

Assessing the Understanding of Expressions: A Qualitative Study of Notional-Machine-Based Exam Questions

JOEY BEVILACQUA, Software Institute, Università della Svizzera italiana, Switzerland

LUCA CHIODINI, Software Institute, Università della Svizzera italiana, Switzerland

IGOR MORENO SANTOS, Software Institute, Università della Svizzera italiana, Switzerland

MATTHIAS HAUSWIRTH, Software Institute, Università della Svizzera italiana, Switzerland

Background and Context. Notional machines are widespread in computing education. While they often are used to explain programming concepts, prior work on visual program simulation demonstrated their use for assessment. This paper presents a qualitative study of the use of a notional machine identified in prior literature—“expression as tree”—as an instrument to assess the understanding of expressions.

Objectives. The study answers the following research questions: What are the mistakes that students make when answering questions based on the “expression as tree” notional machine? What underlying reasons plausibly explain these mistakes? Are there plausible relationships between the mistakes and programming language misconceptions?

Method. We collect a corpus of 542 hand-drawn expression tree diagrams from 12 exams in 6 university programming courses at two different levels over the course of 4 years. We devise and use a coding approach tailored to the qualitative analysis of those diagrams. Our qualitative data analysis approach is unique due to the specific form of the data—hand-drawn diagrams—that admits a wide variety of mistakes, due to the theoretically well-defined programming language constructs that underpin the visualizations, and due to the fact that for each diagram there exists a unique solution that can serve as a reference.

Findings. Our results show that even a single question based on the notional machine is able to reveal a rich variety of mistakes related to the structure, typing, and evaluation of expressions. We identify and categorize 48 mistakes and describe outstanding ones in detail, supported by example diagram snippets. We find mistakes that are plausibly connected to known misconceptions, and others that suggest the existence of new ones.

Implications. Our findings shed light on novel pedagogical strategies to teach programming and provide insights instructors can use to transform their practices. The use of the “expression as tree” notional machine as an assessment instrument can provide valuable insights into students’ understanding of several key aspects of programming.

CCS Concepts: • **Social and professional topics** → **Computer science education**.

Additional Key Words and Phrases: Notional Machines, Assessment, Programming, Expressions

ACM Reference Format:

Joey Bevilacqua, Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2024. Assessing the Understanding of Expressions: A Qualitative Study of Notional-Machine-Based Exam Questions. In *24th Koli Calling International Conference on Computing Education Research (Koli Calling '24)*, November 12–17, 2024, Koli, Finland. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3699538.3699554>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).

Manuscript submitted to ACM

1 INTRODUCTION

When teaching programming, educators often use *Notional Machines (NMs)* [19, 42] to explain the constructs of a programming language. Prior work describing NMs abounds; for example, there is work on NMs focusing on runtime stacks and scoping [15], evaluation by substitution for recursion [48], or object-oriented program execution [5]. Programming teachers may use an entire repertoire of NMs; a recent publication collected 43 NMs used by educators [21].

This paper studies such a NM, which focuses on language constructs that are prevalent but often neglected in educational materials: expressions [13]. The NM is known as “expression as tree”. It focuses on the hierarchical structure, the types, and the evaluation of expression constructs.

While notional machines are often used for *explanatory* purposes, “visual program simulation” systems like UUhistle [43] encourage their use for *assessment* purposes as well. In this paper, we study specifically the use of the “expression as tree” NM for assessment.

The following research questions, inspired by the structure used by Sirkiä and Sorva [41], drive our investigation:

- RQ1** What are the mistakes that students make when answering questions based on the “expression as tree” NM?
RQ2 What underlying reasons plausibly explain key mistakes? Are there plausible relationships between the mistakes and programming language misconceptions?

To answer these questions, we qualitatively analyze 542 expression trees hand-drawn by students in twelve exams of Java programming courses on two different educational levels over four years. Our goal is to provide a foundation on which educators and researchers can design and study new kinds of light-weight assessment items with the “expression as tree” NM to investigate the conceptual understanding of expressions.

2 BACKGROUND AND RELATED WORK

We now discuss related work on the theoretical foundations underpinning this study, on the understanding of expressions in particular, and on notional machines. Then we describe the “expression as tree” NM that is the subject of our study.

2.1 Theoretical Foundations

Theories in computing education research have been characterized [29] as learning sciences and education research theories—such as “cognitive load theory” [44]—or as domain-specific theories of computer science knowledge—such as Nelson et al.’s “formal theory of program tracing knowledge” [30]. Our research is most directly related to the latter kind of theory, the foundational theory underlying programming languages [35, 49]. Unlike natural languages, programming languages are necessarily completely and unambiguously specified. Even in the absence of a formal specification, a language’s compiler or interpreter provides a complete and precise definition of all possible programs in the language. Thus they embody the complete syntax and semantics of the language. However, the implementations of compilers and interpreters are complex—mostly due to their focus on performance—and hard to reason about. To simplify formal reasoning about programming language properties, many mainstream languages have been specified in a theoretical mathematical sense. Such theories exist for example for Java [7], JavaScript [33], and Python [36]. Because even these theoretical specifications still can consist of over a thousand rules, simpler core languages, like Featherweight Java [24] for Java or λ_{JS} [23] for JavaScript, have been developed. Those core languages focus on the most essential language constructs. At the very center of many of these theoretical formalizations sits the lambda calculus [14], a theory of computation and a programming language consisting of only three constructs, all of them expressions.

Unlike many learning science and education research theories, programming language theories represent formal systems, exactly like mathematical theories, which are amenable to formal reasoning and proof. For example, given the rules of a programming language like Java, we can reason unambiguously whether `"Hello".charAt(2)` evaluates to `'l'`. No human interpretation, and no expert opinion is necessary. All that is needed is the mechanical application of the theory, the syntactic and semantic language rules.

2.2 Understanding Expressions

An expression is a term in a program that produces a value. Expressions are written by composing various language constructs, the most basic of which are literals, variable names, and operators. Widely-used languages like Java provide a rich set of expression-related features, such as method calls, type casts, or array accesses, that can be organized into over 25 different expression constructs [13].

Syntax: Understanding Structure. . The source code of a program consists of a sequence of characters. However, correct programs are constrained by the grammatical rules of the programming language. The grammar imposes a tree structure onto the code. While block-based languages like Scratch [37] make that structure explicit, and syntax-directed editors [4, 27, 31] guide the programmer in following that structure, in textual languages that tree structure is invisible. Nevertheless, it is there, and students need to be able to recognize that *structure* in their code [27] in order to properly compose, restructure, and decompose programs.

Static Semantics: Understanding Types. . In a statically typed language such as Java, each expression has a statically-known type [34]. Many misconceptions about types have been documented [12, 47] and various approaches to help students understand types have been described [38]. Understanding the typing of expressions is important: suppose that a student writes the Java expression `Integer.parseInt("10")== "10"`. They try to compile the program, but receive an error: `bad operand types for binary operator ==`. To understand the cause of this error, and to understand how to fix it, students need to be able to reason about the *typing* of expressions.

Dynamic Semantics: Understanding Evaluation. . The purpose of an expression is to be evaluated. Prior work introduced ways to explain the process of expression evaluation: for example, by using ants walking through nested “circles of evaluation” [40], by providing automated visual tutors explaining the evaluation of expressions [26], or by providing tools that allow students to visually simulate program execution [43]. Understanding the evaluation of expressions is important: suppose that a program written by a student contains the expression `o != null & o.m()`. The student intends to avoid invoking the method `m` on `o` when `o` is `null`. They try to execute their program, and they receive a `NullPointerException` error (because the `&` operator always evaluates both sides). To understand and fix this error, students need to be able to reason about the *evaluation* of expressions.

2.3 Notional Machines

The term Notional Machine (NM) was coined in 1981 by du Boulay et al., in the context of teaching Logo, to describe “the idealized model of the computer implied by the constructs of the programming language” [19]. Given the substantial increase in published papers that cite or reference NMs in the mid-2010s, a working group of 12 senior computing education researchers, including du Boulay, published an extensive literature review and analysis of 43 NMs used by educators [21].

NMs sit exactly at the intersection between the theories of content knowledge (programming language theories) and learning sciences and education research theories.

2.4 “Expression as Tree” Notional Machine

“Expression as tree” is one of the NMs described in the aforementioned survey of NMs [21]. It centers around the structure, typing, and evaluation of expressions. The NM is essentially a visualization of the abstract syntax trees that are used to reason about programming languages, both in designing compilers [3] and in the specification of programming language semantics [34]. It is equivalent in structure to pedagogical tools and languages used to teach programming in school, such as the “circles of evaluation” [40]. The visualization resembles that of the structure of expressions in block-based programming languages such as Scratch [28] and Snap! [22]. Those block-based languages can be seen as visual versions of textual syntax-directed editors such as the the Cornell Program Synthesizer [46].

Given the following Java code:

```
public class A {
  public String m(boolean condition) {
    return "condition=" + (condition ? "true" : "false");
  }
}
```

- (1) **Draw an expression tree** for the expression in the return statement of the method.
 - (2) Annotate each node with the **type** of the value produced by the node.
 - (3) Annotate each node with the **value** it produces if the expression is evaluated with `condition = true`.
- Make sure to specify the type of all nodes. Make sure to specify the value of all the nodes that are evaluated.

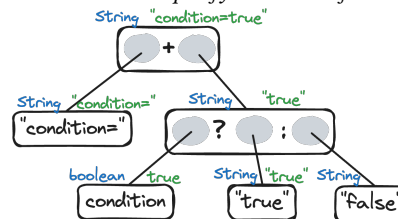


Fig. 1. Exam question E05 and correct answer, using the “expression as tree” NM.

As illustrated in Figure 1, the primary focus of the NM is on the *structure*. The expression is broken down into subexpressions, and each subexpression is represented as a subtree. Nodes can contain “holes” (depicted as little gray circles in Figure 1), which are used as connection points. Edges connect a hole to another node, to indicate that the latter is the root of a subexpression. This representation resembles how expressions are represented in block-based programs; however, the visual “decomposition” of the structure allows each node to be annotated with additional information.

Once the structure is drawn, the *type* of each node is calculated bottom-up¹ and each node is annotated with its type (e.g., in Figure 1 as blue labels at the top left of each node).

Finally, once the expression has been successfully type-checked, one may also *evaluate* the expression. The algorithm for evaluation also proceeds bottom-up, but has a key difference from typing: it does not necessarily traverse all nodes. In the presence of conditional computations (e.g., a short-circuit operator as per Java Language Specification (JLS) 15.23

¹The constructs of Java covered in the studied courses do not require type inference. Determining their types can be done essentially following the typing algorithm described by Pierce [34] for Featherweight Java.

and JLS 15.24, or a conditional expression JLS 15.25), it may skip entire sub-trees. The green labels in Figure 1 exemplify this: the ‘else’ branch of the conditional is not evaluated, and thus no value is shown above the corresponding node.

3 METHODOLOGY

To answer our two research questions, we collected and qualitatively analyzed a corpus of 542 hand-drawn expression trees created by students in different exams of university-level programming courses.

3.1 Context

We studied artifacts from exams because of their authenticity: they were not artificially created for a lab study, but they represent questions that had been asked in a real-world context. Moreover, they were created in a supervised setting, with clear incentives for students to give accurate answers.

The exams took place in six courses at a research university, taught by the same instructor over the course of four years. Three of the courses were second-semester Bachelor courses in object-oriented programming for Computer Science students, and three were courses for existing teachers who got their qualifications to teach computer science in high school. The students in the former were mostly in their first year after graduating from high school, while the students in the latter were existing high school teachers of a multitude of non-CS subjects (STEM and beyond), all with a prior graduate degree in their subject, and some with more than a decade of teaching experience. Because we studied pre-existing artifacts, we do not have detailed demographic information about the students who had produced them.

In all courses, the “expression as tree” NM was used during the course lessons as one of the pedagogical tools to explain expressions, for example to explain nesting and chaining of method calls, or allocating and accessing arrays. Thus, all students acquired familiarity with the notation well before the exam.

All courses had concluded before this research study began. The ethics representative at our university confirmed that we could study past exams, with the caveat of not showing actual student drawings for privacy reasons. We thus refrain from doing that and digitally redraw all the examples shown in this paper.

3.2 Corpus

Our corpus consists of the 542 hand-drawn student answers to all exam questions that used the “expression as tree” NM. Each exam featured only a single question based on this NM, thus each question comes from a different exam.

Table 1 summarizes the 12 exam questions. Together, the questions explore a total of 12 expression constructs. They are the instantiation of classes and arrays (`ClassInstanceCreation` and `ArrayInstanceCreation`), method invocations (`MethodInvocation`), field, array, and variable accesses (`FieldAccess`, `ArrayAccess`, and `SimpleVariableAccess`), operators (`InfixExpression` and `ConditionalExpression`) and literals (`CharacterLiteral`, `NullLiteral`, `NumberLiteral` and `StringLiteral`). These constructs are a subset of all expression constructs included in Java 11, following the naming presented by Chiodini et al. [13].

3.2.1 Question Structure. All exams were paper-based. Students had to draw expression trees in the free space provided on the page showing the exam question. An example of a complete question can be seen for E05 in Figure 1. The question setup was stable throughout the exams, with only minor variations². These differences include the presence of hints on how to handle situations for which the instructor felt that a clarification was needed (e.g., which label to use as

²All the exam question statements are available in the supplementary materials [6].

Table 1. For each question: ID, source code of the expression, number of answers (A), unique language constructs (C), nodes in the expression tree (N), and whether the question required annotating values in addition to drawing the structure of the tree and annotating types. The last row shows the total number of questions (ID), the total number of answers (A), the overall number of unique constructs (C), the total number of tree nodes across all questions (N), and the number of questions requiring value annotations.

ID	Expression Source Code	A	C	N	Values
E01	<code>(d > 1/2)? ("Hi" + '!').length(): new int[s.length()].length + i</code>	8	9	16	
E02	<code>"ar[ix] =" + ((null != ar && ix < ar.length)? (ax[ix] + " "): "no")</code>	13	7	17	✓
E03	<code>gt(a, b)? "a > b" : (a < b ? "a < b" : "a == b")</code>	16	5	11	✓
E04	<code>"a[i] = " + (a != null ? id(a)[i].toString()+ "0" : "X")</code>	19	7	14	✓
E05	<code>"condition=" + (condition ? "true" : "false")</code>	24	4	6	✓
E06	<code>"a[i] = " + ((a != null && a.length > i)? (" " + a[i]): "nothing")</code>	25	7	17	✓
E07	<code>new Runner(v).a().b(c(d))</code>	46	3	6	
E08	<code>"a[i] = " + (a == null ? "X" : id(a[i].toString()))+ '+' + 0</code>	64	9	16	✓
E09	<code>publish(make("this"), make("that"))</code>	71	2	5	✓
E10	<code>1 + rest.len()</code>	84	4	4	
E11	<code>new Cons("A", new Empty()).len()</code>	84	3	4	
E12	<code>height >= 0 ? nameFor(twice(height)): "height < 0"</code>	88	6	8	
12		542	12	124	7

a type for a `NullLiteral`) and the aspects under focus. All the exam questions assessed students on the *structure* and *types* aspects; only some, marked under the ‘Values’ column in Table 1, also assessed *values*.

The example expression tree in Figure 1 includes nodes of four different constructs: one `ConditionalExpression` node, a `SimpleVariableAccess` node (for the variable `condition`), an `InfixExpression` node (with the `+` operator), and three `StringLiteral` nodes (for the literals `"condition="`, `"true"`, and `"false"`).

Some of the expressions in Table 1 may seem too complex. This is just an artifact of using questions from real exams, a choice which otherwise has clear advantages (Section 3.1). The expressions are not intended to resemble typical Java code written by novices; they serve the goal of efficiently assessing the understanding of several programming language constructs (all covered during the courses) in one exam question.

3.2.2 Representativeness of the Corpus. The constructs in our corpus match the top eight expression constructs identified in previous research when studying the prevalence of expression constructs [13, Table 3] in student code extracted from the Blackbox dataset [9]. The only exception is the `Assignment` construct, which does not appear in our corpus as an expression³. In addition to these prevalent constructs, the corpus also includes fundamental literal constructs (`CharacterLiteral` and `NullLiteral`), as well as `ArrayInstanceCreation` (appearing only in one exam) and `ConditionalExpression` (appearing in 8 of the 12 exams), which were found less frequently in the expression prevalence study in Java code written by students.

While understanding how to allocate arrays may reasonably not be considered a high-priority competency [11], the particularly low prevalence of `ConditionalExpression` in the BlackBox dataset (less than 0.1% of expression constructs) indicates that some educators might also consider understanding conditional expressions (`cond ? ifTrue : ifFalse`) less relevant. This is somewhat surprising, because this construct is simply the expression-version of an if-else statement. Conditional expressions are a fundamental construct in programming language design [34]. Their

³Unlike in some other languages, in Java assignments are expressions. Thus they can appear anywhere inside an expression, like in `a + (b = c) + d`. The courses avoided this general use of assignments, and thus refrained from including assignment operators in “expression as trees” exam questions.

evaluation behavior, where only one of the two cases is evaluated, is akin to how if-statements are evaluated. However, conditional expressions are theoretically simpler, because unlike conditional statements, they can be understood without an understanding of mutation and side-effects.

3.3 Qualitative Analysis Approach

Analyzing our corpus of data poses peculiar challenges. On the one hand, the answers are hand-drawn diagrams which leave a lot of room for interpretation and favour an open coding approach. On the other hand, the diagrams represent programming language constructs which are unambiguously defined and thus favor a closed coding approach. The goal of our analysis is to capture all meaningful mistakes, minimizing the chance of missing some relevant ones. Thus we aimed for a coding approach that fulfils the following three requirements:

- (1) Capture every significant deviation of the student’s diagram from a correct diagram;
- (2) Connect observed differences to the related programming language concepts;
- (3) Provide *logical mistakes* instead of merely *physical differences*.

Our approach is related to inductive thematic analysis [8]. However, our context differs from traditional applications of thematic analysis, such as the recent production of a set of tags for Java program construction [10]. In our context, (1) there is a single, known, correct solution to which an artifact can be compared, (2) the constructs used in the artifact are based on an unambiguous theory (the language specification), (3) the data analyzed is of a graphical nature, and (4) the goal is to determine higher-level mistakes.

We had to design an analysis approach that takes into consideration those particularities, integrates inductive and deductive aspects, and enables the analysis of diagrammatical artifacts. Our approach is inspired by Saldaña’s view of coding [39]: “first cycle coding is *analysis*—taking things apart. Second cycle coding is *synthesis*—putting things together into new assemblages of meaning”. Capturing and categorizing the significant deviations is a form of first-cycle coding, while providing logical mistakes can be seen as a form of second-cycle coding.

We proceeded in three phases. The goal of *Phase I* was to find a good structure for coding the diagrams. A five-dimensional structure for codes (detailed in Section 3.4) grew organically during the frequent discussions amongst the coders in this phase. Each coder coded all the answers of at least one exercise. To minimize inter-coder and intra-coder discrepancies, the first phase was followed by a *Consolidation Phase*: each coder analyzed one answer from each exercise already analyzed, including the ones previously coded by the same coder. The discrepancies in codings were discussed and resolved; and as a result, the description of all the codes was improved to increase consistency among coders. The remaining exercises were then coded during *Phase II* without further altering the number of dimensions.

3.4 A Structure for Codes

We recorded every significant deviation of a student’s diagram from the correct diagram using codes with five dimensions: *Aspect - Construct - Operation - Qualification*, and *Mistake*. The first four dimensions describe the physical difference, the last one describes the logical mistake. As a concrete example, the incorrect tree snippet shown on the left of Figure 2 is tagged with the two following codes:

- (1) Structure-SimpleVariableAccess-WasInlined-MethodInvocation, paired with mistake VARIABLEINLINED
- (2) Structure-MethodInvocation-HasInlinedChild-Target, paired with mistake TARGETINLINED.



Fig. 2. Example of an incorrect snippet from a student on the left, and its corresponding snippet from the solution on the right from E01. The target of a MethodInvocation is a subexpression and should therefore be represented with a separate node.

3.4.1 Aspect and Construct. The first two dimensions are related to the programming language: which *aspect* (i.e., structure, types, values) and which *construct* (one of twelve listed in Section 3.2) the code refers to.

3.4.2 Operation and Qualification. The next two dimensions, which emerged during *Phase I* of our process, are loosely related to tree edit distance algorithms. They specify an *operation* (e.g., WasInlined), augmented with a *qualification* (e.g., MethodInvocation), that would fix the observed difference in the given expression tree, turning the incorrect tree into a correct one.

The operations derived from tree edit distance algorithms are deletion (Missing), insertion (Extra), and changing (Replaced) of nodes [50]. These provide a good starting point. However, the structure of diagrams in the NM is more elaborate than that of a simple tree. Thus we include two additional sets of operations. The first set concerns the structure *within* one node. Elements of a node may be (1) “split off” into a child (SplitContent), (2) missing (MissingContent), or (3) extraneous additions (ExtraContent). The second set involves operations on *multiple* nodes. A possible scenario occurs when a node is inlined into its parent, so that the hole of the parent is replaced with the elements of the child (HasInlinedChild and WasInlined). Other scenarios may be described as a “tree rotation” [2] (Rotate) and the relocation of entire subtrees (LostChildRelocated).

We use these operations to describe the *structure*, which is the primary aspect under focus of the NM. Given that *types* and *values* are simply unstructured annotations on the tree, these aspects are coded using three generic operations (Missing, Extra, and Replaced), with the qualification characterizing the replacements.

3.4.3 Mistake. The last dimension of our structured codes is a ‘logical’ mistake, which we recorded whenever we could devise a higher-level characterization of the problem in the tree. A mistake captures semantic information that is not always derivable from the low-level ‘physical’ differences.

3.5 Relationship to Known Coding Methods

Figure 3 relates our coding approach to the underlying established coding methods, as described by Saldaña [39].

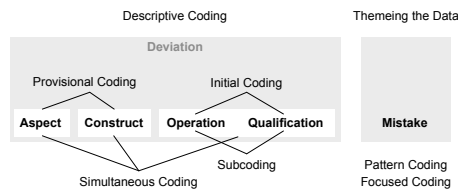


Fig. 3. Our coding approach and its relationship to established coding methods described by Saldaña [39]

To characterize the deviations (*aspect*, *construct*, *operation*, and *qualifications*) from the correct solution, we use DESCRIPTIVE CODING. The *aspect* and *construct* follow the PROVISIONAL CODING (closed coding) method, because the

codes are already known a-priori. The *operation* and *qualification* follow the INITIAL CODING (open coding) method, because the codes emerged from the data during our initial phase. We use SUBCODING to refine the *operations* with *qualifications* (subcodes). Finally, we use SIMULTANEOUS CODING in that the *aspect*, *construct*, and *operation/qualifications* are simultaneous codes describing several aspects of a given deviation.

To move from the descriptive codes of a deviation towards the logical mistakes, we combine the following methods: We use THEMEING THE DATA (Saldaña [39] includes aspects of thematic analysis in this) and PATTERN CODING to group similar deviations. We also use FOCUSED CODING by sorting deviations by their frequencies, their aspects, constructs, operations, and qualifications, to focus our mistake identification on the most frequent and the most salient deviations.

3.6 Connecting Mistakes to Misconceptions

Instructors care about mistakes, because their presence in a tree drawn by a student may be an indication that the student holds one or more misconceptions. In the same way that an illness *causes* a symptom, a wrong belief about the syntax or semantics of the programming language (i.e., a programming language misconception) may *cause* the presence of specific mistakes in an expression tree.

After identifying the mistakes, we identified plausible causal relations between misconceptions and mistakes. Specifically, we considered a publicly available inventory of programming language misconceptions [12]. For this analysis, we perused 74 misconceptions in Java that, at the time of this writing, were either tagged with the concept “expression”⁴ or marked as “expressible in the ‘expression as tree’ NM”⁵, or both. To hypothesize about the causes of the mistakes, we refer to each misconception’s “incorrect statement”. That statement expresses the wrong belief held by a student with such a misconception.

4 RESULTS

We characterized in detail each of the 542 expression trees using the structured coding system described in Section 3. We found 10.1% of the answers to be empty (all nodes were missing). Of the remaining, an additional 6.8% and 6.5% were completely lacking type and value labels, respectively. Because the diagrams were hand-drawn on paper, some of them deviated significantly from the notation. During our analysis, we tried to accommodate variations as far as we could understand them. Of the non-empty answers, 4.9% could not be analyzed for their structure (and consequently also for the types and values); 1.3% and 2.4% were further excluded only from the analysis of types and values, respectively.

Excluding the unassessable cases, we coded 1906 deviations using 404 distinct five-dimensional codes. The number of distinct codes may seem dauntingly large, however, thanks to the structured codes we only needed to develop and use a relatively small number of code components amongst each of the five dimensions: the 3 possible aspects, the 12 expression constructs occurring in the exams, 15 operations, each with possible qualifiers, and 48 mistakes.

4.1 Mistakes (RQ1)

Our qualitative analysis carried out to answer RQ1 revealed 48 mistakes. We organized all the mistakes into a hierarchy, shown in Figure 4. At the top level, we grouped the mistakes according to the three aspects covered by the NM (structure, types, values). The second level uses categories to bring together mistakes that share certain characteristics.

The **Structure** mistakes are categorized into: *Lack of Compositionality* (students did not decompose expressions enough), *Incorrect Decomposition* (students decomposed expressions into too small pieces, which are not expressions

⁴<https://progmiscon.org/concepts/expression>

⁵<https://progmiscon.org/notionalMachines/ExpressionAsTree>

anymore), *Syntactic Mistakes* (other grammatical mistakes), *Extra Content* (tree includes pieces that do not exist in code), and *Missing Content* (tree lacks pieces that exist in code).

The **Types** and **Values** mistakes are categorized into: *Missing* (type or value not specified), *Bad Notation* (incorrect or imprecise notation), and *Typing/Evaluation Mistake* (logical mistake in determining type or value).

4.2 Connection to Misconceptions and Analysis of Key Mistakes (RQ2)

To answer the second research question (RQ2), we analyzed all 48 mistakes and determined their association with previously reported programming language misconceptions [12]. The right side of Figure 4 shows the results. We identified 18 existing misconceptions that are related to the mistakes. Overall, 19 mistakes are connected to at least one misconception. Some mistakes—like `VARIABLEINLINED`—are related to multiple misconceptions, and some misconceptions—like `CALLREQUIRESVARIABLE`—are related to multiple mistakes.

Mistakes that are not related to any misconceptions may represent symptoms of misconceptions that have yet to be studied. For example, the mistakes `METHODNAMESEPARATED` or `TYPENAMESEPARATED` might indicate that students incorrectly consider methods and types to be first-class values in Java.

The rest of this section discusses a subset of five key mistakes in detail. The choice is based on their coverage of all three aspects (structure, types, values) and different constructs, frequency of occurrence, and plausible links to misconceptions.

We describe each key mistake and report one or more exams in which it has occurred, together with the relevant subexpression excerpted from the exam question. We show the snippet of the tree redrawn from the actual answer given by a student and the corresponding snippet from the correct solution, with the differences marked in red.

The discussion of each mistake also includes a section that justifies its importance and an analysis section that reasons about potential causes, including both previously documented and potentially new misconceptions.

***VARIABLEINLINED*. A `FieldAccess` without an explicit target or a `SimpleVariableAccess` is inlined into the parent node.** This mistake describes the inlining of two different Java constructs: `SimpleVariableAccess`, to access a local variable or a parameter, or `FieldAccess`, to access an instance or class field. A `FieldAccess` looks like a “variable” when it does not have an explicit target in a context where that is allowed (e.g., accessing an instance field `f` just using the name `f` instead of `this.f` inside the class body or a class field which was statically imported).

Table 2 shows two exemplary pair of snippets:

- (1) In E01, `s` is a local variable. Students incorrectly inlined the variable into the target hole of a `MethodInvocation`, whereas the correct tree properly represents the variable as a subexpression, which needs to be a separate node.
- (2) In E02, the local variable `ar` is mistakenly inlined into the target hole of an `ArrayAccess`. The target of an `ArrayAccess` can be any expression that evaluates to an array reference (JLS 15.10.3) and does not necessarily have to be just the simple name of a variable holding an array reference.

Importance. This mistake often occurs together with the `TARGETINLINED` mistake, which describes any target (of a `FieldAccess`, `MethodInvocation` or `ArrayAccess`) that is inlined into the parent node. Prior work pointed out that popular Java textbooks also provide incorrect guidance [13], implying that a target of a method invocation expression has to be a variable: e.g., “when you use a non-static method, you use the *object name*, a dot, and the method name” [20], “to call a method [one uses] a *variable* that contains a reference to an object” [17]. This mistake may indicate that students cannot reason properly about common expressions such as method invocation chains (like the one in E07).

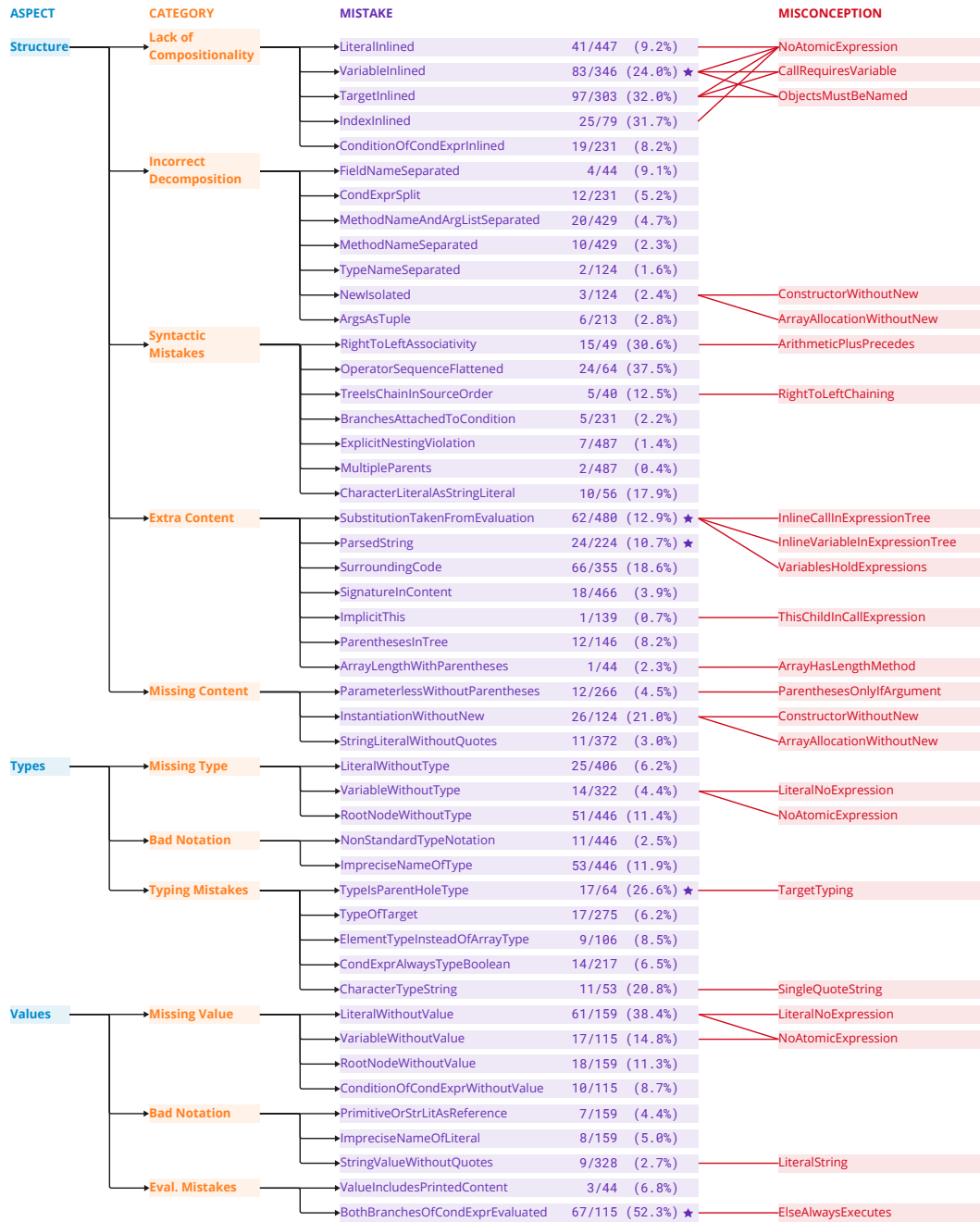


Fig. 4. All the 48 mistakes revealed by our qualitative analysis, grouped by aspect and category. Mistakes are also connected to documented programming language misconceptions [12]. Key mistakes (starred) are extensively discussed in Section 4.2.

Table 2. Examples for the VARIABLEINLINED mistake

Exam	Subexpression	Snippet from student	Snippet from solution
E01	<code>s.length()</code>		
E02	<code>ar[ix]</code>		

Potential causes. This mistake could be caused by the misconception `CallRequiresVariable` (“one needs a variable to invoke a method” [12]). If one needs a variable to invoke a method, then the target of a method invocation cannot be an arbitrary expression (which would be represented with its own subtree) but is constrained to be a name. In that case, it would be correct to represent a method invocation such as `o.m()` with the variable `o` inlined into the method invocation node and not as a separate node. We can generalize this reasoning when considering the misconception `ObjectsMustBeNamed` (“a variable is needed to instantiate an object” [12]). If objects must be named, to represent an array access such as `a[0]` one would not represent the variable `a` as a separate node, but rather inline it into the array access node. Finally, the misconception `NoAtomicExpression` (“expressions must consist of more than one piece” [12]) could also cause this mistake: a variable access would be atomic and therefore considered not an expression.

TYPEISPARENTHOLETYPE. The type of a node is incorrectly specified as the type of the “hole” in its parent node. Table 3 shows the only opportunity in our corpus for this mistake to occur. In that exam, over a quarter of the students made this mistake (17 out of 64):

- (1) E11 makes use of two classes, `Cons` and `Empty`, which both implement `interface Seq { int len(); }`. Together, they model a sequence—also known as a singly linked list—where `Cons` and `Empty` are subtypes of `Seq`. The signature of `Cons`’s constructor is `Cons(String value, Seq rest)`. When typing the expression tree, students mistakenly annotated the `new Empty()` node with `Seq` as a type, rather than `Empty`. Indeed, the second parameter of `Cons`’s constructor is of type `Seq`.

Table 3. Example for the TYPEISPARENTHOLETYPE mistake

Exam	Subexpression	Snippet from student	Snippet from solution
E11	<code>new Cons("A", new Empty())</code>		

Importance. When students take into account typing information from the context (e.g., the parent node), they may be unable to fully reason about subexpressions in isolation and to exploit the composable nature of expressions. For instance, in the above example, `new Empty()` is a valid expression entirely on its own, and `new Empty().m()` would be legal for any method `m` defined in class `Empty`, even if that method did not exist in its supertype `Seq`. This valid behavior would not be allowed if a student incorrectly believes that `new Empty()` is of type `Seq` instead.

Potential causes. This mistake stems from not understanding how types are determined. For the subset of Java covered by the course, the type of an expression is not determined by its surrounding context, but rather by the expression itself. This mistake appears to be a generalization of the `TargetTyping` misconception (“the type of a numerical expression depends on the type expected by the surrounding context” [12]), which only focuses on conversions between primitive types but could easily be extended to cover cases related to conversions between reference types.

***SUBSTITUTIONTAKENFROMEVALUATION.* A node is replaced with a subtree containing code that appears inside methods or constructors called by the expression.** Table 4 shows two exemplary pairs of snippets:

- (1) In E03, the subexpression `gt(a, b)` that calls the `gt` method has been incorrectly replaced with the expression found within the return statement of the called method. The exam question indeed included the definition of the `gt` method: `boolean gt(int a, int b){ return a > b; }`.
- (2) In E10, the `len()` method has an extra child which represents an (incorrect) expression tree for the expression found within the return statement of the recursive `Cons.len()` instance method, whose definition was included in the question: `int len(){ return 1 + rest.len(); }`.

Table 4. Examples of the `SUBSTITUTIONTAKENFROMEVALUATION` mistake

Exam	Subexpression	Snippet from student	Snippet from solution
E03	<pre>gt(a, b) ? "a > b" : (a < b ? "a < b" : "a == b")</pre>		
E10	<code>rest.len()</code>		

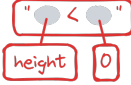
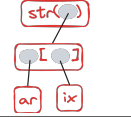
Importance. Understanding the dualism between static and dynamic nature of code is known to be a deep and challenging aspect of programming. For example, prior work [18, 45] documents students’ confusion with variables “holding” unevaluated expressions (static), instead of the result of their evaluation (dynamic).

Potential causes. This mistake indicates a conflation of the static representation of the code, represented by the structure in the expression tree NM, with the dynamic execution of the program that follows method calls. This mistake appears to be a generalization of the misconceptions `InlineCallInExpressionTree` (“The expression tree of an expression involving a call inlines the call’s computation of the returned value.” [12]), `InlineVariableInExpressionTree` (“The expression tree of an expression involving a variable inlines the variable’s definition.” [12]) and `VariablesHoldExpressions` (“= stores an expression in a variable.” [12]).

***PARSEDSTRING.* A string literal is replaced with a subtree representing the “expression tree” of the Java code found in the string.** Table 5 shows two exemplary pairs of snippets:

- (1) In E12, the text contained in the string literal `"height < 0"` is parsed as if it were actual Java code.
- (2) In E02, the string literal `"ar[ix]= "` is replaced by a `str()` method call, with the array access that is written as a string literal parsed as code. This mistake likely originates from Python’s `str()` function. That function is used to construct string representation of values, akin to `Object.toString()` in Java.

Table 5. Examples of the PARSEDSTRING mistake

Exam	Subexpression	Snippet from student	Snippet from solution
E12	"height < 0"		"height < 0"
E02	"ar[ix] = "		"ar[ix]= "

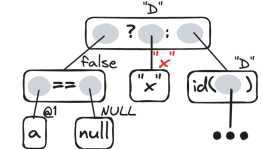
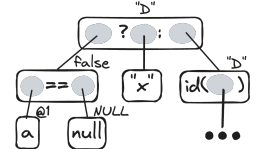
Importance. Writing code fragments within string literals is common, for example, for debugging or logging purposes. Not understanding the difference between a static sequence of characters (i.e., a string), and the tokens of a programming language that are parsed and eventually executed, can lead to incorrect expectations about the program behavior.

Potential causes. This mistake, similarly to `SUBSTITUTIONTAKENFROMEVALUATION`, stems from an inadequate understanding of the difference between static and dynamic semantics. Understanding this difference may be harder for this mistake because of the existence of features in other programming languages such as “string templates” (expressions embedded inside string literals) and dynamic evaluations of code snippets (e.g., Python and Javascript’s `eval`). We could not find a previously documented misconception that captures this problem; this mistake could thus be a symptom of a new misconception.

`BOTHBRANCHESOFCONDEXPREVALUATED`. Both branches of a ConditionalExpression are annotated with values. Table 6 exemplifies one of the exam questions featuring a conditional expression:

- (1) In E08, `a == null` is `false` and therefore the *then* branch is not evaluated; students instead mistakenly evaluated also that subtree. While at first sight `BOTHBRANCHESOFCONDEXPREVALUATED` may not seem like a deep mistake, it can have significant consequences. If Java also wrongly evaluated the *else* subtree when `a == null`, this would trigger a `NullPointerException` when trying to access the value at index `i` of `a`. This idiom of protecting against null pointer dereferencing is extremely common in Java code [32].

Table 6. Example of the `BOTHBRANCHESOFCONDEXPREVALUATED` mistake

Exam	Subexpression	Snippet from student	Snippet from solution
E08	<code>a == null</code> <code>? "X"</code> <code>: id(a[i].toString())</code>		

Importance. The fact that some of the expressions in a conditional computation are not evaluated is particularly important if the evaluation of these expressions were to fail (throwing an exception, such as in `o != null && o.m()` when `o` is `null`) or cause a side-effect (e.g., printing or mutating global state). The evaluation behavior of the `ConditionalOperator` is not an idiosyncrasy of Java. Many other programming languages provide conditional operators with similar behavior. C, C++, C#, and JavaScript provide the exact same operator as Java; other languages—including

ones currently popular in education—provide equivalent conditional expression constructs: Python (`_ if _ else _`), Racket BSL (`((if _ _ _))`), Scala (`if (_)_ else _`), and even the block-based Snap! (`if _ then _ else _`). In languages like Smalltalk/Pharo and the original Self the fact that only one of the branches is evaluated is even made explicit: the arguments of these methods need to be functions (blocks). The paramount importance of the behavior of conditional expressions is also explicitly mentioned in popular introductory programming textbooks [1, 25].

Potential causes. The mistake may be caused by students not understanding that, although evaluation of expressions mostly happens through a post-order depth-first traversal of the tree, when a conditional computation is involved (e.g., in a conditional expression, or with short-circuit operators), the evaluation of an expression does not traverse at all the non-taken branches. This suggests the existence of a new misconception, related to an existing one for conditional statements: `ElseAlwaysExecutes` (“The else branch of an if-else statement always executes” [12]). When drawing expression trees, the problem may be exacerbated by students being also asked to type-check the tree, which needs to be done for all nodes, including the ones that will not be evaluated. The notation adopted by the NM might be at be a contributing factor here due to the similarities between types and value annotations.

Due to space constraints, we could only report a small subset of the 48 mistakes that we identified. The comprehensive list detailing all the mistakes and codes that resulted from our analysis is available in the supplementary materials [6].

4.3 Recommendations for Using “Expression as Tree” in Assessments

Our results show that the “expression as tree” NM, when used as an assessment instrument in real-world exams, can uncover a rich and diverse set of mistakes. Our analysis of the hundreds of student-drawn expression trees is also a source of recommendations for instructors or researchers who plan to use this NM in their own assessments.

Be explicit about notation. Students might be unsure about various aspects of the notation, such as whether to annotate literal nodes with values or types (because the nodes’ contents already contains the exact values), whether to annotate the root node with values or types (because there is no edge going upwards from the root, and students may think of annotations as pertaining to edges instead of nodes), or whether to create nodes to represent parenthesized expressions. At the expense of being more verbose, an assessment item could explicitly state these requirements.

Provide the environment binding names to types and values. If an expression uses a name, the question should define the type and value of that name. This can be done in two ways: either by explicitly providing the names and their types (and/or values), or by providing code that surrounds the expression and defines the necessary names (e.g., by embedding the expression in a method with correspondingly named and typed parameters, and indicating with which argument values the method is called).

Use code context carefully. If an expression is embedded inside a broader piece of code, students might include parts of that context in the expression tree. This can be immediately adjacent tokens, such as the keyword `return` and the statement terminator `;`, or it can be further away code, such as the bodies of methods invoked in the expression. On the other hand, for the very same reasons, providing code context can be useful if one desires to assess the understanding of the difference between the static nature of code and its dynamic execution at runtime.

5 THREATS TO VALIDITY

We now discuss the threats to credibility, transferability, dependability, and confirmability [16] of our results.

Credibility. The corpus we studied is based on exams from real university-level courses. The coding structure was tied to established language constructs and aspects of programming (structure, typing, evaluation). Our data analysis was conducted by four programming language experts with significant experience in programming education. We triangulate the discovered mistakes by relating them to existing programming language misconceptions.

Transferability. We analyzed 542 diagrams from twelve different exams from courses across two different education levels, over four years. All courses were taught by the same instructor; different instructors might introduce the NM in different ways, which could lead to different findings. Moreover, the study focuses on Java, and specifically on the twelve language constructs observed in the exams. The NM would support a broader set of constructs, for which the mistakes one would observe might be different.

Dependability. The analysis was performed by four different coders and took several weeks to complete. To ensure coding consistency, we held frequent consensus finding meetings, often multiple times per day, and we included a consolidation phase in our process, where all four coders analyzed the same set of diagrams and discussed and resolved all discrepancies. Moreover, we captured our codes in a collaborative spreadsheet with multiple automated consistency checks, and we carefully maintained concise descriptions of all dimensions throughout the evolution of our code book.

Confirmability. An artifact with our code book, including the memos describing the codes in detail, the list of the 404 distinct five-dimensional codes, and the tables with the 1906 coded deviations, is available in the supplementary materials [6].

6 CONCLUSION

Expressions are prevalent in code and can be composed of several language constructs. Understanding expressions means understanding not only the syntax, but also the static (types) and the dynamic (evaluation) semantics. In this paper we studied a NM that focuses on expressions as an instrument to assess students' understanding of these aspects.

The peculiar nature of our data required us to devise a structured coding system to qualitatively analyze the trees drawn by students in a systematic way. Our system exploits the fact that the analyzed diagrams deviate from a single, correct solution, and it integrates aspects based on programming language theory with phenomena inferred from the artifacts. Our methodology can be adopted and adapted by researchers to analyze other diagrams and NMs.

Our results show that the “expression as tree” NM, when used as an assessment instrument in real-world exams, can uncover a rich and diverse set of mistakes, which span all three aspects involved in understanding expressions and are connected to misconceptions. We found mistakes that can be associated with misconceptions previously documented and others that suggest the existence of new ones. Moreover, the mistakes we identified provide the basis for designing assessment items and rubrics for practitioners and researchers.

ACKNOWLEDGMENTS

This work was partially funded by the Swiss National Science Foundation project 200021_184689.

REFERENCES

- [1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs, (Second Edition)* (second edition ed.). Vol. 33. MIT Press, Cambridge, MA, USA.
- [2] Georgii Maksimovich Adel'son-Velskii and Evgenii Mikhailovich Landis. 1962. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR* 146, 2 (1962), 263–266.
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

- [4] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–16. <https://doi.org/10.1145/3544548.3580785>
- [5] Michael Berry and Michael Kölling. 2013. The Design and Implementation of a Notional Machine for Teaching Introductory Programming. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education on - WiPSE '13*. ACM Press, Aarhus, Denmark, 25–28. <https://doi.org/10.1145/2532748.2532765>
- [6] Joey Bevilacqua, Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2024. Supplementary Material for the Paper "Assessing the Understanding of Expressions: A Qualitative Study of Notional-Machine-Based Exam Questions". <https://doi.org/10.5281/zenodo.13898594>
- [7] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 445–456. <https://doi.org/10.1145/2676726.2676982>
- [8] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. <https://doi.org/10.1191/1478088706qp0630a>
- [9] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 223–228. <https://doi.org/10.1145/2538862.2538924>
- [10] Neil C. C. Brown, Victoria Mac, Pierre Weill-Tessier, and Michael Kölling. 2024. Writing Between the Lines: How Novices Construct Java Programs. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 165–171. <https://doi.org/10.1145/3626252.3630968>
- [11] Kim B. Bruce, Andrea Danylyuk, and Thomas Murtagh. 2005. Why Structural Recursion Should Be Taught before Arrays in CS 1. *ACM SIGCSE Bulletin* 37, 1 (Feb. 2005), 246–250. <https://doi.org/10.1145/1047124.1047430>
- [12] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafilovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 380–386. <https://doi.org/10.1145/3430665.3456343>
- [13] Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2022. Expressions in Java: Essential, Prevalent, Neglected?. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2022)*. Association for Computing Machinery, New York, NY, USA, 41–51. <https://doi.org/10.1145/3563767.3568131>
- [14] Alonzo Church. 1941. *The Calculi of Lambda-Conversion*. Number 6 in 1. Princeton University Press, Princeton, New Jersey, USA.
- [15] John Clements and Shriram Krishnamurthi. 2022. Towards a Notional Machine for Runtime Stacks and Scope: When Stacks Don't Stack Up. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1 (ICER '22, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 206–222. <https://doi.org/10.1145/3501385.3543961>
- [16] John W. Creswell and Timothy C. Guetterman. 2020. *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research, 6th Edition* (6 ed.). Pearson, Essex.
- [17] Paul J. Deitel and Harvey M. Deitel. 2018. *Java: How to Program Early Objects* (11th edition ed.). Pearson, New York.
- [18] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 57–73. <https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9> arXiv:<https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9>
- [19] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The Black Box inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies* 14, 3 (April 1981), 237–249. [https://doi.org/10.1016/S0020-7373\(81\)80056-9](https://doi.org/10.1016/S0020-7373(81)80056-9)
- [20] Joyce Farrell. 2019. *Java Programming* (ninth edition ed.). Cengage, Australia United States.
- [21] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühlhng, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 21–50. <https://doi.org/10.1145/3437800.3439202>
- [22] Paul Goldenberg, June Mark, Brian Harvey, Al Cuoco, and Mary Fries. 2020. Design Principles behind Beauty and Joy of Computing. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 220–226. <https://doi.org/10.1145/3328778.3366794>
- [23] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 (Lecture Notes in Computer Science, Vol. 6183)*. Springer Berlin, Heidelberg, Berlin, Heidelberg, 126–150. <https://doi.org/10.48550/arXiv.1510.00925> arXiv:[1510.00925](https://doi.org/10.48550/arXiv.1510.00925) [cs]
- [24] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (May 2001), 396–450. <https://doi.org/10.1145/503502.503505>
- [25] Shriram Krishnamurthi. 2023. *Programming Languages: Application and Interpretation* (3 ed.). Electronic textbook, Online.
- [26] Amruth N. Kumar. 2015. The Effectiveness of Visualization for Learning Expression Evaluation. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, Kansas City Missouri USA, 362–367. <https://doi.org/10.1145/2676723.2677301>
- [27] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Do Values Grow on Trees?: Expression Integrity in Functional Programming. In *Proceedings of the Seventh International Workshop on Computing Education Research - ICER '11*. ACM Press, Providence, Rhode Island, USA, 39. <https://doi.org/10.1145/2016911.2016921>

- [28] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. 2010. Learning Computer Science Concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research (ICER '10)*. Association for Computing Machinery, New York, NY, USA, 69–76. <https://doi.org/10.1145/1839594.1839607>
- [29] Greg L. Nelson and Amy J. Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. Association for Computing Machinery, New York, NY, USA, 31–39. <https://doi.org/10.1145/3230977.3230992>
- [30] Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. 2017. 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*. ACM Press, Tacoma, Washington, USA, 2–11. <https://doi.org/10.1145/3105726.3106178>
- [31] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 86–99. <https://doi.org/10.1145/3009837.3009900>
- [32] Haidar Osman, Manuel Leuenberger, Mircea Lungu, and Oscar Nierstrasz. 2016. Tracking Null Checks in Open-Source Java Systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, Osaka, Japan, 304–313. <https://doi.org/10.1109/SANER.2016.57>
- [33] Daejun Park, Andrei Stăfănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 346–356. <https://doi.org/10.1145/2737924.2737991>
- [34] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.
- [35] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2020. *Programming Language Foundations*. Software Foundations, Vol. 2. Electronic textbook, Online.
- [36] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 217–232. <https://doi.org/10.1145/2509136.2509536>
- [37] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. <https://doi.org/10.1145/1592761.1592779>
- [38] Fritz Ruehr. 2008. Tips on Teaching Types and Functions. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education (FDPE '08)*. Association for Computing Machinery, New York, NY, USA, 79–90. <https://doi.org/10.1145/1411260.1411272>
- [39] Johnny Saldaña. 2021. *The Coding Manual for Qualitative Researchers* (second edition ed.). SAGE Publications Ltd, Thousand Oaks, California.
- [40] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. 2015. Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15*. ACM Press, Kansas City, Missouri, USA, 616–621. <https://doi.org/10.1145/2676723.2677238>
- [41] Teemu Sirkiä and Juha Sorva. 2012. Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research - Koli Calling '12*. ACM Press, Koli, Finland, 19–28. <https://doi.org/10.1145/2401796.2401799>
- [42] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13, 2 (June 2013), 1–31. <https://doi.org/10.1145/2483710.2483713>
- [43] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*. ACM Press, Berlin, Germany, 49–54. <https://doi.org/10.1145/1930464.1930471>
- [44] John Sweller. 1994. Cognitive Load Theory, Learning Difficulty, and Instructional Design. *Learning and Instruction* 4, 4 (Jan. 1994), 295–312. [https://doi.org/10.1016/0959-4752\(94\)90003-5](https://doi.org/10.1016/0959-4752(94)90003-5)
- [45] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, Espoo Finland, 151–159. <https://doi.org/10.1145/3230977.3230995>
- [46] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573. <https://doi.org/10.1145/358746.358755>
- [47] Ville Tirronen. 2014. Study on Difficulties and Misconceptions with Modern Type Systems. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITICSE '14)*. Association for Computing Machinery, New York, NY, USA, 303–308. <https://doi.org/10.1145/2591708.2591726>
- [48] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 1023–1028. <https://doi.org/10.1145/3159450.3159479>
- [49] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda*. Electronic textbook, Online.
- [50] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18 (Dec. 1989), 1245–1262. <https://doi.org/10.1137/0218082>