Investigating Expert and Novice Programming Differences on Problems of Varying Complexity

Maria Vorobeva (MariaVorobeva@cmail.carleton.ca)
Suhaylah Carim Bacor (SuhaylahCarimBacor@cmail.carleton.ca)
Mary Alexandria Kelly (Mary.Kelly4@carleton.ca)
Department of Cognitive Science, Carleton University
1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada

Abstract

Programming is a complex problem-solving domain, requiring the coordination of different types of knowledge and skills. The present study investigates expert and novice programming problem solving by analyzing talk-aloud transcripts and the code generated. Based on this analysis a set of basic goal and step components used by novice and expert programmers are identified, which will inform on the generation of cognitive models in the next phase of this research.

Keywords: Programming; Problem Solving; Cognitive Models; Python; Schemas; Knowledge Representation; Algorithms

Introduction

Programming is a complex problem-solving domain that requires both knowledge of how to implement basic constructs as well as high-level strategies of how to compose these into a program. The goal of our research is to develop a model of problem-solving in programming. Such a model can provide a theory as to the knowledge, skills, and strategies acquired by expert programmers that distinguish them from novice programmers. Existing studies investigating the knowledge representations used during programming (e.g., Castro & Fisler, 2020; Rist, 1989) do not formalize their findings in a way that would allow for the construction of cognitive models of programming because they do not present sufficient details on the programming process. Thus, to the best of our knowledge, to date, there does not exist a computational, cognitive model capable of both writing programs and modeling the differences between expert and novice problem solvers, for instance by employing programming schemas to structure problem solving and goal composition.

Computational models have either relied on formalizing behaviour and low-level implementational knowledge (thus avoiding formalizing the underlying representations, e.g., Recker & Pirolli, 1995), and/or focused on parsing completed programs without the ability to produce them (Pirolli, 1986).

A notable exception is the ACT-R Programming Tutor (APT), developed by Corbett (2000), which can write small snippets of programs. APT engages in both knowledge tracing and model tracing, and it is model tracing that allows it to write snippets of code. For model tracing, the tutor uses an underlying production system, called its ideal student model, that contains the full set of rules to solve all of the practice problems. For each student input, once the student has selected their next goal and next step, the model tracer generates a list of all possible, correct next steps and compares it to the student's input. If the student input is correct, problem solving proceeds to the next goal-step combination. If the student's input does not match one of the model's accepted next steps, the tutor provides feedback and encourages the student to correct the mistake. The ACT-R model underlying model-tracing can write the small program snippet solutions as it has the relevant productions, but it does not consider programming strategy and cannot produce solutions to entire programming problems; it also does not model differences between expert and novice programmers.

To address these gaps, the present study aims to identify knowledge representations required for programming, by assessing the programming process of both novice and expert programmers. In future work, we intend to use data from the present study to develop a model of programming expertise.

Background & Related Work

In other, non-programming domains, novice and expert performance differences have been attributed to differences in representational knowledge. In their seminal work using the domain of physics, Chi, Feltovich, and Glaser (1981) showed through a series of studies that experts do not simply have more knowledge than novices, but that they hold inherently different representations. One study presented twenty physics problems that had their superficial features and underlying physics laws crossed. Specifically, the problems included ones that looked superficially similar but required different underlying physics principles to solve, as well as problems that were superficially dissimilar but required the same underlying physics principles to solve. Experts took longer to categorize the problems, engaging with the problem at a deeper level. The results showed that not only do novices and experts use different features when classifying physics problems, but that they have different knowledge structures, with experts interpreting the problems according to their deep knowledge of physics principles (what Chi et al. referred to as schemas).

While much of the research on expertise has focused on physics, algebra, or other similar problem-solving domains (Chi et al., 1981; Gobet & Simon, 1996; Leonard, Dufresne, & Mestre, 1996), there are also studies investigating expert and novice knowledge structures and representations in the programming domain. Programmers' mental representations are pivotal to programming performance and ability. Of particular interest for the present work are studies investigating

In L. K. Samuelson, S. L. Frank, M. Toneva, A. Mackey, & E. Hazeltine (Eds.), *Proceedings of the 46th Annual Conference of the Cognitive Science Society.*

programming expertise (Castro & Fisler, 2020; Soloway & Ehrlich, 1984; Spohrer, Soloway, Elliot, & Pope, 1985).

Experts vs. Novices Programming

Similar to the earlier findings of Chi et al. (1981), research in the programming domain has also shown that experts have their programming knowledge structured differently than novices, and rely on different knowledge representations. Soloway and Ehrlich (1984) identified two representations related to programming expertise, goals and plans, through two studies. Goals were high-level requirements related to the problem specification and could be resolved by implementing plans in a programming language.

Study 1 used a program completion task where both novice and expert programmers had to complete a missing line of code in a program using only the rest of the program as context. Experts were better able to generate the missing code fragment than novices. The authors proposed this was due to a schema experts had for that type of program, which allowed them to infer goals needed to solve the problem.

Study 2 used a program recall task where participants had to recall as many lines of a program as they could after seeing it briefly. Experts were better able to recall lines. Soloway and Ehrlich proposed that experts were able to store program lines in meaningful chunks (i.e., programming plans). This finding mirrors work in other domains. For instance, these programming plans were similar to the schemas identified in physics problem solvers by Chi et al. (1981).

Spohrer et al. (1985) formalized these findings within a tree-like representational framework called GAP (goal and plan networks). GAP trees were used to parse novice programs, categorize their bugs, and identify the misconceptions that led to them. GAP aids in these tasks by decomposing a program into a solution space containing the program's goals, and the plan or set of plans that can implement those goals (often through decomposition into smaller goals and plans). Students who were not able to correctly complete the programming tasks usually had an error in, or the complete absence of, one or more of the GAP tree components. This suggests that novice errors are caused by a missing goal(s), or by incorrect knowledge used to solve the problem. Soloway (1986) proposed that novices have difficulty (1) identifying the goals needed to solve the problem, (2) recalling appropriate programming plans needed to implement the goals, and (3) combining these into a final solution plan.

The GAP tree framework is also used to assess expert programmers (Soloway & Ehrlich, 1984; Spohrer et al., 1985). Soloway (1986) described the expert process as first obtaining an understanding of the goal and plan structure of the problem (i.e., developing a rough GAP tree), then using stepwise refinement to recall prior solutions that can be combined and applied to the present problem.

Novice Representations Rist (1989) analyzed talk-aloud data from a program-generation process of 10 novice programmers to identify how they used simple programming plans to compose larger, more complex plans. A qualitative approach was used to code the talk-aloud transcripts with the plans implemented and their order of implementation. Novice programmers first identified a plan focus, which is the first expression or line of a programming plan that is implemented; the plan focus served as the anchor for a given programming plan. Once the plan focus was implemented, the remainder of the plan was expanded around it.

Castro and Fisler (2020) aimed to identify how and when students move between high level (task-level) programming schema considerations, and low level (code-level) implementational considerations. Here task-level considerations correspond to breaking down and addressing the tasks, analogous to Soloway (1986)'s programming goals of the problem. Code-level considerations are ones that must be made while implementing the actual program. Novices in this study ($N = 138$) learned how to decompose programming problems using the How To Design Programs (HTDP) methodology. HTDP focuses on teaching a high-level approach to students, where prior to programming students first outline a concrete plan. Based on their analysis of novice code and talk-aloud transcripts, three types of styles were identified for shifting between task-level and code-level thinking. The first style was cyclic, where students followed the HTDP style, alternating rapidly between task level thinking to identify goals and code-level thinking to implement them. Students first mentioned programming goals and then wrote code to fulfill the goals, adapting the plans as needed. The second style was code focused. This approach involved jumping directly into code writing; tasks were identified on the fly with no or minimal description of the written code's relation to the task. The third style was a one-way style, where students followed the HTDP style and made a high level plan at the start that they then dutifully translated into code. Unlike the cyclic shifting style, they did not actively adapt the programming plan with code-level considerations to make it suitable for the problem. In comparison to the cyclic style, students who used one-way and code-focused strategies both struggled. In general, the findings highlight that students can struggle with knowing when to focus on implementational aspects versus higher-level task considerations.

Present Study

The goal of the present study is to gather data on the process of program generation by novices and experts and to specifically identify differences in how participants identify goals and implement those goals as steps in Python code. Earlier research showed that programmers used algorithms (schemas) when problem solving (Soloway, 1986; Spohrer et al., 1985), and that there were individual differences in how effectively novice programmers navigated between high-level algorithmic considerations and low level implementational ones (Castro & Fisler, 2020). However, it is unclear from the earlier research exactly how algorithms are structured to aid programming, or how the algorithm is implemented by

programmers with different levels of expertise.

Participants

Participants self-identified as novice or expert programmers. The novices ($N = 12$) were undergraduate students who had completed, or were currently undertaking, one beginner level programming course (or equivalent experience, but no more than that). The experts ($N = 7$) were programmers with a programming-related degree, and/or related work experience; this included graduate students, professional software developers and those with degrees in computer science.

Materials

Python Problems The study involved four programming problems that varied in difficulty. All problems were solved in Python, a high-level, general-purpose programming language. The problems covered concepts such as loops, conditions, variables and other concepts typically covered by a first-year programming course.

Questionnaires A five-item pretest was used to assess basic programming knowledge.

Procedure

The study was done virtually over Zoom. The pretests were graded and used to verify the self-reported expert and novice status of the participants. During this time, participants were given remote access to a code editor (VSCodium), via the Zoom remote user function, and asked to read a general instruction document outlining the study protocol. The instructions specified that the participants must talk out loud as they are problem solving, so that insight into their thought processes could be recorded while they wrote programs. Participants then solved 4 programming problems by writing a program for each problem. They were given up to 15 minutes to complete each problem. During this solution phase, they were asked to not test the code (i.e., by running it). After 15 minutes had passed or they indicated they were done, they were asked to run their program. Verbal utterances and programming actions on the screen during the programming activities were recorded by Zoom. The programming portion took no more than one hour and ten minutes.

Data Processing

The analysis focuses on two of the problems given to the participants: the rainfall problem and the ballot problem. The rainfall problem requires the user to calculate the average rainfall over a certain period of time, using a list of rainfall amounts; negative numbers in the list are to be ignored and the program needed to stop processing the list upon encountering the first -999. The ballot problem asks the user to iterate over vote inputs for either of two candidates (Red and Blue) in three departments (Art, History and Science) and output which of the two candidates won in each department and which candidate won in more than two departments (called the popular winner). The audio files from the Zoom sessions were transcribed. The first pass of the transcription was done by the software Otter.ai. Transcripts were revised by watching each Zoom recording and ensuring that the transcript accurately reflected the verbal utterances. At this point, the transcripts were also supplemented with snippets of the participant's code added at the appropriate location in the transcript, so that they aligned with the verbal protocol. This was done in order to produce a transcript that reflected both verbal data and programming actions in chronological order.

Qualitative Coding of Transcripts

We used a qualitative approach to analyze the data. The transcripts for the rainfall problem were analyzed by the first author, and the ballot problem by the second author, to identify components of the problem-solving process for both problems in order to enable subsequent analysis of problem-solving approaches and novice and expert differences. The coding was based on an approach used in prior work analyzing problem solving in the domain of physics and geometry (Gertner & VanLehn, 2000; Koedinger & Anderson, 1993) as well as programming (Corbett, 2000). In that work, knowledge needed for problem solving was comprised of two key constructs: steps and goals. For the present work, we designed a coding scheme to subsequently label the data with goals and steps, shown in Table 1 and 2, respectively. This was done by reading the transcripts to identify common goal patterns and common implementation steps used. In this scheme, goals corresponded to concrete, verbally stated intentions about a high-level programming action that needed to be performed. A sequence of goals is an algorithm, namely a recipe for solving the problem. Depending on the confidence and experience of the programmer, goals could be very clear and explicit as in "I need to initialize variables for the sum and count" or the goal may be stated in a more confused and uncertain manner, as in "I think I need to go through the list somehow". Utterances not related to the programming process were ignored and not coded. Steps correspond to written code in Python (i.e., in order for something to be labelled as a step, it had to be written in the Python IDE). A step or sequence of steps could immediately follow a goal. For example, once the goal to iterate through the list was expressed, the subsequent code "for x in rains:" corresponds to a step. Steps could also occur in the absence of a goal - in this case steps were identified on the basis of a written Python line, or short written statement (for example a conditional break, which can be written on 2 lines, but it's functionally one expression). Finally, steps could occur later on in the solving process after the corresponding goal.

Qualitative Coding Process

Applying the coding scheme to the transcripts involved reading each utterance and labelling it with a goal tag (see Table 1) if it corresponded to a goal. It also involved labelling the Python code with the step labels (see Table 2). Once the transcripts were coded, the goal and the step codings were extracted and recorded in a separate document tracking their chronological order (i.e., the order that goals were stated and

Table 1: Goals coding scheme

Goal label	Description
Request user inputs	Request user vote input (department/colour)
Iterate user inputs	Use loop to go through user inputs
Identify input	Identify what department, colour or stopping condition was input
Track count - specify by data structure	Increment count by 1 for when target information is met
Stop loop	Break loop when stopping condition is met
Compare count - specify by data structure	Compare counts to find departmental winner or popular winner
Initialize count variable(s)	Set up variable(s) to store data as required by the problem
Print outcome	Output president according to comparisons
Initialize count dictionary(ies) - x	Set up a dictionary of size x to count the number of votes input
Initialize variable total	Set up variable total to store sum of positive numbers
Iterate through list	Iterate through the list of values given
Calculate average	Calculate average of positive numbers in list using sum and count
Track total	Add positive numbers in list to total
Other - idiosyncratic	Goal does not fall into above definitions and is highly idiosyncratic to programmer

Table 2: Steps coding scheme

Steps label	Description
Initialize variable	Create a variable to be used by other steps in the program
User input	Input from user taken
Loop	Iterate until a condition is met or go through all items in a data structure
Condition	A Boolean statement that acts as a condition for another step
Increment variable	Add a value to the variables
Input	Get an input from console
Output	Output information to console
Compare	Find if the value of one variable is bigger than another variable
Calculation	Perform a calculation with a variable
Break	Stop loop
Other - idiosyncratic	Performed a step that did not conform to expected solution

steps were implemented; these did not include time-stamps and only snippets of the original utterances). We refer to the latter codings as chronotranscripts. Once chronotranscripts were made for all of the participants for the rainfall problem, they were compared to each other and the canonical solution and general trends were identified. A qualitative analysis of the chronotranscripts was conducted to identify solution strategies as well as similarities and differences in the problem-solving strategies of experts and novices. To examine how goal decomposition differs between experts and novices, and to identify possible relationships between goals and between goals and steps, we focused on the order of goal identification and step implementation, as well as any variations of the goals and their implementation.

Canonical Solution Canonical solutions for the rainfall and ballot problem were provided by two educational experts within the research team, to which expert and novice solutions were compared. While the solutions are not Pythonic in that they could be more compact, they represent a common strategy taught to students in first year programming classes that has the advantage of being language independent. As the experts had programming experience but were not Python experts, the solutions are appropriate for the present study. The canonical solutions assume that the programs are written in the particular order specified. For the rainfall problem, first the two variables, sum and count are initialized, and are used to track how many positive numbers there are in the list and the sum of the positive list values. Next, the program iterates through the list of rainfall amounts (using a for-loop). Within the loop, the program checks if the current value is the stop signal (-999), which stops the loop; otherwise, the program checks if the current number is positive and if so updates the sum and count. Once all values in the list are processed or the stop value is found, the program calculates the average rainfall. For the ballot problem the canonical solution starts by initializing a dictionary that stores the votes for each department and for each candidate in the department.

Next, a while loop is used to iterate through the votes being input and breaks the loop when -1 is entered as a vote. Within the loop, the vote colour and the department for the vote are requested. To increment the correct dictionary element, the input colour and department are used as the keys for the dictionary and the value for that key pair is incremented. Once the loop is stopped, the program initializes a variable 'victories' that counts the number of departments the Red candidate wins to determine the popular winner. Next, a for-loop is used to iterate through all the departments in the dictionary and comparing the Red and Blue votes to find the winner for each department. For every iteration, the victories variable is incremented if the Red candidate is a departmental winner. The departmental winner is output within each iteration. The next block of code outside the for-loop uses the victories variable such that if it is > 1 this means Red has won in more than 2 departments and is thus the popular winner, otherwise Blue is the popular winner and prints the result.

Results

As noted above, the results are based on a qualitative analysis of expert and novice programming solutions.

Deviations from the Canonical Solution

For the most cases, both novices and experts' final programs were similar to the canonical solution for both the rainfall and ballot problems. When completing the rainfall problem, participants created and initialized variables to track the positive numbers and iterated through the rainfall data list using a loop. Most expert and novice participants addressed the stop signal (-999) correctly and tracked the sum and count of the positive numbers within a single loop over the list. Finally, both experts and novices calculated the average using their variables for sum and count outside of the loop. However, not all participants followed this solution exactly. For instance, participant P117 implemented additional steps not included in the canonical solution. The ballot problem had more deviations from the canonical solution. Most experts used more than one dictionary to store the vote counts. All other participants used a series of simple variables to count the votes. Consequently comparisons to find departmental winners used if else statements instead of for-loops like in the canonical solution. Participants that were able to find the popular winner used two variables to count the number of wins for the candidates instead of one as in the canonical solution.

Novice vs Expert Difference in Programming Strategy

Rainfall Problem When it came to the rainfall problem, most novices (and some experts like participant P115) identified and implemented the goal and step of initializing the variables related to sum and count in sequence. In contrast, the experts were more variable in terms of when they implemented this goal and step. Some experts initialized sum before count, with at least one other non-variable related step or goal in between. For example, expert participant P117 first initialized the variable sum (called rain_sum in their program), then only initialized the variable count (called rain_count) when they had nearly completed their solution. Additionally, some novices had additional sources of variability in their solutions compared to the experts' solutions, due to inefficiencies and inaccuracies in their solution Some novices struggled more with the problem, and would use extra steps and had misconceptions in their approach. To illustrate, participant P109 not only added additional code not required for the problem, which was not used in the canonical solution nor by any other participant, but also made errors when calculating the average. Specifically, rather than keeping a count of the positive values, P109 relied on the built in len function and used that in the average calculation. Thus, their average calculation incorrectly divided the sum of the positive numbers by the total length of the list (that included both the positive and negative numbers). In contrast, experts broadly implemented the standard canonical solution (even though their chronotranscripts deviated from the canonical order). Some deviations did exist, but unlike for the novice deviations they did not represent inaccuracies in the solution process, instead reflecting idiosyncratic variation. To illustrate, expert P117 dealt explicitly with negative values in the list rains, which they handled at the same time as the stop code, and P115 identified the goal to ignore negative numbers but did not implement any steps towards it.

Ballot Problem How votes were stored is an important difference between novices and experts in the ballot problem. Most experts used one or more dictionaries to store and classify the votes and some also used a for-loop to iterate through the dictionaries to compare the votes to find departmental winners like PX and P120. All novices used simple variables to store the votes (see Figure 1 line 18), most often six variables, one for each department and colour pair and used if-else statements to find the departmental winners. Experts tended to identify and implement vote count initialization earlier than novices (see Figure 2 lines 1 to 4). Novices, however, identified iterating user input as their first goal and start coding a while loop and getting the user inputs (see Figure 1 lines 1 to 3). Count initialization was thus a goal they uttered while working on the loop. This shows that the main focus of the novices was on the iteration aspect when reading the problem while experts had a more holistic approach and knew the choice of data structure was a priority. Novices also faced issues related to the count variables. Several novices had trouble visualising how to store the votes and would make several goal changes on how many variables to use. P112 changed the number of counts used 5 times (see Figure 1) at different points in the code generation while working on the while loop. Novices that changed their counts would then run out of time and not complete the problem.

Summary Novices and experts differ quite significantly in their problem solving strategy; both in terms of identifying the goals of the problem, and implementing code to address

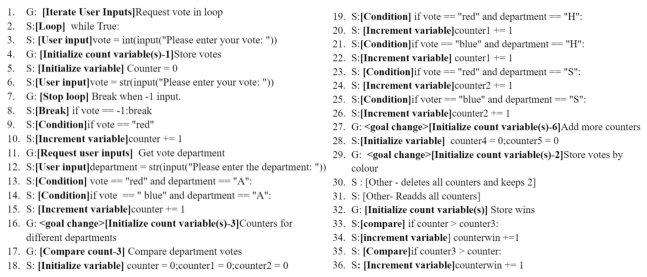

Figure 1: Novice P112 Chronotranscript - Ballot Problem

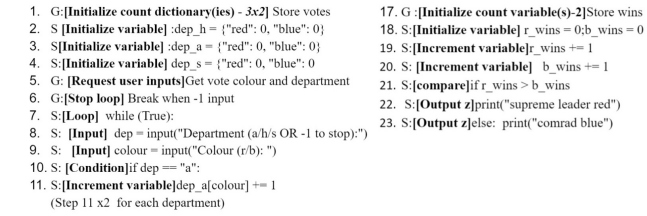

Figure 2: Expert P117 Chronotranscript - Ballot Problem

those goals. These differences become more pronounced as the complexity of the problem increases. Experts are much more reliably able to identify the goals of complex programming problems and are much more flexible in the functions and data structures they choose to use. Differences between experts often reflect different priorities in higher level concerns such as readability or ease of use. These findings support Soloway (1986) notion of programming plans, with experts having accumulated and refined numerous plans (and their associated goals) that they can apply towards a variety of problems. Conversely, even if familiar with complex data structures and functions, novices lack the knowledge to reliably employ them during problem solving. Consequently, they struggle to visualise how to correctly store the data and thus correctly implement their identified goals. Instead they identify iterating through the data as a plan focus (as described by Rist (1989)) around which they try to expand the rest of the code.

Discussion

The main goal of the present study was to gather data on novice and experts' process of program generation in order to construct cognitive models of both expert and novice programming approaches. The proposed cognitive models will use the same goals and steps identified in the present study (formalizing the procedural and declarative memory contents respectively), and will rely on expert and novice strategies identified through this study and through earlier work. Findings from prior work include the findings that show expert programmers rely on algorithms (Soloway, 1986) and that novice programmers expand around a program focus (Rist, 1989). The chronotranscript analysis provides strategy information in the order of the goals identified by novice and expert programmers and how they chose to implement those goals. Implementing expert strategies in a cognitive model that can produce solutions to simple programming problems will further our understanding of how goal-level knowledge is used by expert programmers when generating code to solve problems. Additionally, implementing models of novice approaches to problem solving that rely on strategies identified in prior work (Rist, 1989) and the present study will help formalize the knowledge representational differences between expert and novice programmers. This line of work is still relevant, even with the prevalence of advanced large-language models such as Chat-GPT. While ChatGPT is able to produce code, there are clear indications that the code was not produced by a human. Future work can thus compare the strategies used by ChatGPT in programming problem solving against the strategies by humans on the same problems and using the same sets of goals and steps as in this paper. This will give an insight on some fundamental ways that the model approaches programming problems differently than humans. For the rainfall problem, participants' final solutions tended to include many of the same steps as the canonical solutions, but both experts and novices varied the order that they produced the steps in a way that did not impact the validity of the solution. This demonstrates that participants addressed the problem goals in variable order. For example, both novices and experts varied whether to first stop the loop or increment the sum and count variables within the loop. Novices had other sources of variability that did affect the validity of their solution and reflected gaps and/or inefficiencies in their domain knowledge, such as identifying the index of the stop signal in a separate loop. As for the ballot problem, both expert and novice participants deviated substantially from the canonical solution and each other, relying on different data structures and thus producing vastly different chronotranscripts, with experts using a more complex data structure (dictionaries), and novices relying on sets of single variables. The implications for a cognitive model would be that when modelling more complex problem solving, multiple data constructs may be used, and the one used will reflect the knowledge base and experience level of the programmer. Novices and experts also differ in their approach to problem solving. Experts first took the time to assess the problem and decide on the best data structure to resolve the problem, whether when selecting the simple variables needed for the rainfall problem, or the dictionaries needed for the ballot problem. Conversely, novices would often jump right into implementing the instructions of the problem, only considering the data structure once they had something written down. This is in line with the findings of Chi et al. (1981) with physics problems, where, early in problem solving, experts engage in slower, deeper processing of the problem than novices.

References

Castro, F. E. V., & Fisler, K. (2020). Qualitative analyses of movements between task-level and code-level thinking of novice programmers. In *Proceedings of the 51st ACM technical symposium on computer science education* (pp. 487–493). ACM. doi: 10.1145/3328778.3366847

Chi, M. T. H. (2006). Two approaches to the study of experts' characteristics. *The Cambridge Handbook of Expertise and Expert Performance*, 21-30. doi: 10.1017/CBO9780511816796

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, *5*(2), 121–152. doi: 10.1207/s15516709cog0502_2

Corbett, A. (2000). Cognitive mastery learning in the ACT programming tutor. In *Adaptive user interfaces. AAAI SS-00-01*. Palo Alto, California: The AAAI Press. Retrieved from https://aaai.org/papers/0007-ss00-01-007-cognitive-master-learning-in-the-act-programming-tutor/

Frischkorn, G., & Schubert, A.-L. (2018). Cognitive models in intelligence research: Advantages and recommendations for their application. *Journal of Intelligence*, *6*(3), 34. doi: 10.3390/jintelligence6030034

Gertner, A. S., & VanLehn, K. (2000). Andes: A coached problem solving environment for physics. In G. Gauthier, C. Frasson, & K. VanLehn (Eds.), *Intelligent Tutoring Systems, Lecture Notes in Computer Science* (Vol. 1839, pp. 133–142). Springer Berlin Heidelberg. doi: 10.1007/3-540-45108-0_17

Gobet, F., & Simon, H. A. (1996). Templates in chess memory: A mechanism for recalling several boards. *Cognitive Psychology*, *31*(1), 1–40. doi: 10.1006/cogp.1996.0011

Koedinger, K. R., & Anderson, J. R. (1993). Reifying implicit planning in geometry: Guidelines for model-based intelligent tutoring system design. In *Computers as cognitive tools* (pp. 15–46).

Leonard, W. J., Dufresne, R. J., & Mestre, J. P. (1996). Using qualitative problem-solving strategies to highlight the role of conceptual knowledge in solving problems. *American Journal of Physics*, *64*(12), 1495–1503. doi: 10.1119/1.18409

Pirolli, P. (1986). A cognitive model and computer tutor for programming recursion. *Human–Computer Interaction*, *2*(4), 319–355. doi: 10.1207/s15327051hci0204_3

Recker, M. M., & Pirolli, P. (1995). Modeling individual differences in students' learning strategies. *Journal of the Learning Sciences*, *4*(1), 1–38. doi: 10.1207/s15327809jls0401_1

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*(3), 389–414. doi: 10.1207/s15516709cog1303_3

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858. doi: 10.1145/6592.6594

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595-609. doi: 10.1109/TSE.1984.5010283

Spohrer, J., Soloway, Elliot, & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human–Computer Interaction*, *1*(2), 163–207. doi: 10.1207/s15327051hci0102_4