A Review of Australasian Investigations into Problem-Solving and the Novice Programmer

M. de Raadt[1]

University of Southern Queensland, Australia

Programming is an important skill in computing. This Australasian focussed review compares a number of recent studies that have identified difficulties encountered by novices while learning programming and problem-solving. These studies have show that novices are not performing at expected levels and many novices have only a fragile knowledge of programming, which may prevent them from learning and applying problem solving strategies. The review goes on to explore proposals for explicitly incorporating problem solving strategy instruction into introductory programming curricula and assessment, in attempts to produce improved learning outcomes for novices. Finally, directions suggested by the reviewed studies are gathered and some unanswered questions are raised.

Keywords*: Novice programmers, comprehension, generation, knowledge, strategy, plans, schema, patterns, introductory programming curricula*

1. Introduction

The goal of introductory programming instruction is to train novices in the art of programming. It is usually expected that, by the end of a semester long introduction, novices will be able to write simple programs. Many anecdotal reports suggest failure rates in introductory programming courses are higher than in other courses. A multinational ITiCSE 2001 working group involving researchers from the UK, USA, Israel, Poland and Australia, known as the "McCracken Group", set out to test the competency of novices after a one or two semester course of programming (McCracken et al., 2001). In this study, three related programming exercises were devised and 216 novices were asked to attempt one of the problems. When the attempts by the novices were marked using a common rubric, the average score was 23 out of a possible 110, which led the researchers to conclude that "many students do not know how to program at the conclusion of their introductory courses" (p. 125). With data collected at four participating institutions, in three different countries the study implied that the problems faced by these particular novices were common among novices around the world. The McCracken study did not, however, identify sources of the inadequacies demonstrated by the participating novices' performance, or potential fixes for these problems. What the McCracken study did offer was an opportunity to accept the failings of the past and begin to discover new curricula which could better encourage novices to reach expected standards.

A review by New Zealanders Robins, Rountree, & Rountree (2003) covers areas where cognitive psychology research meets programming and asks the key question: "What are the properties of expert programmers?" (p. 2). Robins et al. begin by categorising the work of researchers, from Australasia and around the world, who have attempted to answer this question. A number of dimensions are introduced by Robins et al. and these dimensions will be used as a context for comparing later works in this review.

- **expert-novice**
  The point at which a programmer is defined as an *expert* may be after 10 years (Winslow, 1996) or it may be when they can produce the best designed solution to a problem (Rist, 1995). The Robins

[1] Authors Address: Department of Mathematics and Computing, University of Southern Queensland, Toowoomba, Queensland, 4350, Australia. Email: deraadt@usq.edu.au

el al. review raises a number of studies that have attempted to uncover the characteristics of expert programmers and how they differ from novices.

- **knowledge-strategy**
  Knowledge involves the declarative nature of a programming language while *strategies* describe how programming knowledge is applied (Davies, 1993). Programming strategies are made up of plans (Soloway, 1986) (or schema or patterns) and the associated means of incorporating these into a single solution. Robins et al. portray strategies as being important but ill-defined in literature.

- **comprehension-generation**
  Novices may be asked to *comprehend* a given piece of code and describe how it works. Alternately they may be asked to *generate* code that solves a problem.

These dimensions are clearly related. For example, experts and novices can be distinguished by how they undertake comprehension (Brooks, 1983) or generation (Rist, 1995). During program generation an expert can rely on a tacit body of programming plans developed through solving past problems (Soloway, 1986) while a novice has traditionally been expected to conceive and apply plans, with varying degrees of success (Rist, 1991). The distinction of expertise by use of strategy is suggested by Bailie (1991, p. 277): "one feature clearly distinguishing the novice from the expert programmer is the ability to plan." One of the significant aims for instructors, raised by Robins et al., is the incorporation of schema instruction in introductory programming curricula. Finally, Robins et al. define a distinction among novices as being *effective* or *ineffective*. Effective novices learn to program with little assistance, while ineffective novices fail to learn how to program, or do so only with a great deal of assistance. Robins et al. suggest the key to encouraging novices to become effective lies in the application of programming strategies rather than acquisition of programming knowledge.

1.1 Australasian Context

Of the 48 universities in Australasia, all but one offers an introductory programming course[2], many offering two or more. An Australasian census of introductory programming courses was conducted in 2001 (de Raadt, Watson and Toleman, 2002) and again in 2003 (de Raadt, Watson and Toleman, 2004). The most recent census uncovered 85 introductory programming courses (71 in Australia and 14 in New Zealand). The census covered aspects of language, tools, paradigm, teaching hours, textbooks, instructor experience and other classifiable features. An interesting finding of the most recent census was that instructors are not in agreement on what constitutes problem-solving instruction in an introductory programming course. The proportion of instruction devoted to problem-solving varied greatly in the courses covered by the census. Some participants indicated that teaching problem-solving strategies was not part of their course; several of these instructors felt problems used in their teaching were not of large enough scale to warrant teaching problem-solving strategies explicitly. Others said that their entire lecture time focussed on teaching of problem-solving strategies. These instructors did not uniformly distinguish teaching of programming strategies from programming knowledge in their teaching. The current review refers to problem-solving specifically in programming (not in a more generic sense) as strategies which are beyond programming knowledge, which are applicable to comprehension and generation, and which are likely to be more developed in an expert than a novice.

[2] In this article, term "course" is used to describe a six month period of study, which might also be referred to as a "subject", "unit" or "paper".

This review examines articles published primarily during a period from the year 2000 to 2007. Authors include researchers primarily from Australia and New Zealand but also, through collaborative works, authors from outside Australasia. Section 2 looks at local studies which have attempted to measure the comprehension and generation abilities of novices. This section considers if novices can move beyond knowledge to strategy. Section 3 reviews studies which have attempted to describe how programming strategies can be incorporated into introductory programming curricula and evaluated. Finally conclusions are made summarising Australasian research in this area and future directions for this research are suggested.

2. Novice Problem Solving

During the 1980s, there was much work in various disciplines aimed at distinguishing novice behaviour from expert behaviour. Computing education researchers borrowed ideas from cognitive psychology. The aim of such early cognitive research was to first develop a theory of expertise in programming, then to see where novices failed to grasp this expertise. The idea of a *schema* or *plan* was established to describe expert strategies. Soloway and Wolf (1980) suggest a definition of plans as problem solutions. The description of plans is given from the perspective of teaching LISP but the authors suggest the description can be generalised to other languages. The potential of a taxonomy of plans is proposed. In a later paper, Soloway, Ehrlich and Bonar (1982) state, "In our work we have assumed that high level plan knowledge is used by expert programmers. Our goal has been to tease out that tacit knowledge and make it explicit" (p. 56). In perhaps what is the definitive *plan* paper, Soloway (1986) suggests the following.

> …language constructs do not pose major stumbling blocks for novices... rather, the real problems novices have lie in “putting the pieces together,” composing and coordinating components of a program. (p. 850)

Soloway then suggests that teaching should reach beyond a focus on syntax as programming knowledge and focus on programming strategies through mechanisms like plans. *Goal/Plan Analysis* is the process of describing an ideal solution, which contains appropriate plans, and comparing this with the solution of a novice. This analysis allows an instructor to see if they have succeeded in learning and applying these plans. Much of the research that followed applied the idea of plans to discover misconceptions novices exhibit (Spohrer, Soloway and Pope, 1985; Spohrer and Soloway, 1986). PROUST (Johnson and Soloway, 1984) was one of a series of intelligent tutoring systems. PROUST could perform Goal/Plan Analysis on a Pascal program and compare its plan structure to one established by an instructor. Johnson (1986) gave a description of the inner workings of PROUST and also, for perhaps the first time, released a catalogue of goals and related plans. During the 1990s investigations by Australian researcher Rist (1991; 1995) revealed how novices expound and apply plans. But in general, the idea of the schema/plan was not used by instructors, until the rise of the object paradigm, which brought with it a new sense of reuse and a new term to computing: *patterns*. According to Clancy and Linn (1999), "learning programming means learning patterns and strategies that enable rapid learning of new programming languages" (p. 37), but novices do not infer patterns naturally, and so instructors should "create appropriate exercises and supports so students extract patterns, reuse patterns, develop a disposition to use patterns, and create patterns of their own" (p. 41).

In Australia the work of Soloway and his colleagues was taken up by de Raadt, Toleman and Watson (2004) who attempted to measure their novices' problem solving ability according to their use of plans in a simple program generation exercise. An experiment was conducted with 42 novices who had not been exposed to the notion of plans and were expected to have learned plans implicitly. At the end of a semester of programming instruction, novices were asked to write a solution to a simple averaging problem, previously used by Soloway (1986), for which a solution in plans was well defined. The solutions created by the novices were analysed using Goal/Plan Analysis. Results were analysed for the presence or absence of the plans. Associated methods for incorporating plans were analysed also. The presence or absence of these aspects is shown in Figure 1.

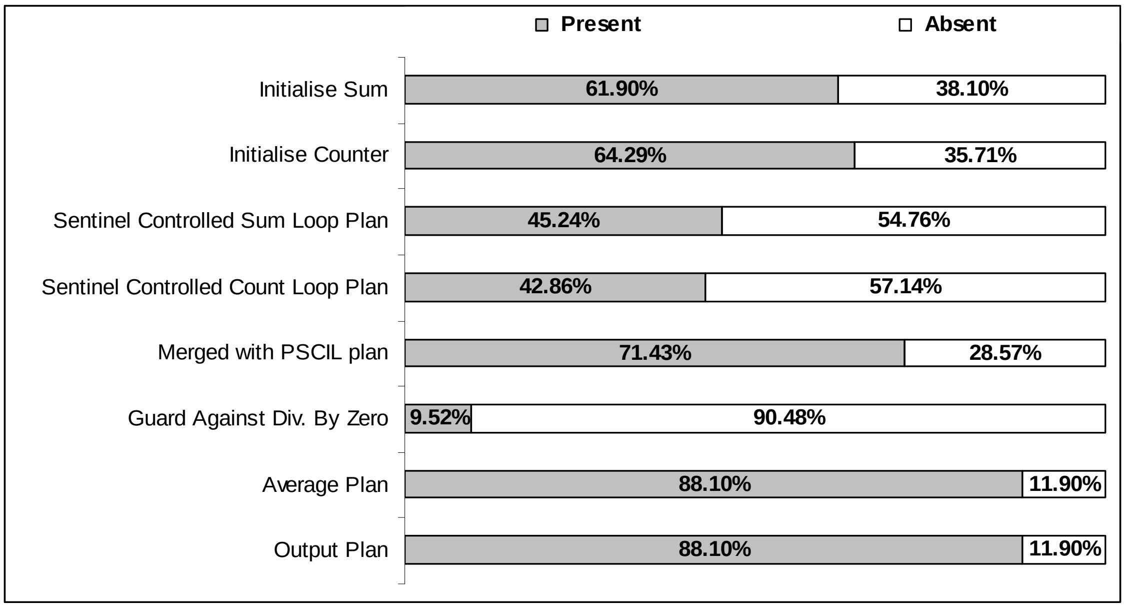

Figure 1. Novice performance in Goal/Plan Analysis (de Raadt, Toleman et al., 2004)

Only one of the 42 participants applied all expected plans. Of the eight measured aspects needed for a correct solution, the novices on average were able to demonstrate use of four. The application of one particular plan *guarded division* was particularly poor; de Raadt, Toleman and Watson admit this may be caused by teaching using problems which avoided boundary conditions, thus shielding novices from implicitly learning such plans. Ignoring guarded division, 23% of students were able to demonstrate use of all remaining plans. The authors claim this study showed weaknesses in the curriculum used at the time and suggest the need to incorporate strategies explicitly in future curricula.

An ITiSCE 2004 working group (Lister et al., 2004), known as the "Leeds group", involved researchers from the UK, USA, Denmark, Finland, Sweden, Australia and New Zealand. This group used a set of multiple choice questions to study the potential of novices to trace through and comprehend existing code or insert missing code segments. The premise of this study was that if novices could successfully trace through the code then this would indicate their programming knowledge was complete and the fault indicated in the McCracken study may lie in novices' problem-solving abilities. If there is no fault with the programming knowledge of novices, we can put this issue aside and focus our attention on how to improve problem-solving strategy instruction. However, if novices were not able to complete the tracing exercises successfully, this would indicate a fragile knowledge of programming concepts which, the Leeds group suggested, are prerequisite for problem-solving. Participating novices did not perform as poorly as those who participated in the McCracken study, nor did they perform universally well. The distinction between the third and fourth quartiles in the Lister study (shown in Table 1) is at 5 correct answers out of the set of 12, which is a performance consistent with guessing.

> Suppose the students who participated in this study were all studying their first semester of programming at a single institution. Suppose further they were given these 12 MCQs as their exam, and the institution regarded a 25% failure rate as the upper limit of what was acceptable. Then students who scored 5 out of 12 on these MCQs would be progressing to the second semester programming course (p. 128).

The Leeds group concluded much of the poor performance of novices demonstrated in the McCracken study may be attributable to novices' possessing only a fragile programming knowledge.

Table 1. Performance in the Leeds study (reproduced from Lister et al., 2004)

Quartile Score	Range	No. of Students	Percent of Students
1st “top”	10 – 12	152	27%
2nd	8 – 9	135	24%
3rd	5 – 7	142	25%
4th “bottom”	0 – 4	127	23%

The questions used in the Leeds study have been criticised for being arbitrary (Whalley et al., 2006). The design of these questions was not driven by past research in order to elicit specifically identified programming knowledge. The study can also be criticised by examining the underlying premise used in the study: that by asking novices to comprehend code, no problem-solving ability was required. If we compare this premise to the dimensions established by Robins et al., comprehension-generation and knowledge-strategy are separate dimensions. There is probably some relationship between these two dimensions, but they are not co-dependant. It is therefore unfounded to conclude that generation exercises require novices to possess programming strategies while comprehension exercises do not. Another possible criticism, admitted by Lister et al., can be found in their assumption that programming knowledge is required before problem-solving can be achieved. While this assumption seems rational, it has never been proven.. Ideally, as suggested at the conclusion of the Leeds study, the relationship between comprehension and generation should be further explored.

A following study, the "BRACElet project" (Whalley et al., 2006), conducted entirely within Australasia, extended the McCracken and Leeds studies by using Bloom's Revised Taxonomy (Anderson et al., 2001), to guide construction of questions. For example, "fixed-code" questions, which asked novices to trace through a piece of code and predict its output, can be categorised at the *execute* sub-category within Bloom's cognitive level *apply*. "Skeleton-code" questions, which asked novices to fill in a blank in code with one of a list of options, can categorised into the *apply*, *evaluate* or *create* levels. The BRACElet study included questions which can be categorised in the Bloom's levels of *apply*, *understand* and *analyse*, with specified sub-categories for each question. The BRACElet study generated a set of questions justified a priori as covering a range of cognitive ability, as defined by the revised Bloom's Taxonomy. The overall results of the multiple choice questions of this study show varied results across the 117 participants. Two questions with the poorest student results had been a priori categorised at the highest cognitive level used from Bloom's Taxonomy level *analyse*. The question with the best student results had been categorised a priori at the lowest cognitive level *understand*. These boundary results seem to confirm the BRACElet group's claims for the cognitive difficulty of the question set. Two questions from the Leeds study were repeated in a revised form with similar results. Overall, the participants' results in answering these better defined multiple choice questions were slightly superior to results from the Leeds group, but again many novices demonstrated gaps in their programming knowledge.

As well as completing the nine multiple choice questions, a subgroup of the BRAClet participants were also shown a piece of code and asked to "In plain English, explain what the following segment of code does" (Whalley et al., 2006, p. 249) (now referred to as Question 10). The responses to this question were categorised according to levels of the SOLO Taxonomy (Biggs and Collis, 1982) which distinguishes levels of understanding. Such a categorisation is a departure from the earlier performance-based categorisations of the McCracken and Leeds studies and produced intriguing results. Responses were categorised as shown in Table 2.

Table 2. SOLO Categorisation of Question 10 responses (reproduced from Whalley et al., 2006)

SOLO category	Description
Relational [R]	Provides a summary of what the code does in terms of the code's purpose.
Multistructural [M]	A line by line description is provided of all the code. Summarisation of individual statements may be included
Unistructural [U]	Provides a description for one portion of the code (i.e. describes the if statement)
Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question
Blank	Question not answered

Approximately 30% of participants in the BRACElet study were able to give a SOLO Relational response, 55% gave a Multistructural response, 13% gave a Unistructural response and a small remaining percentage showed only a Prestructural response. This means that most novices were describing code, at best, line-by-line. Less than a third of novices were able to identify the overall purpose of the code. The BRACElet authors contend "that a vital step toward being able to write programs is the capacity to read a piece of code and describe it relationally" (p. 250). On the comprehension-generation dimension this means that before a novice can generate code, they must first show comprehension at a SOLO Relational level for an equivalent piece of code. If this assertion holds, less than a third of the participating novices could generate code which, as in the comprehension example given, checks if an array of numbers are in ascending order. This description could provide a possible crossing-point between the comprehension-generation and knowledge-strategy dimensions. When comparing the results of the multiple-choice questions with the SOLO category attained for the "explain in plain English" question (Question 10), a correlation was found with stronger students likely to show higher levels of understanding and poorer students showing weaker understanding. The SOLO depth of understanding may relate to the problem-solving ability of a novice programmer. A number of strategies are apparent in the example code used in the experiment. If the novice does not possess the strategies, not only would they be unable to generate code which applies these strategies, they may not be able to comprehend code which applies these strategies. This seems to be an answer to a 20 year old challenge: "by understanding how a programmer, say, goes about comprehending a program, we can then pinpoint where he/she is having difficulties, and thus be in a position to design languages/methodologies/tools that can address these problems" (Soloway, 1985, p. 252).

A follow-up paper from the same study (Lister et al., 2006) asked instructors (as expert programmers) to explain in plain English the same code previously given to students. Their responses were then analysed according to the SOLO categorisation given in Table 2. The results showed that while novices most commonly give Multistructural responses, seven of eight participating instructors gave a Relational explanation of the purpose of the code. This indicates that the ability to comprehend code at a SOLO relational level may be an important component of programming expertise which is lacking in novices. This finding is consistent with Fix, Wiedenbeck and Scholtz (1993) who identified a contrast between the potential of novices and experts on program comprehension. Fix et al. suggested experts can discover goals, relate goals to previous experience, suggest plans, and relate this to the code of a program. The BRACElet group went on to suggest that while their study has examined novice and expert potential to think in abstract ways about code, it does not suggest how novices could be better trained to perform this task, as well as other related tasks.

Oliver et al. (2004) measured the cognitive difficulty of assessments of six courses in an undergraduate computing degree. The six courses measured included three programming courses studied in the first three semesters and three networking courses studied in the second year. Bloom's Taxonomy was used as the basis for this measurement with each course being given a single rating between 1.0 and 6.0 according to the average difficulty of its assessment tasks. This study can be criticised for applying Bloom's Taxonomy as a linear scale; however, it did demonstrate that first year programming courses include assessment tasks that are more cognitively demanding than those encountered in later courses. This seems counter-intuitive to author of this review. The BRACElet project showed programming comprehension can be assessed across a range of cognitive levels. The results of Olivier et al. suggest instructors may be neglecting lower cognitive levels by only using assessment tasks at higher levels of Bloom's Taxonomy.

The studies mentioned thus far have measured novices’ potential in comprehension and generation exercises and compared their performance to expected standards or the level of experts. Results have shown that novices generally are not performing at expected levels. Considering problem-solving ability on the knowledge-strategy dimension, many novices have a fragile programming knowledge, which may prevent them from applying programming strategies. Novices have been shown to perform better in comprehension exercises categorised at lower levels of Bloom's Taxonomy than those at more demanding levels. Novices are more likely to give line-by-line explanations of code than to describe the overall purpose of a piece of code, suggesting they do not possess the strategies used in the generation of the code.

3. Incorporating Strategies Explicitly into Instruction

The *symptoms* are poor results in standardised program generations tests, with many novices having a fragile knowledge and most novices failing to demonstrate programming strategies. When contemplating possible *causes*, two arise. One possibility is we have some bright students, but most students simply don't possess the mental capacity to program at an acceptable level. An alternate possibility is the curricula and assessment methods used in traditional introductory programming courses are lacking and fail to teach most students programming knowledge and strategies. It is likely that both of these causes are contributing in some way; however blaming novices for their own failure will not improve outcomes so we must consider ways of improving curricula to resolve these failings in part.

Traditional programming instruction encourages novices to learn problem-solving implicitly through practical exercises. In a study of second language learning using artificial grammars Reber (1993) showed implicit-only learning can improve a student's performance but does not create an understanding of the underlying systems used. Chick-sexing (determining the gender of new-born chicks) was a skill traditionally taught through implicit-only instruction (Biederman and Shiffrar, 1987). In an experiment the knowledge of an expert chick-sexer was captured and presented explicitly on a single sheet of paper. Novice chick-sexers trained using this explicit approach showed dramatically superior outcomes. According to Baddeley (1997) this demonstrates explicit learning can be more effective than months of implicit learning. Inspired by these studies, de Raadt, Watson and Toleman (2006) developed an approach to teaching strategies explicitly. de Raadt et al. began with an experiment that involved experts solving problems typical of those a novice would be expected to solve at the end of a introductory course in programming. Solutions produced by experts showed consistent use of anticipated plans which had previously been catalogued by Johnson (1986). From this work de Raadt et al. developed a list of plans that can be incorporated explicitly into introductory programming curricula. de Raadt (2007) claims that when compared to a traditional, implicit-only approach, the explicit approach is more likely to produce novices who understand and apply strategies. In interviews novices taught in this way were also observed to use a vocabulary that included plan terminology and showed greater confidence in the correctness of their solutions.

Porter and Calder (2003) suggest *A Pattern-Based Problem-Solving Process for Novice Programmers* in their paper of that title. They describe strategies they believe would normally appear as examples in an introductory programming course "packaged in the pattern format" (p. 235). Porter and Calder show how they have used patterns to enhance their curriculum and pedagogical approach. "Patterns lend themselves to the learning of a skill like programming, because they provide the static knowledge plus the means to apply it" (p. 236). As well as describing the use of patterns to novices, Porter and Calder also suggest a prescriptive approach, driven by their pattern language, for applying patterns to problems and refining solutions. Porter & Calder demonstrate the application of patterns to a problem over multiple iterations of pattern insertion. They tested their approach on a small number of volunteers split into a control and experimental group (Porter and Calder, 2004). Participants were asked to undertake an exercise under test conditions. Results achieved were compared to results of a previous course participants had undertaken. This study showed slightly better outcomes in participants exposed to the patterns and pattern language. However none of the participants in either group demonstrated any obvious use of the patterns or pattern language during testing.

ELP is a tutoring and assessment tool that incorporates a program analysis framework for the static analysis of programs (Truong, Roe and Bancroft, 2004). The framework creates an XML representation of a student's program in the form of an abstract syntax tree. The framework uses the representation to provide quantitative feedback based on software engineering metrics and qualitative feedback based on structural similarity with model solutions that have previously been supplied by an instructor. The software, like the PROUST tool (Johnson, 1986), is able to assess novice's programming strategies, as well as their programming knowledge, and give appropriate feedback. An evaluation of ELP including the program analysis framework was conducted with students in a second year unit who were asked to participate in an attitudinal survey (Truong, Bancroft and Roe, 2005). Sixty-three percent of participants were positive about their use of the program analysis framework as "it was useful in providing them [with] feedback about how … their solution compared to the model solution and [helped] them to think about … alternative solutions when doing exercises" (p. 12).

While this paper is a survey of Australasian work, there is of couse much work elsewhere on explicit instruction of plans. For example, across the world in Finland, Jorma Sajaniemi has been refining an explicit description of the *roles of variables* which is being incorporated in introductory programming curricula (Sajaniemi, 2002). Variables are categorised by their role, for instance *constant*, *stepper*, *most-recent holder* and so on. These roles are taught to novices and when code is shown to students these roles are identified. A standard visualisation of variable roles has also been created. Kuittinen & Sajaniemi (2003) describe an experiment involving novices divided into three groups:

- a control group (receiving traditional instruction without explicit roles of variables),
- a group receiving instruction that explicitly included roles of variables, and
- a group who received explicit instruction in the roles of variables plus animation of roles in examples.

After an exam involving simulation, comprehension and generation exercises, an analysis of results found no significant difference between groups on questions. However, when asked to give explanations of their answers, students in the first group tended to give "operation level descriptions" while students in the second and third groups gave "data level" descriptions. Sajaniemi & Kuittinen (2005) conclude that novices are able to learn the roles of variables and apply them to new situations. They believe this allows novices to generate solutions which demonstrate good programming skills and contain fewer errors.

While these diverse studies show positive steps towards curriculum and assessment change focussed on programming strategies, there appears to be much ground still to be investigated in this area.

4. Conclusions

Introductory programming instruction has been identified as cognitively demanding, with many novices failing to reach expected standards at the end of a period of instruction. Investigations involving computing education researchers from Australasia, with others around the world, have attempted to isolate the weaknesses in novice understandings.

Studies have shown that novices perform very poorly on standardised program generations tests. In program comprehension tests, novice performance is better, but still poorer than expected by instructors. This may indicate that the programming knowledge of novices is fragile and may be partly responsible for poor results in generation exercises. Goal/Plan Analysis of solutions created by novices who had been trained using a traditional curriculum with implicit instruction of problem-solving strategies showed poor use of identifiable strategies. When asked to explain the purpose of a given piece of code only 30% of novices are able to give a SOLO Relational response indicating a possible lack of programming strategy skill. These strategy-related deficiencies could be compounding the effect of poor programming knowledge in generation exercises.

Future computing education theoretical work should attempt to uncover the relationship between the comprehension-generation and knowledge-strategy dimensions. Is novice code generation dependant on, or related to, an ability to comprehend similar code? Is an understanding of programming strategies required before novices can generate code which applies those strategies, or even to comprehend code that applies those strategies? Such investigations may involve analysis using SOLO levels of understanding and analysis of strategy application such as Goal/Plan Analysis which have been seen in Australasian investigations.

Studies have investigated the incorporation of problem-solving strategies explicitly into introductory programming curricula. These curricula need to be more widely implemented and further evaluated in order to properly measure their potential to improve novice outcomes. Programming knowledge is also an important focus area which cannot be ignored; important research is still to be done here also. Programming knowledge and programming strategies, while related, need to be identified separately in curricular materials and assessment. Possible future work might include the publication of strategy guides, possibly incorporated into text books. Testing novices' problem-solving strategies, as a means of assessment in introductory programming courses, is another avenue yet to be explored.

5. References

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, R. & Wittrock, M. C. (Eds.). (2001). *A Taxonomy for Learning, Teaching and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives*. New York, USA: Addison Wesley Longman, Inc.

Baddeley, A. (1997). *Human Memory: Theory and Practice (Revised Edition)*. Erlbaum, UK: Psychology Press.

Bailie, F. K. (1991). Improving the modularization ability of novice programmers. *Papers of the twenty-second SIGCSE technical symposium on Computer science education, 23,* 277 - 282.

Biederman, I. & Shiffrar, M. M. (1987). Sexing Day-Old Chicks: A Case Study and Expert Systems Analysis of a Difficult Perceptual-Learning Task. *Journal of Experimental Psychology: Learning, Memory and Cognition, 13* (4), 640 - 645.

Biggs, J. B. & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies, 18*, 543 – 554.

Clancy, M. J. & Linn, M. C. (1999). Patterns and Pedagogy. *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, 37 - 42.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39* (2), 237 - 267.

de Raadt, M., Watson, R. & Toleman, M. (2002). Language Trends in Introductory Programming Courses. *The Proceedings of Informing Science and IT Education Conference*, 329 - 337.

de Raadt, M., Toleman, M. & Watson, R. (2004). Training strategic problem solvers. *ACM SIGCSE Bulletin, 36* (2), 48 - 51.

de Raadt, M., Watson, R. & Toleman, M. (2004). Introductory Programming: What's happening today and will there be any students to teach tomorrow? *Australian Computer Science Communications, 26* (5), 277 - 284.

de Raadt, M., Toleman, M. & Watson, R. (2006). Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. *Australian Computer Science Communications, 28* (5), 55 - 62.

de Raadt, M. (2007). *Incorporating Strategies Explicitly into Curricula (Working Paper)*. Retrieved May 29, 2007, from (http://www.sci.usq.edu.au/research/workingpapers/sc-mc-0705.ps).

Fix, V., Wiedenbeck, S. & Scholtz, J. (1993). Mental representations of programs by novices and experts. *Proceedings of the conference on Human factors in computing systems.*, 74 - 79.

Johnson, W. L. & Soloway, E. (1984). PROUST: Knowledge-based program understanding. *Proceedings of the 7th international conference on Software engineering*, 369 - 380.

Johnson, W. L. (1986). *Intention Based Diagnosis of Novice Programming Errors*. Los Altos, California, USA: Morgan Kauffman Publishers, Inc.

Kuittinen, M. & Sajaniemi, J. (2003). First Results of An Experiment on Using Roles of Variables in Teaching. *Papers from the Joint Conference at Keele University (EASE & PPIG 2003)*, 347 - 357.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin, 36* (4), 119 - 150.

Lister, R., Simon, B., Thompson, E., Whalley, J. L. & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin, 38* (3), 118 - 122.

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L. & Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33* (4), 125 - 180.

Oliver, D., Dobele, T., Greber, M. & Roberts, T. (2004). This Course Has A Bloom Rating Of 3.9. *Proceedings of the Sixth Australasian Computing Education Conference (ACE2004), 30*, 227 - 231.

Porter, R. & Calder, P. (2003). A Pattern-Based Problem-Solving Process for Novice Programmers. *Fifth Australasian Computing Education Conference (ACE2003), 20*, 231 - 238.

Porter, R. & Calder, P. (2004). Patterns in learning to program: an experiment? *Proceedings of the sixth conference on Australasian computing education, 30*, 241 - 246.

Reber, A. S. (1993). *Implicit Learning and Tacit Knowledge*. New York, USA: Oxford University Press.

Rist, R. S. (1991). Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction, 6,* 1 - 46.

Rist, R. S. (1995). Program Structure and Design. *Cognitive Science, 19,* 507 – 562.

Robins, A., Rountree, J. & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13* (2), 137 - 173.

Sajaniemi, J. (2002). An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02),* 37 - 39.

Sajaniemi, J. & Kuittinen, M. (2005). An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education, 15* (1), 59 – 82.

Soloway, E. & Woolf, B. (1980). Problems, Plans and Programs. *Proceedings of the Eleventh ACM Technical Symposium on Computer Science Education*, 16 - 24.

Soloway, E., Ehrlich, K. & Bonar, J. (1982). Tapping into tacit programming knowledge. *Proceedings of the first major conference on Human factors in computers systems*, 52 - 57.

Soloway, E. (1985). The Psychology of Programming. *Proceedings of the 1985 ACM annual conference on The range of computing*, 252.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29* (9), 850 - 858.

Spohrer, J. C., Soloway, E. & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction, 1*, 163 - 207.

Spohrer, J. C. & Soloway, E. (1986). Novice mistakes, are the folk wisdoms correct. *Communications of the ACM, 29* (7), 624 - 632.

Truong, N., Roe, P. & Bancroft, P. (2004). Static Analysis of Students' Java Programs. *Proceedings of the Sixth Australian Computing Education Conference (ACE2004), 30,* 317 - 325.

Truong, N., Bancroft, P. & Roe, P. (2005). Learning to Program Through the Web. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiSCE 2005),* 9 - 13.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robins, P., Kumar, P. K. A. & Prasad, C. (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006), 52,* 243 - 252.

Winslow, L. E. (1996). Programming Pedagogy -- A Psychological Overview. *SIGCSE Bulletin, 28* (3), 17 - 22.