Validating, Refining, and Identifying Programming Plans Using Learning Curve Analysis on Code Writing Data

Mehmet Arif Demirtaş
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
mad16@illinois.edu

Max Fowler
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
mfowler5@illinois.edu

Nicole Hu
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
nhu6@illinois.edu

Kathryn Cunningham
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
katcun@illinois.edu

ABSTRACT

Background and Context: A major difference between expert and novice programmers is the ability to recognize and apply common and meaningful patterns in code. Previous works have attempted to identify these patterns as programming plans, such as counting or filtering the items of a collection. However, these efforts primarily relied on expert opinions and yielded many varied sets of plans. No methods have been applied to evaluate these various programming plans as far as their alignment with novices' cognitive development.
Objectives: In this work, we investigate which programming plans are learned as discrete skills. To this end, we evaluate how well students transfer their knowledge between problems that test a particular plan. Further, we explore how plan definitions can be improved to better represent student cognition using historical data on student performance collected from a programming course.
Method: We apply learning curve analysis, a method for modeling student improvement on problems that test a particular skill, using programming plans as a skill model. More specifically, we study student submissions on code-writing exercises in Python from a CS1 class for non-majors that includes many small programming problems as well as implicit and explicit instruction on patterns. We compare the learning curves for ten programming plans across seven semesters of the same course.
Findings: Students develop the skill of using some programming plans in their solutions, indicated by consistent declines in error rates on practice opportunities for a subset of plans in multiple semesters with various conditions. Most consistently learned plans have clear and concrete goals that can be explained in natural language, as opposed to having abstract definitions or being explained in terms of language structures.
Implications: We show that learning curve analysis can be used to empirically assess the cognitive validity of proposed programming plans, as well as compare various plan models. However, our work also indicates that instructors should be cautious when assuming that introductory programming students can apply more abstract programming plans to successfully solve new problems, as plans with increased specificity tend to better explain the learning process in our observations.

CCS CONCEPTS

• **Social and professional topics** → *CS1*; • **Information systems** → Data mining.

KEYWORDS

learning curve analysis, CS1, programming plans, programming patterns, knowledge components

ACM Reference Format:
Mehmet Arif Demirtaş, Max Fowler, Nicole Hu, and Kathryn Cunningham. 2024. Validating, Refining, and Identifying Programming Plans Using Learning Curve Analysis on Code Writing Data. In *ACM Conference on International Computing Education Research V.1 (ICER '24 Vol. 1), August 13–15, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3632620.3671120

1 INTRODUCTION

Knowledge of programming plans, which are commonly used pieces of code that achieve a specific goal, differentiates novice and expert programmers [46, 47, 55]. Consequently, it has been argued that programming plans should be central in introductory programming instruction to close the novice-expert gap more quickly. Some argue that programming plans should be taught explicitly [55], while others believe that these common patterns should inform the design of examples and curricular ordering [36, 37]. The goal of this plan-focused instruction is that learners will more easily develop important mental schemata, or "chunks," that they can apply to solve new problems. Studies have shown that decomposing complex problems into such chunks improved problem-solving performance [16], and students with minimal programming experience can successfully solve examples from specialized applications (e.g. web scraping) when plan-like structures are utilized in instruction [13].

While many proposed sets of introductory programming plans exist [4, 7, 14, 23, 25], these sets include small differences due to design choices made by researchers. There has not been an established set of introductory programming plans with consensus from the community. Surprisingly, there are few efforts to understand which plans are actually representative of student cognition. There

ICER '24 Vol. 1, August 13–15, 2024, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0475-8/24/08
https://doi.org/10.1145/3632620.3671120

is reason to doubt that the programming plans generated by expert programmers (even instructors) represent schemata appropriate for learning by introductory programming students. Even the best instructors can suffer from expert blind spot [38], where they can no longer recall what tasks and skills are difficult for novices. Experienced programmers may be likely to believe that overly abstract programming plans are appropriate for novices. If proposed introductory programming plans truly correspond to the key schemata developed by students during programming skill development, we should see evidence that students can apply and transfer knowledge about each programming plan as a discrete skill. Thus, the degree to which a plan is learned as a discrete skill may be an important factor to consider when designing programming plans.

Learning curve analysis, which is an approach for evaluating student skill development and skill models using data from students' problem-solving performance [10], can help identify which concepts in programming correspond to discrete skills. Learning curve analysis is performed with a knowledge component (KC) model, which represents the key skills and abilities students need to solve problems in a given domain [28].

Learning curves show the trends in error rate across attempts on problems that test a KC, which models the development of the skill represented by that KC. In other words, a decreasing error rate on a set of problems that practice a KC indicates a discrete, transferable skill that is being learned by students, and that skill is represented as a knowledge component in the domain. KCs without decreasing error rates could be iteratively refined to represent actual skills.

We propose the use of programming plans as a knowledge component model for learning curve analysis. By looking at which plans are learned as discrete skills that are transferred to new attempts that practice the same plan, we can evaluate plans by their ability to represent student cognition. Moreover, we can provide data-driven design choices for modifying the definitions of programming plans to capture discrete skills on learning curves. While there have been studies that applied learning curve analysis on code-writing data with other KC models, their results have been limited: by either a failure to observe decreasing error rates on most knowledge components [45], or by limited interpretability of the identified knowledge components [51]. Programming plans may be an accurate and interpretable KC model to explain programming skills on code-writing exercises.

In this work, we apply learning curve analysis on open-ended code-writing data with a set of patterns in introductory programming [23] as knowledge components. We annotate a set of 209 short programming exercises by the programming plans they test and use these problems for learning curve analysis on student data from seven semesters collected in a large-scale introductory Python course to validate and refine our plan definitions empirically.

We investigate three research questions:

- **RQ1**: Does knowledge of programming plans explain skill development in introductory programming?
- **RQ2**: Which introductory programming plans have the most evidence as discrete skills?
- **RQ3**: What design implications for improving programming plans can be inferred from learning curve analysis?

We observe that programming plans can partially explain skill development in programming, with evidence that some plans correspond to discrete skills students develop in introductory programming. We note that plans with more concrete definitions and clearer goals tend to be learned more easily by students, such as `counting` and `sum` plans, whereas more abstract plans may be less appropriate to be modeled as discrete skills.

Our work presents implications about the appropriate level of abstraction when designing courses that include implicit or explicit instruction on programming plans. Moreover, we show that learning curve analysis can be an informative tool for monitoring student progress on skills related to the use of programming plans, as well as evaluating proposed sets of programming plans for their alignment with student cognition.

2 BACKGROUND

2.1 Programming plans and expertise

In studies of programming, as well as across many other fields, it has been shown that experts notice meaningful underlying patterns in the content of their domain [11], as well as when to appropriately apply knowledge and strategies [21, 52]. These findings suggest that the difference in performance between novices and experts can be explained by knowledge both in the form of strategies for approaching problems and abstracted solution types, also known as schemata. In programming specifically, these schemata have been termed *programming plans* [18, 54–57].

Programming plans have some variety in their precise definition (as well as alternative nomenclature, like "programming patterns" [24], "templates" [12], "algorithmic patterns" [36], or similar structures like "roles of variables" [49, 58]). Some recent works further distinguish between "plan-schemata" and "standard plans", with the former indicating the mental representation of a typical solution and the latter indicating the instantiation of the solution in a particular programming language [26]. In this work, we adopt a commonly used definition from early programming plan research: a common code pattern associated with a goal that it achieves [44, 55].

Knowledge of programming plans can be used to decompose many programming problem types by enabling problem-solving through an effective top-down approach [18, 44, 47]. In addition to the knowledge of programming plans, the ability to alter and combine plans for a particular problem is another important skill for effective problem-solving [55]. This includes both "tailoring" a plan [55] by modifying parts of it for a given problem, and also plan-composition strategies where multiple plans are combined by concatenating or interleaving them [16, 26]. Bugs in student programs can be explained by students' misunderstanding of plans [60] as well as their errors in composing plans [59], and evidence suggests that students may experience difficulties with complicated plan-composition implementations [19, 20]. While Robins concludes in a recent review [46] "there is considerable evidence that the plan is the basic cognitive unit used in program design and understanding," some authors have critiqued programming plans as not representing deep structure that can span languages [18].

2.2 Programming plans and instruction

While experts can readily notice deep structure in familiar problems, novices struggle to see past surface-level details [9, 48]. Instruction that highlights underlying structure can help students more readily transfer their knowledge to new contexts [53]. Based on the insights from early plan research, pattern-based programming instruction has been proposed as a framework to support schema development during introductory programming education [12, 15]. Later works refined this framework as pattern-oriented instruction (POI) [36] and provided evidence on benefits of POI such as high competence in problem-solving and a positive influence on self-efficacy [37]. An alternative approach identified stereotypical code patterns through a set of variable roles that account for the vast majority of variable usage in novice programs [32, 49]. These roles were shown to act as a helpful tool for improving students' mental models [50, 58].

All pattern-focused instructional approaches depend on a set of programming plans. Many patterns in CS1 have been identified by researchers [5, 7, 14]. Yet, despite efforts to create a comprehensive set of plans for introduction to programming, there is only a partial consensus on which plans should be included in course curricula [23, 25]. Moreover, while these efforts include elaborate reviews of prior work and existing course curricula, the identified sets of plans are still susceptible to expert blind spot [38], as there is limited analysis on the extent to which these proposed programming plans correspond to the cognitive processes of novices. A set of evidence-based programming plans that are empirically supported by student problem-solving data can address this problem.

2.3 The Knowledge-Learning-Instruction Framework

We base our evaluation of programming plans in the Knowledge-Learning-Instruction framework (KLI), proposed by Koedinger et al. [28]. The KLI framework is an instructional theory that defines learning in terms of three types of events. Learning events are the implicit processes that model the changes in student cognition. Instructional events are explicit processes that cause learning events to happen, such as listening to a lecture or reading a textbook. Assessment events are also explicit processes in which a student's current knowledge is evaluated, such as responding to a question. While learning events cannot be directly observed, they can be modeled by monitoring instructional and assessment events. Namely, the KLI framework models learning by analyzing *knowledge components (KCs)*, acquired units of skill that can be inferred through performance on a set of problems. Specifically, by tracking the student's performance on instructional events (i.e. practice problems), it is possible to model which skills are being acquired. The set of KCs that represent the skills being learned in a given domain is called a KC model. These models can guide instructional decisions [28], track student performance, and address misconceptions [22, 40, 43].

A main assumption of the KLI framework is that students will perform better on problems testing a KC as they experience more learning events (e.g. more practice opportunities). If a KC model is accurate, we can track student performance on problems that require the same KC and observe a decreasing error rate through later attempts. Put another way, accurate KCs in a domain can be detected by monitoring which skills show a consistent decrease in error rate on related problems. This process is also known as learning curve analysis [10]. When students' averaged error rates on problems testing a KC are plotted as a function of how many attempts they had, we expect to see a decreasing learning curve as predicted by the power law of practice [39]. Learning curves have been used to detect knowledge components in many domains, including algebra, physics, geometry, and language studies, through online tools and data repositories (e.g. DataShop [27, 29]).

Learning curve analysis can produce empirical evidence of the success of a knowledge component model. A declining curve indicates that students can transfer their knowledge between problems testing a given knowledge component. Thus, a decline in error rate is associated with a *successful* knowledge component i.e. a component that corresponds to a discrete skill. In contrast, a flat curve represents either a failure in instruction or a mismatch between the proposed knowledge components and the set of related problems, as students do not acquire a transferable skill between problems. In such scenarios, KC models can be refined to improve learning curves and capture the underlying skills in the domain.

2.4 Learning curve analysis in programming

Learning curve analysis has been utilized to some extent in intelligent tutoring systems (ITSs) for programming. One of the earliest cognitive tutors, LISP Tutor, used knowledge components derived by experts [3], and increased learning outcomes with individualized problem selection [1]. Learning curves have been used to evaluate the performance of different instructional approaches within ITSs, such as Excel-Tutor and SQL-Tutor, as well as refining the granularity of the KC models in these systems [34]. More recently, data from code comprehension questions was analyzed with learning curves to test expert-annotated KC models [2]. While these efforts all successfully modeled student skill development in programming, they contained highly constrained problems, not the open-ended, code-writing exercises commonly used in programming courses.

Rivers et al. [45] applied learning curve analysis to programming exercises with open-ended code-writing data by breaking each program submission into small steps using abstract syntax tree (AST) nodes. However, their learning curves did not capture the expected decline in error rates, and they concluded that syntax elements may not be an appropriate KC model to represent student skills. Shi et al. [51] took a different approach to open-ended code-writing data by using a neural network to determine a set of programming problems that potentially test the same underlying KC and then inferring the KC from the set of predicted problems. While their learning curves had high fit with the power law of practice, they stated that the interpretability of the model is limited as each KC combines multiple micro-concepts that may not reflect how we reason about programs. Finding a knowledge component model suitable for open-ended code-writing problems while capturing interpretable skills in programming would help us understand students' cognition during programming learning, as well as how to help students learn these skills.

3 METHODS

In this section, we explain our dataset, the mechanics of the learning curve analysis, and evaluation methods for the results.

3.1 Dataset

3.1.1 Data collection. Our data contains instructor solutions and student submissions on code-writing problems from weekly homework of an introductory Python course for non-CS majors at a large public university, collected for seven semesters from Fall 2019 to Fall 2022 with 400 to 700 enrolled students each semester. Students were from a variety of non-Engineering majors, with Business students being the majority and Liberal Arts and Sciences students a large minority. First or second-year undergraduate students made up 75% to 80% of the class. The class was balanced in terms of gender, with an average of 42.6% female student enrollment. 41.8% of students were white, 21.3% were Asian, 16.6% were International, 11.9% were Hispanic, 4.4% were Black/African American, 3.2% were Multi-race, and 0.9% did not disclose.

Weekly homework contains small programming exercises where students write a function for a short task definition, such as summing numbers from a list, as well as other exercises (e.g. multiple choice, Parsons problems) that are excluded from our analysis. Code-writing problems were hosted on the PrairieLearn platform, where each submission is immediately graded on a set of test cases, with infinite attempts to complete homework up to the deadline with no penalty for incorrect answers [63].

The dataset contains 209 problems in total, including 131 unique questions and their isomorphic variants, which are questions generated from existing items to test the same underlying concepts [8, 17, 33]. 117 problems feature at least one of the programming plans we consider (see Section 3.1.2). However, only 50 to 60 of these problems are assigned as homework in a given semester, as assignments were modified between semesters. The instructor solutions for these problems have 6.26 lines of code on average ($SD = 1.94$).

The course features multiple forms of instruction on programming patterns. Implicitly, many homework problems reflect goals commonly identified in pattern literature. Explicitly, the lectures in weeks 9 and 10 provide names to the patterns students have already been practicing in the class.

We opt to treat submissions from each semester as a distinct, uncombined student sample, as students experienced different learning opportunities in each semester. The dataset features sections taught by: a solo instructor (Fa19, Sp20), one pair of co-instructors (Fa20, Sp21, Fa21), a second pair of co-instructors (Sp22), and a third pair of co-instructors including one instructor new to the course material (Fa22). Additionally, instructional methods differed due to COVID (entirely online from halfway through Sp20 to Sp21, hybrid until Fa22). While the course typically has four exams, the Sp22 semester had two exams, which may have impacted student homework and studying behaviors [6, 35].

3.1.2 Plan annotation. We define programming plans as commonly used programming idioms, or *patterns*, that have a particular goal easily explained in natural language and an accompanying implementation. As our initial set of programming plans, we used the patterns identified by Iyer and Zilles in their "pattern census" of introductory programming [23]. Each problem in our dataset was labeled with the programming plans required to implement the solution, using instructor solutions as the ground truth.

For the annotation process, two members of the research team independently labeled each instructor solution with patterns from Iyer and Zilles [23] and the preceding thesis by the same author [24]. Each researcher assigned a set of labels to each instructor solution indicating the plans exercised in the solution. Labeling was performed on the set of all code-writing problems in the course (homework, quiz, and exam questions), which totaled 462 problems. After an initial coding session on 200 problems, inter-rater reliability between two coders using Krippendorff's Alpha [31] was 0.747, using MASI distance for set-valued labels [42], indicating moderate agreement. To increase the accuracy, an iterative annotation approach was adopted where definitions for the programming plans were refined through discussion beyond those provided by Iyer [24] to obtain more precise definitions appropriate for labeling. All disagreements were reconciled through discussion. A third member of the research team independently coded a set of 100 problems using the refined codebook and obtained a Krippendorff's Alpha of 0.857 with the reconciled labels, indicating strong agreement.

A list of programming plans and goals associated with each plan can be found in Table 1. 117 of 209 annotated homework problems were found to require at least one programming plan. We excluded problems that did not require plans from further analysis. Out of 117 problems, 13 of these include two plans and only a single problem includes three plans. We identified 10 plans that were tested in at least one problem in all semesters. However, only four plans (`processAllItems`, `filterACollection`, `findBestInCollection`, and `counting`) were tested with more than three opportunities in all semesters. Including more practice opportunities for all plans may result in a more accurate measurement of student mastery when using learning curve analysis [45]. Figure 1 shows the average number of problems that include a particular plan in all semesters.

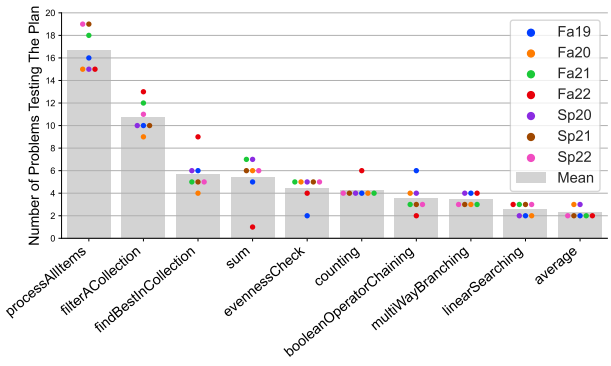


Figure 1: Average number of homework problems that require each plan. Individual semesters are shown with dots.

3.2 Learning curve analysis

After the identification of the programming plans required for each homework problem, learning curves can be computed for each plan by treating plans as a set of knowledge components (KCs) and problem submissions as steps.

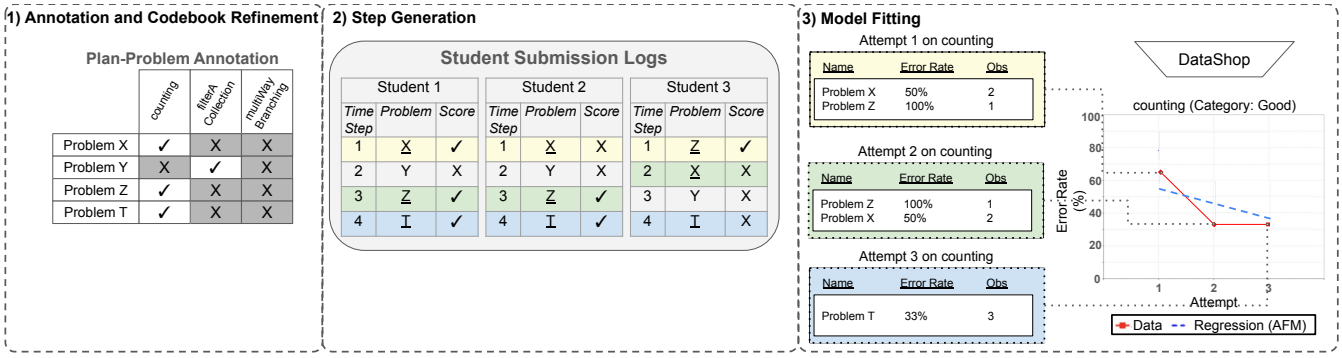


Figure 2: Our process for applying learning curve analysis to data from code-writing homework submissions. First, homework problems are annotated with requisite plans (Section 3.1.2). Next, first submissions on each homework problem that test a plan are labeled as steps (Section 3.2.1). Finally, error rates across attempts are plotted on DataShop (Section 3.2.2).

Table 1: Plans from Iyer and Zilles [23] that are tested by at least one problem in every semester of our course.

Plan	Goal
`processAllItems`	Iterate over the items in a collection
`filterACollection`	Select items from a collection that satisfy a condition
`findBestInCollection`	Find the value that has the greatest value by an arbitrary measure
`sum`	Compute the total of items from a collection
`evennessCheck`	Check if a number is even or odd using the modulus operator
`counting`	Compute the number of items from a collection
`booleanOperatorChaining`	Combine multiple logic expressions to make a decision
`multiWayBranching`	Split into three different branches based on a logic expression
`linearSearching`	Find the first matching item that satisfies a condition
`average`	Compute the mean value of items in a collection

3.2.1 Step generation. Learning curve analysis first requires computing student transaction data, which includes all *attempts* a student made at exercising each KC and the *outcome* of each attempt [28]. In most domains, these attempts are made in the context of an intelligent tutoring system (ITS), where problem-solving is highly scaffolded and broken into small steps [62]. Thus, the term *step* is used to represent the opportunity to demonstrate a KC. The small steps of an ITS mean that KCs in such a scenario can be narrowly focused to quite granular skills (e.g., [30]).

However, code-writing assignments in programming are not decomposed into such atomic steps, as students are expected to submit a complete program without any intermediate checkpoints. Rivers et al. attempted to measure the development of fine-grained skills (use of syntactic structures) in similar code-writing data to ours using a specialized step generation algorithm that created multiple steps for a single homework submission post-hoc, but their results did not show clear learning in almost all cases [45]. By contrast, our programming plan KCs encompass a variety of syntax elements in each step while also representing a higher-level goal.

Our dataset contains many short programming problems with a clear focus on one (or occasionally two) plans. Most of our problems (88%) require only a single plan, and, as a result, the majority of student homework problems only exercise a single KC in our model. Due to this close matching of problems to KCs, we use a straightforward way of generating steps: an entire student submission is an attempt at demonstrating the associated programming plan KC(s). We use the test cases for the problem to understand if the step was completed correctly or not. If the student passed all test cases, we consider that opportunity to be a successful demonstration of knowledge of that programming plan. Missing at least one test case is an unsuccessful demonstration of knowledge. Since students may be able to use information from problem-specific test case results to correct their answer without necessarily learning a plan, we follow the suggestion of prior work and only consider the first attempt on each problem in our transaction data [45].

3.2.2 Model fitting. Transaction data was uploaded to DataShop [27] for learning curve analysis. DataShop is an online data repository for student interaction data that calculates learning curves and performs additive factor modeling automatically. Additive factor modeling (AFM) is a multiple logistic regression method for predicting a student's error rate on a KC [10]:

$$ln(\frac{p_{ij}}{1-p_{ij}}) = \alpha_i + \beta_j + \gamma_i T_{ij} \quad (1)$$

where p_{ij} is the probability of student i correctly exercising knowledge component KC_j with α_i representing student ability for student i, β_j representing the difficulty of KC_j, and $\gamma_i T_j$ represents the ease of learning for KC_j (*learning rate* γ_j) multiplied by number of attempts student had on the topic (T_{ij}).

When a KC represents a discrete skill, we expect the fitted curves calculated with AFM to show a decline in error rate, indicating that students master the underlying KC with more opportunities and are able to solve new examples that practice the same KC (i.e. a high learning rate γ). An overview of this generation process, along with an example learning curve, can be seen in Figure 2.

3.3 Knowledge component evaluation

We use two sets of measures of learning curves to evaluate the validity of the knowledge components. First, we look at the extent to which individual learning curves satisfy the expected behavior explained in Section 3.2.2 by categorizing the decline in error rate.

Second, we look at a set of metrics that assess the complexity of the AFM model and its fit to the underlying data distribution.

3.3.1 Learning curve categorization. Learning curves are categorized according to the number of total attempts, error rates observed in those attempts, and the slope of the AFM curve (γ_i in Equation 1). This slope represents the learning rate of the corresponding KC, and a higher slope indicates a KC that is easily learned. We use the categorization by DataShop following prior work [45], which includes the five categories shown in Table 2. Two of these categories, *Good* and *Still-high*, indicate healthy learning by showing a decline in error rate with more practice (i.e. positive γ_i). We also report γ_i values where available following prior work [2, 34].

Table 2: Datashop's categorization system for knowledge components based on learning curve parameters.

Category	Criteria	Interpretation
Too little data	Fewer than 3 opportunities	Not enough measurements for this concept.
Low and flat	All attempts have an error rate lower than 20%	Students have mastered the concept before their first recorded attempt
No learning	The fitted curve does not have a decreasing trend ($\gamma_i \leq 0$)	Students do not increase their mastery of the KC
Still high	The fitted curve has a decreasing trend, but the error rate for the final attempted problem is higher than 40% ($\gamma_i > 0$)	Students are still learning but need more attempts to master the KC.
Good	The fitted curve has a decreasing trend and the error rate for the final attempted problem is lower than 40% ($\gamma_i > 0$)	Students have mastered the KC

3.3.2 Model fit assessment. We use multiple quantitative metrics for assessing our knowledge component model in addition to categorizing observed trends in error rates. Namely, we use R^2 as a measure of the fit between the AFM model and the underlying data distribution [2, 34]. Previous work considers $R^2 > 0.1$ to indicate partial fit and $R^2 > 0.5$ to indicate strong fit, in line with social science literature in general [41]. A higher R^2 represents that the AFM model explains a greater amount of variance in the data.

Moreover, we use two model comparison metrics: AIC (Akaike information criterion) and BIC (Bayesian information criterion), for evaluating how well a model explains the underlying data distribution while penalizing the number of parameters [29]. A lower AIC/BIC value is preferable as it indicates a model that better navigates the tradeoff between model complexity and model fit.

4 FINDINGS

4.1 Applying learning curve analysis on code-writing data

First, we evaluate how well the set of programming plans proposed by Iyer and Zilles [23] work as a knowledge component model for code-writing on problems requiring those plans. By doing this, we investigate to what extent we can explain skill development in code-writing as development in the ability to use individual plans. We present learning curves that average all KCs for each of the seven semesters in Figure 3. This evaluation approach was used by Rivers et al. in prior work that applied learning curve analysis to code-writing data, so we include that result as a comparison [45]. These curves are created by calculating the learning curves for individual KCs and averaging them all at each attempt. For a successful KC model that accurately models the underlying skills in the domain, we expect this aggregated curve to reflect learning by starting at a high error rate and declining steadily as students master individual KCs with more practice. We further present the fit of AFM curves on aggregated data (R^2) and report averages and standard deviations of learning rates of KCs from that semester ($\bar{\gamma}$).

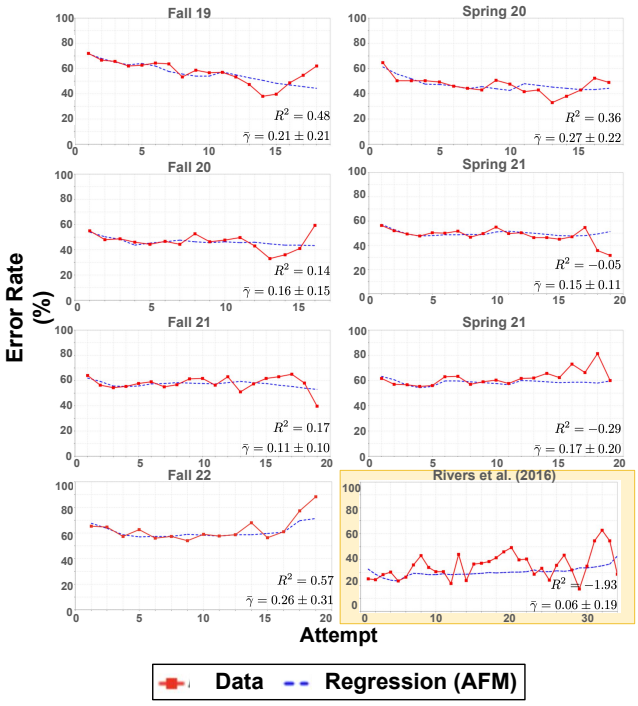


Figure 3: Averaged learning curves for all programming plan KCs, from Fa19 to Fa22. For comparison, we also include the curve for AST node KCs from Rivers et al. [45]. Red line is the student data, blue dashed line is the regression curve fitted using AFM. Moderate model fit ($R^2 > 0.1$) is present in most semesters with small positive slopes (γ) with high variance, indicating that some plans may be successful KCs.

Our knowledge component model using programming plans has a $R^2 > 0.1$ in most semesters, indicating a partial fit. Moreover, we observe small positive learning rates with high standard deviation, pointing to a large difference between the learning rates of individual knowledge components in each semester. Thus, although plans are not sufficient as a complete domain model to explain programming skills, some particular programming plans may capture the development of discrete skills.

While the overall model fit metrics indicate that programming plans are not a complete domain model with high variance across semesters, we observe an improvement on an individual knowledge component basis compared to prior work. We obtain 70 learning

Table 3: Categories of learning curves for each plan across seven consecutive semesters. We observe the majority of learning curves to show learning (*Good* or *Still-high*), with only 4 out of 10 plans having at least one semester with *No-learning*. Repetition plans with concrete goals, such as `counting`, `filterACollection`, and `sum`, are among the best-performing plans with primarily *Good* curves.

KC Name	Fa19	Sp20	Fa20	Sp21	Fa21	Sp22	Fa22
counting	Good	Good	Good	Good	Still-high	Good	Good
filterACollection	Good	Good	Good	Good	Still-high	Still-high	Still-high
sum	Still-high	Still-high	Good	Good	Good	Good	Too little data
findBestInCollection	Still-high	Still-high	Good	Still-high	Still-high	Good	Still-high
evennessCheck	Too little data	Good	Good	Still-high	Still-high	Still-high	Still-high
processAllItems	Still-high	Still-high	Still-high	Good	Good	No-learning	No-learning
linearSearching	Too little data	Too little data	Too little data	Still-high	Still-high	Still-high	Still-high
multiWayBranching	Still-high	Still-high	Still-high	Still-high	No-learning	No-learning	Still-high
booleanOperatorChaining	Still-high	Still-high	Still-high	Still-high	No-learning	No-learning	Too little data
average	Too little data	Still-high	No-learning	Too little data	Too little data	Too little data	Too little data

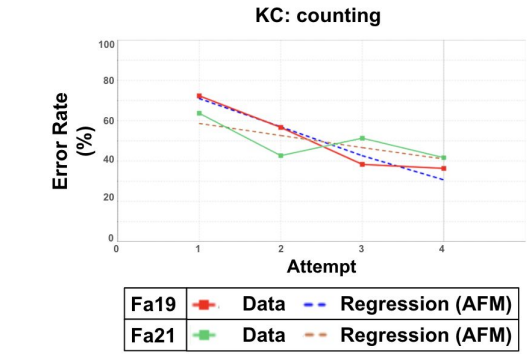
curves by analyzing 10 plans in each of seven semesters: of these, a majority of learning curves obtained across semesters were categorized as *Good* (20 out of 70, 28.6%) or *Still-high* (32 out of 70, 45.7%), indicating some degree of learning ($\gamma > 0$). In contrast, Rivers et al. [45]'s analysis with AST-based knowledge components resulted in single-digit percentages of *Good* (3 out of 48, 6.3%) and *Still-high* (2 out of 48, 4.2%) curves and Shi et al. [51]'s results using AST nodes also yielded a limited number of KC with positive learning rates (5 out of 21, 23.81%).

4.2 Evaluating programming plans based on learning curves

To understand which types of programming plans better model student learning, we evaluate individual programming plans by categorizing their learning curves observed across seven semesters. Table 3 shows the categories for plans in each semester according to the categorization from Table 2. We review these results in three parts: plans with *Good* curves in a majority of semesters, plans with a combination of *Good* and *Still-high* curves that still show learning in each semester, and plans with *No-learning* curves in some semesters. For all plans, we present two instructor solutions exercising the plans (from earlier and later points in the semester), example learning curves (from different categories of curves where possible), and learning rate γ for those curves.

We interpret semesters with *Good* and *Still-high* curves as evidence for a KC that accurately captures an underlying skill, as students' mastery increases on the skill as they get more opportunities to practice that skill ($\gamma > 0$). In particular, *Good* curves indicate both an accurate plan definition that matches the underlying skill and adequate opportunities for the skill to be mastered, whereas *Still-high* curves represent accurately defined plans that need more opportunities to reach mastery. On the other hand, a *No-learning* curve ($\gamma = 0$) can be associated with multiple outcomes. It may indicate that the knowledge component representing the plan does not directly correspond to a discrete skill, as the lack of decline in error rates suggests that students fail to transfer their knowledge between problems associated with the knowledge component. However, it can also indicate that students are failing to learn a skill due to a problem in instruction, or their learning is not captured within the range of the existing practice problems, especially as the course also includes other forms of practice activities (e.g. lectures, multiple choice questions).

4.2.1 Plans showing consistently good learning curves in all semesters. Three out of ten plans have *Good* learning curves in the majority of the semesters: `counting` (6 semesters out of 7), `filterACollection` (4/7), and `sum` (4/7).



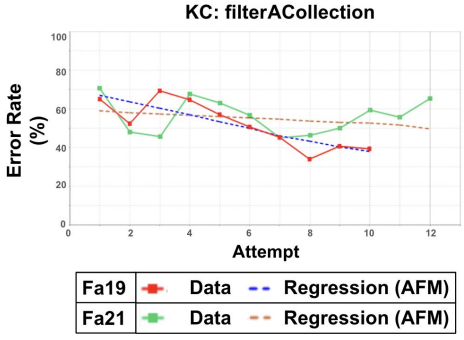

```
def count_over_100(number_list):
    count = 0
    for num in number_list:
        if num > 100:
            count += 1
    return count
```

```
def count_string_vals(a_dict):
    count = 0
    for val in a_dict.values():
        if type(val) == str:
            count +=1
    return count
```

Figure 4: Two examples of the `counting` plan from the instructor's solution set, with an earlier one (top) and a later one (bottom). Learning curves show two semesters: Fa19 (*Good*, $\gamma = 0.65$) and Fa21 (*Still-high*, $\gamma = 0.25$). Both semesters show a clear decrease in error rate across 4 attempts.

counting - With *Good* curves in 6 out of 7 semesters, we observed counting to be one of the most consistently learned plans. In Figure 4, instructor solutions and the learning curves from Fa19 (*Good*, $\gamma = 0.65$) and Fa21 (*Still-high*, $\gamma = 0.25$) are shown. We see that while students in Fa19 have a slope with a higher magnitude, indicating faster learning, both semesters show a decrease in the error rate.

filterACollection - Filtering a collection is another plan that iterates over a collection. In this plan, a new collection is created from the original collection with only the items that satisfy a certain condition. We notice that filterACollection learning curves tend to have a jump around attempt 3-4 (this is explored further in Section 4.4). Example solutions of filterACollection problems along with learning curves from Fa19 (*Good*, $\gamma = 0.15$) and Fa21 (*Still-high*, $\gamma = 0.03$) are presented in Figure 5.



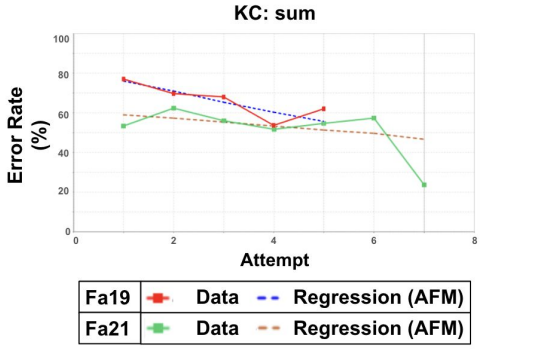

```
def get_subset(sdict, slist):
    nd = {}
    for key in slist:
        nd[key] = sdict[key]
    return nd
```

```
def select_strings_of_sufficient_length(string_list):
    select = []
    for s in string_list:
        words = s.split()
        if len(words) >= 5:
            select.append(s)
    return select
```

Figure 5: Two examples of the filterACollection plan from the instructor's solution set and learning curves from Fa19 (*Good*, $\gamma = 0.15$) and Fa21 (*Still-high*, $\gamma = 0.03$).

sum - Sum also shares a similar structure with the other two plans in iterating over items and comparing items on a criterion. Examples of sum and learning curves from Fa19 (*Still-high* $\gamma = 0.29$) and Fa21 (*Good*, $\gamma = 0.08$) are provided in Figure 6. While the *Still-high* curve has a greater learning rate, five attempts are not enough for the curve to be categorized as *Good*.

4.2.2 Plans showing promising curves with room for improvement in some semesters. We find three more plans with curves that do show a slight decrease in error rate. As these plans had fewer semesters with *Good* learning curves, we found it appropriate to examine these three separately from plans that have primarily *Good* learning curves, even though both categories indicate a decrease in error




```
def sum_lengths(value_list):
    s = 0
    for val in value_list:
        if type(val) == int or type(val) == float:
            s+=val
    return s
```

```
def sum_first_integers(num):
    s = 0
    for i in range(num+1):
        s+=i
    return s
```

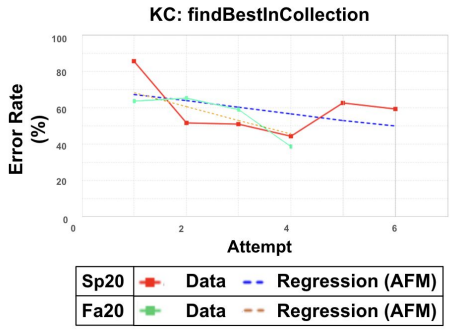
Figure 6: Two examples of the sum plan from the instructor's solution set and learning curves Fa19 (*Still-high*, $\gamma = 0.29$) and Fa21 (*Good*, $\gamma = 0.08$). The plan can include conditions, as shown at the bottom, or involve only summing a subset of a collection, as shown on the top.

rate. These three plans are findBestInCollection (5 semesters with *Still-high* curves out of 7), evennessCheck (4/7), and linearSearching (4/7).

findBestInCollection - This is also a repetition plan, but unlike filter, sum, or count which all produce a new value or sequence, the goal for findBestInCollection is to find an existing item that produces the most extreme value in some comparison function, such as the largest or the smallest of a set of numbers. Two example solutions are given in Figure 7 along with learning curves from Sp20 (*Still-high*, $\gamma = 0.17$) and Fa20 (*Good*, $\gamma = 0.36$).

evennessCheck - Compared to the plans covered so far, evennessCheck is a simpler plan with no high-level goal other than checking whether a number is even or odd. Thus, it is usually used in conjunction with another plan, as 26 of 32 instructor solutions that contain evennessCheck contain one more plan. Two examples of evennessCheck are given in Figure 8 with learning curves from Fa20 (*Good*, $\gamma = 0.45$) and Fa21 (*Still-high*, $\gamma = 0.21$).

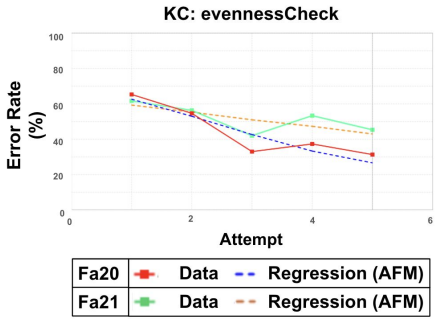
linearSearching - This is another repetition plan similar to findBestInCollection that compares all items to a criterion until an item satisfying the criteria is found. However, this plan is only present in five problems which results in it not having enough data in some semesters. Figure 9 shows two example solutions that use linearSearching with learning curves from Sp21 (*Still-high*, $\gamma = 0.15$). We only include a single learning curve as linearSearching curves are all categorized under *Still-high*.




```
def find_longest_in_data(data):
    ls = ""
    for s in data:
        if type(s) == str and len(s) > len(ls):
            ls = s
    return ls
```

```
def find_largest_csv_val(filename):
    largest = -10000
    fp = open(filename)
    for row in fp.readlines():
        for val in row.split(","):
            if int(val) > largest:
                largest = int(val)
    return largest
```

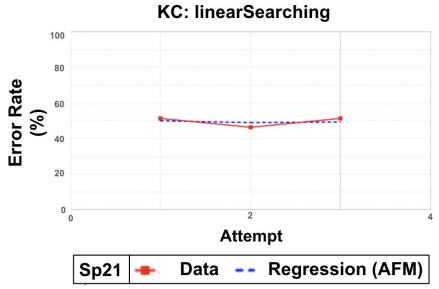
Figure 7: Examples of `findBestInCollection`, learning curves from Sp20 (*Still-high*, $\gamma = 0.17$), and Fa20 (*Good*, $\gamma = 0.36$).




```
def is_odd(input_number):
    if input_number % 2 == 1:
        return True
    return False
```

```
def count_even_integers(num_list):
    count = 0
    for num in num_list:
        if num % 2 == 0:
            count += 1
    return count
```

Figure 8: A problem only `evennessCheck` plan (top), and an example where `evennessCheck` and counting are used together (bottom), with learning curves from Fa20 (*Good*, $\gamma = 0.45$) and Fa21 (*Still-high*, $\gamma = 0.21$).




```
def extract_word_with_given_letter(sentence, letter):
    words = sentence.split()
    for word in words:
        if letter.lower() in word.lower():
            return word
```

```
def employee_lookup(departments, employee):
    for department in departments:
        dep = departments[department]
        if employee in dep:
            return dep[employee]
```

Figure 9: Two examples of the `linearSearching` plan from the instructor's solution set, and the Sp21 learning curve (*Still-high*, $\gamma = 0.15$).

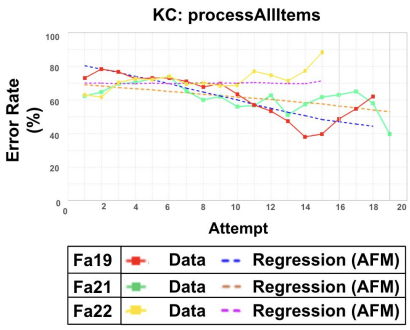
4.2.3 Plans showing mixed results across semesters. Four plans have at least one semester with a learning curve categorized as *No-learning*: `processAllItems` (2 semesters), `multiWayBranching` (2 semesters), `booleanOperatorChaining` (2 semesters), and `average` (1 semester). We exclude `average` from further analysis as it did not have enough data on a majority of semesters. Since *No-learning* indicates no improvement on the problems that include a knowledge component with more attempts, these plans may be too abstract to represent a plan or they might be combining multiple skills in a single knowledge component.

`processAllItems` - Another repetition plan, `processAllItems` is a generic iteration where an operation is performed on all items of a sequence. All other repetition plans can be considered as a special case of this plan. Compared to those special cases, which tend to perform well on the learning curve analysis, `processAllItems` is an outlier that may have an overly abstract definition.

`processAllItems` also has the most variance within its learning curves with three *Still-high* curves, two *Good* curves, and two *No-learning* curves. Two examples are presented in Figure 10 with learning curves from three semesters: Fa19 (*Still-high*, $\gamma = 0.12$), Fa21 (*Good*, $\gamma = 0.04$), and Fa22 (*No-learning*, $\gamma = 0$).

`booleanOperatorChaining` - This plan combines multiple comparisons in a single conditional expression. Due to its versatile definition, it is also usually used with other plans, with only 16 out of 21 usages having at least one more plan in the solution. Examples and learning curves from Fa20 (*Still-high*, $\gamma = 0.19$) and Fa21 (*No-learning*, $\gamma = 0$) are given in in Figure 11.

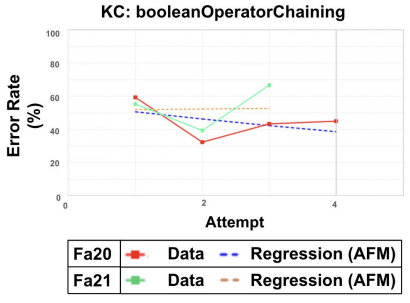
`multiWayBranching` - This conditional plan splits a program into three or more branches, based on a test condition. Due to its broad definition, it can be used for many different goals by changing the test condition and can be combined with other plans. Two examples are given in Figure 12 with learning curves from Fa20 (*Still-high*, $\gamma = 0.09$) and Fa21 (*No-learning*, $\gamma = 0$).




```
def print_dict_keys(dictionary):
    for key in dictionary:
        print(key)
```

```
def increase_elements_by_x(numbers, x):
    nl = []
    for num in numbers:
        nl.append(num + x)
    return nl
```

Figure 10: `processAllItems` examples and learning curves from Fa19 (*Still-high*, $\gamma = 0.12$), Fa21 (*Good*, $\gamma = 0.04$), and Fa22 (*No-learning*, $\gamma = 0$).



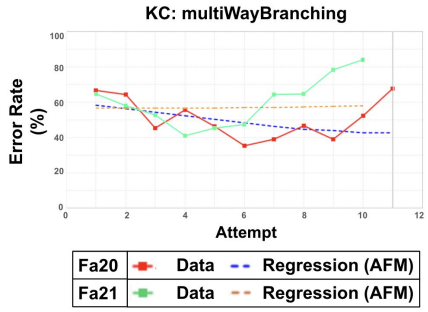

```
def is_a_single_letter(string_arg):
    if len(string_arg) == 1 and 'a' <= string_arg <=
'z':
        return True
    return False
```

```
def finals_room(last_name):
    if 'a' <= last_name[0].lower() <= 'g':
        return 'ROOM 100'
    elif 'h' <= last_name[0].lower() <= 'q':
        return 'ROOM 101'
    else:
        return 'ROOM 201'
```

Figure 11: Examples of `booleanOperatorChaining` with on its own (top), `booleanOperatorChaining` combined with `multiWayBranching` (bottom), and learning curves from Fa20 (*Still-high*, $\gamma = 0.19$) and Fa21 (*No-learning*, $\gamma = 0$).

4.3 Comparing programming plans with varying levels of abstraction

Based on our observations in Section 4.2, we hypothesize that *increased concreteness in plans improves their performance as a knowledge component.* To test this, we performed additional learning curve analysis with two different definitions for `processAllItems`:




```
def assign_grade(exam_score):
    if exam_score >= 90:
        return 'A'
    elif exam_score >= 80:
        return 'B'
    elif exam_score >= 70:
        return 'C'
    elif exam_score >= 60:
        return 'D'
    else:
        return 'F'
```

```
def matching_sign(first, second):
    if first > 0 and second > 0:
        return True
    elif first <= 0 and second <= 0:
        return True
    else:
        return False
```

Figure 12: Two `multiWayBranching` example solutions and learning curves from Fa20 (*Still-high*, $\gamma = 0.09$) and Fa21 (*No-learning*, $\gamma = 0$).

we first propose a counterfactual case by combining an existing plan into `processAllItems` to obtain a more abstract definition, and then we break `processAllItems` into plans that have more concrete definitions.

4.3.1 Counterfactual case: What if `filterACollection` was a part of `processAllItems`? `filterACollection` is not a plan that is widely established. Unlike plans like `sum` and `counting`, which have been commonly acknowledged as separate plans [14, 23, 25], many works either do not mention filtering as a plan [14] or consider it as an extension of other core plans [25]. Iyer and Zilles [23] also define `filterACollection` as a special case of `processAllItems`. Thus, we explore the plan design choice of moving from the abstract (`processAllItems` including filter problems) to concrete (`filterACollection` as a separate plan for filter problems).

Table 4 learning rate γ and the category for learning curves for both original plans and the abstract version `processWithFilter`. While the abstract model has *Still-high* learning curves in all semesters with $\gamma > 0$, we see that on average students learn the combined KC very slowly ($\bar{\gamma} = 0.03$). However, splitting `filterACollection` from `processAllItems` captures multiple *Good* semesters for `filterACollection` and increases the learning rate ($\bar{\gamma} = 0.08$), as well as identifying semesters where `processAllItems` shows *Good* learning or *No-learning*.

Table 4: Learning rate γ and learning curve categories for original plans, and the proposed abstract version: `processWithFilter`. Having a more abstract model hides information on semesters where some parts of the skill are being learned.

Plans	Fa19	Sp20	Fa20	Sp21	Fa21	Sp22	Fa22
				γ			
filterACollection	0.15	0.12	0.07	0.08	0.03	0.09	0.01
processAllItems	0.12	0.06	0.05	0.03	0.04	0.00	0.00
processWithFilter	0.07	0.04	0.03	0.03	0.03	0.00	0.03
				Category			
filterACollection	Good	Good	Good	Good	Still high	Still high	Still high
processAllItems	Still high	Still high	Still high	Good	Good	No learning	No learning
processWithFilter	Still high	Still high	Still high	Still high	Still high	Still high	Still high

Table 5: Learning rate γ and learning curve categories for `processAllItems` and proposed concrete versions. Some new plans, like `processDicts` and `processLists`, show better learning with a higher γ and more *Good* curves.

Plan	Fa19	Sp20	Fa20	Sp21	Fa21	Sp22	Fa22
				γ			
processLists	0.14	0.15	0.12	0.05	0.12	0.00	0.00
processRanges	0.35	0.00	0.00	0.07	0.00	0.00	0.00
processDicts	0.73	0.39	0.41	0.59	0.38	0.15	N/A
processAllItems	0.12	0.06	0.05	0.03	0.04	0.00	0.00
				Category			
processLists	Still high	Good	Good	Good	Good	No learning	No learning
processRanges	Still high	No learning	No learning	Still high	No learning	No learning	No learning
processDicts	Good	Good	Good	Still high	Still high	Still high	Too little data
processAllItems	Still high	Still high	Still high	Good	Good	No learning	No learning

We further compare the counterfactual KC model to the original model in Table 6, showing that having `filterACollection` separately results in a better model fit in all but one semester (Fa20), where the counterfactual model is better according to BIC whereas the original model has the better AIC value, even with the penalty from more parameters.

4.3.2 Case study: Could `processAllItems` *be split by type of iteration sequence?* Inspired by the promising results on plans with concrete definitions, we attempt to break `processAllItems` into even smaller plans, characterized by the data structure of the sequence being iterated. Based on our experience with the course, we suggest that students do not necessarily transfer their understanding of iterating over a list to iterating over a dictionary. Moreover, some data structures require the demonstration of additional skills not related to the core goal of the plan, e.g. using correct parameters for the *range* function or correctly reading from a file. Thus, we break `processAllItems` (found in 33 problems) into four plans: `processRanges` (10 problems), `processLists` (15), `processDicts` (4), and `processFiles` (4).

Table 5 presents the results for four split plans and for the original version. However, we observe a potential problem with more specified plan definitions: `processFiles` was not common enough in our homework assignments to have enough data for analysis, as less than 3 problems testing it were assigned in each semester. Yet, we also see a possible benefit: `processLists` ($\bar{\gamma} = 0.08$) and `processDicts` ($\bar{\gamma} = 0.44$) achieved higher learning rates compared to `processAllItems` ($\bar{\gamma} = 0.04$), and at least marginally improved in the number of semesters with *Good* learning curves, even in semesters where the combined `processAllItems` was *Still-high*. We also see `processRanges` as *No-learning* in many semesters.

Table 6 also evaluates the concrete in comparison to original and counterfactual KC models. We see that splitting `processAllItems` into multiple plans results in improved AIC/BIC metrics in all semesters even with a higher penalty resulting from increased parameter count, indicating this is a better model fit.

4.4 Towards identifying skills not captured by existing plans

Examining the performance of individual KCs may allow us to identify opportunities where existing, even *Good* plans may yet be decomposed to capture different skills. Previous literature on learning curves considers *blips*, which are sudden increases in error rates, to be signs of opportunities to improve KC models as these might indicate a problem testing an additional skill [61]. We explore

Table 6: AIC/BIC comparison for three models across all semesters. Best performing model in each semester is shown in bold. Increased concreteness improves model fit even with a higher parameter count.

		Fa19	Sp20	Fa20	Sp21	Fa21	Sp22	Fa22
AIC	Counterfactual	39,059.27	45,356.85	45,063.24	48,165.21	51,468.36	37,368.87	41,065.03
	Original	38,982.69	45,256.74	45,053.42	48,097.41	51,450.46	37,302.02	40,738.02
	Concrete	**38,745.65**	**44,631.11**	**44,563.54**	**47,955.89**	**51,019.09**	**36,880.72**	**40,496.95**
BIC	Counterfactual	44,534.67	50,552.92	50,704.38	54,321.77	57,825.57	42,065.16	46,168.38
	Original	44,474.85	50,469.77	50,711.47	54,270.97	57,824.80	42,014.82	45,858.11
	Concrete	**44,288.03**	**49,895.00**	**50,272.34**	**54,180.47**	**57,589.81**	**41,643.03**	**45,667.24**

this idea by considering the `filterACollection` pattern from our set of *Good* learning curves. For illustration, we focus on Fa19 curve for `filterACollection`, reproduced and labeled in Figure 13.

Despite being a *Good* curve, `filterACollection` has an unusual upward jump between the second and third opportunities before trending downwards again (see Figure 13). We investigated the difference between the problems measured at the second and third opportunity to see whether these may require different skills.

The questions from the first and second attempts (Label 1 on Figure 13) had relatively simple integer comparisons and were similar to each other. Investigating the questions from the third practice opportunity onward (Label 2 on Figure 13), presents an interesting find. It is at this point, where the learning curve lifts back upward in error rate, that questions start to use more complicated expressions in the conditional. These expressions contained more complex comparisons that required additional functions, such as the *str* methods like *str.split*, or the *type* function. These complex conditionals appear to increase the difficulty of `filterACollection` problems.

However, after this initial spike in error rates, the decrease in error continues steadily. Investigating an example problem from the following attempts gives insights into why this might be the case. The error rate for students who encounter *remove_nonstr* on their fourth attempt at `filterACollection` was 65%, compared to 72% for those who encounter it on their third attempt. The *type* function does not appear in other problems students encounter during their first five attempts on `filterACollection`. Therefore, it appears that students are transferring knowledge on modifying `filterACollection`, even when they have not used *type* before to modify it. In other words, students who have already combined `filterACollection` with a new structure (e.g. *str* methods) perform better when they need to combine `filterACollection` with another function (e.g. *type* function). This reduction in error rate on this specific item indicates that students learn how to modify this

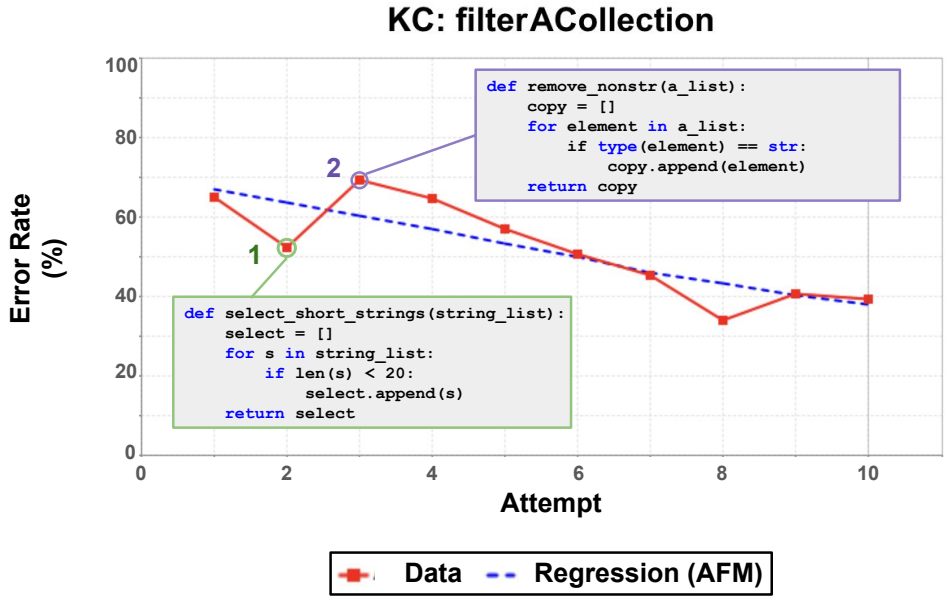


Figure 13: The learning curve for `filterACollection` from the Fa19 semester and the most common problems at the labeled attempts. (1) Before the jump in the learning curve, `filterACollection` problems only used simple operators such as <. (2) After the jump, solutions required more complex functions or operators, like *type()* or *str* methods.

particular plan regardless of the exact modification that needs to be incorporated, demonstrating why the error rates decrease from the third point onward. Therefore, while the plan represents a discrete skill as students still transfer their ability to apply the plan to some extent, variations in student performance can help identify when the plan requires multiple skills, such as tailoring to incorporate complex expressions in addition to implementing the key ideas.

5 DISCUSSION

5.1 RQ1: Does knowledge of programming plans explain skill development in introductory programming?

Our results from Section 4.1 show that Iyer and Zilles's set of programming plans is not a complete domain model that can explain skill development in programming. Our averaged learning curves for semesters (Figure 3) indicate that these programming plans show only a partial fit as a knowledge component model with moderate model fit for most semesters ($0.5 > R^2 > 0.1$). Moreover, the components are highly sensitive to external factors, as evidenced by the variance between and within semesters. This indicates that this set of programming plans, in its current form, is not a complete KC model, even on only the problems that require those plans.

However, a subset of programming plans can model student learning to some extent. We observe an improvement over previous applications of learning curve analysis to code-writing using AST nodes [45, 51] in the ratio of learning curves that capture learning. Moreover, we see large learning rates for some plans in our granular analysis in Section 4.2, such as `counting` and `sum`. These decreases in error rates indicate that students can transfer their knowledge between problems testing these plans. Thus, while programming plans as a whole are not a complete domain model, performance on individual programming plans may be useful for understanding student learning.

5.2 RQ2: Which introductory programming plans have the most evidence as discrete skills?

We find consistent evidence that some programming plans correspond to discrete skills that are transferred between problems. Plans that have mostly *Good* curves (Section 4.2.1) share some characteristics: they are repetition plans that iterate over a collection to produce a type of output (number of items, sum of items, or a filtered subset of items respectively) and have concrete descriptions that focus on goals explained in natural language, intended to solve a clearly defined task (e.g. selecting, counting). These concrete descriptions may limit the variety of problems students encounter, helping them to transfer their knowledge between problems. As reviewed in Section 4.2, these plans have either high enough learning rates to be mastered within a smaller number of attempts (e.g. `sum`, `counting`), or enough attempts to eventually go from *Still-high* to *Good* (e.g. `filterACollection`).

We also see promising results on other repetition plans with concrete goals in Section 4.2.2. Both `findBestInCollection` and `linearSearching` have similar characteristics to previously reviewed repetition plans from Section 4.2.1 and perform reasonably well. While `evennessCheck` also seems to perform on an acceptable level, we note that the learning curves for `evennessCheck` are likely to reflect student competency on other plans as well as this plan is usually implemented together with another plan.

Furthermore, we also observe similarities between plans with less successful learning curves in Section 4.2.3. In comparison to plans with decreasing learning curves from Sections 4.2.1 and 4.2.2, which focus on clear goals, these plans have abstract definitions that focus on language structures (e.g. conditionals, branches). Rather than being designed for accomplishing a concrete goal, these plans describe the execution process of the code: iterating over a collection, branching based on a condition, or combining multiple conditional expressions. This level of abstraction is reflected in the wide variety of problems that require the implementation of these plans, which makes transfer between problems harder and prevents a declining trend in error rates. Thus, our results indicate that forming plans with concrete definitions may better represent student cognition.

We note that there are several possible reasons for the lack of successful learning curves. Most importantly, we notice that *No-learning* semesters are consecutive, which may imply external factors that lead to these results. These effects may be caused by changes to instruction material, e.g. new homework problems introduced in the semester, which were rolled back in the following semesters as instructors observed the struggles of the students. While we did not analyze changes to the course at a fine level of detail, we do have the following context: Changes in the testing process in Sp22 (described in Section 3.1.1) may have had a negative impact with three of four plans having a *No-learning* category in that semester. Fa22 was the first semester where the class returned to fully in-person course meetings with one of two co-instructors brand new to the course, which may have come with instructional differences. Finally, curves that change from *Good* to *Still-high* in Sp-Fa21 may be due to the course's instructional staff swapping in experimental isomorphic questions on some homework, which may potentially be harder than previous questions, increasing the error rates. Thus, while our findings are insightful to show which plans are the most robust against such changes as shown by their consistently decreasing learning curves, the plans with mixed evidence may still produce good learning curves under a more constrained set of conditions.

5.3 RQ3: What design implications for improving programming plans can be inferred from learning curve analysis?

Our case studies from Section 4.3 show that re-defining certain iteration plans with increased concreteness can better represent student learning. For example, examining `filterACollection` as a separate plan compared to including it within a more abstract plan definition paints a more detailed picture of student improvement in the course. Namely, `filterACollection` has a *Good* learning curve in the first four semesters, but the decreasing error rate on questions testing this plan can only be observed when the plan is examined separately. Moreover, we only notice that students are not showing any learning on problems with `processAllItems` in the last two semesters when we separate filter problems from the rest. Thus,

the abstract model *hides* information by removing both positive results (good learning for `filterACollection` in four semesters) and results that may benefit from instructional intervention (no learning for `processAllItems` in two semesters).

The second case study also provides evidence for increasing the concreteness of plans by showing how it may be beneficial for understanding students' learning to isolate skills. Increased concreteness may help identify areas of success that are hidden in the combined KC (e.g. `processLists` and `processDicts` in Fa20), as well as areas for improvement that are hidden (e.g. `processRanges` and `processDicts` in Sp21).

However, the second case study also brings up an important consideration for learning curve analysis when increasing the concreteness of plans, which is to ensure that all plans have enough practice opportunities to measure student mastery accurately. We see this with `processFiles`, where there are not enough measurement opportunities in any semester to infer student learning. Similarly, `average` from Section 4.2.3 provides another example of a concrete definition without sufficient measurement leading to a failure in observing the KC in most semesters. Thus, the concreteness of patterns should be balanced with the number of problems testing each problem when incorporating pattern-based instruction in a class with limited practice opportunities.

This tradeoff of concreteness is also explored in Section 4.4, where we investigate skills that are not included explicitly in a plan definition through learning curve analysis. While `filterACollection` is a *Good* plan with a high learning rate in Fa19, our further exploration shows that students struggle with problems that introduce additional structures along with the core plan. Yet, it is still promising that students adapt to the use of these additional structures with more practice on the core plan after an initial increase in error rates. With our intuition about the course and based on the errors observed on these attempts, what causes the initial difficulty appears to be the idea of combining *any* additional structure with the plan, rather than the use of a *particular* structure. As students are practicing more, they may be also acquiring the skill of modifying their plan to diverse contexts.

6 ASSUMPTIONS

To apply learning curve analysis in a more complicated domain than usual, with open-ended code-writing data as opposed to the highly structured context of intelligent tutoring systems, we adopt a set of simplifying assumptions. We review these below so that readers can better evaluate the implications of our work.

A1. *The use of a programming plan can be modeled as a discrete skill independent of other programming knowledge that may be necessary to tailor the plan to a specific case.* Under this assumption, we consider any mistake in a student's solution as a mistake in applying their plan knowledge, including syntax errors or errors in elements that are not fundamental to the goal of the plan (e.g. use of the *type* function or *str* methods in `filterACollection` as shown in Section 4.4).

Two more assumptions follow from **A1**:

- **A1.1.** *The code from a student with a successful understanding of a plan should pass all test cases in the given question.* This is a strict metric as it requires students not only to have a mental model of the plan (*plan-schemata* from [26]), but also to be able to implement it in a syntactically correct way.
- **A1.2.** *Metrics that quantify the fit of this knowledge component model measure how easy it is to transfer the knowledge of a plan between problems.* Following from A1, we explicitly assume that applying a programming plan is a discrete skill, rather than the combination of many skills that are required to tailor it to different problems. Therefore, successful application of plan KCs across problems that measure those KCs indicates successful transfer of the student's knowledge of the plan under our assumptions.

Overall, our findings show that this simplifying assumption holds to some extent as we observe a number of programming plans with good learning curves, suggesting that the skill to apply these plans to different problems can be learned rather independently from other programming skills. However, the plans for which we failed to observe good learning curves may call for a more fine-grained approach to account for other skills required for correct implementations.

A2. *The first attempt made at a problem accurately represents a student's knowledge of the plan.* We adopt this assumption from Rivers et al. [45], who observed that the knowledge component model with only first attempts provides a better fit compared to using data from all attempts. They further state that traditional learning curve analysis uses the first attempt at a problem to avoid including attempts where students follow the feedback from the tutor without actually learning the material. However, since programming exercises require many skills to be practiced at once, it could be argued that students might miss a small detail or have a syntax error in their program on their first attempt, and this attempt might be incorrectly recorded as a lack of knowledge of the KC. To test this assumption, we did exploratory data analysis with data from Fall 2019, comparing the current first-attempt model to an alternative model that considers whether students get a problem correct within their first two attempts. While this approach reduced the error rates, learning curves for all plans were assigned under the same category in both models. Thus, we did not explore this alternative attempt model further. Still, future work may consider correctness models that incorporate information from multiple attempts to obtain a more nuanced view of student mastery.

A3. *Previously proposed thresholds for categorizing learning curves are applicable to open-ended programming problems.* We use thresholds suggested by DataShop, such as considering a final error rate of 40% as *Still-high*. As these thresholds are set in domains with questions that test more granular skills and may be simpler compared to open-ended programming problems, they might constitute an unrealistic expectation of student success. We still found these categories to provide an informative aggregation for big-picture takeaways, but it should be noted that the domain of programming may require refined thresholds to categorize these curves.

7 THREATS TO VALIDITY

As a consequence of all our assumptions described above, our approach is perhaps conservative in its ability to detect student learning of programming plans. One reason is that students' partial

knowledge of plans is not fully captured. In our model, successful understanding of plans is defined as a student choosing, tailoring, and successfully implementing the "correct" (i.e., the one the instructor used) plan in a syntactically correct way. Small errors, such as minor typos, would mark a student's use of a pattern as incorrect even if the rest of their implementation is flawless. Future work should examine splitting apart pattern selection, plan-composition, and plan implementation from unrelated errors when modeling KCs. Doing so requires answering the complex question of exactly what constitutes plan knowledge.

Secondly, there are multiple potential explanations for why any given KC may not be *Good* beyond the fact that it does not represent a discrete skill. It may be that students do not learn a given KC well due to instruction or it may be that a given KC is simply not measured enough to see learning or movement from a *Still-high* curve to *Good*. Moreover, as indicated by our third assumption, our predefined thresholds may be overly strict for the domain we are working in. If the error cutoffs were higher, we would see more curves categorized as *Good* instead of *Still-high*. Therefore, while we can present some KCs with evidence for validity and comment on the traits of what may make valid KCs, this work cannot identify specific traits about what may be completely inappropriate as KCs or provide a complete skill model.

Finally, assigning problems to specific programming plans was based on instructor solutions. However, there may have been multiple solutions to any given problem in the dataset, students may choose to implement different solutions to problems, and different sets of patterns might be applicable to the same question pool. Given this, there is a possibility for useful plan-based KCs that our work did not locate nor address.

Future work could yield a less conservative estimate of student learning by including a more granular tracking of plan-related skills, thresholds adapted to the domain of open-ended programming exercises, and a solution space that includes other plans that could be applied to solve each exercise. This could result in even better learning curves for a wider range of knowledge components.

8 IMPLICATIONS AND FUTURE WORK

Our findings suggest that students can successfully transfer their knowledge between problems that test more concrete plans. We observe successful transfer between problems of iteration plans with clear and concrete goals, such as `counting` and `sum`. Our work provides evidence of the cognitive validity of these plans as shown by students' ability to transfer their knowledge between problems. Students appear to grasp these concepts within a relatively small number of attempts, and instructors may want to provide fewer examples of these concrete plans.

However, our analysis failed to observe good learning curves for plans with more abstract definitions. For example, when students are presented with a more abstract definition like `processAllItems`, they seem to struggle with transfer. Our evidence suggests that students may learn to iterate over lists and dictionaries, but their ability to process items of these collections does not transfer into processing a collection generated by range. One potential explanation for this finding is that a more abstract goal like 'iterate over a collection' combines multiple tasks that actually require different skills, such as being able to set up the correct parameters for the *range* iterator in addition to the understanding of the loop structure. When we split `processAllItems` into more specific plans such as `processLists`, `processRanges` and `processDicts`, we limit the various skills required to solve problems testing a plan and better represent actual student skill development. These examples may be surprising to instructors who employ plans in their courses, as most works on the topic consider these to be part of the same plan [23, 25]. Thus, it is important that instructors should not assume transfer between plans at a higher level of abstraction (e.g. iterating over any collection) and consider explicitly presenting concrete versions of such plans (e.g. *filtering* the items of a *list*, iterating over a *range*).

Furthermore, we see that concretely defined plans also support tracking student performance more accurately. For example, students seem not to achieve a satisfactory level of performance when given problems from a plan with an abstract definition such as `processAllItems`, which may lead instructors to offer more practice opportunities on that plan. However, with a more fine-grained exploration, we see that students *do improve* on problems testing a subset of skills that are included in the abstract plan (e.g. `processLists`, `processDicts`). Learning curve analysis with knowledge components derived from more concrete plan definitions allows instructors to detect these cases of partial mastery. Such models can better guide the allocation of practice opportunities across topics and prevent instructors from assigning problems that test already mastered concepts.

We also observe that programming plan definitions may not capture some additional skills that are required to be learned for correctly implementing plans. Particularly, students seem to struggle when they need to modify plans by adding an additional structure for the first time, in line with previous findings on struggles with plan-composition (see Section 2.1). For example, we see that students experience difficulties when moving from filter problems with boolean conditions to more complex conditional checks (Section 4.4). It is perhaps unreasonable to claim that modifications to the boolean conditions used to filter are *alone* their own unique, nameable plan. However, just as with the different data structures in Section 4.3.2, there is an observable difference in performance introduced by the more complicated boolean conditions. A potential framing for such additional skills could be *extensions* of existing plans as they share many syntactic structures in implementation, but involve different goals [25]. Plan-based KCs might benefit from further augmentation to capture these extensions, either with details on how a plan is used (e.g. `filterACollection_basic` and `filterACollection_complex`, where the complexity of the filtering condition is considered) or by including plan labels along with some indicators mined from the code, such as information from ASTs. Thus, enhancing KC models based on expert knowledge with automatically detected components (i.e. AST nodes) is a promising research direction. These extensions could also motivate explicit instruction on *tailoring* [55] in addition to teaching plan definitions as a way of modifying existing plans to fit a more extensive set of problems.

9 CONCLUSION

In this work, we evaluated a set of previously proposed programming plans with learning curve analysis to understand the extent to which they correspond to discrete skills that can be transferred between problems. We found that many plans, especially iteration plans with concrete goals, are consistently learned as discrete skills in introductory programming, showing their validity as cognitive structures of novice programmers. However, not all plans in our dataset successfully captured student learning, with plans that are defined with abstract goals and described in terms of language structures yielding poor learning curves. Our work presents important implications for designing instruction around programming plans, as students may not actually learn plans that are at a higher level of abstraction. Our findings imply that plans should include concrete goals to be efficiently transferred across problems by novices. Moreover, we observe that learning curve analysis can model skill acquisition in open-ended programming problems, contrary to prior work on such exercises. This finding suggests that learning curve analysis with plan-based knowledge components can be used to refine instruction, track student improvement on learning how to use various plans, and potentially propose interventions where a skill is not being learned robustly. Applying learning curve analysis with even more accurate knowledge component models to detect complex behavior like tailoring, such as plans combined with automated components like ASTs, could improve support mechanisms for novice learners in code-writing exercises and expand our understanding of skill development in programming.

ACKNOWLEDGMENTS

We thank the University of Illinois Urbana-Champaign Computer Science Department for funding that contributed to this work. We are also grateful to our colleagues, Jinyoung (Jina) Hur, Yoshee Jain, David H. Smith IV, Vidushi Ojha, Victor Zhao, and Craig Zilles for their feedback. We would also like to thank the reviewers for their valuable suggestions.

REFERENCES

[1] Vincent Aleven and Kenneth R. Koedinger. 2013. Knowledge Component (KC) Approaches to Learner Modeling. *Design recommendations for intelligent tutoring systems* 1 (2013), 165–182.

[2] Isaac Alpizar-Chacon, Sergey Sosnovsky, and Peter Brusilovsky. 2023. Measuring the Quality of Domain Models Extracted from Textbooks with Learning Curves Analysis. In *Artificial Intelligence in Education*, Ning Wang, Genaro Rebolledo-Mendez, Noboru Matsuda, Olga C. Santos, and Vania Dimitrova (Eds.). Springer Nature Switzerland, Cham, 804–809. https://doi.org/10.1007/978-3-031-36272-9_75

[3] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray. Pelletier. 1995. Cognitive Tutors: Lessons Learned. *Journal of the Learning Sciences* 4, 2 (April 1995), 167–207. https://doi.org/10.1207/s15327809jls0402_2

[4] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. 1998. Design Patterns: An Essential Component of CS Curricula. In *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98)*. Association for Computing Machinery, New York, NY, USA, 153–160. https://doi.org/10.1145/273133.273182

[5] Owen Astrachan and Eugene Wallingford. 1998. Loop Patterns. In *Proceedings of the Fifth Pattern Languages of Programs Conference*.

[6] Robert L Bangert-Drowns, James A Kulik, and Chen-Lin C Kulik. 1991. Effects of frequent classroom testing. *The journal of educational research* 85, 2 (1991), 89–99.

[7] Joseph Bergin. 1999. Patterns for Selection. https://csis.pace.edu/~bergin/patterns/Patternsv4.html.

[8] Paul D. Bliese, David Chan, and Robert E. Ployhart. 2007. Multilevel Methods: Future Directions in Measurement, Longitudinal Analyses, and Nonnormal Outcomes. *Organizational Research Methods* 10, 4 (2007), 551–563. https://doi.org/10.1177/1094428107301102

[9] John Bransford, Ted Hasselbring, Brigid Barron, Stan Kulewicz, Joan Littlefield, and Laura Goin. 1988. Uses of macro-contexts to facilitate mathematical thinking. *The teaching and assessing of mathematical problem solving* 3 (1988), 125–147.

[10] Hao Cen, Kenneth Koedinger, and Brian Junker. 2006. Learning Factors Analysis – A General Method for Cognitive Model Evaluation and Improvement. In *Intelligent Tutoring Systems (Lecture Notes in Computer Science)*, Mitsuru Ikeda, Kevin D. Ashley, and Tak-Wai Chan (Eds.). Springer, Berlin, Heidelberg, 164–175. https://doi.org/10.1007/11774303_17

[11] Michelene T. H. Chi, Robert E. Glaser, and Marshall J. Farr. 1988. The Nature of Expertise. https://api.semanticscholar.org/CorpusID:143028015

[12] Michael J Clancy. 1996. *Designing Pascal Solutions: Case studies using data structures*. WH Freeman & Co.

[13] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–15. https://doi.org/10.1145/3411764.3445571

[14] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the 11th Australasian Computing Education Conference (ACE 2009)*, Margaret Hamilton and Tony Clear (Eds.), Vol. 95. Bedford Park, South Australia, 45–54.

[15] J. Philip East, S. Rebecca Thomas, Eugene Wallingford, Walter Beck, and Janet Drake. 1996. Pattern Based Programming Instruction. In *1996 Annual Conference*. 1.349.1–1.349.10.

[16] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, Memphis Tennessee USA, 211–216. https://doi.org/10.1145/2839509.2844556

[17] Max Fowler and Craig Zilles. 2021. Superficial Code-Guise: Investigating the Impact of Surface Feature Changes on Students' Programming Question Scores. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 3–9. https://doi.org/10.1145/3408877.3432413

[18] D. J. Gilmore and T. R. G. Green. 1988. Programming Plans and Programming Expertise. *The Quarterly Journal of Experimental Psychology Section A* 40, 3 (Aug. 1988), 423–442. https://doi.org/10.1080/02724988843000005

[19] David Ginat. 2009. Interleaved pattern composition and scaffolded learning. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09)*. Association for Computing Machinery, New York, NY, USA, 109–113. https://doi.org/10.1145/1562877.1562915

[20] David Ginat, Eti Menashe, and Amal Taya. 2013. Novice Difficulties with Interleaved Pattern Composition. In *Informatics in Schools. Sustainable Informatics Education for Pupils of all Ages*, Ira Diethelm and Roland T. Mittermeir (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–67.

[21] B.G. Glaser. 1992. *Emergence Vs Forcing: Basics of Grounded Theory Analysis*. Sociology Press. https://books.google.com/books?id=1ZJiQgAACAAJ

[22] Cyril Goutte and Guillaume Durand. 2020. Confident Learning Curves in Additive Factors Modeling. *International Educational Data Mining Society* (2020).

[23] Vighnesh Iyer and Craig Zilles. 2021. Pattern Census: A Characterization of Pattern Usage in Early Programming Courses. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 45–51. https://doi.org/10.1145/3408877.3432442

[24] Vighnesh Narayanan Iyer. 2020. *An Analysis of Pattern Usage in CS1 Courses*. Thesis. University of Illinois at Urbana-Champaign.

[25] Cruz Izu, Violetta Lonati, Anna Morpurgo, and Mario Sanchez. 2021. An Inventory of Goals from CS1 Programs Processing a Data Series. In *2021 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE49875.2021.9637360

[26] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education (TOCE)* 22, 2 (2021), 1–33.

[27] Kenneth R. Koedinger, Ryan Sjd Baker, Kyle Cunningham, Alida Skogsholm, Brett Leber, and John Stamper. 2010. A Data Repository for the EDM Community: The PSLC DataShop. *Handbook of educational data mining* 43 (2010), 43–56.

[28] Kenneth R. Koedinger, Albert T. Corbett, and Charles Perfetti. 2012. The Knowledge-Learning-Instruction Framework: Bridging the Science-Practice Chasm to Enhance Robust Student Learning. *Cognitive Science* 36, 5 (2012), 757–798. https://doi.org/10.1111/j.1551-6709.2012.01245.x

[29] Kenneth R. Koedinger, Elizabeth A. McLaughlin, and John C. Stamper. 2012. *Automated Student Model Improvement*. Technical Report. International Educational Data Mining Society.

[30] Kenneth R. Koedinger, John C. Stamper, Elizabeth A. McLaughlin, and Tristan Nixon. 2013. Using Data-Driven Discovery of Better Student Models to Improve

Student Learning. In *Artificial Intelligence in Education*, H. Chad Lane, Kalina Yacef, Jack Mostow, and Philip Pavlik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 421–430.

[31] Klaus Krippendorff. 2011. Computing Krippendorff's Alpha-Reliability. 43 (Jan. 2011). https://repository.upenn.edu/handle/20.500.14332/2089

[32] Marja Kuittinen and Jorma Sajaniemi. 2004. Teaching roles of variables in elementary programming courses. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE '04)*. Association for Computing Machinery, New York, NY, USA, 57–61. https://doi.org/10.1145/1007996.1008014

[33] Filip Lievens and Paul R. Sackett. 2007. Situational judgment tests in high-stakes settings: Issues and strategies with generating alternate forms. *Journal of Applied Psychology* 92 (2007), 1043–1055. https://doi.org/10.1037/0021-9010.92.4.1043

[34] Brent Martin, Antonija Mitrovic, Kenneth R. Koedinger, and Santosh Mathan. 2011. Evaluating and Improving Adaptive Educational Systems with Learning Curves. *User Modeling and User-Adapted Interaction* 21, 3 (Aug. 2011), 249–283. https://doi.org/10.1007/s11257-010-9084-2

[35] VT Mawhinney, DE Bostow, DR Laws, GJ Blumenfeld, and BL Hopkins. 1971. A comparison of students studying-behavior produced by daily, weekly, and three-week testing schedules. *Journal of Applied Behavior Analysis* 4, 4 (1971), 257–264.

[36] Orna Muller. 2005. Pattern Oriented Instruction and the Enhancement of Analogical Reasoning. In *Proceedings of the First International Workshop on Computing Education Research (ICER '05)*. Association for Computing Machinery, New York, NY, USA, 57–67. https://doi.org/10.1145/1089786.1089792

[37] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-Oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07)*. Association for Computing Machinery, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[38] Mitchell J. Nathan, Kenneth R. Koedinger, and Martha W. Alibali. 2001. Expert Blind Spot: When Content Knowledge Eclipses Pedagogical Content Knowledge. In *Proceedings of the Third International Conference on Cognitive Science*, Vol. 644648. 644–648.

[39] Allen Newell and Paul S. Rosenbloom. 2013. Mechanisms of Skill Acquisition and the Law of Practice. In *Cognitive Skills and Their Acquisition*. Psychology Press, 1–55.

[40] Huy Nguyen, Yeyu Wang, John Stamper, and Bruce M. McLaren. 2019. *Using Knowledge Component Modeling to Increase Domain Understanding in a Digital Learning Game*. Technical Report. International Educational Data Mining Society.

[41] Peterson K Ozili. 2023. The acceptable R-square in empirical modelling for social science research. In *Social research methodology and publishing results*. IGI global, 134–143.

[42] Rebecca Passonneau. 2006. Measuring Agreement on Set-valued Items (MASI) for Semantic and Pragmatic Annotation. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Nicoletta Calzolari, Khalid Choukri, Aldo Gangemi, Bente Maegaard, Joseph Mariani, Jan Odijk, and Daniel Tapias (Eds.). European Language Resources Association (ELRA), Genoa, Italy.

[43] Jihyun Rho, Martina Rau, and Barry Vanveen. 2022. Investigating Growth of Representational Competencies by Knowledge-Component Model. In *Proceedings of the 15th International Conference on Educational Data Mining*. 346. https://doi.org/10.5281/zenodo.6853077

[44] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* 13, 3 (July 1989), 389–414. https://doi.org/10.1016/0364-0213(89)90018-9

[45] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming: Which Concepts Do Students Struggle With?. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 143–151. https://doi.org/10.1145/2960310.2960333

[46] Anthony V. Robins. 2019. Novice Programmers and Introductory Programming. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V.Editors Robins (Eds.). Cambridge University Press, 327–376. https://doi.org/10.1017/9781108654555.013

[47] Anthony V. Robins, Lauren E. Margulieux, and Briana B. Morrison. 2019. Cognitive Sciences for Computing Education. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V.Editors Robins (Eds.). Cambridge University Press, 231–275. https://doi.org/10.1017/9781108654555.010

[48] Donna Sabers, Katherine Cushing, and David Berliner. 1991. Differences Among Teachers in a Task Characterized by Simultaneity, Multidimensional, and Immediacy. *American Educational Research Journal - AMER EDUC RES J* 28 (03 1991), 63–88. https://doi.org/10.3102/00028312028001063

[49] J. Sajaniemi. 2002. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. 37–39. https://doi.org/10.1109/HCC.2002.1046340

[50] Jorma Sajaniemi and Marja Kuittinen. 2005. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education* 15, 1 (2005), 59–82. https://doi.org/10.1080/08993400500056563

[51] Yang Shi, Robin Schmucker, Min Chi, Tiffany Barnes, and Thomas Price. 2023. KC-Finder: Automated Knowledge Component Discovery for Programming Problems. *International Educational Data Mining Society* (2023).

[52] Herbert A. Simon. 1980. *Problem solving and education: Issues in teaching and research*. Erlbaum, Hillsdale, NJ, Chapter 6, 81–96.

[53] Mark K. Singley and John R. Anderson. 1989. The Transfer of Cognitive Skill. https://api.semanticscholar.org/CorpusID:60731940

[54] Elliot Soloway. 1985. From Problems to Programs via Plans: The Content and Structure of Knowledge for Introductory LISP Programming. *Journal of Educational Computing Research* 1, 2 (May 1985), 157–172. https://doi.org/10.2190/WK8C-BYCF-VQ5C-E307

[55] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594

[56] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (Sept. 1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[57] Elliot M. Soloway and Beverly Woolf. 1980. Problems, Plans, and Programs. *ACM SIGCSE Bulletin* 12, 1 (Feb. 1980), 16–24. https://doi.org/10.1145/953032.804605

[58] Juha Sorva, Ville Karavirta, and Ari Korhonen. 2007. Roles of variables in teaching. *Journal of Information Technology Education: Research* 6, 1 (2007), 407–423.

[59] James C Spohrer, Elliot Soloway, and Edgar Pope. 1985. A Goal/Plan Analysis of Buggy Pascal Programs. *Human–Computer Interaction* 1, 2 (June 1985), 163–207. https://doi.org/10.1207/s15327051hci0102_4

[60] James G. Spohrer and Elliot Soloway. 1986. Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers Workshop*, Elliot Soloway and S. Iyengar (Eds.). Ablex, 230–251.

[61] John C. Stamper and Kenneth R. Koedinger. 2011. Human-Machine Student Model Discovery and Improvement Using DataShop. In *Artificial Intelligence in Education*, Gautam Biswas, Susan Bull, Judy Kay, and Antonija Mitrovic (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–360.

[62] Kurt VanLehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.

[63] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. 26–1238.