concentrate on the general programming concepts and minimise the initial overhead needed before they could write their own programs and get hands-on experience (Wood, 1995). A lightweight dialect of LISP, Scheme, fitted the description, along with some additional benefits, such as a freely available platform and a good textbook (Abelson et al., 1996), and was adopted as the language of choice for the course. The module has been amended and fine-tuned in the course of the last 10 years, while still based on the same language, textbook, and teaching approach: lectures where theoretical principles are immediately illustrated with hands-on examples on the lecturer's laptop, weekly software labs, and two open assessments delivering both code and a description of the solution.

The PoP module is followed by ADS, which plays a dual role at present -- to teach Ada as well as cover the traditional content of a module with this name.[2]

[1] This will change from Oct 2006, when the CCC module will be offered in Week 1/Aut. The module will be offered to all, but is seen as particularly useful to second-year students.

[2] There has been a recent proposal to split these two topics into separate modules to make teaching better structured and more transparent to students.

The LSA practicals (software labs) are based on C. Many students would already have used C, and the rest could opt for the CCC module at the end of Year 1. The language is also taught in a couple of lectures at the start of the module. LSA and CGO used to be taught as one module, Implementation of Programming Languages (IPL), in the Spring and Summer terms of Year 2. IPL was split to make room for a networking module in the second year, which was deemed necessary background for students who would take an industrial placement year at this stage.

Prolog is used to teach LPA in the second year. The AI part of the module has replaced functional programming, which was previously taught with Prolog in the Declarative Languages (DEC) module. The material on functional programming is now taught as a separate third year option (FUN). In this way, by the end of the second year, students will have become familiar with the following programming paradigms: functional programming (Scheme), imperative programming (Ada, some C) and declarative programming (Prolog). In addition, the notions of abstract data types, objects, and methods are discussed in PoP, but several aspects of OOP (class hierarchies and inheritance) are still not covered. A number of students will use Java for their final-year project, but will be expected to learn it on their own. Real Time Systems (RTS) (Spring and Summer/Y3) discusses certain aspects of Java, including OOP, but practicals are based on Ada95.

Students

York CS is a 5** (*aka* 6*) department (5* rated in the last two RAEs), and keeps attracting top students despite the nation-wide decline in applications for CS courses after the peak in 2001. One would expect the vast majority of students to have AAB or comparable A-levels. They will progress to the third and fourth year of their course to take options, which would often be directly related to their lecturers' research. A small number among them will produce a final-year project of publishable quality, and may co-author a peer-reviewed research paper with their supervisor, e.g., (Turner and Kazakov, 2002), (Frisch *et al*., 2006).

The students' background on arrival is very varied: some have years of programming experience in a range of languages (from microprocessor assembly to VB, C++ and Java), others - a substantial minority - will be limited to browsing the WWW and using MS Office. Even the ones who can program may not be able to put their skill in the wider scientific and engineering context of the field. The tutorial on *Computer Science vs Software Engineering* in Appendix A gives a certain insight in the state of mind of the students at the beginning of their first year (this is one of the recommended topics for discussion in the tutorials, in which small groups of first year students - typically 4 - meet their tutor on a weekly basis).

Staff

York CS is a research-driven department, whose teaching has also been praised (judged "excellent" by HEFCE). Individual and research group-wide interests may impose a bias in the teaching of programming in the first two years. For instance, teaching Prolog and concepts of Logic Programming facilitates the teaching of several 3rd and 4th year Artificial Intelligence modules (Natural Language Processing, Constraint Programming, Adaptive and Learning Agents), teaching Scheme also provides for the needs of the Functional Programming group, and the 3rd year module on Real Time Systems makes heavy use of Ada, first introduced in ADS (Year 1). Most members of staff will supervise around 5 undergraduate and MSc projects a year. Many of these projects are related to the staff research interests, and the knowledge of a certain programming language is often a prerequisite for the project.

(Wood, 1995) Wood, A. Principles of Programming. Internal Memo, CS Dept., University of York.

(Abelson et al., 1996) Abelson, H., Sussman, G.J. and Sussman, J. *Structure and Interpretation of Computer Programs*. MIT, Second edition.

(Turner and Kazakov, 2002) Turner, H. and Kazakov, D. Stochastic Simulation of Inherited Kinship-Driven Altruism. *Journal of Artificial Intelligence and Simulation of Behaviour*, p. 183-196, 1(2). [ps | ps.gz]

(Frisch *et al*., 2006) Frisch A., Peugniez, T. Doggett, A. Nightingale, P. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. *Journal of Automated Reasoning*, January 2006.

3. Aims and Philosophy of POP Teaching

Aims

The aims of POP teaching at present are similar to the ones indicated by Wood (1995). POP has to focus on teaching programming concepts, rather than a specific language, and the language chosen should minimise the initial period of passive, lectures-only learning, and allow the students to start getting hands-on experience as soon as possible.

The module should illustrate some of the basic concepts. At the end of the module, the student should be able to write a short program (say, ~100 lines of code) following a clear specification. The student should be able to discuss his/her choice of programming techniques and appreciate issues of time and space complexity. S/he should be able to implement some degree of error handling and have an intuitive (implicit) concept of the debugging process as a systematic effort.

Philosophy

At present, the POP course is designed around the following beliefs:

- writing and executing code in the lectures is more efficient than using the blackboard alone
- lectures have to outline and complement the textbook material, and a substantial amount of self-study is expected
- students need to practice in the labs the material covered in lectures
- the 2 open-book assignments are assessed, but have a substantial formative role and provide feedback, as well as encouragement and motivation to students who may not have had previous programming experience.

Furthermore,

- teaching introductory programming should use a minimum of “magic”. In other words, there should be a minimum of programming machinery that students get to use without understanding its role or the principles on which it is based.

4. The Content of Introductory Programming Teaching

At the moment, the POP module content is, by and large, given by the content of the textbook used to teach it: *Structure and Interpretation of Computer Programs*, by H. Abelson, G. J. Sussman and J. Sussman, MIT Press. The textbook, in turns, was chosen on the basis of a clear argument made by the then POP lecturer (see A. Wood, *Principles of Programming*, 1995). In a nutshell, the choice of an appropriate language was to be supported by an existing textbook and freely available language compiler/interpreter. The language had to have simple and regular syntax, first-class objects, and the ability to implement states. An interpreter would eliminate the need to understand the mechanics of compilation at this stage, and would facilitate experimentation with the code, hence active learning. The document listed the following programming concepts, which were to form the backbone of POP teaching: *Value, Name, Type, Process, Abstraction, Choice, Modularity, State, Sequence, Repetition* (see also the POP outline syllabus in Appendix B).

The Scheme language fitted well this description, and came with a free interpreter and the above mentioned textbook (whose second edition appeared in 1996). The leading concept underlying all material in the textbook is abstraction. It is used to discuss the way in which increasingly complex, compound procedures are built from simpler, and/or built-in ones. Similarly, data is discussed in an increasingly abstract way, starting from primitive data (built-in data types), and progressing to introduce complex (compound) data, tagged (typed) data, and abstract data types (ADT), incl. ADT interfaces.

Scheme allows the lecturer to introduce a *range of programming paradigms*. At the moment, for over half of the term in which the module is taught, a purely *functional subset of Scheme* is presented. In this way, the lecturer can introduce the simpler substitution model of evaluation, and delay the discussion of sequence, assignment, state, scope and environment somewhat. Recursion is taught early, and used to introduce the theoretical aspects of space and time complexity on the case of recursive vs iterative processes.

This opens the road for the Theory of Computation (TOC) module taught in Aut/Year2.

With the introduction of assignment, one can adopt an *imperative programming* style, and immediately point out the pros (meeting the need to represent state) and the cons, such as the need to move to the more complex environment model of evaluation, with its related discussion of scope. The issues related to the order in which formal parameter expressions and local definitions are evaluated are used to demonstrate the need of this new evaluation model.

At this stage, the lecturer has the chance to provide a hook to Implementation of Programming Languages (currently taught as Lexical and Syntax Analysis (LSA, Spring/Year2) and Code Generation and Optimisation (CGO, Aut/Year3)). One can, for instance, discuss the nature of scoping and the use of frames to represent the local bindings of an environment, and speculate about the possible efficient storage techniques, opening the road for the concept of stack.

In the last third of the module, procedures and data are combined in a message-passing programming style to form one of the prerequisites for object-orientated programming. (At this stage, local definitions and scope are discussed as an efficient way of data encapsulation, rather than an issue that the evaluation model has to take into account.) The combination of assignment (allowing the implementation of state) and message-passing style is then used to introduce the *object-orientated programming* paradigm.

The material familiarises the students in considerable detail with the inner workings of an object, including such subtle issues as whether a full copy of all local procedures is made or pointers to the class definitions are stored when an object is created, with the implications on the object's behaviour in the case of a later change in the constructor definition. The persistent nature of an object can be shown as an argument against the use of stack, and, again, one can hint at the need for dynamic memory allocation. The discussion of the usefulness of objects ends with the bid to abstract from all examples seen, and makes the case for a procedure that can be used to define object con-structors. OO languages are then mentioned, as well as the fact that such functionality can be provided in Scheme, as long as one knew how to use to use meta-programming constructs (eval, apply,…) present in the language.

The concept of data is extended to include compound data. All compound data is built out of pairs, although the special case of lists is paid considerable attention. Lists allow great flexibility, and are easily manipulated by recursive procedures.

They can also be easily extended to the more general case of streams (as delayed lists), and so provide the lead for a future discussion of issues related to data input and output (in ADS, Spr&Sum/Year1).

Mutation of the elements of a pair is introduced as an analogue to assignment for primitive data types. This allows one to discuss the issue of pointers, and their potential and related dangers, separating this from the specific issue of pointer arithmetic, which is only relevant to some languages (e.g., C, but not Ada). Here the need to free dynamically allocated memory is mentioned, together with the two major approaches (doing it explicitly or using garbage collection). The danger of memory leaks is pointed out, and students are encouraged to appreciate the advantages of garbage collection, which Scheme, in line with its Lisp roots, faithfully employs.

Pair mutation is also used with association lists, composed out of (key, value) pairs, to implement tables. Tables are then used to implement operator overloading for a range of abstract data types and multiple data representations. In fact, tables are first made available as an ADT and used for the just mentioned purpose well before their implementation is discussed, making in this way a point in favour of ADT as an efficient separation of the issues of what and how. Typed data and overloading also lead to a discussion of coercion and type hierarchies.

Vectors and arrays are not covered in POP, but they are briefly discussed as a specific case of association lists, which permits an efficient implementation, both in terms of demands on memory and access times; here the key is not explicitly stored, but rather reflected in the array member's address, thus making for faster (constant time) access.

Among the ideas that the module does not cover at the moment, but could benefit from, are the following: discuss code testing and debugging; an example of OOP code and how it incorporates the OO ideas; possibly a reference to continuations (cf. third year Principles of Unconventional Programming (PUP)) and their relevance to non-deterministic programs; search and logic programming (a lead to Logic Programming and Artificial Intelligence (LPA, Spr&Sum/Year2)). Of course, all these would have to find place in an already dense curriculum, so some adjustments would need to be done to accommodate these additions, and the trade-off would need to be considered.

5. Instructional Design

The Principles of Programming (POP) module is taken by all ~110 first-year students, including those on joint programmes (such as CS/Math). The module is taught in a combination of lectures and software lab practicals over the 9 weeks of Autumn term. There are 21 x 1h lectures and 7 x 2h practicals. Two open-book assessments are handed out on the Monday of Weeks 5 and 9, and collected on the Friday of the same week. There is no closed-book assessment.

The instructional design as outlined was inherited from the previous lecturer. The module was rated highly by the students, and this success seems to have been preserved after the hand-over. The module is built around a weekly cycle in which two lectures introduce one or two concepts, which are consolidated in the third and last lecture of the week. Except for the 2 weeks of the assessments, there is a weekly practical that should allow the students to practice the new material (and receive feedback and help from demonstrators) with a minimum delay. Due to the choice of programming language, students are able to start coding in their practicals from the first week of teaching (sometimes, less than 24h after the first lecture...).

The size of the class makes lectures the only feasible form of presenting the new material to students. (For comparison, the students are usually split into three separate groups for the practicals due to lab size limitations.) To keep their interest, and allow them to see coding and execution of software first hand, the lectures combine the use of slides (transparencies) and demonstrations of examples of code, which are written on the lecturer's notebook, then executed to demonstrate the effect of evaluating each expression. This presents an image of programming as a process, with its errors, considerations of the different merrits of alternative solutions, and one which is guided by testing. The Scheme interpreter used is set to produce a log of everything done, which is made available to the students after the lecture, so they can study the examples and repeat them for themselves. As mentioned in Section 3 (Aims and Philosophy), the use of Scheme avoids the need to use programming concepts and language constructs before they are defined. The very first “programs” (Scheme expressions) must appear familiar to the students regardless of their background, as it is a question of writing simple arithmetic expressions (addition, multiplication, etc.) and finding the result of performing the operations, all in a way similar to the one in which a calculator would perform the task. (The only unusual aspect here is the use of reverse Polish notation.)

The process of typing an expression and obtaining a result is familiar *per se*, but for the large majority of the present generation of students, who have grown up using a computer through a GUI for information processing, communication, and interactive games, the notion implied by the semantics of the word `computer' is unfamiliar and appears over-restrictive. Exposing these students to this cultural shock early on seems an essential element of the demistification of the computer perceived as an active agent with anthropomorphic qualities, in order to replace this perception with one of a tool that is used to externalise the mental processes of its programmer.

In this way, the use of functional programming as the first programming paradigm facilitates the initial introduction of the concept of programming, and allows the students to experience the comfort stemming from being operationally successful (able to achieve results) through actions that are well understood. This is not necessarily the case if one was to begin with, say, coding imperative programs in C or Java where the student would be told to type a number of lines on top and bottom of the actual code being taught, with the only likely explanation of “that's how it works”. (It is not implied here that those students will never understand the role of that extra code, simply that they would have to suspend their belief until these details become clear.) However, the same students would have acquired from the beginning the notion of programming as the process of writing a program that is usually editted (and compiled if needed), then executed to produce a number of changes in the computer state and output as a result. On the other hand, a POP student needs at present to modify his/her concept of programming as a whole a number of times, from the typing and immediate evaluation of an expression to the writing of a number of such expressions in a file, which can then be loaded and all expressions in it evaluated sequentially in a batch mode, a step that can arbitrarily be followed by further evaluations of expressions typed directly into the interpreter. Introducing imperative and OO programming changes and extends this concept further still.It is not clear whether POP students at present are consciously aware of this gradual shift, and, while they may understand the individual concepts in the new material, one may have to consider the need for an explicit reminder of the way in which the top level concept of programming is ammended.

The assessments, which are an important element in the instructional design, are discussed in the next section.

Artefact: a typical lecture (slides, notes, log).

6. Principles of Programming Assessment

While they are assessed, the two assessments have a strong formative element, and are aimed at providing the students with the opportunity (and incentive) to address a problem unassisted, and gain confidence, something which those with no programming experience will need to face to overcome the peer pressure of those who have programmed before. The assessments also serve as a revision for the first, resp. second half of the module. Having only one assignment would unacceptably delay this challenge, and would not provide the chance to test/assess alternative programming styles, as it is at present (purely functional programming without the use of state or assignment in the first, a mixture of imperative and object-oriented programming in the second). Having more than 2 assessments would make marking too laborious, if the assessments preserve their 2 components – code, submitted electronically, and a printed description of the solution, including all code.

Students often feel insecure about their interpretation of the assignment, and try to contact the lecturer for reassurance. I have found that providing the students with a small initial draft of part of the code, and a transcript of the expected behavour of the solution help a lot to reassure them and avoid misunderstandings.

The assessment provides feedback about the students' level of understanding and ability to work independently well beyond the work in the carefully guided step-by-step practical sessions. It is intended also to give students a feel of their real rather than perceived degree of understanding. The large volume of submissions to mark can delay the process considerably. In Aut/2004, an automated test returning a single signal (A, B,...,F) was first used to provide students with a quick partial feedback. This was extended in Aut/2005 to include signals on individual tests (there were around 15 such tests in total). Rather than publishing the tests themselves, the students were told of the code properties (part of specification) that these tests evaluated (See the feedback sheet artefact, as well as the 2 exam papers)

This is still rather late, as the first feedback becomes available after 2/3 of the term has passed. Automated weekly tests (involving minimum or no lecturer's intervention) would be the ideal solution. Instant feedback to student voting in lectures (*Who wants to be a millionaire* style) would also help, although they could not replace individual code submissions.

Another issue worth mentioning is the need to avoid or at least minimise the amount of cultural/linguistic references used in the assignment. For instance, I know (from bitter experience as a student) that even decks of cards, an otherwise handy example of objects with properties, such as *suit* and *rank,* can be confusing: I was baffled when my Czech lecturer in Statistics started to talk about red, black and *green* suits! Another example is the typical POP assignment where the program has to spell out a number (e.g. 145 → one hundred *and* fourty five). Students whose mother tongue is not English may find formulating the rules more challenging than the rest, despite the fact they have been provided with a number of examples.

Summary
Five of the most important aspects of assignment are:

1. Its formative nature, hence: open, hands-on, done in 2 rounds. The assignments forces the students to program "for real", without consulting the answers that a textbook or the practicals provide (although the latter are published with a one week delay). Comments like "*it was an eye opener*", "*I've learned more in the assignment than in the practical*s" are common in the students' feedback.

2. It should motivate students: at least a third of them have never programmed. This is the very first assessed assignment in Year 1, and it can have a great impact on the students' perception of assessments in general.

3. Provide timely feedback to lecturer and students. At the moment, the first formal (as opposed to demonstrators' comments in practicals) feedback students see, is the automated test results from their first assignment, which arrives quite late, around the sixth week of teaching.

4. It should assess real code to encourage hands-on skills (hence the automated test), but give marks for a "near-miss" solution (paper submission). The initial draft of part of code + transcript of SW's expected behavour have proved very helpful.

5. Difficult to avoid cultural/linguistic references:
    - Decks of cards not necessarily the same
    - Spelling out numbers requires a level of proficiency in English.

7. Evaluation

There several channels through which one can obtain feedback about the quality and impact of one's teaching. There are listed below, then each of them is discussed.

1. Assessment results
2. Student feedback forms (per module)
3. End of term forms
4. Peer review
5. Performance review
6. Direct contact with students in lectures and practicals
7. Contact with students in first-year small-group tutorials

Assessment results: the proof is in eating the pudding. Provided the assessment is well designed, it should provide a good picture of the students' level of understanding. While testing the submitted code is the most critical probing stone, reading the written report permits to identify the areas and concepts that are particularly problematic, and judge programming style as well as performance. On another level, reading through the code can be a way of spotting potential issues of plagiarism and collusion. Experience shows that assignments of similar style (and, of course, covering the same material) tend to repeatedly highlight certain topics as the ones most likely to pause problems to students.

Among these is recursion, especially when used in combination with lists, implementing counters in a functional programming context (without the use of state), implementing counters in objects (when local state *has* to be used), or distinguishing between data types (e.g., the number 5 and the list containing it as its only member).

Some of the difficulties are particularly common among different groups of students: the ones with programming experience, typically in an imperative or OO programming language, can find it very challenging to manage without the use of state, whereas the ones with no programming experience may struggle to express their knowledge in a rigorous form that meets the exact specification, and may feel that completely failing an automated test because of data type mismatch is too harsh and injust.

There is only so much one can do to provide additional support for these topics, and it is a truism that some concepts are inherently more complex and difficult to learn than others. This means that many (most) aspects of the difficulties with the assessment will reappear and can be anticipated. As a result, marking a large number of reports may not be very informative, and will considerably delay the feedback of the marker to students.[3] An interesting alternative may be to change the first assessment to only require the submission of code. The results of an automated test can be provided almost instantly, together with a single summary of the most common problems and good practice examples. This summary can be prepared ahead of the assessment, and possibly fine-tuned on the basis of a sample of all submissions. This will relieve the lecturer from the burden of marking and give him more time to interact with students.

[3] At the moment, open -book assessment feedback has to be provided to students within 4 weeks. To meet this deadline at a time of intensive POP teaching, additional markers have to be used.

The maximum overall POP mark is 50 = 2 x 25 marks (for each of the assessments). Fourty percent of each assessment come from the automated tests, the rest is split between the quality of the report (40%) and the quality of the solution (20%). Appendix D presents the guidelines for marking the second POP assessment report (the paper submission). These are similar to the ones used for the first assessment.

Student feedback forms: Students fill in online feedback forms for each module, including POP. These usually cover a number of aspects (delivery, content, availability of materials, facilities, feedback, etc.). Students can also add free-form entries.

End of term forms: Students meet their supervisors (tutors) at the end of each term and fill in a form that has separate entries for each module. Comments are then summarised and passed to lecturers, or, less often, discussed directly between lecturer and supervisor.

Peer review: All members of staff go through an annual peer review of their teaching. Not all modules are necessarily covered in one year, but individual years focus on different teaching styles – large group teaching, small group teaching, etc., which means that POP is covered biennially in the average. The lecturer passes on the main aims of the teaching session to his/her reviewer; written feedback is returned after, which is then discussed and a mutually agreed record is filed at departmental level.

Performance review: Teaching is covered as part of the annual performance review. The lecturer has the opportunity to discuss various aspects of his teaching with his reviewer, who is another member of staff. This is first done in writing, then the two meet to discuss the lecturer's performance review (report). They also agree on suggestions regarding the improvement of teaching (research, etc.), which are passed on to the Head of Department.

Direct contact with students in lectures and practicals: Having taught second-year students for a number of years, it was surprising to discover that first years' seem readier to engage in a dialog – ask or answer questions or help the lecturer with suggestions while coding an example.

Contact with students in first-year small-group tutorials: The lecturer meets with his first-year tutorial group of four students on a weekly basis. While most tutorials are on a dedicated topic, the meeting provides a chance to obtain informal feedback on POP teaching. In addition, some of the first tutorials in the autumn (such as the one in Appendix A) help gauge the students' understanding of basic concepts related to POP.

8. Delivery

At the moment, most of the new material is first seen in lectures. The material is outlined on OHP slides, then the lecturer's laptop is used to demonstrate examples of coding and execution on the data projector screen. The previous lecturer handed virtually all currently used material, including extensive notes, which outline the course of the lecture in great detail. OHP slides are often gradually completed to illustrate an example. This makes their scanning and posting problematic. It is considered gradually to transfer the material in the form of Power Point slides, where animation can be used with a similar effect.

Students are often prompted to answer questions or make suggestions. It is difficult to get actively involved such a large audience, but coding and running examples helps keep them focussed. Taking a 'vote' in answer to a question can get the students involved, and allows one to gauge their knowledge and confidence. Arguing the pros and cons of a statement reduces the chances of them being embarassed if proved wrong, as the lecturer shows there is a certain rationale behind either position.

POP is one of the first modules the students take, and a foundation stone for much of the subsequent teaching. The perceived `esoteric' nature of the programming language used can mislead students into thinking that this is material that can be covered, and forgotten once the exam has been sat. It is important to make it clear that this is not the case. To that purpose, it is often pointed out in lectures that certain topic will be taken further by another module, much along the lines of the `hooks' suggested in Section 4.

When they enter university, students often identify Computer Science with writing programs (see Appendix A). In the same way, they are likely to use POP as a prime example of what they will be taught in the rest of the BSc or MEng programme. It would be a pity if the links with other areas of knowledge or art, which are present in the Departmental staff research, are not pointed out in our undergraduate teaching. POP presents several oportunities for this, and, when taken, students' feedback suggests they tend to draw their point home. For instance, while teaching recursion, one can mention Hofstadter's "Gödel, Escher, Bach: An Eternal Golden Braid". (A student who read the book wrote an enthusiastic comment on the POP feedback form.) The use of `(define ...)` in Scheme to introduce aliases is taken to an extreme to replace the language vocabulary (including `` `define' `` itself) and highlight the difference between vocabulary and syntax. One can link this to families of human languages sharing syntactic similarities, such as the default order of Subject, Verb and Object in the sentence (e.g., SVO for English and Bulgarian, SOV for Japanese).

9. Reflections

Taking part in the Programming Commons meetings and writing this portfolio exposed the author to prime examples of originality and professionalism in teaching, and made him consider the future course of the POP module. Some of the examples of good practice can be adopted with minimum effort, while others require initial investment in time or money; others still appear tempting, but need to be judged in the context of this Department's teaching. There are also the author's own ideas, some new, some contemplated before, but never in the clear form that results from writing them down. Here are some specific examples of the above categories.

Debugger for the programming language of choice: A visit to Dundee (see Jim Bown's Commons portfolio) has revealed that a tailor-made environment can be used to provide programming support with the teaching examples that goes beyond the feedback provided by a standard parser. Templates written for each programming examples guide the student to check for the presence of the main structural components in the code, and suggest their addition if not detected. The observed tool was written for Java. However, the excellent facilities for metaprogramming that Scheme provides mean this could be done to serve POP teaching in York. A proposal for a third-year project aiming at developing such tool has been written (Appendix E), and a student will start work on it in October 2006.

Wireless Voting Devices: A number of wireless handheld sets allowing the students to vote for one of N (N<10) options, *Who Wants to Be a Millionaire* style, can help take an instant poll and collect statistics of students' understanding of material. The devices have been used to teach CS in Glasgow, and an enquiry at the CS Department in York has shown that a number of other lecturers would also use them in their teaching.

MP3 recordings of lectures: This is an easy and low-cost way of providing additional support for visually impared students or those with dyslexia. Again inspired by the Commons, the idea has got the support of the departmental disability officer.

A cross-reference index linking POP-related concepts with other subjects: This would play a two-fold role. Firstly, it will provide students with a resource linking the main concepts in POP with the other modules in the curriculum, reinforcing in this way the importance of these concepts and helping make the connection with previously covered material when another related module is taught. Similarly, it will help new lecturers get their bearings in the host of modules, and help all members of staff avoid redundancy in the material, as well as discrepancies in style and content. Implementing this index as a Wiki Web site will permit any interested lecturer in the department to add their references and maintain the resource up to date.

Study of the correlation of marks across modules: At present, several statistics are collected for the marks of each module at departmental level. Personal experience with second-year teaching has shown that, in the past, these statistics were closely correlated for some modules (IPL and NDS when both were taught in Year 2). It would be interesting to see how POP marks are correlated with related modules in Year 1 and 2, and if such correlation is discovered, consider its significance, e.g., whether it indicates that similar skills are assessed in the two modules, or knowledge of one module has helped understand the other, or it indicates the student's gift for the subject.

Use of rote-learning: Rote learning is an essential aspect of learning to speak a human language well (e.g., one French teacher used to make students repeat commonly used phrases involving complex verb moods and tenses, rather than have them think each time about the use of Subjunctive in the phrase “I have to go”). It is clear that active coding, rather than reading about the programming language syntax alone, is an important part in becoming a good programmer. It would be interesting to experiment with the closest equivalent of rote learning – copying examples of code illustrating the same concept and parsing them (or executing/evaluating them) to verify they've been copied correctly. For instance, one can supply a number of examples of tail recursion and see whether the students are more likely to write correctly a control example from a specification. It is an interesting observation that the use of self-reference in recursion can be used with success to design a “win-win” didactic problem. The student may be asked to write 50 times the computer code equivalent of “Write *this* 50 times”. By typing repeatedly the code of this recursive function, they are likely to memorise its structure. However, it would be acceptable if they wrote a recursive function that printed this code 50 times, as this would be a proof that they have mastered recursion...

Appendix A: Computer Science vs Software Engineering First Year Tutorial

Suggested by
Dimitar Kazakov

Linked to courses
BSc/BEng, MEng

Suggested timing
Suggested timing: Early in Autumn term (W2-4)

Purpose of the exercise
Initiate a discussion on the nature of Computer Science and Software Engineering, as understood by the students in the group, which should lead to a mutual understanding of the expectations about the nature of the course (programme).

Outline of the exercise
The BSc/BEng degree some of the students are on is in "Computer Science", while others are probably studying for the MEng degree in "Computer Systems and Software Engineering". Your choice of current studies is the result of months - if not years - of research and careful considerations. Surely, you should know what it is about... So,

- What is Computer Science?
- What is Software Engineering?

Suggest definitions for these two disciplines, and use examples to illustrate their scope. Write keywords/phrases/ on the blackboard in an attempt to cover all relevant aspects, and reach consensus.

Solution/notes for the tutor
Students often arrive without a clear idea of the balance between the scientific and engineering aspects in the 3 and 4 year programmes. While some expect to spend most of their time coding - and assume that an ad hoc approach to it is all right, others may believe that (given our emphasis on good grades in Maths), most of the material in the programme will be approached in the rigorous framework of formal methods and proofs.

Despite the fact that interviews routinely include questions about the nature of our programmes, and what students expect from them, when asked about the scope of Computer Science, students in their first tutorial have repeatedly stated that it deals with various aspects of software. If they are asked subsequently what Software Engineering is about, they are often puzzled, as the obvious answer has been used up, and the answer surprisingly often is "about hardware"! Significantly, in groups where the questions are asked in the opposite order, the answers tend to pair CS with hardware, and SWE with software. Clearly, this is not a satisfactory state. Before joining the discussion more actively though, it would be useful to let the students describe their different views. It usually provides a good amount of insight into the personal experience with computers and programming, as the aspects the student perceives as desirable are usually highlighted. In this way, they can also share some personal information relevant to the course (programme) they are on.

Let the students start a discussion on HW vs SW, but don't let them slip into it for too long. Instead, once the true dichotomy is outlined, suggest to get back to the HW/SW topic in another practical. You can feed the discussion in a variety of ways (do not try to cover all):

- Look at the first and second-year modules (POP, ADS, TOC, LPA, LSA, MSD, RDQ...), outline the content in a few words and invite students to suggest whether it falls mostly under CS or SWE, and why. A side effect of this is a better understanding of the first 2 years' programme content, and how these modules are related (and feed into) each other.
- Briefly describe the typical course of a final-year project (lit.review, design and implementation, testing, evaluation). Give a CS-ish project as an example (e.g., a fictitious one implementing and comparing 2 sorting algorithms), then try to lead the students see the difference between the CS content, and the SWE-ish life-cycle framework.
- Show the group the outline of D. Knuth's published (and planned) volumes of The Art of Programming, then suggest that all that is covered, is Computer Science, and the rest - SWE. (While this is a somewhat exaggerated, hence contentious statement, it makes a point that is easy to remember - and not far from the truth...)
- Discuss the pros and cons of teaching introductory programming concepts through a variety of languages, which are not necessarily popular with industry (e.g., Scheme), vs. using an industrial-strength platform from the very beginning (e.g., VB.net at the time of writing (2006)). Highlight the importance of putting an emphasis on sound CS theory early on.

Suggestions for further work/discussion

Definition of Computer Science (Wikipedia)

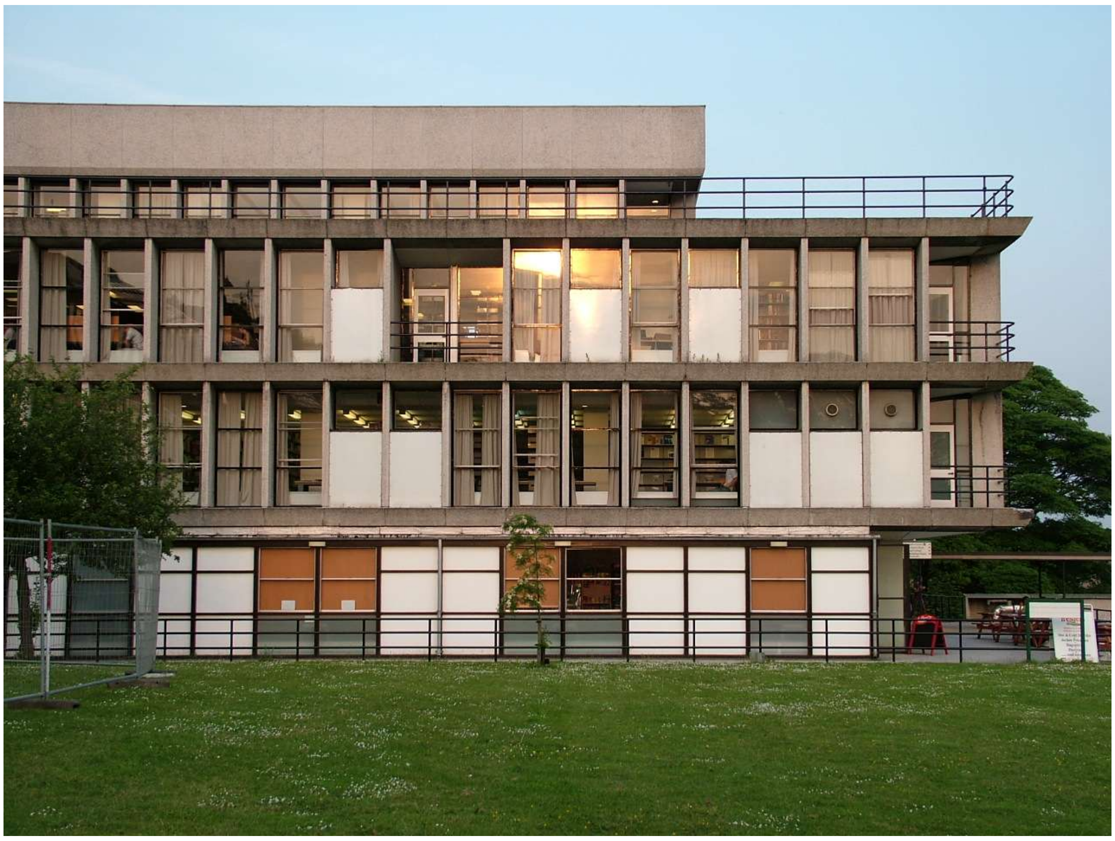
Definition of Software Engineering (Wikipedia)

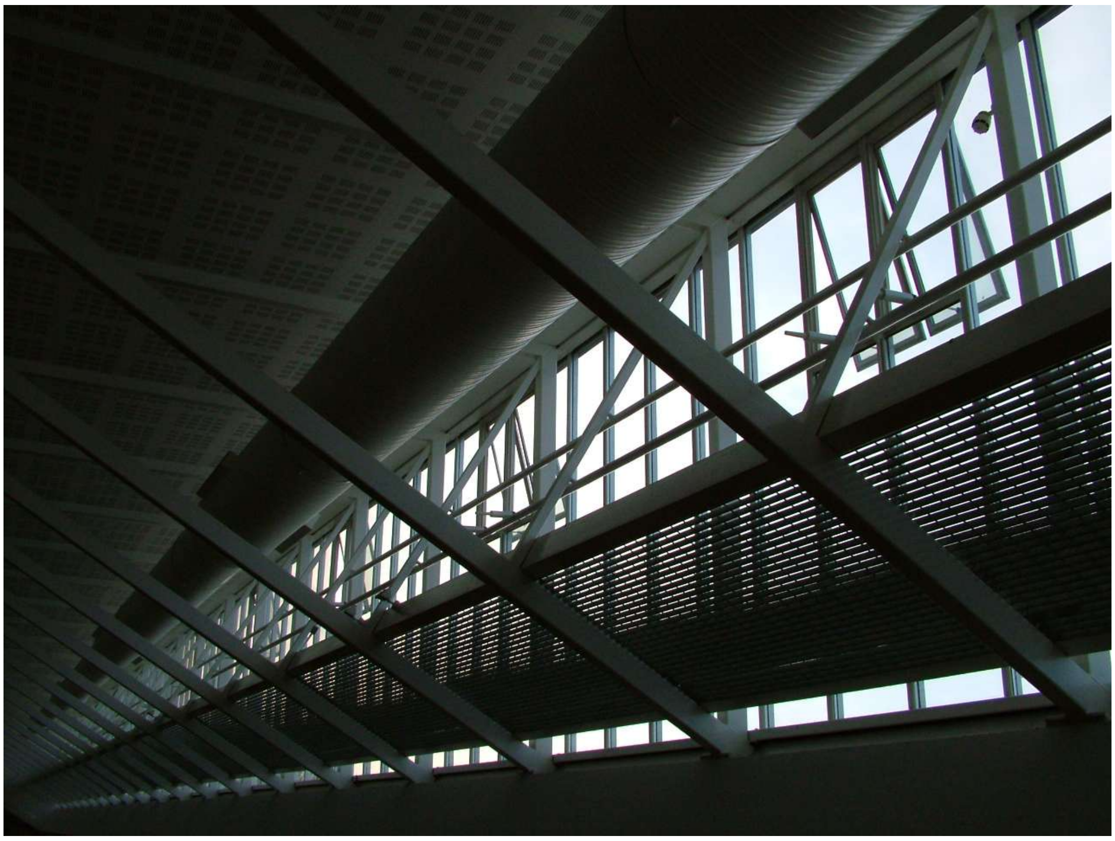
Appendix: POP Outline Syllabus (by A. Wood)

Wk	Lecture			SICP
2	**1**	**2**	**3**	1.1
	Names, Variables, Values and Expressions Numbers; symbols (`quote`); representation *v.* value; Definitions	**Procedures** Procedure definition; `lambda` as a procedural value; Procedureal abstraction; Choice (`if` and `cond`; Substitution model; Evaluation order;	*Consolidation* [Ex: Zeller, leap-year]. Abstraction; Value; Function; Name; Variable; Expression; Operational model	
3	**4**	**5**	**6**	1.2 (skip 1.2.5 & 1.2.6) 2.2.1
	Repetition Recursion [Ex: factorial]; [Ex: Newton's method]	**Data (1)** Abstraction; Compound data; Lists; [Ex: `last`, `reverse`] *{SICP 2.2.1}*	*Consolidation* [Ex: Binary numbers]; local definitions; Repetition; Recursion; Idioms; Function; Scope; Abstraction	
4	**7**	**8**	**9**	1.3
	Computational Processes Iterative process [Ex: fibonacci]; Orders of Growth (intro);	**Higher-order procedures (1)** Procedural parameters	*Consolidation* Iteration; Big-O (basic) Eficiency; Process; `lambda`; Value; Abstraction	
5	**10**	**11**	**12**	1.3, 2.1 - 2.3 (skip 2.2.2 - 2.2.4, 2.3.3 & 2.3.4)
	Higher-order procedures (2) Procedural results	**Data (2)** Interfaces; Multiple representations; [Ex: Rational Numbers]	*Consolidation* [Ex: Symbolic Differentiation]	
6	**13**	**14**	**15**	2.4
	Abstract Data Mutliple representations;	**Types** Need for; Manifest; Coercion; Higher-order procedures (revisited)	*No Lecture* [Open Assessment]	
7	**16**	**17**	**18**	2.5
	Generics Data-directed style; Table-driven [Ex: Complex numbers]	**Generics** Polymorphism [Ex: Combining number types]	*Consolidation* [Ex: Coercion (explicit, and implicit)]	

8	**19**	**20**	**21**	3.1, 3.2
	State and Assignment Assignment: `set!`; Environments;	**Objects and Message Passing** Constructor functions; Local state; Access functions; Interface procedures [Ex: The Blood Donor]	*Consolidation* [Ex: Bank accounts]	
9	**22**	**23**	**24**	3.3
	Mutation Box-and-popinter diagrams; List mutators; Equality;	**Mutation** Tables;	*Consolidation* Summary of *The Principles of Programming*	
10	**25**	**26**	**27**	
	No Lecture Second	**No Lecture** Open	**No Lecture** assessment	

Appendix C: A Sense of Place – The York Campus





Appendix D: Marking Guidelines (Second Assessment)

1. Solution features

1.1 Overall design:

A – Clearly and systematically follows a given design paradigm (e.g., top-down, bottom-up); where alternative solutions are possible, (some) explanation is provided for/against the choice made; shows a sense of style, i.e., avoids the eclectic use of alternatives if one can be used throughout the program.

B – almost all of the above (resp. the above is almost fully applicable to the submission).

C – the majority of the listed in A.

D – some of the listed in A.

E – almost none of the listed in A.

F – none of the listed in A.

1.2 Error case handling:

A – Demonstrates confident use of error handling techniques ensuring software robustness and appropriate feedback generation (for instance, through a range of tests that gradually refine the information about the source of error, e.g., (number X) then (integer X), then (interval X min max) in the case of type checking); present in a number of procedures, including the ones expected to be most commonly used and all for which error handling is explicitly requested in the assignment.

B – F: same as above (repreated for all following cases).

1.3 Appropriate use of assignment:

A – Assignment (set!) should only be used where needed, i.e., to change the state of an object.

1.4 Use of abstraction:

A – Effectively abstracts from the SW specification to generate reusable code and minimise redundancy, striking a balance between initial effort and payoff with respect to the programming effort required, while also considering the impact on the resulting code clarity.

1.5 Use of objects:

A – Encapsulates and keeps away from the user as much as possible of the details of its implementation; implemented in a uniform way including (1) constructor returns a pointer to a procedure, (2) message passing used to invoke local procedures (object methods), (3) a consideration is given to the value returned by local procedures changing the object state.

1.6 Elegance:

A – There is harmonious simplicity in the choice and arrangement of steps; the solution is ingenious, convenient, and effective.

1.7 Code clarity:

A – Uses mnemonic names in a consistent fashion; appropriate layout; related procedures listed as closely to each other as possible.

2. Report features

2.1 General quality:

A – Clear, concise, to the point, combines well portions of code with the narrative, uses a straightforward layout (as instructed).

2.2 Procedure descriptions:

A – All non-trivial procedures are explained; the narrative complements the code rather than repeat it literally; variable names are given an appropriate meaning (and not discussed as arbitrary symbols); (small) illustrative examples used where appropriate; procedure description related to its use in the larger context of the program.

2.3 Description of testing:

A – Considers and provides evidence of appropriate SW behaviour in a substantial number of cases, including the correct behaviour of all procedures explicitly defined in the assessment, and some of the most commonly expected erroneous uses.

Appendix E: Third Year 2006 Project Proposal – Debugging Support Tool for Teaching Scheme

Supervisor: Dimitar Kazakov
Prerequisites: Scheme (to the extent covered in POP)

Aims: Implement a debugger that provides first time Scheme programmers with support in the form of type checking and schematic analysis signalling whether a range of expected features is implemented for a given problem class.

Background: Scheme is the language used to teach the introductory Principles of Programming module in this department. While having many didactic advantages, the minimal debugging support can pause a problem, especially to first-time functional programmers. On the other hand, Scheme, similarly to any other LISP dialect, allows the programmer to treat data and programs interchangeably, and use metaprogramming (e.g., the procedures eval and apply) to implement an evaluator (interpreter) that could serve as a debugger and provide the user with additional feedback.

Methodology: in its simplest form, the process of handling what is a de facto code as data, and implementing type checking and evaluating the correct expressions, is introduced in POP Lecture 12 on the case of symbolic differentiation: (follow the link file:///n/course/pop/Lectures/2005/12 for the implementation (all *.scm files) and a transcript of use, e.g. file:///n/course/pop/Lectures/2005/12/16-Nov-2004.txt).

The project will test and visualise the type for each submitted expression: procedure arity, value type, and argument type(s) - where possible. It will also focus on specific programming styles and tasks that appear in the POP module (e.g., tagged data or object orientated programming) and detect whether certain aspects of the implementation, such as type checking or message-passing implementation of objects, are present in the code. Appropriate feedback should help the user produce code that more closely matches the style shown in the lectures. In Aut 2006, POP students will be asked to log a transcript of their practical sessions:

```
> (transcript-on "myfile")
```

then these examples will be fed into the debugger to see in how many of the cases where the student got stuck, the debugger provided (appropriate) help. The appropriateness would be judged by a predefined list of criteria.

Reading

- Structure and Interpretation of Computer Programs (especially Chapter 4: Metalinguistic Abstraction), Abelson, Sussman and Sussman. Second edition, MIT, 1996. URL: http://mitpress.mit.edu/sicp/full-text/book/book.html
- GNU Emacs Lisp Reference Manual, Section Debugging Lisp Programs, http://www.info.uni-karlsruhe.de/~i44www/html-docu/elisp/elisp_16.html.
- Scheme programs for debugging Scheme programs. http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/scheme/code/debug/0.html.

Other Artefacts:

1. Alan Wood. Principles of Programming. Internal Memo, CS Dept., University of York, 1995. (PDF)
2. 2006 POP programming assessment No 1. (PDF, 5p.)
3. POP practical on *Project Allocation* (aka 2005 2nd POP programming assessment). (PDF, 5p.)
4. A sample run of the project allocation sample solution.
5. Sample student submissions for the above 2 exam papers (anonymised).
6. SICP textbook URL: http://mitpress.mit.edu/sicp/full-text/book/book.html
7. Notes and slides for POP lecture 23.
8. POP module description.
9. Assessment results (anonymised).