Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies

Michael de Raadt, Richard Watson
Department of Mathematics and Computing
University of Southern Queensland
Toowoomba, Queensland, 4350
{deraadt,rwatson}@usq.edu.au

Mark Toleman
Department of Information Systems
University of Southern Queensland
Toowoomba, Queensland, 4350
markt@usq.edu.au

Abstract

This study examines the problem solving strategies used by expert programmers. Past studies of the cognitive processes of expert programmers suggest the existence of plans that describe the problem solving strategies held by these programmers. To date such strategies, which are relevant to novice programmers, have not been explicitly incorporated into the curricula of introductory programming courses. In revisiting these earlier studies and confirming the existence of the strategies held by experts, this study argues for inclusion of explicit strategy instruction.

Keywords: novice, expert, programming, problem solving strategies, explicit instruction

1 Introduction

Novice programmers have traditionally acquired problem solving skills through implicit, rather than explicit, instruction. Such skills are developed by the student while attempting to solve practice problems, in the absence of a systematic framework or methodology that can be used as a guide to choosing an appropriate problem solution.

In a number of independent studies, unrelated to programming instruction, implicit instruction in solving a problem has been shown to be result in poor learning outcomes. (These are reported in Section 3 of this paper.) It is reasonable to expect that implicit-only instruction of programming problem solving strategies may also be less effective than instruction that includes explicit instruction of these strategies.

Whatever form of instruction is devised, students need to learn the kind of skills that expert programmers possess. The work of Soloway (1986) provides a concrete and rigourous framework for classifying these strategies, and so is an attractive base for an explicit strategy instruction curriculum. Before proceeding to do so, it is necessary to test the Soloway model to ensure that the expert knowledge he describes is indeed exhibited by proficient professional programmers. This is the primary aim and of this paper.

The paper is organised as follows. Section 2 offers background justification for this study. Section 3 explores past studies of explicit instruction, and introduces the goal/plan framework described by Soloway. Section 4 describes the experimental methodology followed by this study. Section 5 presents the results of the study. Finally, conclusions are drawn in Section 6.

2 Background – Knowledge and Strategies

Traditional instruction of programming has focused on providing a novice with programming knowledge. This includes the constructs and facilities available in a language and the rules that describe how to combine these constructs and facilities. Most programming texts devote the majority of their content to programming knowledge. A distinction can be made between programming knowledge and strategies (Davies 1993, Robins, Rountree & Rountree 2003). Programming strategies can be described as understanding how to apply programming knowledge appropriately to solve problems. An instructor may present a problem to illustrate some programming knowledge, say a looping construct. The instructor may then display and describe a coded solution to the problem. A novice may say "I understand how the *for* loop works and I can see your program solves the problem, but I don't think I could have dreamed up that solution myself." This novice has distinguished between their programming knowledge and their programming strategies (or lack thereof).

Programming problem solving strategies relate specifically to a programmer designing and implementing

a solution to a stated problem. These strategies are developed in a novice's early study of programming. These strategies might be contrasted with other programming strategies applied by programmers, such as systems analysis, code maintenance, teamwork, version control and so on.

In a census of eighty-five introductory programming courses from Australian and New Zealand universities, participants were asked to estimate what proportion of time was spent in face-to-face classes teaching problem solving strategies (de Raadt, Watson & Toleman 2004). Estimates varied greatly with some participants indicating problem solving strategies were not part of their course while others claimed their entire lecture time focussed on teaching of problem solving strategies. Such variation suggests many instructors were not distinguishing teaching of problem solving strategies from teaching of programming knowledge. Several participants felt problems used in their teaching were not of a large enough scale to apply teaching of problem solving strategies explicitly.

An analysis of the forty-nine texts found by the census to be in use showed varying degrees of problem solving content. Only a small number of texts introduced problem solving and attempted to integrate this teaching throughout the text using case studies of problems being solved. Some texts offer a brief mention of problem solving in an early chapter, but this teaching is not obviously integrated in the remainder of the text. Some texts avoid problem solving, as a specific topic, altogether.

When a novice has completed a study of programming they are expected to have developed some expertise in problem solving strategies. The most common form of problem solving instruction begins with a worked example. Students are shown a simple problem specification and its corresponding solution, together with some explanation from an instructor (or a textbook author) of why this particular solution was adopted. Following exposure to a range of simple problems and their solutions, a student is given a problem definition, usually not too different from those already encountered, and expected to devise a solution. A novice is expected to build strategies by undertaking the problem solving process, applying reasoning about the examples presented. Typically no framework to assist in building understanding is presented to the student. This is an implicit approach to acquisition of problem solving strategies. By contrast, explicit problem solving strategy instruction presents concrete techniques that a student can use to transform a problem definition to its solution.

3 Explicit Instruction

A quantitative difference between teaching through explicit and implicit approaches is shown by Biederman and Shiffrar (1987). Sexing of day-old chicks is performed by experts employed by commercial egg producers. The distinction between male and female chicks remains hard to determine until one month of age. Being able to determine the gender of chicks early avoids feed wastage on unwanted males. Professional sexers classify over 1000 chicks per day and can identify gender in less than a second with 98% accuracy. Traditionally training of sexers takes place over six to twelve weeks of implicit instruction (observation, trial-and-error) in special schools. Sexers can then take years to master the task and achieve the required accuracy. A group of volunteers with no sexing experience were asked to identify the gender of 18 chicks from genitalia photographs. Performance of these subjects was 60.5%, slightly greater than chance. The same photographs were shown to five expert sexers who achieved an average performance of 72%. A sexer with vast experience (50 years and 55 million chicks) was recruited to identify the gender of a number of chicks from a series of photos. Afterwards the expert was asked to identify the visual aspects that prompted his decisions. From this interview an information sheet was created describing key visual aspects. The volunteers were split into a control and experimental group. The information sheet was given to the experimental group to study for one minute. After instruction volunteers were retested using a second set of randomly ordered photographs. Control subjects showed no improvement in their accuracy. Those volunteers in the experimental group who had read the sheet averaged performance of 84%, which was higher than experts. According to Baddeley (1997) this demonstrates explicit learning can be more effective than months of implicit learning.

Studies have shown that implicit-only learning can improve a student's performance but does not create an understanding of the underlying systems used. Another study, closer to programming, that examines implicit learning is derived from the study of language acquisition. Children learn the majority of their native language through implicit means. Second language instruction is usually achieved through explicit study of the grammar of a new language. Reber (1993) used a small, finite state artificial grammar to test the effectiveness of implicit learning of a second language. In this experiment an experimental group was shown sequences generated from the grammar without being shown the rules of the grammar. A control group was shown randomly generated sequences. Both groups were then shown 44 sequences; half were grammatically correct and half not. Subjects were asked to determine which were correct. Experimental subjects achieved 79% accuracy while members of the control group showed no capacity to accurately distinguish sequences. This showed that the experimental group were able to recognise sequences from the grammar, however when asked to describe the grammar they had been exposed to, experimental subjects were unable to show any understanding of the rules used to generate sequences. A similar study (Berry and Dienes 1993) asked subjects to learn the workings of a simulated transport system through implicit instruction only. After a series of tests, subjects showed an improvement in performance operating the system, but no increase in their understanding of the underlying rules of the system.

The previously described experiments clearly indicate the weaknesses of implicit learning. Novice programmers do

learn problem solving strategies over time though implicit instruction. But can instruction of problem solving strategies be improved by incorporating explicit instruction? Could this improve not only the speed of learning, but also create a more structured understanding of how problem solving strategies are applied?

If explicit instruction of programming problem solving strategies is to be included in introductory programming courses, such explicit strategies must first be captured. As in the chick sexing experiment such strategies could be captured from the tacit knowledge of expert programmers. Two considerations are: (1) what expert strategies are relevant to novice programmers and (2) how can these be captured?

When considering strategies relevant to novices it is possible to see programming problem solving strategies existing at several levels. The highest level is the system level. Here well established strategies have been formulated for designing and implementing systems, usually following a waterfall software development process (analyse, design, implement, test, maintain); new processes such as extreme programming also fall into this level. This level of programming problem solving strategy is beyond the novice in their initial study of programming because they are limited by the programming knowledge they possess. At a lower level are algorithmic problem solving strategies. This level would include established strategies such as sorting, searching and the application of data structures; patterns as applied to object-oriented programming may also fall into this level. A novice may be able to start using such strategies at the end of a semester-long study. A lower sub-algorithmic level is conceivable that relates to the strategies applied to the most basic processes such as applying a condition to guard a division or applying a loop appropriately to capture input until a sentinel is found. This level of programming problem solving strategies is particularly relevant to novices as they are developing an initial programming knowledge of basic constructs and features.

3.1 Goals and Plans

When investigating how to capture expert strategies the literature reveals a number of cognitive studies of programming problem solving strategies performed on expert programmers during the 1980s (a number are covered in Hoc *et al* 1990, and also in Koenemann 1991, Soloway 1982). Similar studies of novice programmers have been performed (Soloway 1980, Spohrer 1986) and the distinction between novices and experts has also been studied (Fix & Wiedenbeck 1993, Gugerty & Olson 1986). Of note was the emerging idea of plans or schema which can be seen as sub-algorithmic problem solving strategies.

Soloway (1986) described how expert programmers possess a tacit body of programming strategies developed through solving problems. When presented with a novel problem an expert is able to adapt a canned solution from this tacit body of strategies. Soloway proposed a theory that experts recognise *goals* present in a problem statement and apply *plans* that will achieve each goal. A catalogue of goals and plans used for PROUST (an Intelligent Tutoring System) is found in Johnson (1986). An expert programmer must also know how to integrate these plans to form a whole solution. Take, for example, the following problem.

> *Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.*

This problem is likely to have many potential solutions. However, there are a number of critical goals that can be identified:

- Input the numbers
- Compute the sum of the numbers
- Compute the count of the numbers
- Calculated the average from the sum and the count (keeping in mind that the count of values could be zero)
- Output the average

Figure 1 shows a possible list of plans and how they could be integrated. The input of numbers is achieved in a Sentinel Controlled Input Sequence (3) in which are *nested* an input to prime the loop test (4) and another (7) for possible subsequent inputs (these inputs could be combined depending on implementation). To capture the count of values a variable is used that must be initialised (1). The count accumulates in a Sentinel Controlled Count Loop (5). So that input is only to be captured once, this loop needs to be *merged* with the input sequence. A similar initialisation (2) and integrated loop (6) is required to capture the sum of values. When input is complete the average is calculated. As this involves a division, the count should be tested to see it is not zero before performing the division (8). The result, or some message explaining the lack of a result, will then be output (9). The ordering of plans is critical, so *abutment* is used to integrate the plans into a single sequence.

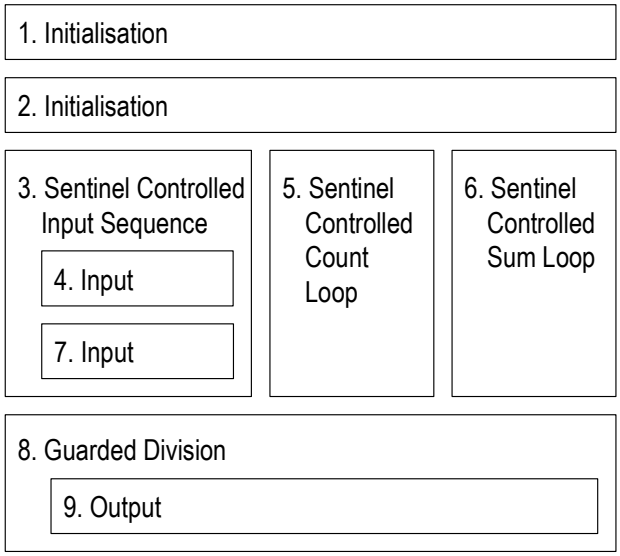


Figure 1: Plans and how they are integrated

Spohrer and Soloway (1985) used the goal/plan framework to evaluate solutions created by novice programmers in Pascal to discover which "bugs" were

being commonly created. The application of goal/plan analysis seems to be applicable to all languages with an imperative component with most plans relating to structured programming constructs. Although Soloway and his colleagues intended to develop a goal/plan based introductory programming curriculum, this was never completed (Soloway 2003).

de Raadt, Watson and Toleman (2004) applied goal/plan analysis to test if students' solutions to the above problem. Participating students had completed one semester's instruction where problem solving strategies were taught implicitly only. The results showed significant weaknesses in initialisation, creating a sentinel controlled loop, merging plans and guarding division. This demonstrated that novices were not learning several important problem solving strategies through implicit only instruction.

If expert programming problem solving strategies can be extracted it may then be possible to integrate explicit instruction of these strategies into an introductory programming curriculum. Soloway's goal/plan theory is a reasonable model of how experts apply programming problem solving strategies, but is it actually how experts think? This study attempts to verify that experts do exhibit plans in their solutions to problems of the scale relevant to novice programmers.

4 Methodology

An experiment was conducted with experts. As Winslow (1986) suggests there are several levels between a novice and an expert. In this study participants were considered qualified if they were generating code on a regular or daily basis.

As with the chick sexing experiment, participants were asked to solve problems on paper, away from a computer. The focus of analysis was not on the syntactical correctness of solutions, but on how experts solved the problems. Using paper was a means of enforcing this focus.

Participants were timed to see how long they took to solve each problem. Participants were asked not to rush or, where more than one programmer was participating simultaneously, not to compete.

The aim of this experiment was to discover sub-algorithmic strategies possessed by experts which are relevant to novice programmers as they begin their study of programming. The strategies being elicited would be used by experts on a regular basis within solutions to greater problems. Three problems were chosen that a novice would be expected to solve at the end of an initial semester of programming. For each, the problem statement, identifiable goals and expected plans are shown below. The problems increase slightly in complexity from Problem 1 to Problem 3. The problems are sufficiently generic to permit solutions from a broad range of languages.

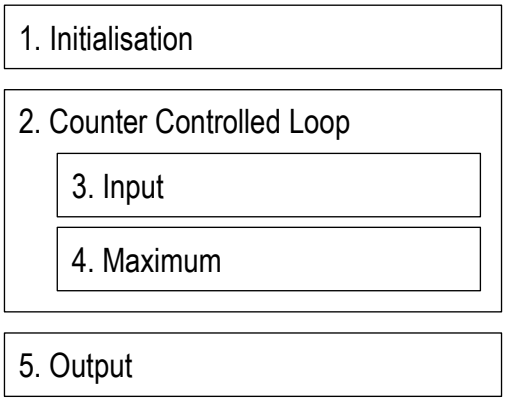
4.1 Problem 1

Read in 10 positive integers from a user. Assume the user will enter valid positive integers only. Determine the maximum.

4.1.1 Goals

- Input 10 numbers
- Determine maximum
- Output maximum

4.1.2 Plans



4.2 Problem 2

Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.

Goals and plans as shown above in section 3.1.

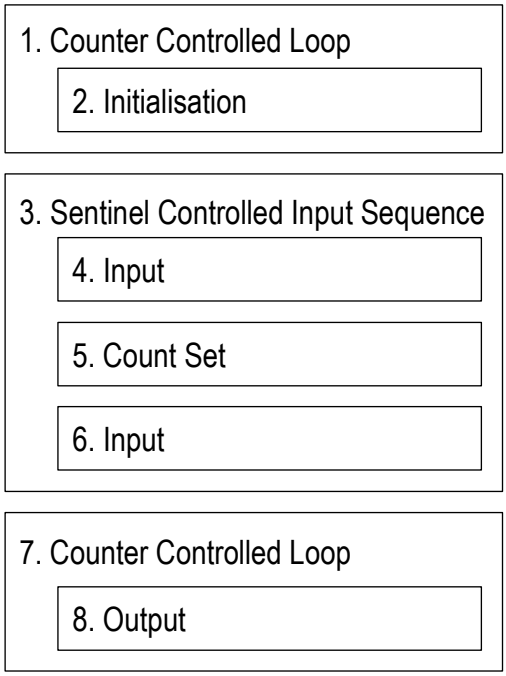
4.3 Problem 3

Input any number of integers between 0 and 9. Assume the user will enter valid integers only. Stop when a value outside this range is encountered. After input is concluded, output the occurrence of each of the values 0 to 9.

4.3.1 Goals

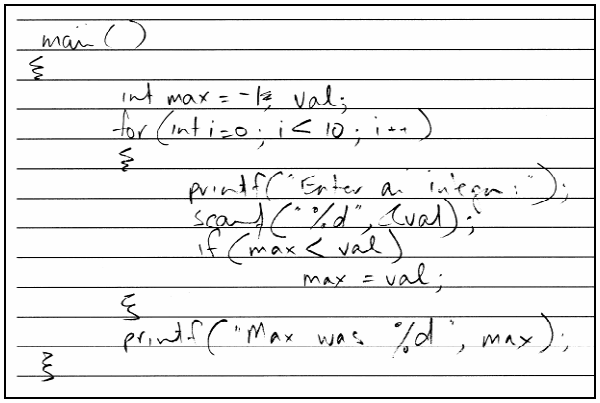
- Input numbers
- Count set
- Output set

4.3.2 Plans



4.4 Goal/Plan Analysis

Results were analysed by checking for the presence of each of the plans above, as well as nesting and merging in appropriate locations and an overall correctness measure of abutment. In most cases the presence of a plan is easily determined. For instance when searching for a maximum plan, look for a test comparing the current maximum with a new candidate and then an assignment if appropriate. With counter controlled loops, only loops including a test of an incrementing counter variable were accepted.



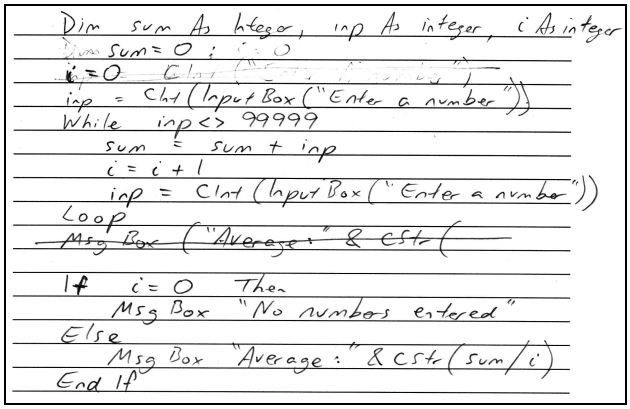
```
main ()
{
    int max = -1, val;
    for (int i=0; i < 10; i++)
    {
        printf("Enter an integer:");
        scanf("%d", &val);
        if (max < val)
            max = val;
    }
    printf("Max was %d", max);
}
```

Figure 2: A participant's solution to Problem 1

In Figure 2 a solution to Problem 1 created by a participant is presented. When performing goal/plan analysis the following features were identified.

- The maximum is initialised; the first input will become the new maximum.
- There is a counter controlled loop; the for loop will repeat 10 times regardless of user input.
- The user is able to enter input.
- Each input is compared with the current maximum and kept if greater.
- The maximum is output at the end of the program.
- The input and maximum elements are nested inside the counter controlled loop.

Sentinel controlled loops were only considered present if the looping construct tested that the first input could have been the sentinel and did not include the sentinel as an input.

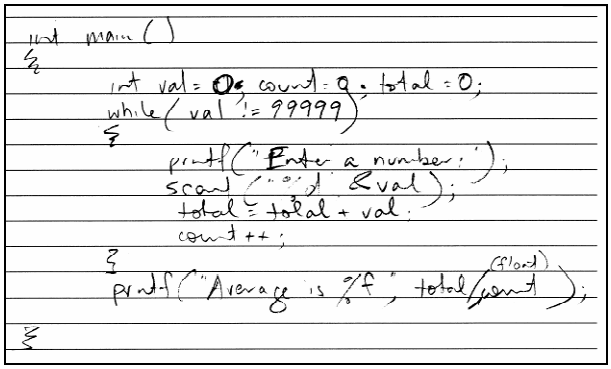


```
Dim sum As Integer, inp As Integer, i As Integer
sum = 0 ;
i = 0
inp = CInt(InputBox("Enter a number"))
While inp <> 99999
    sum = sum + inp
    i = i + 1
    inp = CInt(InputBox("Enter a number"))
Loop
MsgBox("Average:" & CStr(
If i = 0 Then
    MsgBox "No numbers entered"
Else
    MsgBox "Average:" & CStr(sum/i)
End If
```

Figure 3: An acceptable solution to Problem 2

Figure 3 shows a participant's sol solution to Problem 2. From this solution the following features can be identified.

- The sum and count are initialised.
- The loop is a sentinel controlled loop as the test is primed by an initial input and the sentinel will not be included in the sum or count.
- If the count of inputs is zero, the calculation of the average (including a division by the count) will not be performed.



```
int main ()
{
    int val = 0, count = 0, total = 0;
    while (val != 99999)
    {
        printf("Enter a number:");
        scanf("%d", &val);
        total = total + val;
        count++;
    }
    printf("Average is %f", total/(float)count);
}
```

Figure 4: A poor solution to Problem 2

Figure 4 shows another participant's solution to Problem 2. This solution fails to demonstrate a number of elements that were being identified.

- The loop is not a sentinel controlled loop. The input value used in testing is initialised but it is not primed with user input (which could be the sentinel). The sum and count will include the sentinel.
- The division operation used to calculate the average is not guarded.

For a set counting plan only methods of classifying and counting inputs, as opposed to capturing and keeping the

user's input, were acceptable. In some languages initialisation of variables and arrays is done automatically; where this was the case participants were seen as having fulfilled the initialisation components of the plans.

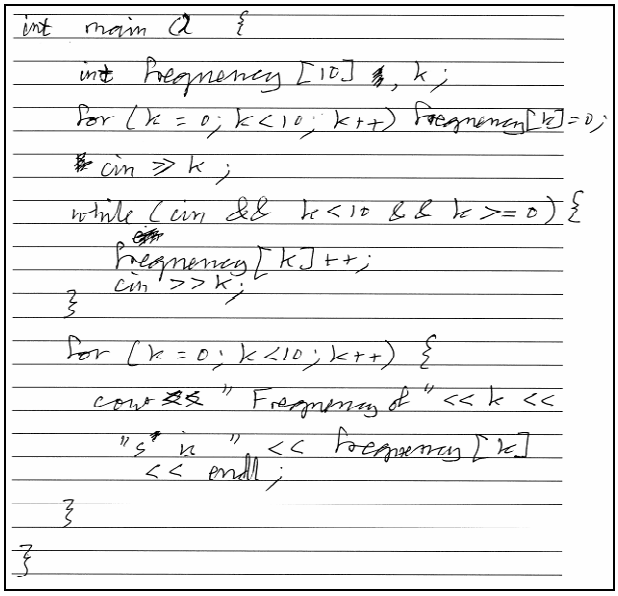


Figure 5: A participant's solution to Problem 3

In Figure 5 a participant's solution to Problem 3 is shown. From this solution the following features can be identified.

- A count of numbers 0 to 9 is being kept in an array. The array elements are initialised to zero using a counter controlled loop at the start of the program.
- The loop is a sentinel controlled loop as the test is primed by an initial input and the sentinel (any value outside the range 0 to 9 will not be counted.
- Each input is counted using the input as an index into the array.
- The counts are output using a counter controlled loop at the end.

5 Results and Discussion

Solutions from 25 experts were gathered for each problem. This included 11 academics involved in the teaching of programming and 14 professional programmers.

In one instance a participant used an event driven paradigm to solve the problems. With this case goal/plan analysis could not be applied as many of the underlying constructs, such as loops, were not used. This demonstrates that experts do think in different ways and that goal/plan analysis is not completely universal. The solutions of this participant were not used in analysis. In three instances participants created a solution to some different problem. In these cases it was clear they had misread the instructions rather than being unable to solve the set problem. These solutions were disqualified from analysis, but the remaining solutions from these participants were used.

	Prob. 1	Prob. 2	Prob. 3	Overall
Academics	4:50	4:52	6:17	15:58
Professionals	5:33	5:17	6:16	17:06
All	5:13	5:05	6:16	16:34

Table 1: Average times for problems by expert type in minutes and seconds

Table 1 shows times taken by participants to solve the problems. The average time to complete all three problems was 16min 34sec. To give a comparison, the aforementioned previous study (de Raadt, Watson and Toleman, 2004) showed novices at the end of a semester's instruction taking at least 20min to solve Problem 2 alone. There was a difference in times between participants who were academics and those who were professional programmers, although it was not proven to be significant in this sample. The six fastest times were contributed by academics. This may have been due to the simplified nature of the problems which would be familiar to academics but less so to professionals. Also, taking professionals away from their normal coding environment may have had an impact here. There was no real difference in average times for problem three, so it may be assumes that, by this stage, professional programmers had adapted to solving simple problems on paper. There was not a significant difference in the presence of plans between academics and professionals (3% difference in overall plan used).

All experts achieved correct abutment (the correct ordering of plans). In other words, no expert, for instance, placed the output of a maximum before the calculation of the maximum.

Plan	Presence
Max Initialised	100%
Counter Controlled Loop	100%
Input Plan	100%
Maximum Plan	100%
Output Plan	87%
Input Nested in Counter Controlled Loop	100%
Max Plan Nested in Counter Controlled Loop	100%

Table 2: Presence of plans for Problem 1

Problem 1 showed almost universal conformity to the set plans. Three participants included no output and this may be due to the wording of the problem that asked for the maximum to be determined but did not specifically ask for an output.

Plan	Presence
Sum Initialised	92%
Count Initialised	100%
Sentinel Controlled Input	92%
Sentinel Controlled Count	92%
Sentinel Controlled Sum	92%
Guarded Division	33%
Output Plan	92%
Loop Plans Merged	100%
Inputs Nested in Sentinel Controlled Loop	92%
Output Nested in Guarded Division	33%

Table 3: Presence of plans for Problem 2

Problem 2 showed most participants conforming to the expected plans. In some cases individual participants failed to show one plan. Where a person failed to show a sentinel controlled loop, the looping plans merged with this loop were considered as not being present, even though they may have attempted to capture a count or sum. One obvious deficiency is shown by the absence of a guarded division. Only one third of participants' solutions contained a guarded division plan.

Plan	Presence
Counter Controlled Loop (for Initialisation)	91%
Array Initialisation	100%
Sentinel Controlled Input	86%
Count Set Plan	95%
Counter Controlled Loop (for Output)	86%
Output Plan	100%
Initialisation nested in Counter Controlled Loop	91%
Inputs nested in Sentinel Controlled Loop	91%
Count Set nested in Sentinel Controlled Loop	86%
Output Nested in Counter Controlled Loop	95%

Table 4: Presence of plans for Problem 3

Problem 3 showed most participants conforming to the expected plans. This problem encouraged the greatest variation in solutions; this difference was found in how the data was stored (an array was expected, but some participants used variables), initialisation of the data store (where an array was used, a counter controlled loop containing element initialisations was expected, but some participants used set notation to initialise the array) and set counting (the user's input could have been used as index to the array, but some participants used a 'switch'-like construct to increase counts).

The problem statements were simplistically worded and this may have affected the results. For instance, in Problem 1, participants are asked to "determine" but not output the maximum, which was part of an anticipated solution. Some participants did not output the maximum and it is difficult to determine if this was because of the wording of the problem or that they simply forgot to do so.

The poor showing of guarded division may also have been a product of simplistic problem statements. Participants may also have been affected by being out of their normal programming environment and without the tools they would use for testing such boundary conditions. After participants completed the three problems they were told what plans were expected. At this stage the participants' solutions had not been analysed, however in some cases participants admitted neglecting to include a guarded division and saw that it was required. This might be contrasted to a novice who might not apply a guarded division because they are unable to or unaware that they need to.

6 Conclusions

Two main conclusions are drawn from this experiment.

Experts exhibit the plans catalogued by Soloway in their solutions to problems.

The results show that in most instances experts produce solutions that exhibit plans that can be anticipated from the catalogue of plans created by Soloway and his colleagues. These findings are constrained to the problems shown here, but may be transferable to other problems of the scale relevant to novices in their first study of programming.

The teaching of Goal/Plan based strategy instruction or similar is of benefit for novice programmers.

The authors believe that these results are a good indication that programming problem solving strategies applied by experts can be explicated. It is likely that similar plan/schema ideas other than those presented by Soloway *et al* could also be explicated through a similar study. With an understanding of the strategies applied by experts, it should be possible to create a curriculum of study that explicitly involves programming problem solving strategies. When compared to the chick sexing experiment discussed earlier, this study has extracted the equivalent of 55 million chicks worth of expert programmer strategies in preparation to trial such explicit instruction on novices.

At this stage it would be unjustified to claim that this would definitively improve instruction and outcomes in student learning, but it would be remiss not to attempt to incorporate such teaching. When compared to implicit instruction of problem solving strategies, the potential benefits of explicit strategy instruction could be:

- faster learning of problem solving strategies,
- better performance by novices in solving problems,
- a better understanding of the underlying processes involved in solving a problem at the sub-algorithmic level, and
- providing novices with an informal vocabulary for discussing and learning problem solving strategies.

The next stage of this study will be to incorporate programming problem solving strategies explicitly into an introductory programming curriculum. Such instruction would include the correct and incorrect application of the following strategies.

- Average plan
- Divisibility plan
- Swap plan
- Guarded exception plans (including guarded division plan)
- Sentinel controlled loops and counter controlled loops and the distinction between
- Min/Max plans
- Sm and count plans
- Stream input/output plans (including user input and file input)
- Validation plan
- Recursion plans (single- and multi-branching)

Such strategies would build upon the associated knowledge components throughout the period of instruction. This explicit instruction would then be reinforced with paper and computer exercises. Potential to comprehend and construct programs that rely on these strategies would be assessed.

Once such incorporation has taken place, this study will then attempt to judge the impact of explicit strategy instruction on the understanding of novices and their potential.

7 References

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

Berry, D.C. & Dienes, Z. (1993) *Implicit Learning: Theoretical and Empirical Issues*. Lawrence Erlbaum Associates Ltd., East Sussex, UK.

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.

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

de Raadt, M., Toleman, M. & Watson, R. (2004) Training Strategic Problem Solvers, *SIGSCE Inroads 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? *Proceedings of the Sixth Australasian Computing Education Conference (ACE2004)*, Dunedin, New Zealand, Australian Computer Society.

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., Amsterdam, The Netherlands, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gugerty, L. & Olson, G.M. (1986). *Debugging by skilled and novice programmers*. Conference proceedings on Human factors in computing systems, Boston, MA USA, ACM Press, New York, NY, USA.

Hoc, J.-M., Green, T.R.G., Samurcay, R. & Gilmore, D.J. (1990) *Psychology of Programming*. Academic Press Ltd., London, UK.

Koenemann, J. & Robertson, S.P. (1991) Expert Problem Solving Strategies for Program Comprehension, *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*. ACM Press, New York, USA.

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

Robins, A., Rountree, J. & Rountree, N. (2003) Learning and teaching programming: A review and discussion. *Computer Science Education*, **13**(2), 137 – 172.

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

Johnson, W.L (1986) *Intention-Based Diagnosis of Novice Programming Errors*. Pittman Publishing Limited, London, UK.

Soloway, E., Ehrlich, K. & Bonar, J. (1982). *Tapping into tacit programming knowledge*. Proceedings of the first major conference on Human factors in computers systems, Gaithersburg, Maryland, United States, ACM Press, New York, NY, USA.

Soloway, E. & Woolf, B. (1980). *Problems, Plans and Programs*. Proceedings of the Eleventh ACM Technical Symposium on Computer Science Education, Kansas City, Missouri, United States, ACM Press, New York, NY, USA.

Soloway, E. (2003) Personal communication, email, 27/05/2003.

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.

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