# Reexamining Learning Curve Analysis in Programming Education: The Value of Many Small Problems

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

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

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

## ABSTRACT

Analyzing which skills students develop in introductory programming education is an important question for the computer science education community. These key skills and concepts have been formalized as knowledge components, which are units of knowledge that can be measured by performance on a set of tasks. While knowledge components in other domains have been successfully identified using learning curve analysis, such attempts on students' open-ended code-writing assignments have not been very successful. To understand why, we replicated a previously proposed approach, which uses abstract syntax tree (AST) nodes as knowledge components, on data collected across multiple semesters of a large-scale introductory programming course. Findings from our replication show that, given sufficient measurement opportunities, a significant subset of AST nodes provide a viable knowledge component model for learning curve analysis to understand student learning, contrary to earlier findings. In addition to providing evidence for the validity of certain AST-based knowledge components, we recommend a set of conditions for programming courses that may enable knowledge components generated using AST nodes to be successfully observed using learning curve analysis. Our findings suggest that learning curve analysis can yield useful insight for instructors on skills related to language elements, and can be integrated into any environment that collects code-writing data using our step generation method.

## Keywords

knowledge components, learning curve analysis, programming syntax, computing education

## 1. INTRODUCTION

Understanding the skill development of students who are learning to write programs is a key challenge in the field of computer science education [7, 23, 32, 34, 37, 19, 10]. Learning curve analysis offers a methodology to empirically validate domain models that describe such skills [15], however, attempts to apply learning curve analysis in the context of programming education have yielded few results so far. While programming education is rich in data collected during code-writing, thanks to numerous learning environments that capture and automatically grade student program submissions [8, 17, 36], applications of learning curve analysis on such data have produced a limited number of validated knowledge components [30], or knowledge components that are difficult to interpret [33].

Applying learning curve analysis on code-writing data is a difficult problem. Unlike the highly constrained and focused problems typical of intelligent tutoring systems, where learning curve analysis is most often applied [16], programming assignments are open-ended, leaving learners many choices to make at once. Such problem types prevent an easy way to define independent attempts at practicing a knowledge component (KC). In addition to the difficulty of identifying independent steps, the open-ended form of programming assignments means that not all correct submissions use the same surface-level features as the canonical solution. The open-ended nature of these questions also motivate automated techniques for identifying KCs, as common techniques such as having expert annotations for skills practiced in each question would become time-consuming and infeasible. Thus, traditional KC modeling methods must be modified in order to perform learning curve analysis with programming data.

Rivers et al. [30] attempted to apply learning curve analysis to programming data by adapting the definition of a step and step correctness for the context of student code submissions. They tested a knowledge component model of students' ability to use appropriate language features, which was implemented by tracking the relevant abstract syntax tree (AST) nodes in student programs. This enabled them to efficiently and automatically detect KCs in each submission, although not all KCs may perfectly align with an expert's domain model. They hypothesized that using syntax elements correctly is a foundational skill students need to solve programming assignments, making them an appropriate model of KCs. However, their analysis resulted in very few satisfactory learning curves that showed a decrease in error rate across attempts. They questioned whether learning curve analysis could be applied to open-ended code-writing data and whether their choice of the domain model (syntax structures) was representing student skill development.

M. A. Demirtas, M. Fowler, and K. Cunningham. Reexamining learning curve analysis in programming education: The value of many small problems. In B. Paaßen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 53–67, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society.


https://doi.org/10.5281/zenodo.12729774

We perform a replication of Rivers et al.'s work in order to address both concerns. We hypothesize that Rivers et al.'s attempt to model knowledge components was limited by the nature of their data, which were collected over a single semester and included data from relatively few students and problems. The limited number of problems may not have provided enough clarity about student skill development, which may be particularly important in the context of open-ended programming problems where a single "step" (a whole program submission) represents a large number of KCs. The small number of students may have simply led their earlier analysis to be underpowered. As a result, the conclusion that syntax elements are an inappropriate or "naive" [33] domain model for learning of introductory programming may be premature.

In this work, we replicated the learning curve analysis from Rivers et al. [30], using the same domain model, on a new data set containing over twice as many problems, over four times as many students, and data collected across an entire introductory programming course from seven different semesters. We answer the following questions:

- **RQ1:** Can student progression in introductory programming be modeled with language features as knowledge components using learning curve analysis?
- **RQ2:** Which properties of a course can affect the success of this learning curve analysis?
- **RQ3:** Which language elements may be best identified as knowledge components using AST nodes?

Our findings suggest that, in fact, a domain model focused on students' use of appropriate syntax structures *can* capture student learning in introductory programming to a considerable extent, contrary to previous findings. We further observe that students from different semesters exhibit consistent performance on many of the identified KCs, providing further evidence for the validity of this KC model. We provide suggestive evidence that having a problem set where students have a larger number of attempts for showing their mastery of each KC is one of the major reasons for this finding. Moreover, these improvements are present even in small samples with fewer students than prior works, indicating that learning curve analysis with AST nodes can be applied in any programming classroom that provides a significant amount of practice opportunities.

# 2. RELATED WORK

## 2.1 Language elements as a domain model for programming

A commonly proposed set of foundational skills in introductory programming is the ability to use each of the various programming language elements to write code (e.g., to use an *if statement* correctly, to use a *for loop* correctly). This approach has intuitive appeal: all programs are made up of these various syntax structures, so learners must apply them in some way. Most computing education material also follows this principle by organizing textbook chapters and grouping exercises to cover individual programming language elements (e.g. if statements, for loops). Frameworks for categorizing language-independent concepts in computing education, such as FCS1 [34], include many topics that overlap with abstract syntax tree nodes such as variables, logical operators and definite loops (for).

Furthermore, a domain model based on language features reduces the need for annotation from experts with domain knowledge as they can be automatically extracted, making it a feasible model to apply in even small classrooms without putting an extra load on the instructor. Language features in student submissions, as represented by abstract syntax tree (AST) nodes, have been automatically extracted for providing scaffolding in programming exercises [31, 28] and predicting student success on further exercises as a proxy of mastery of the concept [35]. If language features, as represented by AST nodes, can be validated as a domain model, systems that automatically extract these nodes can be used to model student learning.

## 2.2 Modeling with knowledge components

Knowledge components (KC) are defined by Koedinger et al. [15] as acquired units of skill or knowledge of that can be measured using a set of tasks. KCs are used as a way of representing the mental abilities required to complete unit tasks in a domain. A successful KC model captures the required skills and abilities in a domain and supports data-driven instructional decisions [15].

As knowledge components are measured using performance on a set of tasks, they constitute a framework for empirically validating domain models. A common method for this validation is learning curve analysis [6]. Learning curve analysis is based on the power law of practice, stating that students will achieve a lower error rate in problems related to a KC as they have more attempts to practice the skills represented by that KC [24]. Learning curves have been shown to be a viable method for improving domain models [20], and platforms implementing analysis tools for knowledge components have been successfully used to identify KCs in algebra, physics, geometry, and language studies using learning curves [16, 13]. Nguyen et al. used KC models derived from expert knowledge to modify mathematics exercises to address common difficulties [25]. Goutte et al. proposed a method for tracking skill acquisition in students and applied it on geometry tutoring to categorize the difficulty of skills [12]. Rho et al. used KC models for representational competencies in engineering education, showing that the dynamic development of student ability can be tracked with KC modeling [29]. While valid KC models for many domains exist, a comprehensive KC model for programming has not yet been identified.

## 2.3 KC models for programming data

KC-based learner models have been used in early tutoring systems in programming environments, such as the LISP tutor [3], for individualized problem selection [1]. Recently, Alpizar-Chacon et al. used concepts extracted from textbooks to manually annotate code reading exercises as a possible domain model with promising results [2]. However, these KC models were time-consuming and limited in scalability as they required manual annotation of all problems by an expert. Moreover, they were not compatible with open-ended programming assignments, which require students to

show mastery of multiple skills at once to complete.

Rivers et al. [30] was the first work that applied learning curve analysis on code-writing data from open-ended programming assignments to empirically validate an automatically extracted KC model for programming. Their data was collected from 40 introductory programming practice problems and submissions from 89 students. The goal of their analysis was to identify which language concepts were more challenging for students by using AST nodes in program submissions as a KC model. Since AST nodes could be automatically extracted from a given program, this model could be scaled to larger classrooms more easily and recognize solutions partially correct implementations. However, they failed to observe learning curves that show a decrease in error rate for a large portion of AST nodes and attributed it to inadequate KC modeling.

Shi et al. [33] used a neural network to generate knowledge components from scratch. They attempted to interpret them afterwards by reviewing the student submissions including identified KCs. They tested their model using submissions from 410 students on 50 programming problems. While their KCs show the expected learning curves by construction, they cannot be interpreted as easily as a KC model based on syntax elements. In particular, it weakens their utility for informing curricular decisions, such as focusing on skills that are more difficult to students.

Our replication addresses the limitations of these two studies by applying the interpretable AST-based KC model on a dataset to more students and more problems to analyze which conditions can lead to a better evaluation of language features as a domain model.

# 3. METHOD

We replicate the learning curve analysis from Rivers et al. [30] on a dataset of programming problem submissions that was collected in classrooms with more students, includes more questions, and has submissions from the entirety of multiple semesters. Through this replication, we revisit the question of whether the ability to use syntax structures correctly is an appropriate model of the skills acquired when learning to program, and further, examine the conditions under which this skill model is validated by learning curve analysis.

In this section, we first review the method of Rivers et al. [30] for preparing code-writing data to be used in learning curve analysis where usage of particular syntax structures serve as knowledge components. Then, we highlight the differences between our implementation and the original work. Finally, we explain the properties of our dataset that differentiate our analysis from prior work.

## 3.1 Step generation for code-writing data

Writing code is an important learning activity in introductory programming, but code-writing data presents unique challenges for learning curve analysis. In most domains, items that are used to assess knowledge components consist of atomic steps that test a single KC at a time, such as entering a value in one short-answer box for a subpart of a problem in an intelligent tutoring system. This allows a straightforward way of analyzing attempts made for each knowledge component, e.g. by extracting the *steps* a student takes while interacting with an intelligent tutoring system. However, such systems are less common in programming learning. Instead, students are expected to practice many skills at once in each submission of a code-writing assignment. This requires an alternative method to break the submission into smaller steps, where each step represents an attempt the student made at exercising a KC, and the outcome of the attempt at that KC (whether the KC is used correctly or not). A step generation algorithm that breaks assignment submissions into steps compatible with learning curves is required to track each KC independently in open-form programming assignments.

In this analysis, we follow the step generation algorithm proposed by Rivers et al. [30]. Figure 1 shows the overall process for step generation. The algorithm starts by computing the AST of the student submission, and ASTs of all the student submissions that passed all test cases and the instructor solution, using Python's AST library (Figure 1.a). While there might be a large number of correct submissions, this process maps all solutions that have the same syntax structure to a single AST, reducing the size of the solution space. By including correct student submissions in addition to the instructor solution, we end up with a set of ASTs that capture different valid implementations that satisfy the test cases for the given problem. Moreover, this method can be used without an instructor solution to automatically annotate KCs required in a programming problem by only using the set of student submissions that pass the test cases.

Then, we compare the student submission AST to the set of correct ASTs to find the solution that is most similar to what the student submitted (Figure 1.b). Ideally, we are trying to find the implementation the student envisions, even if their program is not complete at the time of submission. In this step, Rivers et al. use the ITAP algorithm [31] to search a solution space and find the correct solution that is most similar to the student submission. The ITAP algorithm computes the set of edits to transform the student submission into the correct solution, and the correct solution with the least number of required edits is selected as the most similar.

In our implementation, we followed a different method for identifying the most similar solution as we did not have access to the ITAP algorithm. Instead, we compared the student submission AST to the correct ASTs in the solution space using TF-IDF distance, as proposed by Mokbel et al. [22]. If the student submission could not be parsed, an empty AST is used as a substitute. For the distance calculation, each AST is represented by a vector, containing frequencies of each node type in the tree. Then, by comparing the vectors, we find the correct solution that has the most similar distribution of nodes to the student submission.

After identifying the correct solution that is most similar to the student submission, ASTs from the student submission and the correct solution are compared using depth-first search. Both trees are traversed starting from the root node, which is the same AST node in all trees by construction. If trees diverge at a node, all nodes traversed so far are considered correctly used. Nodes from the subtree rooted at the

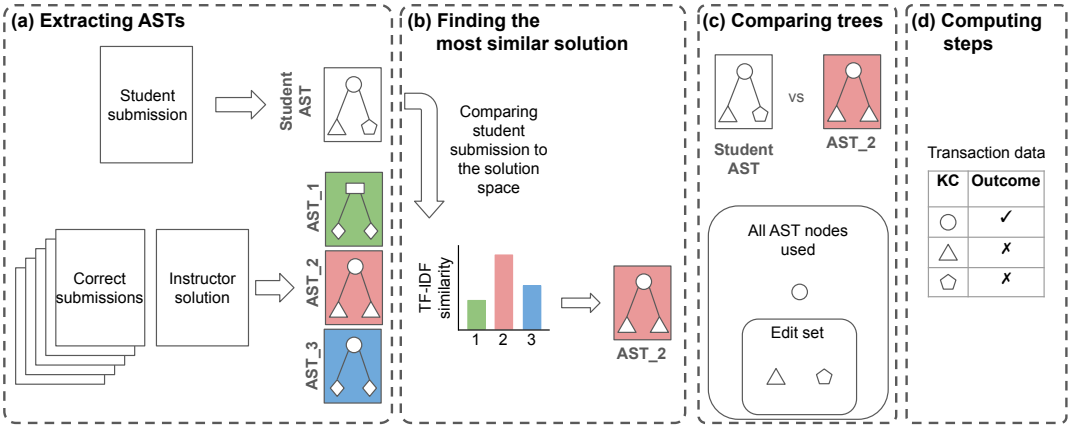


**Figure 1: Step generation algorithm from code-writing submissions. (a) First, abstract syntax trees (AST) for the student submission, all correct submissions for the given problem, and the instructor solution are extracted. (b) Second, the student submission AST is compared against all the correct ASTs using the TF-IDF metric, identifying the most similar correct solution. (c) Then, the ASTs from the student submission and the identified correct solution are compared to form an edit set of the AST nodes that have to be inserted or removed in order to correct the student submission. (d) Finally, incorrectly used KCs (nodes from edit set) and correctly used KCs (rest of the nodes) are recorded as the transaction data.**

divergent node are added to an *edit set*: nodes that need to be either inserted in or removed from the student submission to reach the AST of the correct solution (Figure 1.c).

We identify the outcome of a step using the modified step generation rules suggested in Rivers et al. [30]:

- Attempt at KC is classified as INCORRECT if the node occurs in the edit set
- Attempt at KC is classified as CORRECT if the node is included in the student submission and not in the edit set
- Other KCs are skipped

Using this criteria, we create a transaction table that includes a separate step for each KC included in the submission, or missing in the submission and present in the edit set (Figure 1.d). By combining transaction tables from student submissions to all of the problems, we obtain steps representing student attempts made at all of the KCs, across all homeworks. Note that we only use data from the first submission of each student for a given problem, as suggested by Rivers et al. [31]. This eliminates the the impact of improvements made by following hints or other feedback from the learning environment (e.g., results of test cases), which may not necessarily demonstrate mastery of the underlying knowledge components.[1]

## 3.2 Additive factor modeling for learning curve analysis

After generating the transaction data, consisting of each KC associated with each problem and whether it is correctly used or not in a given attempt by a student, learning curves were generated on DataShop [14]. DataShop is an online data repository that allows researchers to upload interaction data and compute learning curves. Each learning curve plots the average error rate of students at each attempt at practicing a knowledge component, as shown in Figure 2. DataShop also fits a curve that represents the overall trend in the learning curve using additive factor modeling (AFM). We opted to use DataShop instead of implementing our own AFM algorithm to avoid introducing differences in model fitting compared to the analysis in Rivers et al. [30].

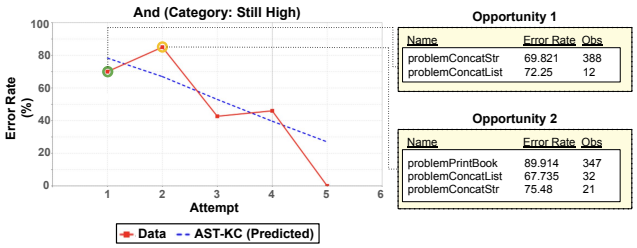


**Figure 2: An example learning curve from DataShop. The learning curve generated from interaction data (red) and the fitted AFM curve (blue) is plotted. This curve is categorized as 'Still high' based on the AFM values (shown at top). On the right, information about the data used to generate the first two points on the curve is shown, including a list of problems testing this KC and average error rate on the each problem.**

DataShop assigns knowledge components into one of five categories based on the results from observed error rates and slopes from fitted learning curves using AFM:

- **Too little data.** *Criteria:* Fewer than three attempts are made at this KC.
- **Low and flat.** *Criteria:* All attempts at this KC have an error rate lower than 20%. *Interpretation:* Students have mastered the concept before their first recorded attempt.
- **No learning.** *Criteria:* The fitted curve does not have a decreasing trend for this KC. *Interpretation:* Students

[1]We provide our implementation's source code at: https://github.com/marifdemirtas/ast-kc-step-generation

do not increase their mastery of the KC despite having a significant amount of attempts.

- **Still high.** *Criteria:* The fitted curve has a decreasing trend for this KC, but the error rate for the final attempt is higher than 40%. *Interpretation:* Students are still learning and need more attempts to master the KC.
- **Good.** *Criteria:* The fitted curve has a decreasing trend for this KC and the error rate for the final attempt is lower than 40%. *Interpretation:* Students have mastered the KC.

AFM values are also used for evaluating how well a KC model explains data using AIC (Akaike information criterion), BIC (Bayesian information criterion), and RMSE values based on two types of cross-validation, where the model's predictive capability on unseen students and unseen transactions are measured [16]. A lower AIC/BIC score means the model explains the data better with less parameters, whereas a lower RMSE means that the model can generalize to a new dataset from the same tutoring environment.

Note that some prior work utilized the power law of practice to compare calculated learning curves to an ideal curve with exponential decay [33]. In our work, we follow Rivers et al. [30] and look for decline trends in the fitted AFM curves, representing a decrease in error rate and a process of 'mastering' the KC in further attempts instead of calculating the deviation from an exponential curve. While this is a more relaxed constraint for evaluating learning curves, we found it to be more meaningful on code-writing data as the step generation process differs from traditional learning curve calculation and the generated curves rarely show clear exponential decay. For completeness, we provide the model fit $R^2$ and the slope of the curve for all knowledge components in Appendix A.

## 3.3 Dataset

We hypothesize that limitations in enrolled student count and number of problems hampered previous applications of AST nodes. Therefore, we used a large dataset collected from homework in an introductory Python course for non-CS majors from a large Midwestern research university. With IRB approval, data was collected from 7 semesters starting from Fall 2019 to Fall 2022, with 400 to 700 students each semester. Students from 9 to 11 unique academic colleges were enrolled in the course, with a majority of students coming from Business and Liberal Arts and Sciences. Enrolled students were mostly in their first or second year of undergraduate. Furthermore, the sample was diverse in terms of gender, with females comprising, on average, 42.6% across 7 semesters. In terms of race/ethnicity, on average, 21.3% were Asian, 4.4% were Black/African American, 11.9% were Hispanic, 16.6% were International, 3.2% were Multi-race, 41.8% were White, and 0.9% were unknown.

To have comparable samples for all semesters and to not exceed the computational requirements of DataShop, we created two random samples from our dataset with 400 students and 40 students respectively for all semesters. Unless explicitly mentioned, we use the larger sample in our experiments. We opted not to create samples including students from different semesters. This is because students in each semester go through different experiences during each offering of the course, and so their learning opportunities are distinct. In particular, the dataset features three sets of co-instructors, as well as semesters where the course was primarily to entirely online due to the COVID pandemic: entirely online from halfway through Spring 2020 through Spring 2021 and retaining online meetings for lectures until Fall 2022.

Our analysis is focused on the programming questions from the class. These consist of auto-graded programming exercises where students write a function for a small task, such as summing all the numbers in a list. As the grading is based on the outcome of test cases, these open-ended questions could have multiple valid solutions. The students' homework was hosted on the PrairieLearn platform, from which student submission data was collected [36]. The dataset includes student submissions from 129 unique questions and 80 isomorphic variants [4, 18], questions generated from other existing questions to test the same underlying concepts following the strategy proposed by Fowler and Zilles [11]. These questions appeared on homework during the 7 semesters from which data was gathered. Critically, among these questions are multiple opportunities to assess the same underlying KCs. However, not every question was used in every semester as assignments were modified between course offerings[2] and not every student necessarily attempted every question.

The dataset also included curated, instructor written solutions for all questions. These served as a baseline solutions with which to compare student submissions to for the classifying KC attempts (see Section 3.1). Our analysis of instructor solutions showed that an average solution has 5.16 lines and contains 40 tokens. A problem involves 14.44 knowledge components on average as measured by the number of unique AST nodes in the solution. However, some of these AST nodes do not map to a meaningful language element as they appear in all submissions that satisfy a trivial set of conditions. These are "**Module**" which is the root node for any AST in Python, "**Name**" and "**Load**" which appear in the AST for any operation that involves a variable, and "**FunctionDef**", "**arg**", and "**arguments**" which appear in the ASTs for any program that includes a function signature. We retain these 6 elements in our analysis to be comparable with the original study by Rivers et al. [30]. Figure 3 shows how many problems test each KC in each semester, as represented by the instructor solution of a problem. We see that all semesters have similar distributions of problems per KCs, and more than half of KCs are tested by more than 10 problems. More detailed explanations for each AST node can be found in the Python documentation[3] and in Appendix B.

# 4. RESULTS

In this section, we describe the results of our replication of the learning curve analysis of Rivers et al. [30], as well as additional analyses that explore the fit of the model and the conditions under which valid learning curves are generated.

[2]The exact number of questions that appear on homework in a given semester varies between 90 and 98.
[3]https://docs.python.org/3/library/ast.html

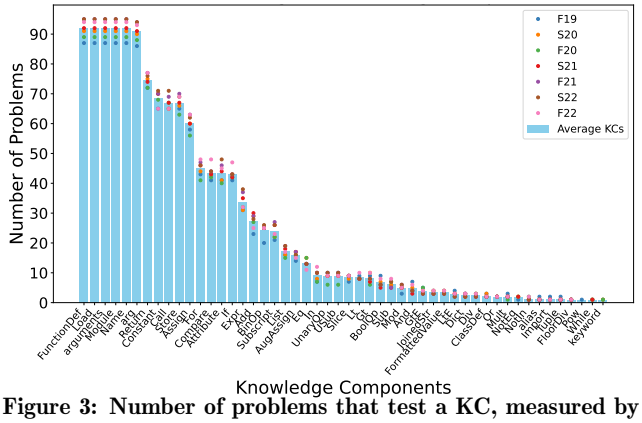


**Figure 3: Number of problems that test a KC, measured by the number of instructor solutions that use the KC.**

## 4.1 Evaluating AST nodes as a domain model

To understand how well a knowledge component model where KCs are represented by AST nodes (which we term the *AST-KC model*) can represent student skill development, we evaluate the learning curve fit for the seven semesters of our data. We generate two types of learning curves: those for all KCs in the model, and those for individual KCs.

### *4.1.1 Validity of the domain model*

To evaluate the overall fit of the domain model, we first look at learning curves that incorporate all knowledge components. These curves are generated by first creating individual learning curves of all KCs and then taking the average error value across KCs at each attempt. As with any learning curve, these curves are expected to start higher (representing the high error rate of students who are beginners) and decrease steadily as students master individual KCs.

Learning curves for all KCs combined across the seven semesters on the large sample (n=400) are shown in Figure 4. Table 1 shows the initial error rate, final error rate, and the slopes of the AFM curves fitted to these averaged learning curves for both small (n=40) and large (n=400) samples.

**Table 1: Error rates at initial and final attempts, and the estimated slope of the AFM curves across all KCs, showing a clear decline in all semesters compared to previous work [30] with no remarkable difference between our sample sizes.**

| Large sample (n=400) | | | | | | | |
|---|---|---|---|---|---|---|---|
| | F19 | S20 | F20 | S21 | F21 | S22 | F22 |
| Initial Error | 51.507 | 45.386 | 41.957 | 40.687 | 44.781 | 41.557 | 47.831 |
| Final Error | 27.463 | 19.983 | 24.401 | 25.99 | 25.057 | 28.761 | 27.788 |
| Slope | -0.273 | -0.273 | -0.197 | -0.162 | -0.208 | -0.136 | -0.216 |
| Small sample (n=40) | | | | | | | |
| | F19 | S20 | F20 | S21 | F21 | S22 | F22 |
| Initial Error | 52.544 | 47.853 | 43.079 | 39.482 | 45.562 | 39.685 | 48.464 |
| Final Error | 28.194 | 21.836 | 22.965 | 26.174 | 28.095 | 26.42 | 21.031 |
| Slope | -0.277 | -0.280 | -0.226 | -0.146 | -0.184 | -0.141 | -0.312 |
| Δ Slope | -0.003 | -0.007 | -0.029 | 0.015 | 0.024 | -0.005 | -0.096 |
| Rivers et al. [30] (n=89) | | | | | | | |
| Initial Error | 29.697 | | | | | | |
| Final Error | 40.611 | | | | | | |
| Slope | 0.341 | | | | | | |

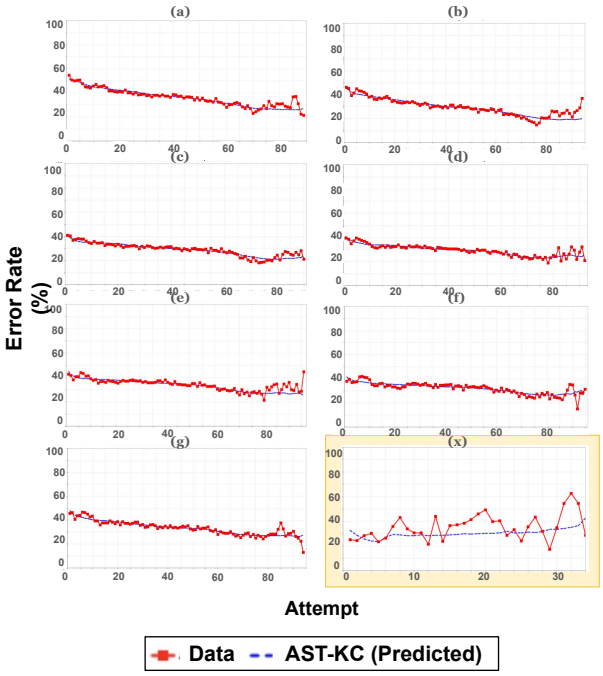


**Figure 4: Learning curves showing average error rate on all KCs from each semester between Fall 2019 to Fall 2022 (a-g), and from Rivers et al. [30] (x, highlighted yellow).**

Our learning curves for all KCs combined show a steady decrease in the error rate in all seven distinct semesters (see Figure 4). This indicates that students increase their mastery of knowledge components from the AST-KC model as they get more attempts, in contrast to the results from Rivers et al. [30] (see Figure 4(x)). Table 1 also shows that the combined curves for every semester have a negative slope greater than 0.1 in magnitude, confirming the decrease visible in Figure 4. The decrease in error rate varies between 13 percentage points and 24 percentage points, as opposed to the increase in error rate observed in the data from Rivers et al. [30]. Moreover, the same decrease is visible in the small sample, with an average difference of only -0.014 compared to the large sample's slope. This indicates that the larger number of students in our dataset is not the main reason for the improvement in learning curve analysis.

The decrease in averaged KC error rates shows that a majority of components from AST-KC model produce satisfactory learning curves. The consistency of the result across seven semesters implies that AST-KC can be a valid domain model that explains the skills developed in introductory programming to an extent.

Figure 4 also shows how all of the curves exhibit a smooth decrease in contrast to the jumps seen in other works applying KC models using AST nodes [30, 33]. Note that the red lines oscillate or increase in the last section of the plots. As some KCs are practiced across more problems than others, later attempts in the aggregated curve have fewer KCs averaged in comparison to earlier attempts, leading to the oscillation. While the AFM curve (blue dashed line) compensates for this to some extent, this visualization may not be an optimal way to model domains where the number of opportunities for each KC differs. Therefore, while these

curves have been used to compare KC models in the study we replicate, we present other comparison methods that might produce more informative results.

To further validate the AST model, we compare our results to two generic baselines from prior work. *Single-KC* defines a single KC for all problems, representing the mastery of programming in general [27]. *Random-KC* randomizes the set of KCs associated with all the questions so each question has a new set of KCs assigned to it [33].

**Table 2: Means and standard deviations for AIC, BIC and RMSE values of AST-KC and baselines Single-KC and Random-KC across all semesters (lower is better).**

| | | Mean | SD |
|---|---|---|---|
| AIC ↓ | AST-KC | 543,477.73 | 26,650.51 |
| | Single-KC | 581,475.19 | 29,631.02 |
| | Random-KC | 579,877.72 | 25,475.15 |
| BIC ↓ | AST-KC | 549,578.24 | 26,674.01 |
| | Single-KC | 585,915.64 | 29,647.67 |
| | Random-KC | 585,978.22 | 25,495.69 |
| RMSE (student) ↓ | AST-KC | 0.458 | 0.008 |
| | Single-KC | 0.476 | 0.009 |
| | Random-KC | 0.472 | 0.010 |
| RMSE (item) ↓ | AST-KC | 0.448 | 0.009 |
| | Single-KC | 0.468 | 0.010 |
| | Random-KC | 0.473 | 0.010 |

We compare AST-KC and the generic baselines using mean AIC/BIC scores, item-blocked RMSE and student-blocked RMSE across all semesters in Table 2. The results show that our KC model outperforms both Single-KC and Random-KC, as it achieves a lower AIC/BIC score. A lower score was achieved in all semesters using AST-KC. The model also has a lower RMSE in all semesters for both predicting unseen students and unseen items.

### 4.1.2 Validity of individual knowledge components

As a more fine-grained look into which KCs are being learned well by the students, Figure 5 shows the categories of KCs observed in Rivers et al. [30] across all semesters in our dataset. All three KCs identified to have 'Good' learning curves in the original study (usage of function arguments, usage of function definition, and comparison operator) are also consistently identified as 'Good' in our study. Moreover, we observe 'Good' learning for language elements that were categorized as 'No learning' or did not have enough data in the previous work.

Out of 48 knowledge components pictured in Figure 5, we observed 39 KCs that had enough data in all semesters to be successfully analyzed. 22 of these KCs were either 'Good' or 'Still high' in all semesters, indicating a decrease in error rate. Since the difference between 'Good' and 'Still high' categories depends on the final error rate, learning curves with the same decrease in error may be classified in either category due to a variance in initial ability. Thus, we argue that both 'Good' and 'Still high' categories indicate satisfactory learning. The complete results on model fit ($R^2$) and the slope, representing the learning rate, can be found in Appendix A.

| | F19 | S20 | F20 | S21 | F21 | S22 | F22 | Rivers et al. |
|---|---|---|---|---|---|---|---|---|
| arg | Good | Good | Good | Good | Good | Good | Good | No learning |
| arguments | Good | Good | Good | Good | Good | Good | Good | Good |
| Dict | Good | Good | Good | Good | Good | Good | Good | Too little data |
| FunctionDef | Good | Good | Good | Good | Good | Good | Good | Good |
| Module | Good | Good | Good | Good | Good | Good | Good | Low and flat |
| Tuple | Good | Good | Good | Good | Good | Good | Still high | Too little data |
| BoolOp | Still high | Good | Good | Good | Good | Good | Good | No learning |
| Compare | Good | Good | Good | Good | Still high | Good | Still high | Good |
| Or | Good | Good | Good | Good | Good | Still high | Still high | Too little data |
| In | Good | Good | Good | Still high | Good | Good | Still high | Too little data |
| For | Good | Good | Still high | Good | Good | Still high | Good | No learning |
| Return | Good | Still high | Still high | Good | Good | Good | Good | No learning |
| And | Still high | Still high | Good | Good | Good | Good | Good | No learning |
| If | Good | Still high | Good | Still high | Still high | Good | Good | Low and flat |
| Eq | Still high | Still high | Good | Good | Still high | Good | Good | Still high |
| Constant | Good | Good | Good | Still high | Still high | Still high | Still high | No learning |
| Call | Good | Still high | Still high | Good | Still high | Still high | Good | Still high |
| Mod | Good | Still high | Still high | Good | Good | Still high | Still high | No learning |
| GtE | Still high | Good | Still high | Still high | Still high | Good | Good | Too little data |
| Expr | Still high | Still high | Still high | Good | Good | Good | Still high | Too little data |
| BinOp | Good | Still high | No learning | Good | Good | Good | Good | No learning |
| Div | Still high | Still high | Still high | Still high | Still high | Still high | Still high | Too little data |
| LtE | Still high | Still high | Still high | Still high | Still high | Still high | Still high | Too little data |
| List | Still high | Still high | Still high | Good | No learning | Good | Still high | Too little data |
| Attribute | Still high | Still high | Still high | Still high | No learning | Good | Good | No learning |
| Add | Good | Good | Good | No learning | No learning | Good | Good | No learning |
| Name | Good | Still high | Still high | Still high | No learning | No learning | Good | No learning |
| Gt | Still high | Still high | Still high | Still high | No learning | No learning | Still high | No learning |
| Assign | Good | Still high | Good | No learning | No learning | No learning | Still high | No learning |
| Store | Still high | Still high | Good | No learning | No learning | No learning | Good | No learning |
| Sub | No learning | Still high | No learning | Still high | Still high | Good | No learning | Too little data |
| Load | Good | Still high | Still high | No learning | No learning | No learning | No learning | No learning |
| Lt | Still high | Still high | No learning | No learning | Good | No learning | No learning | No learning |
| Mult | Still high | Still high | Too little data | Too little data | Too little data | Too little data | No learning | No learning |
| NotIn | No learning | Good | No learning | Low and flat | No learning | No learning | Good | Too little data |
| Import | No learning | Too little data | Too little data | Still high | Still high | No learning | No learning | Too little data |
| NotEq | No learning | Good | Good | No learning | No learning | No learning | No learning | Too little data |
| While | Too little data | Too little data | Too little data | Too little data | Good | Too little data | Too little data | Too little data |
| Subscript | Still high | Good | No learning | No learning | No learning | No learning | No learning | No learning |
| JoinedStr | No learning | Too little data | Too little data | Too little data | Too little data | No learning | Still high | No learning |
| alias | No learning | Too little data | Too little data | No learning | Still high | No learning | No learning | Too little data |
| Slice | No learning | No learning | No learning | No learning | No learning | No learning | Still high | Too little data |
| UnaryOp | No learning | No learning | No learning | No learning | No learning | No learning | No learning | Too little data |
| USub | No learning | No learning | No learning | No learning | No learning | No learning | No learning | Too little data |
| FloorDiv | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data | No learning |
| Not | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data |
| Pow | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data | Too little data | No learning |

**Figure 5: KCs observed in Rivers et al. [30] and their category, as assigned by DataShop, across the seven semesters in our dataset.**

On this note, we also present the percentage of KCs that have satisfactory learning compared to all observed KCs in each semester in Table 3. On average, 45.6% of AST-KC components have learning curves with either 'Good' or 'Still high' categories on the large sample (SD 6.2%). Similar results are obtained on the small sample with an average of 44.4% (SD 5.9%). By contrast, in previous works only single digit counts of KCs with satisfactory learning curves were detected using AST nodes [30, 33].

In addition to the evidence for the validity of AST-KC model on data from a higher number of students, learning curve analysis of the code-writing data we collected shows much more support for the AST-KC model than Rivers et al. [30] even at half the student size (40 vs 89). This further confirms that our increased sample size was not the primary reason for stronger evidence on the validity of the AST-KC model.

**Table 3: Percentages of KCs with satisfactory learning curves (as represented by a decreasing error rate) across all semesters, and from previous work [30] (in yellow).**

| Large sample (n=400) | | | | | | | |
|---|---|---|---|---|---|---|---|
| | F19 | S20 | F20 | S21 | F21 | S22 | F22 |
| %KC | 50.65% | 52.56% | 44.00% | 38.96% | 45.95% | 36.36% | 50.67% |
| Small sample (n=40) | | | | | | | |
| | F19 | S20 | F20 | S21 | F21 | S22 | F22 |
| %KC | 52.17% | 51.43% | 43.66% | 39.13% | 40.30% | 37.14% | 47.06% |
| Δ % | 1.52% | -1.14% | -0.34% | 0.17% | -5.65% | 0.78% | -3.61% |
| Rivers et al.[30] (n=89) | | | | | | | |
| %KC | 10.41% | | | | | | |

## 4.2 Examining the impact of course properties on learning curves

Our analysis found significant fit of a knowledge component model based on AST nodes, in contrast to prior work. Since our results show this improvement is present even in small samples of our dataset, in this section we focus on a key property of our dataset and explore how its large problem set interacts with the success of learning curve analysis.

To understand to what extent having more code-writing problems in our data set may explain our improvements over previous work, we explored the relationship between the number of problems that test a particular KC and the amount of learning seen on those KCs. To find the number of problems that test a given KC, we counted how many problems contained that KC somewhere in the solution space. Then, we looked at the distribution of numbers of problems for KCs with a 'Good', 'Still high', and 'No learning' categorization (see Figure 6).

We see that students have a lot more opportunities to practice knowledge components categorized as 'Good' and 'Still high', with median values of 68 and 64 problems, respectively. On the other hand, AST nodes with less practice opportunities tend to be in the 'No learning' category with a median of 30 problems per knowledge component.

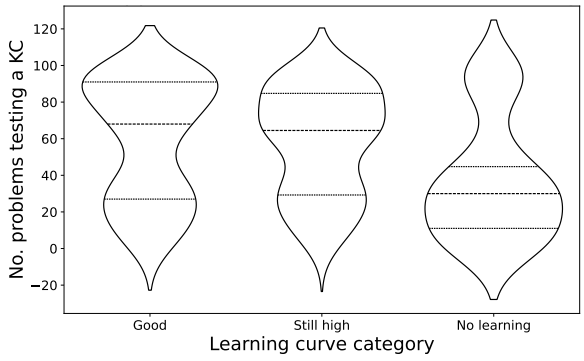


**Figure 6: Distribution of number of practice opportunities for KCs from 'Good', 'Still high', and 'No learning' categories.**

This relationship suggests that a larger number of measurement opportunities may increase the chances to observe learning on a given AST-based KC. Without enough measurement opportunities, we are more likely to observe no decrease in error rates ('No learning'), but once given enough opportunities to practice a KC, we are more likely to observe some decrease in error rates ('Still high' or 'Good'). This could explain why our learning curve analysis produces much better results compared to prior works that used AST nodes as knowledge components, as our dataset includes twice as many problems for each semester.

## 4.3 Case studies of successful knowledge components

Our application of the AST-KC model showed learning on a large variety of components represented by AST nodes. When we look more deeply at some of these KCs, we see patterns along the lines of different types of syntax structures. We present two case studies below that may help explain which language elements are more appropriate to be measured using AST nodes as a KC model.

Note that the red lines often oscillate towards the end of the curve, as they include data from fewer students who attempted more problems than the majority of the class.

### *4.3.1 Control flow structures*

We found that KCs representing control flow structures, such as for loops, if statements, and return statements, show a decrease in error rates across all semesters in our dataset. (A notable exception is while, which was not covered in this course.) The large number of attempts on these skills as well as the nature of the relevant AST nodes may explain this result. To gain more insight, we look more closely at the "**For**" and "**If**" KCs.

Students have many attempts to practice the use of "**For**" and a decreasing error rate is achieved in all semesters (see Figure 7). This KC has learning curves categorized as either 'Good' or 'Still high' in all semesters. This is in stark contrast to the findings of Rivers et al. [30], which placed "**For**" in the 'No learning' category with only 4 attempts.

Loops are heavily emphasized in the introductory curricula of this course, as on average half of the coding problems in a given semester test "**For**" (see Figure 3). Students have over 45 attempts at this KC each semester.

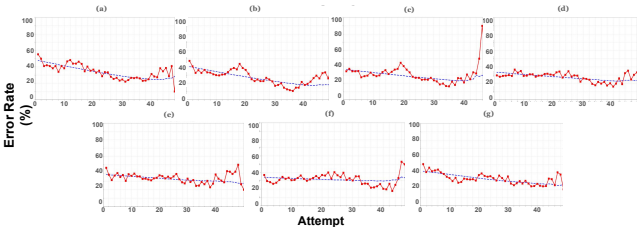


**Figure 7: Learning curves from Fall 2019 (a) to Fall 2022 (g) for the KC "For" show 'Good' learning in five semesters and is 'Still high' in two semesters (c, f).**

Another control flow structure, "**If**", also shows consistent learning with 3 semesters in 'Good' and 4 semesters in 'Still high' learning. Similarly to "**For**", "**If**" also has a large number of practice opportunities in each semester, with 43 problems on average testing it. Moreover, "**If**" was categorized as 'Low and flat' in the analysis of Rivers et al. [30], indicating that students had already mastered the concept by the first recorded attempt. With a dataset that includes more opportunities to observe mastery early in the semester, we also see that we can capture the initial learning period with a decrease in error rates.

In addition to the large number of opportunities, we attribute the satisfactory learning in control flow structures to the nature of AST tree construction. Since AST nodes for these elements appear at the start of a block, a student may master the KC for one of these structures even if they make small mistakes in the body of the control flow statement. For example, the KC "**For**" measures the ability to identify where an iteration is needed to be applied, and not the correctness of the body of the loop. The ability to write a for loop in an accurate location in a program is a more specific

skill than the ability to write correct for loops in general, and this constrained KC seems likely to reflect a discrete skill where learning can be measured.

#### 4.3.2 Data structure declarations

In our analysis, we found that data structure AST nodes, such as "**Dict**", "**Tuple**", and (to a lesser extent) "**List**", tend to show satisfactory learning. However, in contrast to control structures, there are many fewer measurement opportunities for each of these KCs.

Learning curves for "**Dict**" show consistently strong evidence of learning, even though students only practice this KC for few attempts (see Figure 8). "**Dict**" is measured by 2.57 problems on average in a given semester, and other data structure AST nodes such as "**Tuple**" (1.14) and "**List**" (17.14) are used in many fewer problems compared to control flow structures, but also show satisfactory learning curves (see Figure 5).

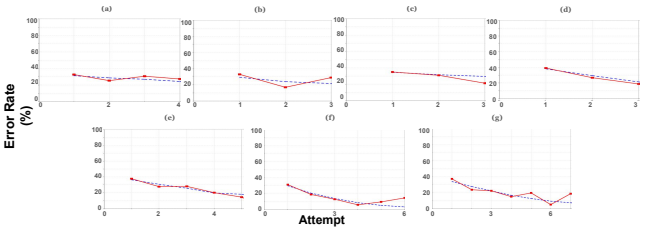


**Figure 8: Learning curves from Fall 2019 (a) to Fall 2022 (g) for the KC "Dict" show 'Good' learning in all seven semesters.**

While this is surprising, the construction of AST trees provides a potential explanation. In Python, the node for a data structure only appears for the declaration of that data structure in the AST. In other words, the KC "**Dict**" in the AST-KC model measures whether or not a student *declares* a dictionary correctly, not whether they perform other dictionary operations correctly, such as updating a value or removing a key. It appears that students master the skill of declaring a data structure in rather few attempts, making this a KC with a satisfactory learning curve.

# 5. DISCUSSION

## 5.1 AST nodes as knowledge components

In this section, we answer **RQ1**: *Can student progression in introductory programming be tracked using language features as knowledge components?*

Our results show that language features, represented by AST nodes, can model student learning as knowledge components to a significant extent. Both the overall model fit (see Section 4.1.1) and the fit of many of the individual KCs (see Section 4.1.2) show that students' error rates decrease as they practice these KCs during the course of the semester. Furthermore, these results are consistent across seven different semesters. Students do improve at using many individual AST nodes as they get more attempts to practice them, confirming that AST nodes are a viable candidate for modeling students' domain knowledge in programming.

This finding is contrary to the results of previous attempts to use AST nodes as knowledge components. Rivers et al. [30] failed to observe a decreasing trend in their averaged knowledge component curves, and the baseline results from Shi et al. [33] that used AST nodes as KCs also could not obtain decreasing trends in learning curves. Furthermore, in Table 3 we see that between 35% to 55% of the KCs in each semester have satisfactory learning rates, compared to only 5 KCs showing satisfactory learning out of 48 in Rivers et al. [30] and "only a small subset" with valid learning rates in Shi et al. [33]. Our results from Table 2 also confirm the validity of the AST-KC model on the basis of model fit (AIC/BIC) and predictive power (RMSE), in contrast with Shi et al. [33] who observed lower scores for model fit compared to the randomized baseline.

## 5.2 Course properties that support learning curve analysis

Our finding that AST nodes can serve as viable knowledge components in the analysis of code-writing data is in stark contrast to prior work. This motivates **RQ2**: *Which properties of our course lead to successful learning curves?*

There are two key features of our dataset that differentiate our analysis from prior work. First, our student samples include 400 students, which is approximately 5 times larger than the sample of Rivers et al. [30] (89 students) and comparable to the sample of Shi et al. [33] (410 students). Second, our question pool includes at least 90 unique problems each semester, compared to 40 problems from Rivers et al. [30] and 50 problems from Shi et al. [33].

Initially, we hypothesized that increasing the student sample size might improve learning curve analysis and address the jagged, highly varying behavior of AST-based learning curves in previous works [30, 33]. However, we observed that the size of the student sample does not have a remarkable affect on our learning curves (See Section 4.1). These results indicate that a large number of students in a course is not a prerequisite for learning curve analysis of code-writing data.

Based on this, we believe having a large number of questions to test knowledge components is a more important condition for successful learning curve analysis. Our data supports this argument in a few ways. Section 4.2 shows how KCs with declining error rates have a larger number of practice opportunities compared to 'No learning' KCs. Moreover, some KCs with no learning in Rivers et al. [30] were consistently observed to have a positive learning rate in our data, such as "**For**" (Figure 7). We see that these KCs are tested through more attempts in our course compared to previous works.

While we did not have access to problems and instructor solutions from previous studies, we were able examine a partial set of student submissions used in Rivers et al. [30], available on DataShop. In an analysis of these submissions, we observed that more than two thirds of KCs included in the study of Rivers et al. were tested by less than 10 problems. In comparison, our dataset had more than 10 problems for over 60% of KCs.

Based on these observations, we hypothesize that a larger question pool with more attempts for each KC may support learning curve analysis on code-writing problems because it mediates the impact of other features of each problem. In

code-writing problems, where many skills must be combined in a single submission, a confusing or challenging aspect of a problem may cause the student to submit an incorrect program, even if they understand some relevant KCs. However, with more attempts, our learning curve analysis may become more resistant to such outliers.

## 5.3 Language elements that are better represented by AST-based KCs

Finally, we answer **RQ3**: *Which language elements may be best identified as knowledge components using AST nodes?*

Some types of AST nodes were consistently learned by students in all semesters. While our dataset includes seven semesters with similar instructional material, the course featured different instructors, the enrolled student cohort varied in prior experience, and some semesters were affected by the COVID pandemic. Thus, the consistency of learning curve analysis across semesters gives us additional insight into which KCs best represent student skills. Looking into the language elements with consistently successful learning curves, we observe that language elements that usually are used in similar contexts tend to have satisfactory learning curves, whereas elements that can be used in many different contexts are harder to track with AST-based KCs.

The most appropriate language elements to be KCs may be control flow structures. We noticed these structures, such as if statements, for loops, and return statements, are easily and accurately detected from ASTs. As control flow structures are practiced in a fairly large portion of opportunities, students are likely to show satisfactory learning on these KCs. Moreover, these structures are likely to be used in similar contexts in small programs. The mix of copious practice opportunities and the ease with which correct control flow structure can be drawn from ASTs may make them ideal language features to use in this type of learning curve analysis.

A second class of consistently good KCs comes from data structures such as dictionaries and tuples. These KCs are good despite the small number of practice opportunities. These may be less useful KCs to use, as a deeper inspection showed that AST nodes for these structures only measured the mastery of declaring a data structure and did not capture the use of data structures elsewhere in the program, limiting the various contexts a data structure could be used in. Therefore, it is important to note that AST nodes for data structures cannot be interpreted as a sign of mastery in all the skills required for the underlying language element.

Some language elements did not show consistent validity as KCs. Operators are a surprisingly inconsistent source of KCs, with some high-quality curves (e.g. **"BoolOp"**) and some consistently underperforming curves (e.g. "**UnaryOp**", "**Not**"). We attribute this to the flexibility of these elements as they can be used in various contexts that have less surface-level similarity, preventing students from transferring their knowledge to new problems.

Thus, while the methodology of KC generation based on AST nodes enables automatic labeling of KCs, it also presents challenges in how the resulting KCs should be interpreted. Most importantly, AST-KC should not be seen as a complete domain model that can explain all skills relevant to programming, but as a promising approximation to some of the key skills related to using fundamental structures in programming in some contexts without time-consuming expert annotations. Developing KCs that can capture context-independent skills is grounds for future study.

## 5.4 Implications for practice

### 5.4.1 Teaching

The evidence we found for the AST-KC model during introductory programming learning implies that instructors *can* track student skill development in terms of individual programming language features—with some caveats. The nature of the AST-based methodology for measuring successful use of language features requires that we use caution when interpreting what this type of learning curve analysis tells us about student skill development.

For instance, our analysis shows that students do improve their ability to use individual control structures like for, if, and return over time. Instructors should not over-interpret this finding, however. More specifically, we found that across the course of a semester, students improve their ability to put for loops, if statements, and return statements in *correct locations in their code*. The AST-KC model does not tell us about students' ability to implement the internals of such control structures correctly.

When computing instructors colloquially describe students as "learning for loops", they mean that students not only learn when to use a for loop, but also how to use it accurately in various situations. Our results suggest that students' ability to use a variety of language features in appropriate locations are valid KCs even if they do not capture all the skills related to the language features. Instructors may find it useful to teach and assess the skill of knowing *when* to use a language feature independently of the skills of knowing *how* to use the language feature.

Further, learning curve analysis serves as a potentially useful diagnostic tool for understanding where a given course succeeds and where improvements may be necessary. Learning curves may identify KCs that are taught consistently well or insufficiently. This comes with a caveat, in that our KCs are based on ASTs and thus skills that do not directly involve the use of syntactic elements cannot be diagnosed with AST-based KCs. Regardless, KCs with consistently good curves can provide instructors with some confidence that the related syntactic elements are well covered in the course, while KCs with underperforming curves can identify possible rooms for improvement.

### 5.4.2 Learning Environments

We have shown that our step generation algorithm can indeed support learning curve analysis on open-ended code-writing data. As many programming learning environments already collect the necessary data from code-writing exercises (students' code and its correctness), and the step generation process and step labeling for AST-based KCs can be entirely automated, our approach to learning curve analysis can be readily added to existing programming learning

environments. Automatically generating learning curves for AST-based KCs as part of learning environments already in use would make learning curve analysis accessible to all programming instructors, without the need to develop potentially costly intelligent tutoring systems.

Our finding that learning curve analysis with the AST-KC model can provide evidence of skill development even at small student samples implies that such systems may be beneficial even for small classrooms. However, our findings also suggest that a large number of small programming problems may be necessary to support learning curve analysis in introductory programming. Given this, it is important that learning environments contain a sufficient number of code-writing exercises that measure each KC.

One mechanism for creating *many small problems* that measure the same KC is the creation of *isomorphic* questions. Isomorphic questions are question variants designed to test the same programming concepts while modifying surface-level details [4, 18]. Isomorphic questions have been used in computing education both to assess learning and expand the size of question pools for assessments [5, 38, 26, 21, 11].

### 5.5 Limitations and future work

While our work is a replication of Rivers et al. [30] and follows the assumptions from that original work, our implementation differs in the method of finding the most similar AST for step generation. We use a metric based on node frequencies, while Rivers et al. used the ITAP algorithm [31] (see Section 3.1). Since node frequencies may not be as sensitive as the ITAP path construction while determining the most similar correct submission, our method may mark some KCs that students use correctly as incorrect. However, this difference is likely to cause *false negatives*, that is, KCs that were practiced correctly and classified as incorrect, rather than *false positives*, KCs that were practiced incorrectly and classified as correct. Thus, future work may explore other similarity measures with a higher sensitivity to observe even better learning curves by detecting these false negatives that underestimate student mastery.

Another limitation of applying learning curve analysis on only code-writing data is the possibility of missing learning opportunities in different forms. The course we analyzed includes many other learning opportunities beyond coding activities in homework, such as online textbook activities, quizzes, exams, and non-coding activities in the homework. Devising an updated step generation model to capture the increase in mastery in these other activities (e.g. multiple choice questions, Explain in Plain English tasks [9]) in addition to code-writing attempts could result in learning curves that capture a more accurate picture of the student progression while mastering these KCs.

An interesting avenue for future work could be generating and validating AST-based KCs that contain more contextual information. We show that the most straightforward construction of an AST-based KC model, where individual AST nodes represent KCs, does show partial validity as a model for programming learning. However, some of these KCs are difficult to interpret, or are used in a variety of contexts. More sophisticated KCs that represent combinations of AST nodes (e.g., an if statement inside a for loop, or an if statement with a modulo operator inside) may yield additional insight about what skills students develop as they learn programming.

## 6. CONCLUSION

We replicated a previously proposed approach to apply learning curve analysis to open-ended code-writing data, and, contrary to prior results, we found that abstract syntax tree nodes can model skill acquisition in introductory programming education to a certain extent. While interpretation of the meaning of these skills should be done with caution, we observe that students increase their mastery in many language elements across the course of a semester, such as for loops, if statements, return statements, declarations of data structures, and boolean operators. Analysis of learning curve results across different slices of our dataset suggests that having many measurement opportunities for a KC leads to learning curves that show a decrease in error rate. The viability of this method for automatically extracting information about students' successful or unsuccessful use of language elements for use in learning curve analysis means that such analysis could be used as a diagnostic tool in any programming course to observe if students are learning to use language elements correctly and modify the curriculum based on topics students have difficulties with.

## 7. ACKNOWLEDGMENTS

We thank the University of Illinois Urbana-Champaign Computer Science Department for funding that contributed to this work. We also thank Cindy Tipper from the Datashop development team for their support.

## 8. REFERENCES

[1] V. Aleven and K. R. Koedinger. Knowledge component (KC) approaches to learner modeling. In *Design Recommendations for Intelligent Tutoring Systems*, volume 1, chapter 15, pages 165–182. US Army Research Laboratory, 2013.

[2] I. Alpizar-Chacon, S. Sosnovsky, and P. Brusilovsky. Measuring the Quality of Domain Models Extracted from Textbooks with Learning Curves Analysis. In N. Wang, G. Rebolledo-Mendez, N. Matsuda, O. C. Santos, and V. Dimitrova, editors, *Artificial Intelligence in Education*, pages 804–809, Cham, 2023. Springer Nature Switzerland.

[3] J. R. Anderson and B. J. Reiser. The LISP tutor. *Byte*, 10(4):159–175, 1985.

[4] P. D. Bliese, D. Chan, and R. E. Ployhart. Multilevel methods: Future directions in measurement, longitudinal analyses, and nonnormal outcomes. *Organizational Research Methods*, 10(4):551–563, 2007.

[5] L. Butler, G. Challen, and T. Xie. Data-Driven Investigation into Variants of Code Writing Questions. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10, Nov. 2020. ISSN: 2377-570X.

[6] H. Cen, K. Koedinger, and B. Junker. Learning Factors Analysis – A General Method for Cognitive Model Evaluation and Improvement. In M. Ikeda, K. D. Ashley, and T.-W. Chan, editors, *Intelligent*

*Tutoring Systems*, Lecture Notes in Computer Science, pages 164–175, Berlin, Heidelberg, 2006. Springer.

[7] M. Corney, R. Lister, and D. Teague. Early relational reasoning and the novice programmer: Swapping as the "hello world" of relational reasoning. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 95–104, AUS, Jan. 2011. Australian Computer Society, Inc.

[8] S. H. Edwards and M. A. Perez-Quinones. Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, page 328, New York, NY, USA, June 2008. Association for Computing Machinery.

[9] M. Fowler, B. Chen, and C. Zilles. How should we 'explain in plain english'? voices from the community. In *Proceedings of the 17th ACM conference on international computing education research*, pages 69–80, 2021.

[10] M. Fowler, D. H. Smith IV, M. Hassan, S. Poulsen, M. West, and C. Zilles. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education*, 32(3):355–383, July 2022.

[11] M. Fowler and C. Zilles. 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*, SIGCSE '21, page 3–9, New York, NY, USA, 2021. Association for Computing Machinery.

[12] C. Goutte and G. Durand. Confident Learning Curves in Additive Factors Modeling. *International Educational Data Mining Society*, 2020.

[13] K. Koedinger, K. Cunningham, A. Skogsholm, and B. Leber. An open repository and analysis tools for fine-grained, longitudinal learner data. In *Educational Data Mining 2008*. Citeseer, 2008.

[14] K. R. Koedinger, R. S. J. d. Baker, K. Cunningham, A. Skogsholm, B. Leber, and J. Stamper. *A Data Repository for the EDM Community: The PSLC DataShop*. CRC Press, 2010.

[15] K. R. Koedinger, A. T. Corbett, and C. Perfetti. The Knowledge-Learning-Instruction Framework: Bridging the Science-Practice Chasm to Enhance Robust Student Learning. *Cognitive Science*, 36(5):757–798, 2012.

[16] K. R. Koedinger, E. A. McLaughlin, and J. C. Stamper. Automated Student Model Improvement. Technical report, International Educational Data Mining Society, June 2012.

[17] M. Z. Last, M. Daniels, V. L. Almstrum, C. Erickson, and B. Klein. An international student/faculty collaboration: The Runestone project. *ACM SIGCSE Bulletin*, 32(3):128–131, July 2000.

[18] F. Lievens and P. R. Sackett. Situational judgment tests in high-stakes settings: Issues and strategies with generating alternate forms. *Journal of Applied Psychology*, 92:1043–1055, 2007.

[19] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 101–112, New York, NY, USA, Sept. 2008. Association for Computing Machinery.

[20] B. Martin, A. Mitrovic, K. R. Koedinger, and S. Mathan. Evaluating and improving adaptive educational systems with learning curves. *User Modeling and User-Adapted Interaction*, 21(3):249–283, Aug. 2011.

[21] R. Millar and S. Manoharan. Repeat individualized assessment using isomorphic questions: a novel approach to increase peer discussion and learning. *International Journal of Educational Technology in Higher Education*, 18(1):22, Apr 2021.

[22] B. Mokbel, S. Gross, B. Paaßen, N. Pinkwart, and B. Hammer. Domain-independent proximity measures in intelligent tutoring systems. In S. K. D'Mello, R. A. Calvo, and A. Olney, editors, *Proceedings of the 6th International Conference on Educational Data Mining, Memphis, Tennessee, USA, July 6-9, 2013*, pages 334–335. International Educational Data Mining Society, 2013.

[23] G. L. Nelson, B. Xie, and A. J. Ko. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 2–11, New York, NY, USA, Aug. 2017. Association for Computing Machinery.

[24] A. Newell and P. S. Rosenbloom. Mechanisms of skill acquisition and the law of practice. In *Cognitive Skills and Their Acquisition*, pages 1–55. Psychology Press, 2013.

[25] H. Nguyen, Y. Wang, J. Stamper, and B. M. McLaren. Using Knowledge Component Modeling to Increase Domain Understanding in a Digital Learning Game. Technical report, International Educational Data Mining Society, July 2019.

[26] M. C. Parker, L. Garcia, Y. S. Kao, D. Franklin, S. Krause, and M. Warschauer. A pair of aces: An analysis of isomorphic questions on an elementary computing assessment. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ICER '22, page 2–14, New York, NY, USA, 2022. Association for Computing Machinery.

[27] P. I. Pavlik, H. Cen, and K. R. Koedinger. Performance Factors Analysis – A New Alternative to Knowledge Tracing. In *Proceedings of the 14th International Conference on Artificial Intelligence in Education*, 2009.

[28] T. W. Price, Y. Dong, and T. Barnes. Generating Data-Driven Hints for Open-Ended Programming. Technical report, International Educational Data Mining Society, 2016.

[29] J. Rho, M. Rau, and B. Vanveen. Investigating Growth of Representational Competencies by Knowledge-Component Model. In *Proceedings of the 15th International Conference on Educational Data Mining*, page 346, 2022.

[30] K. Rivers, E. Harpstead, and K. Koedinger. 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, pages 143–151, New York, NY, USA, Aug. 2016. Association for Computing Machinery.

[31] K. Rivers and K. R. Koedinger. Automating Hint Generation with Solution Space Path Construction. In S. Trausan-Matu, K. E. Boyer, M. Crosby, K. Panourgia, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, and G. Weikum, editors, *Intelligent Tutoring Systems*, volume 8474, pages 329–339. Springer International Publishing, Cham, 2014.

[32] K. Sanders, J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, L. Thomas, and C. Zander. Threshold concepts and threshold skills in computing. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 23–30, New York, NY, USA, Sept. 2012. Association for Computing Machinery.

[33] Y. Shi, R. Schmucker, M. Chi, T. Barnes, and T. Price. KC-Finder: Automated Knowledge Component Discovery for Programming Problems. Technical report, International Educational Data Mining Society, 2023.

[34] A. E. Tew and M. Guzdial. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 97–101, New York, NY, USA, Mar. 2010. Association for Computing Machinery.

[35] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to Represent Student Knowledge on Programming Exercises Using Deep Learning. Technical report, International Educational Data Mining Society, June 2017.

[36] M. West, G. L. Herman, and C. Zilles. PrairieLearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. In *2015 ASEE Annual Conference & Exposition*, pages 26–1238, 2015.

[37] B. Xie, D. Loksa, G. L. Nelson, M. J. Davidson, D. Dong, H. Kwik, A. H. Tan, L. Hwa, M. Li, and A. J. Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3):205–253, July 2019.

[38] D. Zingaro and L. Porter. Tracking student learning from class to exam using isomorphic questions. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, page 356–361, New York, NY, USA, 2015. Association for Computing Machinery.

# APPENDIX
# A. ADDITIONAL RESULTS

This appendix includes the full results for the knowledge components included in Figure 5, including the fit of the AFM model to the underlying data ($R^2$) and the slope parameter representing the learning rate.

| | F19 | | S20 | | F20 | | S21 | | F21 | | S22 | | F22 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $R^2$ | Slope | $R^2$ | Slope | $R^2$ | Slope | $R^2$ | Slope | $R^2$ | Slope | $R^2$ | Slope | $R^2$ | Slope |
| arg | 0.178 | 0.010 | 0.363 | 0.014 | 0.148 | 0.007 | 0.108 | 0.007 | -0.140 | 0.004 | 0.001 | 0.003 | 0.103 | 0.006 |
| arguments | 0.273 | 0.011 | 0.428 | 0.014 | 0.135 | 0.007 | 0.137 | 0.005 | -0.075 | 0.003 | 0.004 | 0.002 | 0.091 | 0.005 |
| Dict | -0.061 | 0.140 | -0.522 | 0.311 | -0.381 | 0.248 | 0.033 | 0.461 | -0.173 | 0.261 | 0.782 | 0.480 | 0.583 | 0.295 |
| FunctionDef | 0.273 | 0.011 | 0.428 | 0.014 | 0.135 | 0.007 | 0.139 | 0.005 | -0.075 | 0.003 | 0.005 | 0.002 | 0.089 | 0.005 |
| Module | 0.641 | 0.014 | 0.605 | 0.018 | 0.502 | 0.012 | 0.399 | 0.009 | 0.305 | 0.010 | 0.384 | 0.008 | 0.429 | 0.011 |
| Tuple | 0.894 | 0.278 | 0.735 | 0.400 | -1.557 | 0.134 | 0.667 | 0.315 | 0.081 | 0.086 | 0.256 | 0.190 | 0.228 | 0.107 |
| BoolOp | -0.797 | 0.087 | -0.762 | 0.046 | 0.816 | 0.099 | 0.823 | 0.206 | 0.797 | 0.135 | 0.544 | 0.102 | -0.062 | 0.155 |
| Compare | 0.596 | 0.026 | 0.364 | 0.027 | 0.154 | 0.015 | -0.104 | 0.018 | 0.082 | 0.009 | 0.013 | 0.015 | 0.595 | 0.020 |
| Or | 0.613 | 0.217 | 0.531 | 0.041 | 0.799 | 0.171 | 0.942 | 0.633 | 0.473 | 0.445 | 0.438 | 0.203 | -0.001 | 0.243 |
| In | -0.359 | 0.311 | 0.883 | 0.282 | -0.066 | 0.251 | 0.001 | 0.175 | -0.673 | 0.101 | -0.199 | 0.065 | 0.579 | 0.137 |
| For | -0.090 | 0.030 | 0.298 | 0.030 | 0.133 | 0.019 | 0.238 | 0.013 | 0.189 | 0.010 | 0.178 | 0.006 | 0.388 | 0.019 |
| Return | 0.464 | 0.014 | 0.136 | 0.013 | 0.121 | 0.007 | 0.031 | 0.008 | -0.056 | 0.002 | 0.059 | 0.004 | 0.156 | 0.006 |
| And | -1.129 | 0.047 | 0.160 | 0.149 | 0.900 | 0.219 | 0.806 | 0.137 | 0.436 | 0.052 | 0.514 | 0.092 | 0.180 | 0.242 |
| If | 0.606 | 0.023 | 0.092 | 0.026 | 0.018 | 0.013 | 0.147 | 0.018 | -0.186 | 0.011 | 0.438 | 0.016 | 0.632 | 0.020 |
| Eq | -0.879 | 0.078 | 0.284 | 0.077 | 0.710 | 0.067 | 0.094 | 0.140 | -0.754 | 0.082 | 0.669 | 0.130 | 0.302 | 0.087 |
| Constant | 0.284 | 0.015 | 0.181 | 0.018 | 0.354 | 0.012 | -0.035 | 0.008 | -0.018 | 0.005 | 0.190 | 0.008 | 0.225 | 0.013 |
| Call | 0.358 | 0.015 | 0.239 | 0.019 | 0.065 | 0.010 | 0.380 | 0.007 | -0.010 | 0.003 | -0.180 | 0.006 | 0.130 | 0.009 |
| Mod | -0.382 | 0.054 | -1.591 | 0.417 | -1.185 | 0.402 | -0.856 | 0.260 | 0.035 | 0.171 | -0.248 | 0.234 | -0.164 | 0.243 |
| GtE | -0.015 | 0.055 | -0.620 | 0.230 | 0.011 | 0.180 | -1.193 | 0.102 | -2.779 | 0.321 | 0.976 | 0.644 | 0.909 | 0.503 |
| Expr | -0.589 | 0.025 | 0.008 | 0.042 | 0.412 | 0.026 | -0.134 | 0.020 | -0.013 | 0.009 | -0.107 | 0.007 | 0.629 | 0.025 |
| BinOp | 0.708 | 0.078 | 0.623 | 0.048 | 0.012 | 0.000 | -0.072 | 0.004 | 0.084 | 0.007 | 0.569 | 0.035 | 0.293 | 0.023 |
| Div | -3.910 | 0.186 | 0.714 | 0.907 | -0.435 | 0.293 | -5.409 | 0.953 | -2.921 | 0.549 | -0.885 | 0.108 | -1.153 | 0.333 |
| LtE | -0.582 | 0.124 | 0.589 | 0.178 | -0.274 | 0.157 | -0.220 | 0.018 | -1.052 | 0.108 | -1.908 | 0.234 | 0.719 | 0.183 |
| List | -0.610 | 0.040 | 0.023 | 0.054 | -0.129 | 0.023 | 0.379 | 0.024 | -0.010 | 0.000 | -0.069 | 0.003 | -0.220 | 0.009 |
| Attribute | 0.065 | 0.015 | 0.195 | 0.017 | 0.164 | 0.008 | 0.129 | 0.005 | 0.022 | 0.000 | 0.059 | 0.003 | 0.331 | 0.009 |
| Add | 0.644 | 0.048 | 0.154 | 0.032 | 0.311 | 0.003 | -0.122 | 0.001 | -0.291 | 0.000 | 0.155 | 0.025 | 0.128 | 0.008 |
| Name | 0.535 | 0.011 | 0.229 | 0.010 | 0.040 | 0.002 | 0.060 | 0.002 | -0.016 | 0.000 | -0.003 | 0.001 | -0.023 | 0.001 |
| Gt | -0.391 | 0.046 | -0.108 | 0.071 | -0.108 | 0.040 | -0.831 | 0.106 | 0.081 | 0.000 | 0.279 | 0.000 | 0.573 | 0.064 |
| Assign | -0.196 | 0.008 | -0.181 | 0.009 | 0.061 | 0.002 | 0.183 | 0.000 | 0.000 | 0.000 | -0.218 | 0.000 | 0.017 | 0.004 |
| Store | -0.282 | 0.010 | -0.289 | 0.011 | 0.069 | 0.005 | 0.029 | 0.000 | -0.075 | 0.000 | -0.326 | 0.000 | 0.133 | 0.007 |
| Sub | -0.913 | 0.000 | -1.905 | 0.167 | -0.138 | 0.000 | -0.057 | 0.031 | 0.295 | 0.154 | -0.141 | 0.021 | 0.026 | 0.000 |
| Load | 0.542 | 0.011 | 0.225 | 0.009 | 0.041 | 0.002 | 0.075 | 0.001 | -0.017 | 0.000 | -0.003 | 0.001 | -0.024 | 0.001 |
| Lt | -0.707 | 0.024 | -0.023 | 0.030 | -0.140 | 0.000 | 0.194 | 0.000 | 0.209 | 0.052 | -0.138 | 0.000 | 0.279 | 0.000 |
| Mult | 0.697 | 0.607 | -2.091 | 0.360 | -0.162 | 0.000 | -0.241 | 0.000 | 0.477 | 0.340 | -1.702 | 0.165 | -0.535 | 0.000 |
| NotIn | -1.033 | 0.000 | 0.741 | 1.141 | 0.004 | 0.000 | 0.917 | 0.481 | -0.873 | 0.000 | -0.464 | 0.000 | -1.161 | 0.300 |
| Import | -0.089 | 0.000 | 0.468 | 0.000 | 0.997 | 0.150 | -1.128 | 0.031 | -0.250 | 0.088 | -0.353 | 0.000 | -1.093 | 0.000 |
| NotEq | -0.176 | 0.000 | -0.418 | 0.100 | -0.061 | 0.043 | 0.467 | 0.000 | 0.009 | 0.000 | -0.534 | 0.000 | -0.167 | 0.000 |
| While | -0.945 | 0.000 | -0.531 | 0.000 | -0.515 | 0.000 | 0.357 | 0.158 | -0.907 | 0.197 | 0.633 | 0.548 | 0.635 | 0.624 |
| Subscript | -0.441 | 0.006 | -0.269 | 0.008 | 0.171 | 0.000 | 0.091 | 0.000 | -0.097 | 0.000 | 0.063 | 0.000 | 0.043 | 0.000 |
| JoinedStr | N/A | 0.000 | -0.210 | 0.000 | -3.949 | 1.000 | N/A | 0.000 | -0.836 | 0.000 | 0.368 | 0.000 | -0.739 | 0.010 |
| alias | -0.089 | 0.000 | -0.219 | 0.000 | 0.485 | 0.000 | -0.816 | 0.000 | -0.250 | 0.088 | -0.297 | 0.000 | -1.069 | 0.000 |
| Slice | -0.043 | 0.000 | -0.159 | 0.000 | -0.357 | 0.000 | 0.061 | 0.000 | -0.764 | 0.000 | -0.167 | 0.000 | -0.759 | 0.086 |
| UnaryOp | 0.146 | 0.000 | -0.045 | 0.000 | -0.049 | 0.000 | -0.210 | 0.000 | -0.199 | 0.000 | -0.293 | 0.000 | -0.647 | 0.000 |
| USub | -0.753 | 0.000 | 0.006 | 0.000 | -0.263 | 0.000 | -0.193 | 0.000 | -0.185 | 0.000 | -0.177 | 0.000 | -0.256 | 0.000 |
| FloorDiv | -0.731 | 0.000 | -0.844 | 0.000 | -0.723 | 0.000 | -0.246 | 0.000 | -0.877 | 0.000 | 0.160 | 0.000 | 0.053 | 0.000 |
| Not | 1.000 | 46.230 | -0.906 | 0.000 | -0.231 | 0.000 | -0.904 | 0.000 | -1.190 | 0.000 | -Inf | 0.000 | -0.956 | 0.000 |
| Pow | -1.124 | 0.000 | N/A | 0.000 | N/A | 0.000 | N/A | N/A | N/A | N/A | N/A | 0.000 | N/A | N/A |

## B. AST NODE DESCRIPTIONS

This appendix includes descriptions for all AST nodes used in Figure 5, along with an example usage of the given node in a code piece. In all examples, language elements that are represented by the given node are given in italics.

| Name | Description | Code Example |
|---|---|---|
| arg | Each single function argument. | f(*a*, b) |
| arguments | Argument list for a function. | f(*a, b, c*) |
| Dict | Declaration of a Dictionary object. | *{"a":1, "b": 2}* |
| FunctionDef | A function definition. | *f(a, b): ...* |
| Module | The Python module that is the root node for all ASTs. | N/A |
| Tuple | Declaration of a Tuple object. | *(1, 2, 3)* |
| BoolOp | Expressions with the boolean operations 'Or' or 'And'. | *x or y* |
| Compare | Expressions with a comparison operation. | *x < y* |
| Or | The boolean operator 'Or'. | x *or* y |
| In | The comparison operator 'In'. | 'x' *in* ['x', 'y'] |
| For | The control flow structure 'For'. | *for x in y: ...* |
| Return | The return statement in a function. | def f(a, b): *return 4* |
| And | The boolean operator 'And'. | x *and* y |
| If | The control flow structure 'If'. | *if x: ... / else: ...* |
| Eq | The comparison operator 'Eq' (equal). | x == y |
| Constant | Constant literal objects. | *123* |
| Call | A function call. | x = *f(a)* |
| Mod | The binary operator for modulus. | 15 *%* 3 |
| GtE | The comparison operator 'GtE' (greater than or equal to). | 5 >= 3 |
| Expr | An expression that is not assigned to any variable. | *-5* |
| BinOp | Expressions with a binary operator. | *5 - 3* |
| Div | The binary operator for division. | 10 */* 2 |
| LtE | The comparison operator 'LtE' (less than or equal to). | 3 < 5 |
| List | Declaration of a List object. | *[1, 2, 3]* |
| Attribute | Accessing the attribute of an object. | *object.name* |
| Add | The binary operator for addition. | 1 *+* 2 |
| Name | A named variable. | *a* |
| Gt | The comparison operator 'Gt' (greater than). | 1 > 2 |
| Assign | An assignment. | *a = 5* |
| Store | A context node that shows the value of a variable is being modified. | *a* = 5 |
| Sub | The binary operator for subtraction. | 5 - 2 |
| Load | A context node that shows the value of a variable is being accessed. | b = *a* |
| Lt | The comparison operator 'Lt' (less than). | 1 < 2 |
| Mult | The binary operator for multiplication. | 5 * 3 |
| NotIn | The comparison operator 'NotIn'. | 2 *not in* [4, 5, 6] |
| Import | An import statement. | *import ast* |
| NotEq | The comparison operator 'NotEq' (not equal). | 5 *!=* 3 |
| While | The control flow structure 'While'. | *while True: ...* |
| Subscript | The indexing operator for a collection. | *my_list[0]* |
| JoinedStr | A f-string (formatted string). | *f"Hello {name}"* |
| alias | The aliasing operator 'as'. | import x *as* y |
| Slice | The slicing operator for a collection. | *my_list[1:3]* |
| UnaryOp | Expressions with an unary operator. | *not x* |
| USub | The unary operator for minus sign (-). | -5 |
| FloorDiv | The binary operator for floor division. | 10 *//* 2 |
| Not | The unary operator for negation. | *not* x |
| Pow | The binary operator for exponentiation. | 5 **** 3 |