Intent-KG Review: A Knowledge Graph for Region-Grounded Program Intent Analysis

Rikuto Sasaki[1], Ren Shimoide[1] and Kazuhiro Takeuchi[1]

[1]Osaka Electro-Communication University, Neyagawa, Osaka 572-8530, Japan

Abstract

Large language models (LLMs) allow learners to generate programs and obtain explanations with little effort, but being able to use a program does not necessarily mean understanding how that program solves a problem. Such understanding requires knowledge of how individual code elements contribute to intermediate sub-goals and how those sub-goals, in turn, contribute to the overall solution. Similarly, when an LLM is asked to explain a program from a particular perspective or to implement a specific function, appropriate context and framing may be necessary. This paper proposes an intent model for a non-LLM analysis module that provides explicit program context to an LLM as a step toward automated code review. Here, intent does not refer to a programmer's psychological intention; rather, the model describes, as a knowledge graph, the roles that code Regions are expected to play within a solution. We refer to this graph as an Intent Knowledge Graph (IKG). We construct this model without relying on LLM interpretation, instead combining non-LLM processing modules grounded in programming-education and code-analysis research. The resulting model is not primarily intended to generate human-readable explanations; instead, it represents acquired relations between code Regions and local problem-solving roles, functional sub-goals, problem requirements, and solution plans. Experiments on CodeWorkout problems characterize how this structural and intent knowledge is acquired incrementally. We further apply the resulting non-LLM analyzer to Java programs not used during IKG acquisition, instantiate Region-grounded IKG paths for the selected Regions, and use these paths as context for relation generation by a commercial LLM. In this exploratory evaluation, the IKG context increased mean pairwise Jaccard similarity from 0.546 to 0.689 and mean three-run consensus from 0.383 to 0.583, suggesting that explicit program-intent context can reduce variation in the perspective and granularity of subsequent LLM analysis.

Keywords

Knowledge Graph, Code Review, Intent Knowledge, Programming Education

CEUR-WS.org/Vol-4316/pd-short-7.pdf

1. Introduction

Large language models (LLMs) have become familiar support tools in programming education. Learners can provide a problem statement to generate code, or submit their own code and ask for an explanation. While such use makes executable code examples and natural-language explanations readily available, these explanations do not necessarily make explicit how local code elements participate in the problem-solving structure of a program. For example, an explanation may describe what an `if` statement or loop does locally without clarifying which functional sub-goal it supports, which problem requirement that sub-goal addresses, or how it contributes to the overall solution plan.

Figure 1 illustrates the distinction addressed in this work. In programming education, knowledge tracing (KT) primarily models learners' mastery of knowledge components from their problem-solving histories, typically representing the knowledge required for programming tasks as identifiable components, although relations among such components may also be modeled. Toward model-driven automated code review, we develop a non-LLM analysis module that draws on prior programming research to represent Region-grounded problem-solving intent as a hierarchy distinct from the program's syntactic structure. Program intent represents what a selected part of the program contributes to the solution, from a local problem-solving role to a functional sub-goal and its relation to a problem requirement.

IJCKG 2026 Posters, Demos and Workshops, November 19–21, 2026, Bangkok, Thailand

takeuchi@osakac.ac.jp (K. Takeuchi)

0009-0006-4307-1767 (K. Takeuchi)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-06

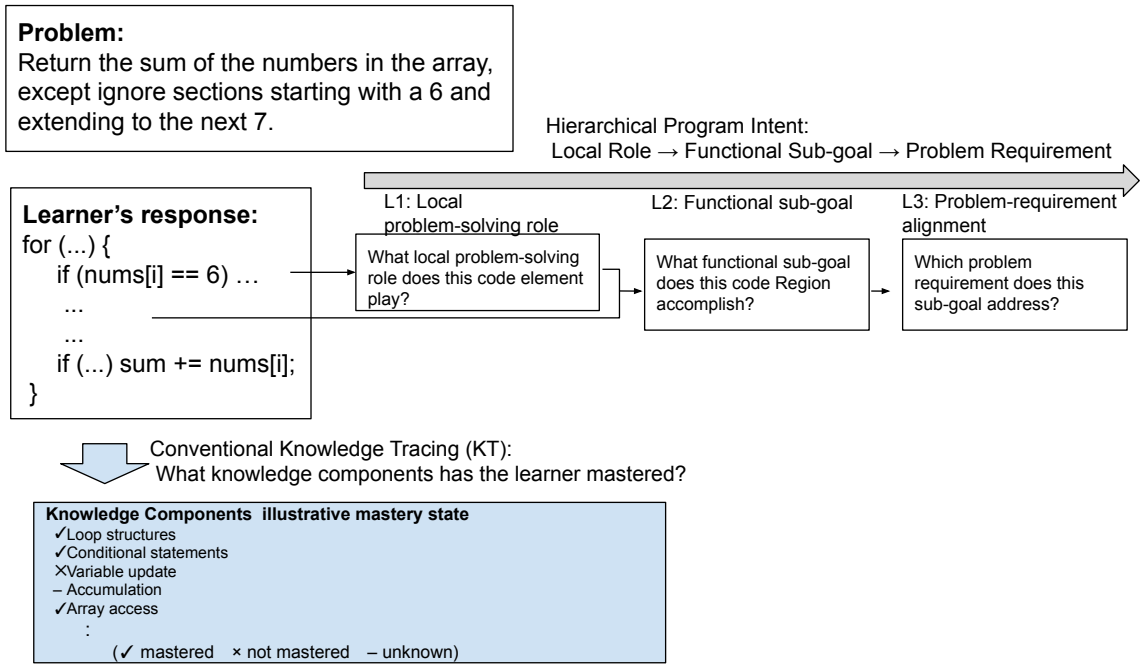


Figure 1: Comparison between knowledge tracing and hierarchical program-intent analysis. The IKG represents acquired intent relations across L1 local roles, L2 functional sub-goals, and L3 problem-requirement alignments.

This paper makes three contributions. First, we define a five-level Region-grounded program-intent model connecting parser-derived structure to local roles, functional sub-goals, problem requirements, and solution plans. Second, we present a non-LLM process that incrementally acquires these relations as an IKG. Third, we apply the resulting analyzer to post-acquisition Java programs and evaluate the generated Region-grounded IKG paths as context for structured LLM-based relation generation.

2. Region-Grounded Program Intent Model

In this study, program intent is treated neither as a mere functional classification of code elements nor as the programmer's internal psychological state. Passonneau and Litman studied intention-based discourse segmentation, in which discourse segments are characterized in terms of communicative intention and related to observable linguistic cues [1]. Motivated by the idea of identifying intentional structure from observable evidence, we operationally define program intent for source code as an evidence-grounded answer to a question about the role or purpose of a specific code Region. For a code Region r, we represent program intent as

$$I(r) = \{(q, a, e) \mid q \in Q(r),\ a = A(q, r, C, K),\ e \in E(C, K),\ e \models a\}.$$

Here, q is a question about the role or purpose of Region r, a is an answer derived from the code structure C and relevant knowledge K, and e is structural or knowledge-based evidence that supports the answer. K includes both problem-side knowledge and knowledge already acquired in the IKG. Program intent is therefore represented not as unconstrained natural-language explanation, but as an explicit knowledge unit that connects a question about a code region with an answer and the evidence supporting that answer.

In programming education, knowledge tracing (KT) has developed as a framework for estimating learners' mastery of knowledge components (KCs) from their problem-solving histories and for predicting future performance. Recent programming KT methods increasingly incorporate information from problem statements and submitted code in addition to sequences of correct and incorrect responses. For example, DPKT represents problem-text understanding difficulty and KC difficulty using an LLM

Table 1
Non-LLM analysis and Q/A mechanisms, generated relations, and prior-work foundations of the code-side IKG levels. Problem requirements are represented separately and are linked to code-side sub-goals at L3.

Level	Component	Meaning or generation mechanism
L0	Focus	Syntactic classification of code evidence. The implicit question is: *What syntactic construct does this Region instantiate?*
	Analysis and representation	Source code is parsed using Tree-sitter Java version 0.23.5 [3]; Java SE 21 is used as the fixed normative language-specification reference [4]. The parser output provides typed nodes, source spans, and nesting information, which are deterministically normalized using a CodeOntology-inspired vocabulary [5].
L1	Focus and Q	Local problem-solving roles: *What local problem-solving role does this code element play?*
	Q/A generation	Deterministic rules interpret parser-derived structural evidence together with initialization, update, condition, and control contexts, and generate `code_element has_role local_role`. Variable-role categories are informed by Roles-of-Variables research, whereas control-, condition-, and return-role rules are operational extensions introduced in this study [6].
L2	Focus and Q	Functional sub-goals: *What functional sub-goal does this code Region accomplish?*
	Q/A generation	Explicit composition rules detect combinations of acquired L1 roles within a problem solution and construct functional sub-goal candidates. Each candidate is grounded in one or more supporting Regions through the L1 evidence used in its construction. The use of functional sub-goals is informed by subgoal-labeled worked-example research; the deterministic composition rules are our operationalization [7, 8].
L3	Focus and Q	Alignment with problem-side knowledge: *Which problem requirement does this sub-goal address?*
	Q/A generation	Deterministic rules align an L2 sub-goal with independently represented problem requirements and generate relations such as `realizes_requirement`, `guards_requirement`, `contributes_to_requirement`, and `uses_requirement_property`. This level is informed by the problem–KC/Q-matrix perspective of KC-Finder, while its neural KC-discovery model is not executed [9].
L4	Focus and Q	Ordering and dependency among sub-goals: *How do these sub-goals work together to form the solution?*
	Q/A generation	A plan candidate is constructed only when its required acquired sub-goals are present. Project-defined deterministic rules use the heuristic sequence `guard/state/traverse/check/update/return` to generate `precedes_or_enables` dependencies. The interpretation of solution planning is informed by programming problem-solving and self-regulation research, whereas the specific ordering rules are our operationalization [10, 11].

and CodeBERT and incorporates these representations into learner knowledge-state prediction [2]. Our work is related to this direction in that it also uses information from programming problems and source code, but analyzes it at the level of individual code Regions and their roles in problem solving.

The objective, however, is different. Rather than predicting learner states, we investigate how far local program intent can be analyzed without LLM-generated interpretations. We operationalize prior analysis methods as explicit intent-analysis modules and compose them into a staged pipeline from syntactic structure to solution planning.

We organize the analysis into five practical levels, from parser-grounded structure (L0) through local roles (L1), functional sub-goals (L2), problem-requirement alignment (L3), and solution planning (L4). Table 1 summarizes their analysis mechanisms, generated relations, and prior-work foundations.

At L0, the question corresponding to syntactic classification is implicit. For L1–L4, the Question Generator specifies the relation to be examined, and the Answer Generator deterministically derives a candidate relation and supporting evidence from code structure, problem-side knowledge, and previously acquired relations. Thus, Q/A serves as a non-LLM knowledge-acquisition interface for representing explicit program-intent relations in the IKG.

3. Non-LLM Intent Knowledge Graph Acquisition

The mechanisms described in Section 2 are not used as independent classifiers. Instead, they are composed into a staged acquisition process that constructs an IKG from a problem statement and its corresponding source code. A central design property is that the acquisition of L0–L4 does not use LLM-generated interpretations. The acquired relations are derived from problem-text evidence, parser-derived code evidence, and explicit deterministic rules that operationalize the analytical foundations described in Section 2. Consequently, the resulting IKG can be maintained independently of LLM-generated explanations and used as explicit context for subsequent LLM-based analysis.

Let each code instance be represented as

$$d_i = (p_i, c_i), \tag{1}$$

where p_i is a problem statement and c_i is its corresponding program code. Let B_t denote the set of instances introduced at acquisition stage t, and let

$$G_t = (V_t, E_t) \tag{2}$$

denote the IKG after completing that stage.

At L0, the corresponding question is implicit. For each parser-derived Region, the system identifies the syntactic construct instantiated by that Region and records the resulting structural classification. Let $G_{t,0}$ be the graph obtained by adding the L0 structural classifications derived from B_t to G_{t-1}.

For L1–L4, the levels are processed sequentially. At level l, a Question Generator R_l specifies the type of knowledge relation to be examined from the graph available after processing the lower levels:

$$Q_t^l = R_l(B_t, G_{t,l-1}), \qquad l \in \{1, 2, 3, 4\}. \tag{3}$$

An Answer Generator Φ_l then derives a structured answer candidate from the same available evidence:

$$A_t^l = \Phi_l(Q_t^l, B_t, G_{t,l-1}), \qquad l \in \{1, 2, 3, 4\}. \tag{4}$$

For L1–L4, Φ_l is entirely non-LLM and uses parser-derived structure, problem-side knowledge, and lower-level acquired relations. Its output is treated as a candidate knowledge relation rather than unconstrained explanatory text.

An acquisition operator H_l performs schema-level validation of the Q–A result and converts it into new graph relations:

$$\Delta E_t^l = H_l(Q_t^l, A_t^l, B_t, G_{t,l-1}), \qquad l \in \{1, 2, 3, 4\}. \tag{5}$$

Here, schema-level validation checks the relation type, evidence references, and applied rule identifier, and merges duplicate candidates; it does not denote human validation of semantic correctness. The graph is updated within the stage as

$$G_{t,l} = G_{t,l-1} \oplus \Delta E_t^l, \qquad l \in \{1, 2, 3, 4\}, \tag{6}$$

where $\oplus$ inserts the validated relations and their endpoint nodes while preserving existing graph elements. The graph after acquisition stage t is therefore

$$G_t = G_{t,4}. \tag{7}$$

This formulation makes explicit that an Ll candidate may use the previously accumulated graph and the lower-level candidates already acquired at the same stage, but not relations that have yet to be generated at higher levels.

Among the 50 CodeWorkout [12] problems analyzed during IKG design, 47 with available source code were introduced in the recorded acquisition order. For each problem, at most one working and

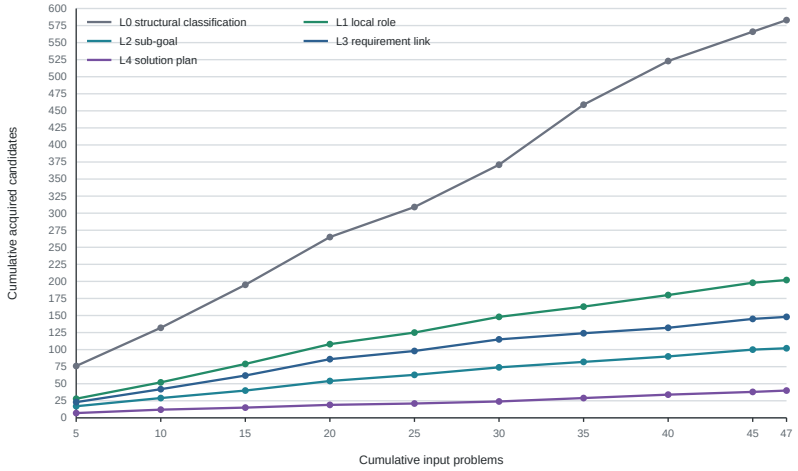


Figure 2: Cumulative acquisition of L0–L4 candidates from 47 CodeWorkout problems using the non-LLM pipeline.

correct code example was used. Figure 2 shows the cumulative L0–L4 candidates acquired by the non-LLM analysis modules. Because acquisition order can affect when a relation first appears, the curve characterizes the recorded order rather than order-invariant convergence.

The larger growth at L0 mainly reflects parser-derived structural Regions, whereas only a subset contributes to higher-level intent relations. Plateaus at higher levels indicate only that no additional candidates were generated under the current rules and recorded order; they do not imply IKG completeness or convergence.

This experiment characterizes structural and intent-knowledge accumulation, not learner performance or IKG quality. Section 4 applies the resulting analysis framework to code not used during acquisition.

4. Demo and Exploratory Evaluation

The interface synchronizes the problem statement, source code, parser-derived structural Regions, and the Region-grounded intent path. Each gutter marker indicates the highest IKG level reachable from a Region, while Regions for which no higher-level relation has been acquired remain explicitly visible.

In Fig. 3, the condition on line 6 is selected. Its highest directly grounded intent level is L1, and it is connected, through an L2 sub-goal and L3 requirement alignment, to an L4 solution plan. The labels “direct: L1” and “connected: L4” distinguish the highest level directly grounded in the selected Region from higher-level knowledge reachable through the IKG. All displayed links are produced by the non-LLM analyzer; the interface neither infers intent levels from Region size nor fills missing links with generated explanations.

As an exploratory test of the module’s downstream use, we evaluate whether adding Region-grounded IKG context makes an LLM select more consistent Region relations across repeated runs. As shown in the L1–L4 entries of Table 2, the supplied IKG path provides higher-level intent context, while the LLM must still derive the requested Region-to-Region relations from the code.

The evaluation used ten selected correct Java programs created after IKG acquisition; none was used during acquisition. They form a transfer-oriented feasibility set rather than a balanced sample of Java programs or naturally occurring learner submissions. For each program, one Region was selected and evaluated under the two conditions in Table 2. The model generated at most three relations using the controlled predicates `controls`, `guards`, `feeds`, `terminates`, `enables`, and `completes`. Three independent runs were performed for each condition using `gpt-5.6-sol`, with a fresh ephemeral context for every run. The evaluation therefore comprised 60 responses (10 programs $\times$ 2 conditions $\times$

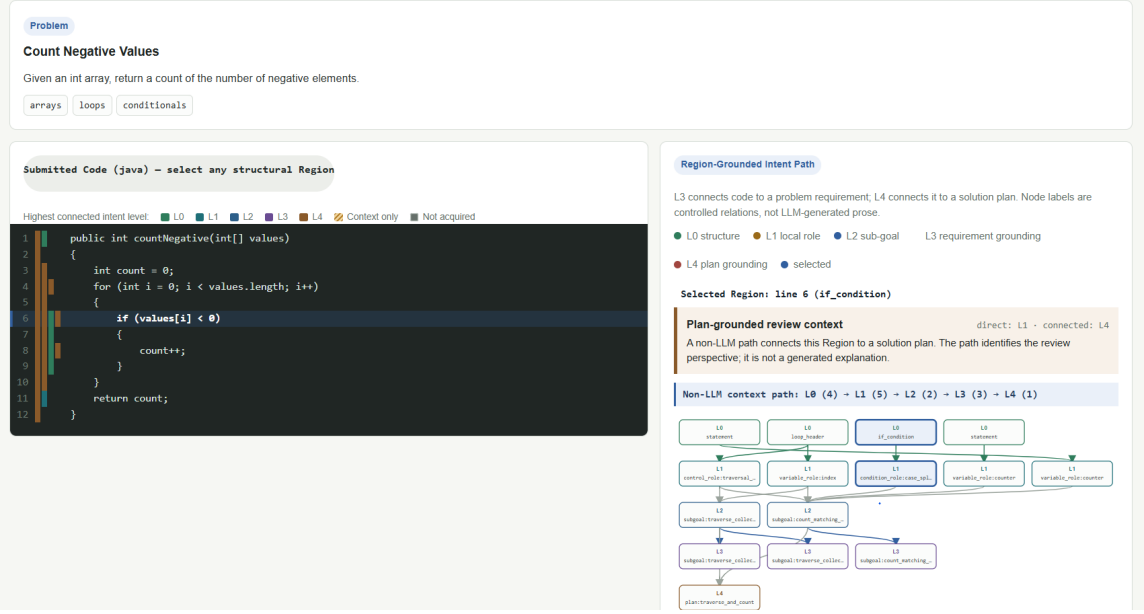


Figure 3: Demo interface for a post-acquisition program not used in IKG acquisition. The selected condition has L1 as its highest directly grounded intent level and is connected through L2 and L3 knowledge to an L4 solution plan. The interface exposes this non-LLM context without generating a natural-language explanation.

3 runs).

Natural-language wording in the `statement` field was excluded from scoring. Each valid relation was normalized as a tuple consisting of the sorted unique source Region set, a controlled predicate, and the sorted unique target Region set. For each program and condition, the three runs produced relation sets $S^{(1)}, S^{(2)}, S^{(3)}$. We measured repeatability using the mean pairwise Jaccard similarity and three-run consensus:

$$J = \frac{1}{3}\sum_{i<j}\frac{|S^{(i)} \cap S^{(j)}|}{|S^{(i)} \cup S^{(j)}|}, \qquad C = \frac{|\bigcap_{j=1}^{3} S^{(j)}|}{|\bigcup_{j=1}^{3} S^{(j)}|}.$$

The reported values are macro-averages over the ten programs. Stable and union relation counts are obtained from the corresponding per-program intersections and unions. Schema-level grounding validity requires non-empty source and target Region sets, a predicate from the controlled vocabulary, and Region IDs present in the supplied Region map. All generated relations satisfied these constraints, and no run produced an empty relation set. This validation does not establish semantic correctness.

As shown in Table 3, the selected IKG context increased mean pairwise Jaccard from 0.546 to 0.689 and mean three-run consensus from 0.383 to 0.583. Relations appearing in only one run decreased from 17 to 11. Six of the ten programs improved on both repeatability measures, three were unchanged, and one decreased. The decreased case was `containsCat`, for which pairwise Jaccard changed from 0.556 to 0.167 and three-run consensus from 0.333 to 0. For *Count Negative Values*, all three Code + Region runs produced `R6 guards R7`, but one run additionally produced the coarser relation `R6 controls R5`. With IKG context, all three runs produced only `R6 guards R7`. Thus, the principal local relation was already identifiable without the IKG; the observed difference was reduced variation in the granularity and perspective of the generated relations. These results do not establish semantic correctness or show that the IKG contains knowledge unavailable to the LLM. Rather, they provide exploratory evidence that a Region-grounded IKG path can constrain the analysis perspective and make generated relations more convergent for the selected transfer cases. The demo therefore illustrates both the transferable reach of the current non-LLM intent analysis and its use as context for subsequent LLM-based analysis.

Table 2
Prompt structure for the two evaluation conditions.

Code + Region (baseline)	Code + Region + selected IKG path
Problem statement Given an int array, return a count of the number of negative elements.	**Problem statement** Same.
Java implementation `countNegative(...)`	**Java implementation** Same.
Region information Complete Region map (`R0–R8`), including type and containment.	**Region information** Same.
Selected Region `R6: if_condition`	**Selected Region** Same.

Additional IKG context
None.
	Additional IKG context
	`L1: R6 has_role condition_role:case_split`
	`L2: R0 accomplishes count_matching_cases`
	`L3: count_matching_cases realizes_requirement count_matching_elements`
	`L4: plan:traverse_and_count traverse_collection precedes_or_enables count_matching_cases`

Review instruction Review only how the selected Region participates in the program's higher-level behavior.	**Review instruction** Same.
Output constraint Identify at most three important behavioral relations involving the selected Region.	**Output constraint** Same.
Allowed predicates `controls, guards, feeds, terminates, enables, completes`	**Allowed predicates** Same.
Region constraint Use only Region IDs in the supplied map, preferring the most specific supported Regions.	**Region constraint** Same.
Output format `source_regions + predicate + target_regions + statement`	**Output format** Same.

Table 3
Repeatability of Region-relation generation over ten post-acquisition programs. Stable and union counts are summed over the ten per-program comparisons.

Metric	Code + Region	IKG context
Mean pairwise Jaccard	0.546	0.689
Mean three-run consensus	0.383	0.583
Stable relations / union	8/32	12/25
Relations appearing once	17	11
Schema-level grounding validity	100%	100%

5. Conclusion

We presented Intent-KG Review, a non-LLM program-intent analysis module developed as a step toward model-driven automated code review. The L0–L4 IKG is constructed without LLM-generated interpretation by combining deterministic analysis modules grounded in prior work.

The CodeWorkout experiment characterized how these structural and intent relations accumulate as additional problem and code evidence is introduced. The demonstration further showed that the

resulting analysis framework can be applied to post-acquisition programs to generate structured context for subsequent LLM-based Region-relation generation. The exploratory evaluation suggests that such context can improve the repeatability of generated relations, while not establishing their semantic correctness.

Declaration on Generative AI

ChatGPT was used to translate text originally drafted by the authors in their native language into English, and for linguistic polishing and proofreading. The authors reviewed the final manuscript and take full responsibility for its content.

Acknowledgments

This research was partially supported by JSPS KAKENHI Grant Number JP23K28152.

References

[1] R. J. Passonneau, D. J. Litman, Discourse segmentation by human and automated means, Computational Linguistics 23 (1997) 103–139. URL: https://aclanthology.org/J97-1005/.

[2] L. Yang, X. Sun, H. Li, R. Xu, X. Wei, Difficulty aware programming knowledge tracing via large language models, Scientific Reports 15 (2025) 11475. doi:10.1038/s41598-025-96540-3.

[3] Tree-sitter contributors, Tree-sitter Java, GitHub repository, 2026. URL: https://github.com/tree-sitter/tree-sitter-java.

[4] Oracle, The Java Language Specification, Java SE 21 Edition, Oracle documentation, 2023. URL: https://docs.oracle.com/javase/specs/jls/se21/html/index.html.

[5] M. Atzeni, M. Atzori, CodeOntology: RDF-ization of source code, in: The Semantic Web – ISWC 2017, Springer, 2017, pp. 20–28. doi:10.1007/978-3-319-68204-4_2.

[6] J. Sajaniemi, M. Kuittinen, An experiment on using roles of variables in teaching introductory programming, Computer Science Education 15 (2005) 59–82. doi:10.1080/08993400500056563.

[7] L. E. Margulieux, B. B. Morrison, A. Decker, Design and pilot testing of subgoal labeled worked examples for five core concepts in CS1, in: Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Association for Computing Machinery, 2019, pp. 548–554. doi:10.1145/3304221.3319756.

[8] L. E. Margulieux, B. B. Morrison, A. Decker, Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples, International Journal of STEM Education 7 (2020) 19. doi:10.1186/s40594-020-00222-7.

[9] Y. Shi, R. Schmucker, M. Chi, T. Barnes, T. W. Price, KC-Finder: Automated knowledge component discovery for programming problems, in: Proceedings of the 16th International Conference on Educational Data Mining, International Educational Data Mining Society, 2023, pp. 28–39. doi:10.5281/zenodo.8115671.

[10] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, M. M. Burnett, Programming, problem solving, and self-awareness: Effects of explicit guidance, in: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, 2016, pp. 1449–1461. doi:10.1145/2858036.2858252.

[11] D. Loksa, A. J. Ko, The role of self-regulation in programming problem solving process and success, in: Proceedings of the 2016 ACM Conference on International Computing Education Research, Association for Computing Machinery, 2016, pp. 83–91. doi:10.1145/2960310.2960334.

[12] S. H. Edwards, K. P. Murali, CodeWorkout: Short programming exercises with built-in data collection, in: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, Association for Computing Machinery, 2017, pp. 188–193. doi:10.1145/3059009.3059055.