srcML-DKT: Enhancing Deep Knowledge Tracing with Robust Code Representations from srcML

Maciej Pankiewicz
University of Pennsylvania
Warsaw University of Life Sciences
mpank@upenn.edu

Yang Shi
Utah State University
yang.shi@usu.edu

Ryan S. Baker
University of Pennsylvania
ryanshaunbaker@gmail.com

ABSTRACT

Knowledge Tracing (KT) models predicting student performance in intelligent tutoring systems have been successfully deployed in several educational domains. However, their usage in open-ended programming problems poses multiple challenges due to the complexity of the programming code and a complex interplay between syntax and logic requirements embedded in code development. As a result, traditional Bayesian Knowledge Tracing (BKT) and more advanced Deep Knowledge Tracing (DKT) approaches that use binary correctness data find limited use. Code-DKT [26] is a knowledge tracing approach that uses recurrent neural networks to model learning progress leveraging information extracted from the student-generated code, incorporating abstract syntax tree (AST)-based code features, but its reliance on parsable code limits its effectiveness; unparsable submissions may constitute a substantial part of code submitted for evaluation within platforms for automated assessment of programming assignments. To overcome the ASTs limitations, we propose srcML-DKT, an extension of Code-DKT that utilizes srcML-based code representations, enabling feature extraction from both parsable and unparsable code. By capturing syntactic and structural details directly from the code text, srcML-DKT enables including all student code submissions, regardless of syntax errors. Empirical evaluations on a dataset of 610 students and six programming tasks focused on conditional statements demonstrate that srcML-DKT consistently outperforms both Code-DKT and traditional DKT models, achieving higher AUC and F1-scores across first and all attempts. These results highlight the model's ability to track student knowledge progression more accurately, in environments where trial-and-error learning is common.

Keywords

Knowledge Tracing, CS1, Programming, Deep Knowledge Tracing, Deep Learning.

1. INTRODUCTION

Knowledge tracing has been used in intelligent tutoring systems (ITS) to monitor and support student learning across various domains [21,23,26]. By capturing how students progress in mastering specific skills, knowledge tracing models have helped educators personalize instruction and feedback [30]. A key aspect of these models is the concept of knowledge components (KCs [14])—discrete units of knowledge or skills—used to track student mastery at a granular level [6]. While certain subjects such as algebra have well-established KCs [23], defining and identifying KCs in programming education remains more elusive due to the complex and multifaceted nature of coding tasks [7,24,27].

In response to these challenges, machine learning (ML) approaches for knowledge tracing in programming have been proposed. One such approach is Code-DKT [26], which leverages Abstract Syntax Trees (ASTs) to represent student code submissions [16]. ASTs can capture structural and syntactic features of code, making them suitable for modeling student competencies and predicting performance on future tasks. However, Code-DKT's reliance on AST generation comes with a key limitation: since the standard AST construction process requires parsable code, any unparsable submissions are excluded. This is a non-trivial concern in educational settings. Simply discarding these submissions risks favoring more advanced students who can already write error-free code, and neglecting novice learners who often struggle with syntax and other basic mistakes.

Given that novices potentially benefit the most from targeted instructional support in introductory programming courses, there is a need to develop a knowledge tracing model that can incorporate the full range of code submissions, including code that cannot be parsed into an AST. To address this gap, we propose srcML-DKT, an approach that integrates all code submissions via srcML [5]. Unlike previous AST generation techniques that fail when code does not parse, srcML can represent code structure more flexibly, accommodating code structure from incomplete or erroneous submissions across multiple programming languages (C, C++, C# and Java). By incorporating these submissions, srcML-DKT can provide a more comprehensive representation of student knowledge states, improving the model's applicability to more code submissions.

In the following sections, we describe details of our approach. We first summarize related work on knowledge tracing and AST-based methods in programming education. Next, we describe the technical underpinnings of srcML-DKT, illustrating how srcML is integrated into the workflow to handle all code submissions. We then present our experimental setup and results, highlighting the improvements achieved by including these overlooked submissions. Finally, we discuss future implications and outline the next steps for advancing knowledge tracing in programming education.

2. CONTEXT OF THE STUDY

2.1 Compiler Errors

A compiler error occurs when code fails to adhere to the syntactic rules of a programming language, preventing it from being translated into an executable program. These errors detected by a compiler are typically accompanied by messages indicating where the code deviates from the required syntax.

Novice programmers frequently encounter compiler errors as they navigate the complexities of learning syntax. This stage is critical in programming education, as many researchers have emphasized that overcoming these early challenges significantly impacts future

Maciej Pankiewicz, Yang Shi, and Ryan S. Baker. srcML-DKT: Enhancing Deep Knowledge Tracing with Robust Code Representations from srcML. In Caitlin Mills, Giora Alexandron, Davide Taibi, Giosuè Lo Bosco, and Luc Paquette (eds.) Proceedings of the 18th International Conference on Educational Data Mining, Palermo, Italy, July, 2025, pp. 541–548. International Educational Data Mining Society (2025).

https://doi.org/10.5281/zenodo.15870306

success in computer science [2,3,12,13]. Gaining the ability to interpret and fix compiler errors builds a foundation for tackling more advanced programming tasks, while repeated struggles can hinder progress, erode confidence, and limit skill development. And for novices this may be a challenging task. A study on the use of an online programming tool for a CS1 course reported that more than one-third of student submissions failed to compile [20]. An even higher failure rate was observed in classroom settings – a study analyzing compilation events from CS1 laboratory sessions found that 60% of compilation attempts were unsuccessful [28].

2.2 Abstract Syntax Trees

Abstract syntax trees (ASTs) have long served as an instrument for analyzing the structure and evolution of programming code in educational contexts [16]. Researchers have employed AST-based methods to investigate programming style, detect plagiarism, and even offer automated feedback on student submissions [22]. However, generating an AST requires code that can be parsed successfully. As a result, researchers often discard non-compilable submissions when assembling large-scale datasets for analysis [7,17,25,32]. This limitation is particularly pronounced in novice programming contexts, where syntax errors are frequent, and code rarely compiles on the first attempt. Disregarding this code means losing a significant portion of real-world submissions, thus creating a gap in our understanding of how learners grapple with fundamental syntax rules.

2.3 srcML

srcML [5] is an XML-based representation of source code designed to facilitate the analysis, transformation, and querying of program structure and semantics. By encoding syntactic elements of code into a hierarchical XML format, it enables advanced tools and algorithms to process source code while preserving its original structure. This approach has been used in software engineering research and applications, including program comprehension, refactoring, and code generation [8,29]. srcML supports languages commonly used in introductory programming courses, such as C, C++, C#, and Java, making it a suitable tool for educational and academic purposes. srcML represents source code while preserving all original text, including comments and whitespace. Unlike strictly compiler-oriented tools, srcML can accommodate incomplete or erroneous code, making it especially valuable for analyzing novice submissions.

2.4 Challenges in Knowledge Tracing for Programming Education

Bayesian Knowledge Tracing (BKT) is one of the foundational approaches to modeling student learning [6]. It predicts whether a student has mastered specific skills based on their performance over time. BKT uses a probabilistic framework with the following parameters: guess (success without mastery), slip (failure despite mastery), and the probability of transitioning from unlearned to learned states. While effective in some domains [23], traditional BKT models treat student responses as binary (correct or incorrect), which limits their applicability to complex problem-solving domains like programming. These domains often require more nuanced analysis to account for variations in student understanding reflected in their solution attempts.

Deep Knowledge Tracing (DKT) is a neural network-based approach that initially attempted to improve predictive accuracy, while still requiring binary correctness. Introduced by [21], DKT employs long short-term memory (LSTM) networks – a more complex variant of RNNs – to capture sequential dependencies in students' learning trajectory. The model processes sequences of student attempts, encoded as problem-response pairs, and outputs a probability distribution indicating the likelihood of a correct response on the next attempt. By leveraging deep learning, DKT identifies complex, nonlinear patterns in student learning, offering a more flexible and scalable framework for modeling educational trajectories. Recent advancements in DKT, including attention-based mechanisms [11,18], have enhanced the model's predictive accuracy by incorporating richer contextual features, such as problem text and student-generated responses. DKT, in contrast to BKT, automatically discovers relationships among problems, and no longer needs a pre-specified knowledge component model. However, it still relies on binary correctness indicators, missing the information embedded in student responses, which may be valuable in such domains as programming education. Here, the nature of incorrect or correct solutions can reveal essential details about a student's conceptual understanding or misconceptions.

The recently proposed Code-based Deep Knowledge Tracing (Code-DKT) [26] addresses these limitations by incorporating domain-specific features from students' code submissions. Code-DKT extends the DKT framework by analyzing the actual content of programming solutions, providing a richer representation of student knowledge. It achieves this by utilizing the code2vec model [1], which represents programming solutions as abstract syntax trees and extracts paths between AST nodes to create meaningful embeddings. These embeddings are weighted using an attention mechanism informed by the correctness of prior submissions, allowing the model to identify the most predictive aspects of the code.

Code-DKT's integration of code content marks a departure from traditional domain-general approaches. While DKT and its variants are effective on large datasets with simpler problems (e.g., multiple-choice, or short-answer questions), they fail to capture the intricacies of domains like programming. By incorporating the structural and semantic features of code submissions, Code-DKT enhances the predictive accuracy of KT models, enabling them to account for subtleties like syntax errors, conceptual misunderstandings, or inefficient logic, which binary correctness data cannot capture.

An empirical evaluation of Code-DKT demonstrated its advantages over traditional BKT and DKT models [26]. In a study involving 410 students across five introductory programming assignments, Code-DKT consistently outperformed DKT by 3-4% in AUC ROC (Area Under the Receiver Operating Characteristic Curve) across all assignments. This improvement is on par with advancements achieved by state-of-the-art domain-general models, like SAINT [4] and SAKT [18], in other domains. Code-DKT is particularly effective in identifying patterns of student learning when problems share common programming constructs or learning objectives, although its performance can vary for novel or unique tasks.

Despite these advancements, challenges remain. Code-DKT, like other deep models, requires considerable data to learn effectively, which can be a limitation in smaller datasets. Additionally, while the inclusion of code features improves predictions, further research is needed to optimize the representation and integration of such features. Exploring alternative code representation models, like CodeBERT [9] or ASTNN [15], may provide further enhancements. Additionally, as we investigate here, Code-DKT cannot analyze student solutions if the AST cannot be created. Nevertheless, Code-DKT demonstrates the potential of combining domain-specific features with deep learning to improve knowledge tracing,

particularly in domains where problem-solving complexity extends beyond binary correctness.

3. METHODS

3.1 Platform for Automated Assessment of Programming Assignments

The study utilized a programming code dataset from an online platform, RunCode [19], which provides an automated environment for the execution and testing of programming code. RunCode has been actively used since 2017 by computer science students at a large European University. Although usage of the platform was voluntary and performance did not count toward the final grade, it remained highly popular among students, with a participation rate of around 90% over the years. The platform features a diverse set of a few hundred programming assignments, covering fundamental programming topics: types and variables, conditional statements, recursion, loops, arrays, and bitwise operations, but also more advanced topics around object-oriented programming. Students submit their programming code via an online code editor.

3.2 Assessment Process

The platform's assessment process involves compiling and testing the submitted code. Students receive comprehensive feedback to support their learning and debugging, which includes:

- Compiler messages: detailed information on errors, including the line number, error message, and error ID. Code editor highlights the lines, where compiler errors occurred.
- Unit test results: unit tests are executed to verify if the code logic is correct. Outcomes for each unit test executed on the submitted code are presented as a list with green or red marks requiring clicks to access detailed feedback. Detailed feedback includes input values, the expected output, and the actual output generated by student code. A test fails if the value generated by a student code does not equal the expected output.
- Overall score: a percentage (0–100%) representing the proportion of successfully passed unit tests.

This detailed feedback is provided to allow students to iteratively refine their solutions and deepen their understanding of programming concepts.

3.3 Dataset

The dataset originates from a study that involved first-semester computer science students enrolled in an Introduction to Programming (CS1) course, conducted during the Winter Semester at a large European university over four academic years (2020–2024). The course, which uses C# as the programming language, is a mandatory component of the computer science curriculum. The dataset comprises submissions from N = 610 students on six programming assignments designed to practice conditional statements. These assignments were completed on an automated assessment platform, progressively increasing in complexity by introducing new operators and logical constructs (Table 1). Consent was obtained from students prior to joining this study. Due to the COVID pandemic, in academic year 2021-2022 the course was fully online, and the assignments were made available during the semester, as topics were introduced weekly during online sessions

In this paper, we analyze a selected set of tasks focusing on conditional statements. These tasks were selected for the following reasons: 1) Foundational yet approachable: Conditional statements are the first topic that introduces logic into programming. Unlike more complex topics, such as loops, it may be easier to interpret underlying "knowledge components". 2) High frequency of syntax errors: Therefore an approach that can parse solutions with syntax errors could be valuable. 3) Syntax errors beyond slips: At this early stage, many syntax errors likely stem from misconceptions rather than accidental slips, offering valuable insights into student understanding. 4) Incremental difficulty: Each task incrementally builds on previous ones. The gradual progression in task complexity allows for clearer interpretation of student learning patterns across consecutive attempts.

Table 1. Summary of the number of students and total submissions to the selected "if" tasks – categorized as non-compiling (NC), compiling but incorrect and correct – across the years 2020-2023.

	2020	2021	2022	2023
Submissions	2329	1052	1626	1465
NC	828 (36%)	247 (23%)	221 (14%)	190 (13%)
Incorrect	310 (13%)	121 (12%)	373 (23%)	374 (26%)
Correct	1191 (51%)	684 (65%)	1032 (63%)	901 (61%)
Students	189	109	161	155

The six assignments were:

T1: Create a function that returns true if the input number is zero and false otherwise (*Key concepts: equality comparison*).

T2: Create a function that returns true if the input number is positive and false otherwise (*Key concepts: greater-than comparison*).

T3: Create a function that returns true if the input number is even and false otherwise (*Key concepts: equality comparison, modulo operator*).

T4: Create a function that returns true if the input number is not divisible by 3 and false otherwise (*Key concepts: equality comparison, modulo operator*).

T5: Create a function that returns true if the input number is both positive and even. This task builds on earlier functions by requiring students to combine conditions using logical operators (*Key concepts: logical AND, modulo operator, equality comparison, greater-than comparison*).

T6: Create a function that returns true if the input number is positive, even, and not divisible by 3. Students must integrate multiple conditions and reuse previously defined functions (*Key concepts: logical AND, modulo operator, equality comparison, greater-than comparison*).

These tasks were designed to progressively challenge students by requiring them to apply foundational programming concepts and build on their earlier work to solve more complex problems. Across these tasks, we recorded **6,472 submissions**, of which **2,018 were non-compiling** and **4,454 were successfully compiled and tested**. 3,808 (59%) submissions were correct (Table 2).

3.4 srcML-DKT

srcML-DKT is an approach to extending the deep knowledge tracing architecture with student code, which addresses the limitations of abstract syntax tree (AST)-based code representations, particularly when dealing with these non-compilable student submissions that cannot be parsed into an abstract syntax tree. The original Code-DKT approach [26] leverages AST-based representations of student code, which relies on successfully parsing code into complete syntax trees, which can be problematic when students submit non-compilable code. The majority of the non-compilable submissions in our dataset are also non-parsable (66%) and therefore AST generation for such submissions fails.

Table 2. Summary of the number of students and total submissions for selected 'if' tasks (T1–T6), categorized as non-compiling (NC), compiling but incorrect, and correct.

	T1	T2	T3	T4	T5	T6
Submissions	1356	1063	968	918	1073	1094
NC	635 (47%)	252 (24%)	297 (31%)	157 (17%)	368 (34%)	309 (28%)
Incorrect	39 (3%)	162 (15%)	34 (3%)	133 (15%)	99 (9%)	179 (16%)
Correct	682 (50%)	649 (61%)	637 (66%)	628 (68%)	606 (57%)	606 (56%)
Students	604	594	580	578	570	564

An example of the AST represented for a code snippet is shown in Figure 1 (a). In Code-DKT, code paths are extracted as code features, and an example of a code path is shown in red in Figure 1. They are represented as sequences of text and embedded as vectors for next-step processing. For example, in the red code path of Figure 1 (a), the path is represented as `['input', 'method', 'body', 'doSomething', 'input']`.

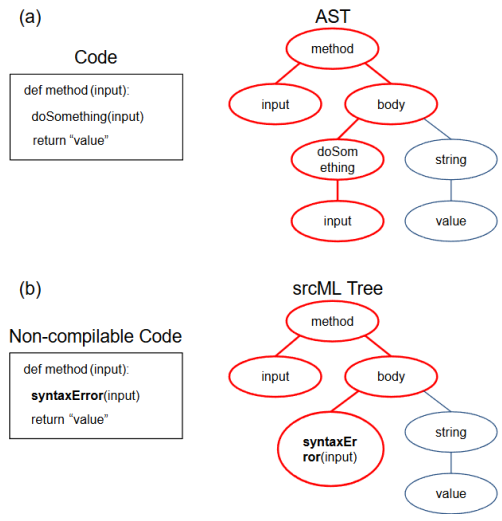

Figure 1. Example of AST (a) and srcML tree (b) and the extracted code path features for Code-DKT specified in red. In srcML, uncompilable code can be represented in an XML format even if it is unparsable and thus represented as a tree for srcML-DKT feature extraction.

To overcome the issue with AST generation process not parsing code, srcML-DKT replaces AST-based code feature extraction with a srcML-based approach, which provides a more flexible and robust representation of student submissions, regardless of whether they can be parsed. An example of a non-compilable, unparsable code and the corresponding srcML tree is shown in Figure 1 (b). In case the code cannot be parsed due to a syntax error in the second line (bolded), srcML still captures both the syntactic and structural elements of source code preserving faulty parts' plain textual format, allowing for a comprehensive analysis of both complete and incomplete programs. srcML parses programming code into XML format, which can be further transformed into a tree (similarly to AST, but without strict syntax rules, where the AST generation would fail). This representation ensures that the learning model has access to meaningful structural features even in the presence of syntax errors, improving its ability to track students' evolving knowledge states.

srcML-DKT leverages srcML-based features instead of the AST-based features in the knowledge tracing framework. It encodes the structural information of student code at each attempt. Each code submission c_t at submission t has a set of R_t codepaths (example codepath is marked red in Figure 1), and similar to Code-DKT, they are embedded as vectors through attention mechanisms [31]. These code features ($\mathbf{[z_1, z_2, \ldots, z_T]}$) are combined with student prior correctness traces (denoted as $\mathbf{[x_1, x_2, \ldots, x_T]}$), and are then processed using a long short-term memory (LSTM) architecture to model the temporal progression of student learning [10,21]. Note that in case of programming assignments, a single problem may involve different skills across multiple attempts. Since correct solutions are not strictly defined, we follow the approach used in Code-DKT and consider every attempt in this model for tracing [26]. By incorporating a more flexible code representation, srcML-DKT enhances the applicability of deep learning-based knowledge tracing in programming education, making it more resilient to non-compilable code, allowing it to more fully capture the progression of student learning.

4. RESULTS

In this section, we present results obtained for the srcML-DKT model and compare it to the Code-DKT and DKT models.

4.1 Hyperparameter tuning

The analysis was conducted by partitioning the dataset at the student level into training, validation, and testing sets in a 3:1:1 ratio that also ensured that submissions from the same student did not appear in multiple sets. The validation set was used for hyperparameter tuning, where ten runs were performed to identify the optimal parameter configuration. For each run, we calculated the AUC (ROC) metric and for each configuration of hyperparameter set we averaged obtained AUCs to find a configuration that maximizes the averaged AUC. Specifically, the learning rates tested included 0.0001, 0.0005, 0.001, 0.005, and 0.01, while the number of epochs varied within the range of 80 to 180 in increments of 20. Once the best-performing hyperparameters were established on the validation set, the model was evaluated on the test set 10 times to assess its final performance. For each of these 10 runs, the model was randomly initialized, trained from scratch using the training dataset, and then evaluated on the test dataset. This approach accounts for variability in neural network training and ensures that the reported results are averaged over multiple runs for a more reliable performance estimate. The final AUC/F1 performance of models are reported as the average of the 10 runs.

Table 3 presents the performance comparison of srcML-DKT, Code-DKT, and the baseline DKT model, evaluated using AUC and F1-score metrics for both first attempts and all attempts. srcML-DKT consistently outperforms both Code-DKT and DKT across all evaluation metrics, demonstrating the efficacy of incorporating srcML-based code representations into the knowledge tracing framework. For first attempts, srcML-DKT achieves an AUC of 0.8355 and an F1-score of 0.8278, representing an improvement of 1.65 percentage points in AUC and 0.53 percentage points in F1-score over Code-DKT (AUC = 0.8190, F1 = 0.8225). Compared to the baseline DKT model (AUC = 0.7931, F1 = 0.8132), srcML-DKT exhibits a substantial gain of 4.24 percentage points in AUC and 1.46 percentage points in F1-score. This indicates that the srcML-based feature extraction method provides richer and more reliable information about student knowledge, particularly in predicting performance on new, unseen problems.

Similarly, when considering all attempts, srcML-DKT achieves the highest performance with an AUC of 0.8467 and an F1-score of 0.8053. This reflects an improvement of 1.61 percentage points in AUC and 0.88 percentage points in F1-score over Code-DKT (AUC = 0.8306, F1 = 0.7965), and a 2.90 percentage point increase in AUC and 1.32 percentage point improvement in F1-score compared to DKT (AUC = 0.8177, F1 = 0.7921). These results highlight srcML-DKT's enhanced capability to model student learning trajectories.

Table 3. Performance Comparison of srcML-DKT, Code-DKT, and DKT Using AUC and F1-Score for First and All Attempts.

	First attempts		All attempts	
	AUC	**F1**	**AUC**	**F1**
srcML-DKT	**0.8355**	**0.8278**	**0.8467**	**0.8053**
Code-DKT	0.8190	0.8225	0.8306	0.7965
DKT	0.7931	0.8132	0.8177	0.7921

Overall, the consistent performance gains observed in srcML-DKT across both first and all attempts suggest that srcML-based code feature extraction not only addresses the limitations of ASTs in handling non-compilable code but also provides a more robust and generalizable approach to knowledge tracing in programming education.

4.2 Case Study: The Impact of Non-Compiling Code Features on srcML-DKT Predictions

To investigate how incorporating features from non-compiling code submissions improves predictive performance, we conducted a case study comparing srcML-DKT, Code-DKT, and DKT models.

4.2.1 Prediction Visualization

We utilized prediction heatmaps to visualize model performance. In these heatmaps, each cell represents a specific student's attempt on task T4 (after initially solving tasks T1, T2, and T3 in 5 total attempts: two non-compilable and 3 compilable submissions). The numerical value inside the cell indicates the ground truth outcome—1 for a correct (successful) submission and 0 for an incorrect (unsuccessful) one. The cell color corresponds to the model's predicted probability of success: green indicates a higher likelihood of a correct submission, while red denotes a lower probability of success.

Figure 2 presents the prediction heatmaps generated by the three models for this student who made four submission attempts on a task. This example was selected to highlight how incorporating features from non-compiling code may impact prediction accuracy. In this case, the first three submissions were incorrect and non-compilable—a common scenario among novice programmers. Only the fourth attempt was compilable – it was also the final and correct submission.

4.2.2 Code Evolution Across Attempts

Alongside the heatmaps, we analyzed the student's code modifications (pairs of consecutive submissions with changes tracked in a git-style presented on Figure 2) to understand the sequence of corrections made by this student while creating a function to check if a number is divisible by 3.

Attempt 1: The initial submission resulted in a compiler error due to a missing semicolon (compiler reports one error).

Attempt 2: The student added the missing semicolon, but the code still failed to compile (compiler reports one error) due to a misunderstanding of the `/=` operator. In C#, `/=` is a division assignment operator, dividing the left-hand operand by the right-hand operand and assigning the result back to the left-hand operand (e.g., `x /= y` is equivalent to `x = x / y`).

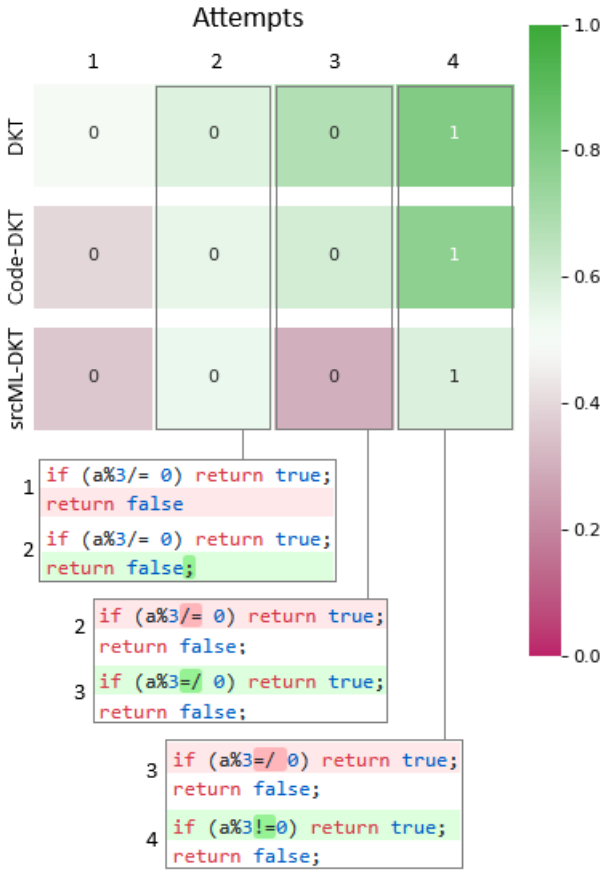

Figure 2. Heatmap Comparison of DKT, Code-DKT, and srcML-DKT Predictions with Annotated Student Code Changes Across Four Consecutive Attempts on a Task T4.

Attempt 3: The student attempted to fix the error by changing `/=` to `=/`, which is syntactically incorrect, causing another compilation failure (compiler reports one error).

Attempt 4: The student correctly replaced `=/` with the inequality operator `!=`, resulting in a compilable and correct submission.

4.2.3 Model Predictions and Analysis

Initial predictions for the first attempt of the student on task T4 varied across models. The DKT model, which does not utilize code features, was overly optimistic, predicting a high probability of success (indicated by light green). In contrast, both Code-DKT and srcML-DKT, which incorporate code-based features, correctly predicted the student's failure on the first attempt. This highlights how even basic code feature inclusion can improve predictive accuracy for novice programmers prone to syntax errors.

As the student progressed through subsequent attempts, the predicted probability of success increased for all models. However, key differences emerged: DKT showed a gradual increase in success probability but lacked the contextual understanding of code correctness, leading to less accurate predictions in earlier attempts. Code-DKT, while leveraging code features, could not utilize non-compiling submissions, limiting its ability to learn information about the student's progress when the student's code failed to compile. srcML-DKT demonstrated superior performance by incorporating features from non-compiling code. Notably, it correctly predicted the failure on the third attempt, where the syntax error (`/=`) persisted. This illustrates the model's enhanced ability to interpret student performance and learning, even when submissions contain compilation errors

This case study demonstrates that integrating non-compiling code features significantly enhances prediction accuracy in knowledge tracing models. srcML-DKT's ability to process incomplete or erroneous code may provide a more nuanced understanding of student learning trajectories, particularly in early programming education where syntax errors are prevalent.

5. DISCUSSION

Programming poses unique challenges for knowledge tracing due to the multi-layered nature of errors, which can occur at the syntax level (code does not compile), runtime level (code compiles but crashes during execution), or logic level (code runs but produces incorrect results). Additionally, when solving programming assignments, students often require multiple compilation attempts before arriving at a correct solution. These complexities make modeling student knowledge in programming environments particularly difficult and previous research has shown that traditional Bayesian Knowledge Tracing and more advanced Deep Knowledge Tracing approaches struggle with data from programming assessment platforms. Code-DKT – a model that integrates features extracted from student code has been found to outperform both BKT and DKT, but it does not include code features if the AST cannot be generated. To address this limitation of the Code-DKT method, we explored the use of srcML as a structured representation of programming code and selected a set of tasks for evaluation. In our investigation, we focused on tasks involving conditional statements, a fundamental concept introduced early in programming courses. While these tasks are relatively simple, many novices struggle with syntax, making these tasks a suitable choice for this initial analysis.

In this paper, we introduced the srcML-DKT approach, hypothesizing that improved code feature extraction from both parsable and unparsable code can enhance knowledge tracing performance. We leveraged the srcML format to generate code features for all submissions, including all non-compiling ones, ensuring that the full range of student attempts contributes to model training. We evaluated our approach on a subset of conditional statement tasks, demonstrating improved performance over Code-DKT and DKT in terms of the AUC and F1 metrics. Additionally, we investigated a set of submissions from a student who struggled with the material in detail, finding that incorporating features from the full range of code submissions improved prediction accuracy. This suggests that our model better captures the nuances of novice programmers' submissions and is more responsive to students who experience difficulties. These findings highlight the robustness of the srcML-DKT approach and its potential value in introductory programming courses, where students frequently experience wheel spinning or struggle with syntax comprehension. By accounting for non-compiling submissions, our model offers a more comprehensive view of student learning, with the potential to ultimately supporting more effective interventions for struggling learners.

The tasks selected for this analysis all focused on the concept of conditional statements, which enhances the interpretability of our results. We assume that the model can learn from submissions made on assignments covering the same concept. However, its performance when a student transitions to a different topic—such as moving from conditional statements to recursion or loops—has not yet been evaluated. Investigating this aspect is a key next step in our research. Although students were free to choose the order in which they completed tasks on the platform, most followed the sequence in which the tasks were listed. This behavior may have also influenced model performance, and further analysis is needed to assess the robustness of our approach. Fortunately, our dataset allows for such an evaluation, as the platform supports randomized task ordering for individual students, and this feature was enabled for selected modules. Examining model performance under these conditions is also part of our planned future work.

Our study has several limitations. The dataset used in this research includes submissions from N=610 students, which, while larger than some previous studies, remains relatively small for deep learning applications. This limited sample size may affect the generalizability of our findings, particularly when applying the model to more diverse student populations or different educational contexts. Future research should focus on validating srcML-DKT's performance using larger datasets to ensure broader applicability and robustness. The scope of the tasks in our dataset is constrained, consisting of only six assignments, all centered around conditional statements. In addition, this model tracks every submission from students, and this is different from the typical setting of DKT. This is because of the nature of open-ended programming. Students may practice different skills in the same assignment, and the detailed definition of skills in open-ended programming is still under discussion [24,27]. While this focus allowed for a controlled exploration of the model's capabilities, it limits the ability to generalize the findings to other programming concepts, such as loops, functions, or data structures. As a next step, we plan to further investigate the model's performance across a wider variety of programming tasks and concepts to more fully assess its effectiveness and usefulness.

This study suggests that adopting a more robust approach to code feature generation may improve knowledge tracing for programming education. We hope that as a result it will ultimately provide benefits to introductory courses, supporting more effective and tailored interventions for struggling learners, ultimately fostering greater success in early CS education.

6. REFERENCES

1. Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL. https://doi.org/10.1145/3290353
2. Brett A. Becker. 2016. A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 296–301. https://doi.org/10.1145/2899415.2899463
3. Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 177–210. https://doi.org/10.1145/3344429.3372508
4. Youngduck Choi, Youngnam Lee, Junghyun Cho, Jineon Baek, Byungsoo Kim, Yeongmin Cha, Dongmin Shin, Chan Bae, and Jaewe Heo. 2020. Towards an Appropriate Query, Key, and Value Computation for Knowledge Tracing. In *Proceedings of the Seventh ACM Conference on Learning @ Scale* (L@S '20), 341–344. https://doi.org/10.1145/3386527.3405945
5. Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance*, 516–519. https://doi.org/10.1109/ICSM.2013.85
6. Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction* 4, 4: 253–278.
7. Mehmet Arif Demirtas, Max Fowler, and Kathryn Cunningham. 2024. Reexamining Learning Curve Analysis in Programming Education: The Value of Many Small Problems. In *Proceedings of the 17th International Conference on Educational Data Mining*, 53–67. https://doi.org/10.5281/zenodo.12729774
8. Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering* 27, 3: 63. https://doi.org/10.1007/s10664-022-10118-5
9. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139
10. Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8: 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
11. Shuyan Huang, Zitao Liu, Xiangyu Zhao, Weiqi Luo, and Jian Weng. 2023. Towards robust knowledge tracing models via k-sparse attention. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2441–2445.
12. Matthew C. Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, 73–84. https://doi.org/10.1145/1151588.1151600
13. Matthew C. Jadud and Brian Dorn. 2015. Aggregate Compilation Behavior. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, 131–139. https://doi.org/10.1145/2787622.2787718
14. 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: 757–798.
15. Ye Mao, Yang Shi, Samiha Marwan, Thomas W Price, Tiffany Barnes, and Min Chi. 2021. Knowing When and Where: Temporal-ASTNN for Student Learning Progression in Novice Programming Tasks. *14th International Conference on Educational Data Mining, EDM 2021*: 172–182.
16. Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, 1–5.
17. Linus Östlund, Niklas Wicklund, and Richard Glassey. 2023. It's Never too Early to Learn About Code Quality: A Longitudinal Study of Code Quality in First-year Computer Science Students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (SIGCSE 2023), 792–798. https://doi.org/10.1145/3545945.3569829
18. Shalini Pandey and George Karypis. 2019. A self-attentive model for knowledge tracing. In *12th International Conference on Educational Data Mining, EDM 2019*, 384–389.
19. Maciej Pankiewicz. 2020. Measuring task difficulty for online learning environments where multiple attempts are allowed — the Elo rating algorithm approach. In *Proceedings of the 13th International Conference on Educational Data Mining, EDM 2020*, 648–652.
20. Maciej Pankiewicz and Ryan S Baker. 2024. Navigating Compiler Errors with AI Assistance - A Study of GPT Hints in an Introductory Programming Course. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (ITiCSE 2024), 94–100. https://doi.org/10.1145/3649217.3653608
21. Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. In *Advances in Neural Information Processing Systems*, 505–513. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2015/file/bac9162b47c56fc8a4d2a519803d51b3-Paper.pdf
22. Chris Piech, Mehran Sahami, Daphne Koller, Steve

Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (SIGCSE '12), 153–160. https://doi.org/10.1145/2157136.2157182

23. Steven Ritter, John R Anderson, Kenneth R Koedinger, and Albert Corbett. 2007. Cognitive Tutor: Applied research in mathematics education. *Psychonomic Bulletin & Review* 14, 2: 249–255. https://doi.org/10.3758/BF03194060

24. 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), 143–151. https://doi.org/10.1145/2960310.2960333

25. Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27: 37–64.

26. Yang Shi, Min Chi, Tiffany Barnes, and Thomas Price. 2022. {Code-DKT}: A Code-based Knowledge Tracing Model for Programming Tasks. In *Proceedings of the 15th International Conference on Educational Data Mining*, 50–61. https://doi.org/10.5281/zenodo.6853105

27. 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*.

28. Emily S Tabanao, Ma. Mercedes T Rodrigo, and Matthew C Jadud. 2011. Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the Seventh International Workshop on Computing Education Research* (ICER '11), 85–92. https://doi.org/10.1145/2016911.2016930

29. Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How Software Refactoring Impacts Execution Time. *ACM Trans. Softw. Eng. Methodol.* 31, 2. https://doi.org/10.1145/3485136

30. Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational psychologist* 46, 4: 197–221.

31. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, \Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (NIPS'17), 6000–6010.

32. Mengxia Zhu, Siqi Han, Peisen Yuan, and Xuesong Lu. 2022. Enhancing programming knowledge tracing by interacting programming skills and student code. In *LAK22: 12th International Learning Analytics and Knowledge Conference*, 438–443.