Sound Notional Machines

A Foundation and Its Applications

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Igor Moreno Santos

under the supervision of
Matthias Hauswirth

September 2023

Dissertation Committee

Patrick Eugster	Università della Svizzera italiana, Switzerland
Antonio Carzaniga	Università della Svizzera italiana, Switzerland
Johan Jeuring	Utrecht University, The Netherlands
Tobias Wrigstad	Uppsala University, Sweden

Dissertation accepted on 28 September 2023

Research Advisor	PhD Program Director
Matthias Hauswirth	**The PhD program Director *pro tempore***

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Igor Moreno Santos
Lugano, 28 September 2023

Abstract

A notional machine is a pedagogic device that abstracts away details of the semantics of a programming language to focus on some aspects of interest. A notional machine should be *sound*: it should be consistent with the corresponding programming language, and it should be a proper abstraction. Notional machines found in the computer science education literature are usually not evaluated with respect to their soundness. To address this problem, we first introduce a formal definition of soundness for notional machines. The definition is based on the construction of a commutative diagram that relates the notional machine and the aspect of the programming language the notional machine is focused on. Leveraging this formalism, we present a methodology for constructing sound notional machines and a similar methodology to reveal potential inconsistencies in existing notional machines. We apply these methodologies to build sound-by-construction notional machines and find inconsistencies in existing ones as well as propose solutions to these inconsistencies. Finally, we show that the same commutative diagram that describes a notional machine can be used also to design experiments to evaluate that notional machine as an educational assessment instrument.

Acknowledgements

I first want to thank Nate Nystrom, with whom I started the PhD and worked with for the first three years. I had done my master's thesis with Nate about an approach to program inversion and continued in the PhD studying the same topic under his supervision. I really like this topic and I also liked working with Nate. I learned a lot from him.

Nate helped me to find another advisor before he left the university and Matthias was the best fit but there was one catch: I had to change topics. So I did but fortunately, I could still use my PL background in the new topic. Most notably, I learned about Milner's simulation, which I use to characterize the relationship between a notional machine and the aspect of the programming language under its focus, from the paper Refactoring Pattern Matching, by Jeremy Gibbons, that I read while working with Nate.

I want to thank my advisor Matthias Hauswirth, of course. He always valued my input and my opinion long before I started the PhD, always making me feel I had something interesting to contribute. It's funny to look back, for example, at the time he was asking me for my insights on some aspects of his new grant proposal connecting computing education and programming languages, and now to think that I'm working on that very project. Moreover, I want to thank Matthias for his patience and somehow for managing to actually get me to graduate, with measured doses of challenge, nudging, and vast encouragement throughout the process.

I also want to acknowledge the absolutely essential support Nate and Matthias gave me when I was suffering from a severe case of epicondylitis. I even thought I had to abandon computer science because I couldn't type but thanks to them I managed to continue in the program by using a dictation software, which I still rely on to this day.

I also would like to thank the Committee for their time and feedback especially Johan Jeuring, with whom I had many meetings while developing the ideas that turned out to be central to this dissertation.

I would like to thank my colleagues Luca, Andrea, and Joey that are part of

our research lab. I co-advised Joey in his bachelor's and master's thesis and now he's a PhD student in the team (how time flies!). He has been instrumental in the development of ExpressionTutor as a whole not only in the aspects I describe in this dissertation. Working with Luca (the little monster) has been especially pleasurable. We collaborated intensely in the ideas and implementation of the improvements to ExpressionTutor for Java among many other things that are not part of this dissertation. It's great to work with someone that cares about understanding things deeply. Andrea didn't have as much influence on the content of my research but he definitely helped me to keep sane in the last few months. Thank you, Andrea.

Taking the course Foundations of Software by Martin Odersky, based on Benjamin Pierce's Types And Programming Languages (the TAPL book), was a water-shattering experience for me. I remember when saw the Curry-Howard correspondence and thought to myself: "how did I never hear about this before?". What amazed me wasn't only proving properties about programming languages but that programming languages themselves could be mathematical entities. It also confirmed my intuition that some "things" in programming languages were more fundamental than others.

I also want to thank Richard Eisenberg. I met Richard at ZuriHac. He was always very kind and welcoming. He suggested I go to ICFP as a student volunteer. It was a great advice. I loved the experience and the community and became really motivated to pursue a career in research.

I'm deeply inspired by Conal Elliott's work on denotational design. I like to think of my approach to the soundness of notional machines as denotational, although I don't claim it to be in the dissertation. Of course, I undermine this idea by adopting an operational definition of the languages I describe in the appendix. In reality, I don't have enough experience with denotational semantics and it is convenient and appealing to follow the well-known sequence of languages described in the TAPL book. But the idea of approaching a problem by first precisely specifying what it means, so eloquently described by Conal, is at the heart of what I'm trying to do.

Jeremy Gibbons' work had also a big influence on me from the time I was working with Nate researching about program inversion. I had the opportunity to talk to him briefly at ICFP and he was very kind. He also gave me important advice later when I was transitioning from working with Nate to working with Matthias. It was reading one of Jeremy's papers that I learned about Milner's simulation, which I use to characterize notional machines. He is, of course, a famous proponent of reasoning equationally about programs, which I use extensively in my description of notional machines that are sound by construction.

I want to thank Brent Yorgey and the maintainers of the Diagrams library, which I'm using to generate most of the concrete representations of notional machines in this dissertation. Thanks to the diagrams library, the images are generated with little Haskell programs written inside the tex files that call directly into our Haskell artifact.

I also would like to acknowledge many people that had an influence in this work not academically but by having an important influence on me personally. One of them is Monika Maslikowska. I would like to thank Monika for all the support she gave me during the first part of my PhD. I'm hopeless in trying to describe unemotionally the effect she had on me personally because it is intangible, wide, deep, intense, and long-lasting.

I want to thank Boyan Beronov. I met Boyan in the middle of the master's program and he had a big influence on the way I think about science in general and programming languages in particular and ultimately in my interest to pursue research. We had many discussions about the core ideas I was developing in my master thesis which were foundational to the research I conducted in the first half of my PhD. Not to mention the countless hours talking about life and all ranges of personal matters. He's a great friend.

Tomás Cardoso is not only a good friend but he also introduced me to Ludwig Wittgenstein's work on philosophy of language. Although not used directly in my research, these ideas, and certainly my discussions with Tomás about them, have left a long-lasting impact which manifests itself in the form I approached the research and think about the problems. Thank you, Tomás.

I would like to thank Corinne Häberling for being a great friend and for the support she gave me especially in the last months. I really appreciate her measured response and advice during challenging times and her particular perspective which is often surprisingly different than what I hear from most people. Besides that, she has been a great inspiration and influence on me in many ways, particularly in my late-in-life discovery of the immense benefits of sports holistically in one's life.

There are many more friends that I'm not acknowledging here but are near and dear to my heart. Thank you for your friendship.

Last but not least, of course, I cannot express enough gratitude toward my family, who have supported me in many more ways than I can describe.

My father taught me to program when I was a child. It was a lot of fun. When I was finishing high school, I remember fondly the day he advised me to study Computer Science and not Math, which is particularly funny given he is a mathematician. It turned out that what really blew my mind and motivated me to do a PhD was a combination of both: the world of mathematically rigorous

approaches to programming languages. Thank you, dad.

I would like to thank my mother, of course, for her invaluable and unwavering support throughout so many personal challenges I have faced throughout the PhD. Many times the difficulties seemed unsurmountable but I could always count on her support. In the most tricky and intricate issues I would often think to myself "At least there is someone who understands". Priceless! Thank you, mom.

My dear daughter Sofia always brings a smile to my face. She's great! She is a big source of motivation, helps to keep me centered, and brings out the best in me. Thank you, honey.

Contents

Contents ix

List of List of Figures xiii

List of List of Tables xvii

1 Introduction 1
- 1.1 Definition of Notional Machine . . . 1
- 1.2 Examples of Notional Machines . . . 2
- 1.3 Quality of Notional Machines . . . 3
- 1.4 Soundness of Notional Machines . . . 6
- 1.5 From Theory to Practice . . . 7
- 1.6 Contributions . . . 7

2 Designing Sound Notional Machines 9
- 2.1 Isomorphic Notional Machines . . . 10
  - 2.1.1 Illustrative Example . . . 10
  - 2.1.2 Soundness via Commutative Diagrams . . . 11
- 2.2 Interlude: Abstract vs. Concrete Syntax of Notional Machines . . . 12
- 2.3 Monomorphic Notional Machines . . . 14
  - 2.3.1 Illustrative Example . . . 15
  - 2.3.2 Commutative Diagram . . . 15
- 2.4 A Notional Machine to Reason About State . . . 18
  - 2.4.1 Designing a Notional Machine . . . 19
  - 2.4.2 Illustrative Example . . . 21
  - 2.4.3 Commutative Diagram . . . 21
- 2.5 A Notional Machine to Reason About Types . . . 24
- 2.6 Conclusion . . . 26

3 Analyzing Existing Notional Machines **27**
3.1 Reduct 28
3.1.1 Illustrative Example 30
3.1.2 Commutative Diagram 30
3.2 Alligator Eggs 33
3.2.1 Illustrative Example 34
3.2.2 Commutative Diagram 35
3.2.3 From Proof to Property-Based Testing 36
3.3 The Concrete Syntax of Alligators 39
3.3.1 Designing a Concrete Representation 40
3.4 Conclusion 42

4 A Family of Notional Machines for Expressions **43**
4.1 Why Focus on Expressions? 44
4.2 Expressions in Java 45
4.2.1 Grammar is Not Enough to Identify Constructs 46
4.2.2 Java Expression Constructs 47
4.3 ExpressionTutor for Java: A Typing Activity 50
4.3.1 Programming Language Layer With JDT 50
4.3.2 From Java Expressions to ExpressionTutor Diagrams 50
4.3.3 Notional Machine Layer as a Student Activity 53
4.3.4 Discussion 58
4.4 ExpressionTutor for Java: A Parsing Activity 58
4.4.1 Commutative Diagram for Parsing 59
4.4.2 Distractor Nodes 61
4.5 Automatic Generation and Assessment of Activities 67
4.5.1 Single Activity Generation 68
4.5.2 Personalized Activity Generation 68
4.5.3 Automatic Assessment of Activities 68
4.6 Conclusion 75

5 Notional Machines as Assessment Instruments **77**
5.1 The Instructor's Perspective 78
5.2 Experiment Design Methodology 79
5.3 Pilot Study Design 80
5.3.1 Designing Ground Truth and Notional Machine Questions 81
5.3.2 Analyzing Explanations 85
5.3.3 Results: Compare and Analyze Answers 88
5.3.4 Question A 92

5.3.5 Question B 103
5.4 Discussion 111

6 Related Work **113**

7 Future Work **115**
7.1 Notional Machines for Data Structures 115
7.2 Proven Sound by Construction 116
7.3 Expression Tutor Improvements 117
7.3.1 Generalized Automatic Assessment 118
7.4 Experiment to Evaluate ExpressionTutor 118
7.5 Misconception Detection 119

8 Conclusion **121**

A Programming Language Definitions **125**
A.1 UntypedLambda 125
A.2 TypedArith 126
A.3 TypedLambdaRef 127

B Exam Questions **129**
B.1 Midterm Exam Question 130
B.2 Final Exam Question 131

Bibliography **133**

List of Figures

1.1 Excerpt taken from the LOGO manual [Du Boulay and O'Shea, 1976] 3
1.2 The "Expression as Tree" notional machine and the "Recursion Role Play" notional machine captured by [Fincher et al., 2020]. 4
1.3 The "Array as Row of Spaces in Parking Lot" notional machine captured by [Fincher et al., 2020]. 5

2.1 Evaluation of $(\lambda x.\ \lambda y.\ x)\ a\ b$ in programming language (top) and ExpTree notional machine (bottom). 10
2.2 The soundness condition for notional machines shown both as a commutative diagram and in algebraic form. 11
2.3 Instantiation of the commutative diagram in Figure 2.2 for the notional machine ExpTree and the programming language UntypedLambda. 13
2.4 The omega combinator in UntypedLambda (top) and (incorrect) representations in ExpTutorDiagram notional machine (bottom). 15
2.5 Instantiation of the commutative diagram in Figure 2.2 for the notional machine ExpTutorDiagram 17
2.6 Simplified version of the commutative diagram for the notional machine ExpTutorDiagram shown in Figure 2.5 18
2.7 TAPLMemoryDiagram for TypedLambdaRef for TAPL Exercise 13.1.1 21
2.8 Trace of `(λr:Ref Nat.(λs:Ref Nat.s := 82; !r) r) (ref 13)` in TAPLMemoryDiagram for TypedLambdaRef 22
2.9 Instantiation of the commutative diagram in Figure 2.2 for the notional machine TAPLMemoryDiagram and the programming language TypedLambdaRef. 23
2.10 First attempt at instantiating the commutative diagram in Figure 2.2 for a notional machine that focuses on type-checking using the programming language TypedArith. 24

2.11 Second attempt at instantiating the commutative diagram in Figure 2.2 for a notional machine that focuses on type-checking using the language TYPEDARITH. The notional machine now exposes the inner workings of the typing algorithm. 25
2.12 One step in the notional machine EXPTUTORDIAGRAM as it types the term `if iszero 0 then succ 0 else succ succ 0` in the language TYPEDARITH. 26

3.1 Evaluation in programming language (left) and REDUCT notional machine (right). 31
3.2 Evaluation of `(λt.(λf.t)) a b` in the untyped lambda calculus (top) and ALLIGATOR notional machine (bottom). 34
3.3 Old alligator in `(λa.y) ((λb.b) c)` 35
3.4 First attempt at instantiating the commutative diagram in Figure 2.2 for the notional machine ALLIGATOR. 37
3.5 Second attempt at instantiating the commutative diagram in Figure 2.2 for the notional machine ALLIGATOR. 38
3.6 Both program and notional machine have abstract and concrete representations 40
3.7 Different concrete representations of the same Alligator family in the ALLIGATOR notional machine. 41
3.8 Confusion due to suboptimal concrete representation in the ALLIGATOR notional machine. 41

4.1 Instantiation of the diagram in Figure 2.2 for ExpressionTutor focused on typing Java expressions. The diagram serves simultaneously as a conceptual view of the tool's architecture. 51
4.2 ExpressionTutor diagram for the expression `"As String: " + new Object(){ int m() {` whose AST contains descendants that are not sub-expressions. . . 52
4.3 Excerpt of the ExpressionTutor reference page for Java, which contains examples of diagrams for each expression construct in Java, up to Java 11. 53
4.4 Current interface for a typing activity about the expression `1 / 2 / 3`. 54
4.5 Improved interface for a typing activity about the expression `1 / 2 / 3`. 55
4.6 Typing activity showing to the student the code context of the expression. 57

4.7 Instantiation of the diagram in Figure 2.2 for ExpressionTutor focused on parsing Java expressions. The diagram is also a conceptual view of the tool's architecture. 59
4.8 Distractor nodes generated for the expression `"a[i] = " + (a == null ? "X" : id(a[i]` 66
4.9 Feedback automatically generated for a student's answer to an automatically generated activity based on the student's code. . . . 71
4.10 Overview report shown for answers to an activity used in a quiz. . 72
4.11 Incorrect nodes report for answers to an activity used in a quiz. Answers are grouped by wrong nodes. 73
4.12 Assessment report shown to an instructor about a set of student submissions. 74

5.1 Beginning of the page in the Moodle quiz as shown to students containing an ExpressionTutor question for a parsing activity. . . . 84
5.2 Correctness of answer to multiple-choice (MC) questions vs. correctness of answers to ExpressionTutor questions. 91
5.3 Information given to students and expected solution to ExpressionTutor activity in question A. 93
5.4 ExpressionTutor page as seen by students when coming from the Moodle question A. 94
5.5 Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.1. 95
5.6 Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.2. 96
5.7 Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.3. 99
5.8 Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.4. 100
5.9 Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.5. 102
5.10 Information given to students and expected solution to ExpressionTutor activity in question B. 104
5.11 Comparison between answers to ExpressionTutor question B and answers to multiple-choice item B.1. 105
5.12 Comparison between answers to ExpressionTutor question B and answers to multiple-choice item B.2. 108
5.13 Comparison between answers to ExpressionTutor question B and answers to multiple-choice item B.3. 110

A.1 The untyped lambda calculus (UntypedLambda). 125
A.2 Syntax and reduction rules of the TypedArith language. 126
A.3 Syntax of types and typing rules of the TypedArith language. . . 127
A.4 TypedLambdaRef: Syntax and Evaluation 128

List of Tables

2.1 Notional machines, programming languages, and notional machine focuses used throughout the chapter. 9

2.2 Abstract and concrete representations of a programming language and a notional machine. 14

3.1 The two notional machines used in this chapter, both described here focusing on evaluation of untyped lambda calculus terms. In the original paper, REDUCT focuses on JavaScript. For simplicity, here we use untyped lambda calculus. 27

3.2 Metaphors used in REDUCT 29

4.1 The variations of ExpressionTutor for Java shown in this chapter. 43

4.2 Java expression constructs. The meta-variable e denotes subexpressions. The bold symbols follow the conventions of EBNF. The colored tokens denote tokens of the Java language. The remaining meta-variables are described in Table 4.3. 48

4.3 Meaning of `MetaVariables` used in Table 4.2 with reference to relevant section(s) of the Java Language Specification. 49

4.4 Categories of mistakes made by students when drawing expression trees in paper-based exams. 64

4.5 Categories of mistakes and number of occurrences of mistakes in each category found in two exam questions about expression trees. 65

5.1 Examples of answers and explanations given by students for the question “Is **`return`** an expression?” classified with each category. 86

5.2 Examples of answers and explanations given by students for the question “Is **`return`** an expression?” showing that a wrong explanation can be given for a correct answer. 87

5.3 Examples of answers and explanations given by students for the question “Is `return` an expression?” showing that the same explanation can be given for both a correct and a wrong answer. 88
5.4 For each multiple-choice (MC) item, a Figure with the corresponding ExpressionTutor (ET) question, its stem, and the Figure summarizing the comparison between the answers to that item and the corresponding expression trees. 90
5.5 Aggregated results for multiple-choice items A.3 (shown and Figure 5.7) and A.4 (shown and Figure 5.8). 97
5.6 Table of Answers and Explanations 106

8.1 Instantiations of the commutative diagram in Figure 2.2 that represents the soundness condition for notional machines. For each instantiation, we show the different dimensions we explored. . . 123

Chapter 1

Introduction

Programs are expressed in a programming language (be it a text-based language or not) so learning to program involves learning the syntax and semantics of a programming language (or at least part of it). For novices, the semantics of a program is often not obviously apparent from the program itself. Instructors then often use a *notional machine* [Fincher et al., 2020] to help teach some particular aspect of programs or programming, and also to assess students' understanding of said aspect.

1.1 Definition of Notional Machine

The term notional machine was coined by Du Boulay [1986], who defined it as "the idealised model of the computer implied by the constructs of the programming language". This definition was part of the work on LOGO and their efforts to teach it to children and teachers [Du Boulay and O'Shea, 1976]. The modern use of the term notional machine, which somewhat differs from du Boulay's original definition, is due to two prominent works in the field of computing education. Both refer to du Boulay's work but Robins et al. [2003] define a notional machine to be a "model of the computer as it relates to executing programs", while Sorva [2013] defines a notional machine to be "an abstraction of the computer in the role of executor of programs of a particular kind".

In 2020, a working group of prominent researchers in the field of computing education, including du Boulay, conducted an extensive literature review on the topic of notional machines [Fincher et al., 2020] while capturing and cataloguing examples of notional machines in use. They first distinguished between (i) mental models, which are internal, personal to a learner, idiosyncratic, incomplete, unstable, (ii) conceptual models, which are precise and complete representations

consistent with scientifically accepted knowledge, and (iii) notional machines that are a special kind of conceptual model. The authors then establish a definition of notional machine, which is the one we will adopt here:

> A notional machine is a pedagogic device to assist the understanding of some aspect of programs or programming.

They refine the definition with a set of definitional characteristics of notional machines:

Pedagogical Purpose: The purpose of a notional machine is for use in teaching to support student learning of computational concepts. A crucial aspect of a notional machine is that it should simplify an actual concept or skill as an aid to understanding.

Function: The generic function of a notional machine is to uncover something about programming, computers or computation, or to draw attention to something, that is not obviously apparent in the artefact the student is using.

Focus: A notional machine typically focuses on a particular aspect of programs and their behaviour. As well as programs, a notional machine's focus can also be concerned with computers as places where programs can be built, run, and stored.

Representation: A notional machine will have a representation and this representation will draw attention to certain aspects of the focus and possibly ignore others.

Two of these aspects are of particular importance for our work: (i) a notional machine focuses on some aspect of programs and (ii) it has a representation that draws attention to this aspect under focus.

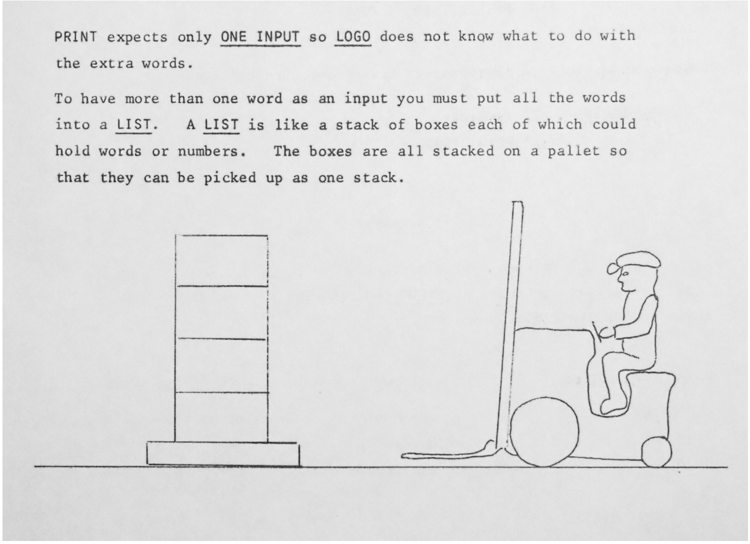
1.2 Examples of Notional Machines

The LOGO manual [Du Boulay and O'Shea, 1976] contains what are probably the first examples of notional machines. Figure 1.1 shows part of it, where a list is explained by an analogy with a stack of boxes.

As part of the effort by Fincher et al. [2020] to systematically capture and catalog examples of notional machines, they collected over 50 notional machines[1],

[1] The entire collection is kept up to date on a website: `https://notionalmachines.github.io/notional-machines.html`

capturing each one using a template card with mandatory elements. Two notional machine examples described using this template are shown in Figure 1.2. The one on the left is the “Expression as Tree” notional machine, focused on “expression structure, how evaluation proceeds, how types are determined”. The one on the right is the “Recursion Role Play” notional machine, an example of a notional machine that is not a diagram but is meant to be enacted with the students in the class. Notice in both examples the correspondence between concepts in the programming language (PL) and elements in the representation of the notional machine (NM), something critically important for our work.


PRINT expects only ONE INPUT so LOGO does not know what to do with the extra words.

To have more than one word as an input you must put all the words into a LIST. A LIST is like a stack of boxes each of which could hold words or numbers. The boxes are all stacked on a pallet so that they can be picked up as one stack.

Figure 1.1. Excerpt taken from the LOGO manual [Du Boulay and O’Shea, 1976]

1.3 Quality of Notional Machines

Given the extensive use of notional machines, and that they are intended to help students when learning, it is important to look at their quality. Obviously, a notional machine should be tested in practice to answer questions such as “how well do students understand a particular aspect after studying it using a particular notional machine?” and “what is the cost associated with introducing the notional machine (how much time had to be invested to obtain a benefit)?”. But even before experimenting with a notional machine, one should make sure that the notional machine is *sound*: it is in some sense a faithful representation of the

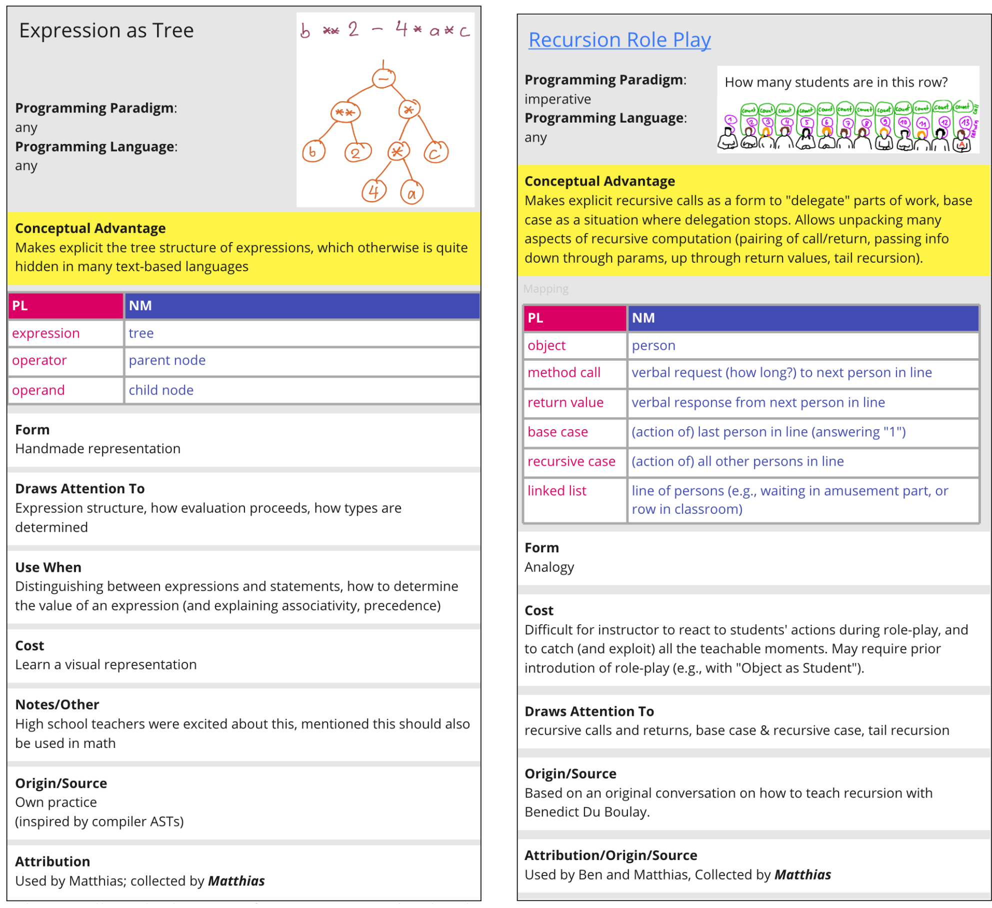
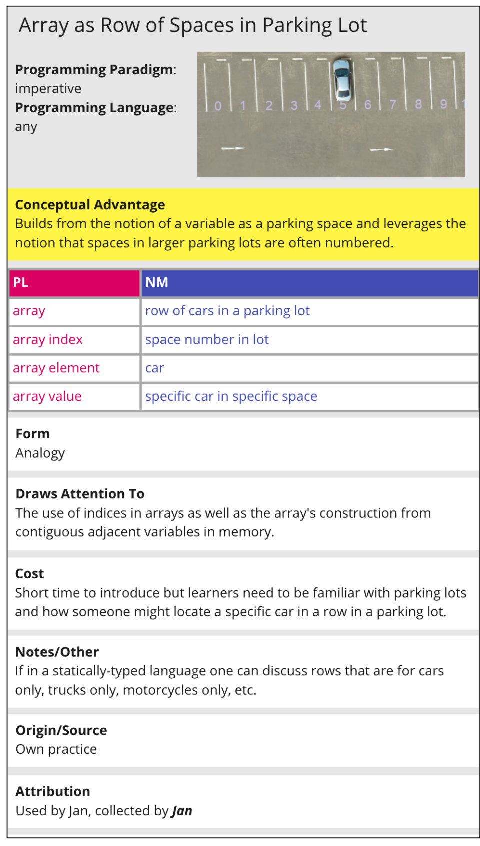


Figure 1.2. The "Expression as Tree" notional machine and the "Recursion Role Play" notional machine captured by [Fincher et al., 2020].

aspect of programs it is meant to represent. Anecdotal evidence of using unsound representations in education goes back a long way. Richard Feynman eloquently stated [Feynman, 1985], after reviewing "seventeen feet" of new mathematics schoolbooks for the California State Curriculum Commission:

> [The books] would try to be rigorous, but they would use examples (like automobiles in the street for "sets") which were almost OK, but in which there were always some subtleties. The definitions weren't accurate. Everything was a little bit ambiguous

Ambiguously specified notional machines and notional machines with imperfect analogies to programming concepts are a problem. Educators may mischaracterize language features and students may end up with misconceptions [Chiodini et al., 2021] instead of profoundly understanding the language.



Array as Row of Spaces in Parking Lot

Programming Paradigm: imperative
Programming Language: any

Conceptual Advantage
Builds from the notion of a variable as a parking space and leverages the notion that spaces in larger parking lots are often numbered.

PL	NM
array	row of cars in a parking lot
array index	space number in lot
array element	car
array value	specific car in specific space

Form
Analogy

Draws Attention To
The use of indices in arrays as well as the array's construction from contiguous adjacent variables in memory.

Cost
Short time to introduce but learners need to be familiar with parking lots and how someone might locate a specific car in a row in a parking lot.

Notes/Other
If in a statically-typed language one can discuss rows that are for cars only, trucks only, motorcycles only, etc.

Origin/Source
Own practice

Attribution
Used by Jan, collected by ***Jan***

Figure 1.3. The “Array as Row of Spaces in Parking Lot” notional machine captured by [Fincher et al., 2020].

For example, Fincher et al. [2020] describe the “Array as Row of Spaces in Parking Lot” notional machine, summarised in Figure 1.3. Let’s consider a language like Java. In Java, when an array of objects is allocated, all its slots contain `null`, which means these slots don’t contain a reference to any object. This would be reasonably represented in the notional machine as an empty parking lot. But if instead of an array of objects, we have an array of `int`s, for example, then when we instantiate an array, all its slots contain `0`, which is not the absence of a number but a number like any other. A student could also reasonably question whether one can park a car in a slot that is already occupied by another car, or whether one has to remove a car from a spot to park another car in the same spot. In fact, the authors point out that, “The effectiveness of the analogy depends on [...] how well that models the semantics of the programming language.”

1.4 Soundness of Notional Machines

To avoid these issues, we need to make sure that a notional machine is indeed an accurate abstraction. Although the definition of notional machines as "pedagogic devices to assist the understanding of some aspect of programs or programming" makes no direct reference to programming languages, programs are expressed with programming languages so we will look at these "aspects of programs" through the lens of how they are realized by some programming language[2]. If a notional machine represents a part of the operational semantics of a programming language, for example, then this representation should be sound, in the sense that steps in the notional machine correspond to steps in the operational semantics of the programming language. We say, informally, that a notional machine is *sound* if it is:

- a proper abstraction: it represents one or more aspects of the programming language and no superfluous aspects;
- consistent with the corresponding programming language: steps in the notional machine correspond to steps in the language semantics, and lead to similar results.

To show the soundness of a notional machine, or the lack of it, we need (1) a formal description of the programming language (or aspect of it), (2) a formal description of the notional machine, and (3) a formal description of the relationship between them. Notional machines that are heavily used in practice are rarely based on a comprehensive formal definition. One of the contributions of this research is the introduction of an approach to design sound notional machines by using formal descriptions.

Showing the soundness of a notional machine amounts to demonstrating that the notional machine *simulates* (in the sense described by Milner [1971]) the aspect of programs under the focus of the notional machine. This aspect under focus can be, for example, the operational semantics of a programming language. This property can be given in the form of a commutative diagram. We will present several such diagrams throughout this work.

Milner's simulation was also used by Hoare [1972] to establish a definition of the correctness of an 'abstract' data type representation with respect to its

[2]That is not to say that we will only restrict ourselves to notional machines focused on program execution but that we will restrict our analysis to aspects of programs that can be expressed in terms of the syntax or semantics of a programming language. Later, we will clarify that our approach to soundness of notional machines is, in principle, even more general.

corresponding 'concrete' data type representation. There are two interesting things about this interpretation of simulation: (1) it also captures the relationship between a notional machine and the underlying programming language because a notional machine is indeed an *abstraction* over some aspect of interest, and (2) it hints at the generality of the approach as a way to formally describe any notional machine.

1.5 From Theory to Practice

The definition of soundness for notional machines that we provide is valuable not only as a measure of the quality of a notional machine but also because it gives us a principled approach to reason about the relationship between a notional machine and the aspect of programs under its focus. We will show many applications of this reasoning framework, from the design and analysis of notional machines to the implementation of tools to support teaching using notional machines, and even the evaluation of the effectiveness of notional machines as assessment instruments.

Although we try to keep these applications well grounded in the theory, many of them stem from using this reasoning framework to reason informally about notional machines and instructors are certainly not required to write formal proofs to benefit from reasoning about notional machines using this reasoning framework.

1.6 Contributions

The contributions of this research are as follows:

- A formal definition of sound notional machine.

  This definition, presented in Section 2.1.2, gives us a principled way to reason about notional machines and their relationship with the aspect of the programming language they focus on.

  This formalism gives rise to three methodologies (*M1*, *M2*, and *M3* below).

- A methodology (*M1*) for designing notional machines that are sound by construction.

  In Chapter 2, we show how we can use the formal definition of sound notional machine to systematically design sound notional machines. We

demonstrate the methodology by applying it to a combination of various notional machines, small programming languages with well-known formalizations, and aspects of programming language semantics.

- A methodology (*M2*) for analyzing notional machines with respect to their soundness.

  In Chapter 3, we demonstrate the methodology by analyzing existing notional machines, pointing out inconsistencies, and suggesting directions for improvement. Then in Chapter 4, we apply the methodology to ExpressionTutor: a family of notional machines focused on various aspects of expressions. Differently from the previous chapters which used small programming languages for their arguments, there we use Java.

- A complete description of expression constructs in Java 11.

  Although there exist several formal descriptions of various subsets of Java, there seems to be no complete and concise description of Java's expression constructs (up to Java 11), which we describe in Chapter 4 as part of the analysis and refinement of ExpressionTutor.

- A methodology (*M3*) for evaluating the effectiveness of a notional machine as an educational assessment instrument.

  Experiments to evaluate the effectiveness of a notional machine as an assessment instrument can be systematically derived from the commutative diagram that defines the notional machine. In Chapter 5, we demonstrate the methodology by applying it to the design of a pilot study to evaluate ExpressionTutor for Java.

Finally, Chapter 6 briefly discusses related work, Chapter 7 outlines various directions for future work, and Chapter 8 concludes.

Chapter 2

Designing Sound Notional Machines

A notional machine draws attention to a particular aspect of a programming language. So we can only begin to talk about the soundness of a notional machine if we have a formal description of the programming language the notional machine is focused on. We consider a set of small programming languages with well-known formalizations described in Pierce [2002] that explore different aspects of programming language semantics. Throughout this chapter, these languages are used in various examples that explore different aspects of the design of notional machines. Table 2.1 lists the notional machines we use in this chapter as well as the corresponding programming language and aspect of the semantics of the programming language that the notional machine focuses on.

We model each programming language and notional machine in Haskell. The models are executable, so they include implementations of the programming languages (including parsers, interpreters, and type-checkers), the notional machines, and the relationship between them[1]. The soundness proofs presented in this chapter are done using equational reasoning [Gibbons, 2002; Bird, 1989].

[1] The artefact is at `https://github.com/LuCEresearchlab/sound-notional-machines`.

Section	Notional Machine	Programming Language	Focus
2.1	ExpTree	UntypedLambda	Evaluation
2.3	ExpTutorDiagram	UntypedLambda	Evaluation
2.4	TAPLMemoryDiagram	TypedLambdaRef	References
2.5	ExpTutorDiagram	TypedArith	Types

Table 2.1. Notional machines, programming languages, and notional machine focuses used throughout the chapter.

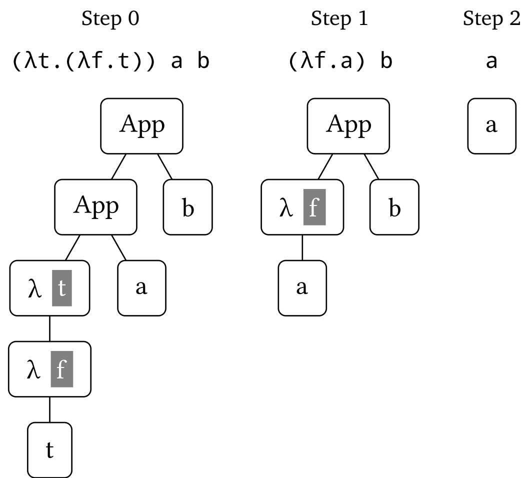


Figure 2.1. Evaluation of $(\lambda x.\ \lambda y.\ x)\ a\ b$ in programming language (top) and ExpTree notional machine (bottom).

2.1 Isomorphic Notional Machines

As a first straightforward example, let's look at a notional machine for teaching how evaluation works in the untyped lambda-calculus (we will refer to this language as UNTYPEDLAMBDA[2]). While most research papers discuss the lambda-calculus using its textual representation, textbooks sometimes illustrate it using tree diagrams [Pierce, 2002, p. 54]. We use this as an opportunity to define a simple notional machine which we call EXPTREE[3].

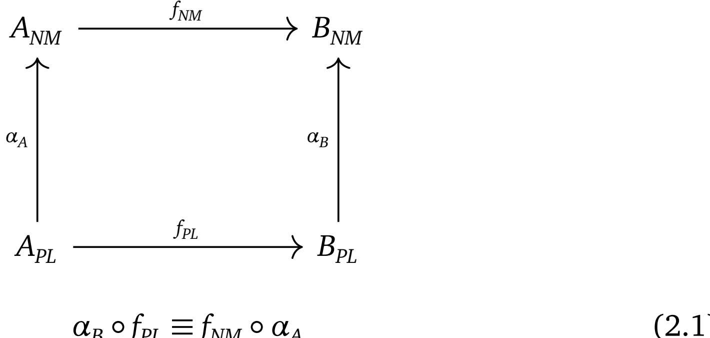
2.1.1 Illustrative Example

Figure 2.1 uses EXPTREE to demonstrate the evaluation of a specific lambda expression, which happens in two reduction steps. The top of the figure shows the terms in the traditional textual representation of the programming language, while the bottom shows the terms as a tree.

Whether or not using this notional machine indeed helps in teaching how terms in the untyped lambda-calculus get evaluated is an open question. It could be that the cost of introducing an additional (visual) representation is bigger than the benefit such a notation might provide. This question is valid also for richer and more complex notional machines. However, independent of how effective

[2]The syntax and reduction rules for UNTYPEDLAMBDA are reproduced in Appendix A.1.

[3] We use capitalized typesetting whenever we are referring to formally defined programming languages and notional machines.



$$\alpha_B \circ f_{PL} \equiv f_{NM} \circ \alpha_A \tag{2.1}$$

Figure 2.2. The soundness condition for notional machines shown both as a commutative diagram and in algebraic form.

they might be, notional machines are used in practice. Here we instead focus on a different question: how does one know whether a given notional machine is sound, and how does one design a sound notional machine.

2.1.2 Soundness via Commutative Diagrams

In general, a notional machine is sound if the diagram in Figure 2.2 commutes. We call the commutativity of this diagram the *soundness condition* for a notional machine. In this diagram, the vertices are types and the edges are functions.

The bottom layer (A_{PL}, f_{PL}, B_{PL}) represents the aspect of a programming language[4] we want to focus on. A_{PL} is an abstract representation of a program in that language. In our example, that is the abstract syntax of UNTYPEDLAMBDA (given by the type $Term_{U\lambda}$). The function f_{PL} is an operation the notional machine is focusing on. In our example, that would be *step*, a function that performs a reduction step in the evaluation of a program according to the operational semantics of the language, which in this case also produces a value of type $Term_{U\lambda}$.

The top layer of the diagram (A_{NM}, f_{NM}, B_{NM}) represents the notional machine. A_{NM} is an abstract representation of the notional machine (its abstract syntax). In our simple example, that is a type *ExpTree* trivially isomorphic to $Term_{U\lambda}$ via a simple renaming of constructors. The function f_{NM} is an operation on the notional machine which should correspond to f_{PL}. Connecting the bottom layer to the

[4]Although we refer to the bottom layer of the diagram as the programming language layer and we restrict ourselves to analyzing aspects of the syntax and semantics of programming languages, for which we have well-established formalizations, that is not an intrinsic restriction of the approach. In principle, the bottom level of the diagram can be whatever aspects of programs or programming the notional machine is focused on.

top layer, there are the functions α_A and α_B from the abstract representation of a program in the programming language to the abstract representation of the notional machine. α is also called an abstraction function.

Definition. Given the notional machine (A_{NM}, B_{NM}, $f_{NM} :: A_{NM} \rightarrow B_{NM}$), focused on the aspect of a programming language given by (A_{PL}, B_{PL}, $f_{PL} :: A_{PL} \rightarrow B_{PL}$), the notional machine is *sound* iff there exist two functions $\alpha_A :: A_{PL} \rightarrow A_{NM}$ and $\alpha_B :: B_{PL} \rightarrow B_{NM}$ such that $\alpha_B \circ f_{PL} \equiv f_{NM} \circ \alpha_A$

If the abstract representation of the programming language (A_{PL}) is isomorphic to the abstract representation of the notional machine (A_{NM}), we can construct an inverse mapping α_A^{-1} such that $\alpha_A^{-1} \circ \alpha_A \equiv id \equiv \alpha_A \circ \alpha_A^{-1}$ (we use the symbol $\equiv$ to denote equivalence, as opposed to the symbol $=$ used for definitions). In that case, we can always define a correct-by-construction operation f_{NM} on A_{NM} in terms of an operation f_{PL} on A_{PL}:

$$
\begin{aligned}
&f_{NM} :: A_{NM} \rightarrow B_{NM} \\
&f_{NM} = \alpha_B \circ f_{PL} \circ \alpha_A^{-1}
\end{aligned}
$$

In such cases, the diagram always commutes and therefore the notional machine is sound:

$$
f_{NM} \circ \alpha_A \equiv \alpha_B \circ f_{PL} \circ \alpha_A^{-1} \circ \alpha_A \equiv \alpha_B \circ f_{PL} \tag{2.2}
$$

Instantiating the commutative diagram for ExpTree and UntypedLambda yields the diagram in Figure 2.3. A dashed line indicates a function that is implemented in terms of the other functions in the diagram and/or standard primitives.

We call these isomorphic notional machines because they are isomorphic to the aspect of the programming language they focus on. Of course not every notional machine is isomorphic so throughout the next sections we will move further away from this simple example, arriving at various other instantiations of this commutative diagram.

2.2 Interlude: Abstract vs. Concrete Syntax of Notional Machines

The example we have shown of ExpTree in Figure 2.1 uses concrete images to represent the trees. But there is a distinction between the data structure that represents the notional machine and how it is visualized as well as a difference

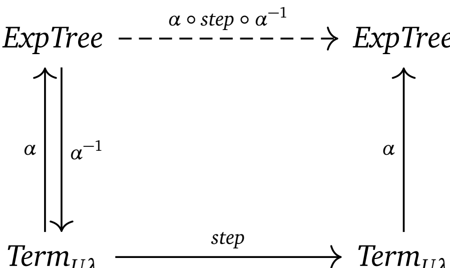


Figure 2.3. Instantiation of the commutative diagram in Figure 2.2 for the notional machine ExpTree and the programming language UntypedLambda.

between the operations on that data structure and how those transformations are enacted. Those differences are akin to the difference between the concrete and the abstract syntax of a language. Like a programming language, a notional machine has a concrete and an abstract syntax as well. We also refer to those as concrete and abstract representations of a notional machine. Notice that the concrete representation of a notional machine may not only be a diagram or image that can be depicted on paper but it could also be made of physical entities in the case of notional machines that are enacted in the real world. In fact, many notional machines are ludic in nature or are built around a metaphor, so the concrete representation of a notional machine is very important.

Table 2.2 shows the different layers at play here. The term `(λa.a) b` is shown using the concrete and abstract syntax of UNTYPEDLAMBDA. The abstract syntax is a value $a_{PL} :: Term_{U\lambda}$. The abstract representation of EXPTREE, in the notional machine layer, is the corresponding value $a_{NM} = \alpha\ (a_{PL}) :: ExpTree$ and the concrete representation is the visual representation of a_{NM}.

In this visual representation, function applications are made explicit and there is a visual distinction between places where names are introduced, shown with a gray background, and places where names are used, shown with a white background.

We will explore further concrete representations of notional machines in Section 3.3, where we describe an existing notional machine with two different concrete representations.

Programming Language		Notional Machine	
Concrete Syntax	**Abstract Syntax**	**Abstract Representation**	**Concrete Representation**
$(\lambda a.a)\ b$	*App* (*Lambda* "a" (*Var* "a")) (*Var* "b")	*AppBox* (*LambdaBox* "a" (*Box* "a")) (*Box* "b")	App λ a b a

Table 2.2. Abstract and concrete representations of a programming language and a notional machine.

2.3 Monomorphic Notional Machines

Notional machines can also serve as the basis for so-called “visual program simulation” [Sorva et al., 2013] activities, where students manually construct representations of the program execution. This effort often is supported by tools, such as interactive diagram editors, that scaffold the student’s activity. Obviously, instructors will want to see their students creating correct representations. However, to prevent students from blindly following a path to a solution prescribed by the tool, the visual program simulation environment should also allow *incorrect* representations.

ExpressionTutor[5] is such an educational tool to teach the structure, typing, and evaluation of expressions in programming courses. ExpressionTutor allows, among other things, for students to interactively construct expression tree diagrams given a source code expression. The tool is language agnostic so each node can be freely constructed (by the instructor or the student) to represent nodes of the abstract syntax tree of any language. Nodes can contain any number of holes that can be used to connect nodes to each other. Each hole corresponds to a place in an abstract syntax tree node where an expression would go. The tool allows for nodes to be connected in a variety of ways, deliberately allowing for incorrect structures that not only may not be valid abstract syntax trees of a given programming language but may not even be trees. Even the root node (labeled

[5] expressiontutor.org

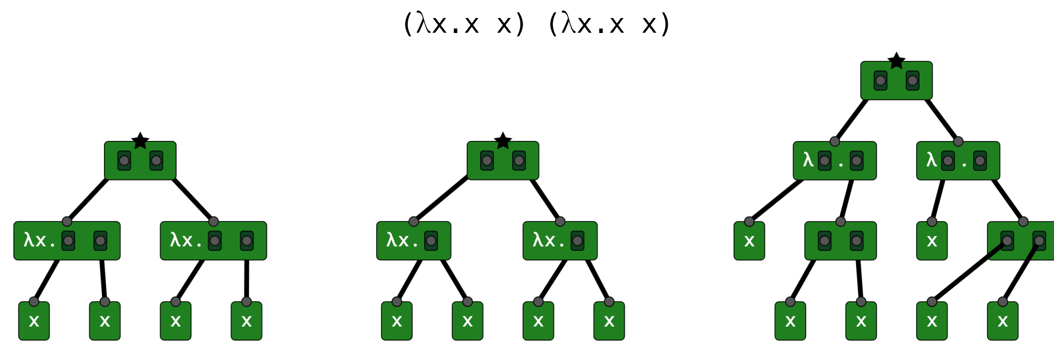


Figure 2.4. The omega combinator in UntypedLambda (top) and (incorrect) representations in ExpTutorDiagram notional machine (bottom).

with a star) has to be explicitly labeled by the student, so it is not guaranteed to exist in every diagram.

We define the notional machine EXPTUTORDIAGRAM, which models the behavior of ExpressionTutor. The fact that ExpressionTutor allows students to construct incorrect expression tree diagrams means that the abstraction function α is not bijective, as was the case of EXPTREE's α. Such incorrect diagrams do not correspond to programs, thus α is deliberately not surjective.

2.3.1 Illustrative Example

Figure 2.4 uses EXPTUTORDIAGRAM to represent the omega combinator. The top shows the textual form on the level of the programming language. Below that are three different incorrect representations students could produce. The left tree collapses the `x x` applications into the lambda abstraction. The middle tree similarly does this, but it preserves the structure of the lambda abstraction node, while violating the well-formedness of the tree by plugging two children into the same hole. The right tree shows a different problem, where the definition of the name is pulled out of the lambda abstraction and represented as a variable node instead.

2.3.2 Commutative Diagram

In general, if the mapping α_A, from the abstract representation of the programming language (A_{PL}) to the abstract representation of the notional machine (A_{NM}), is an injective but non-surjective function, we can still define the operations on A_{NM} in terms of the operations on A_{PL}. For this we define a function $\alpha_A^{\circ} :: A_{NM} \rightarrow \mathit{Maybe}\ A_{PL}$

to be a left inverse of α_A such that $\alpha_A^\circ \circ \alpha_A \equiv \mathit{return}$ (we use *return* and *fmap* to refer to the *unit* and *map* operations on monads). Here we modeled the left inverse using a *Maybe* but another monad could be used, for example, to capture information about the values of type A_{NM} that do not have a corresponding value in A_{PL}. The top-right vertex of the square (B_{NM}) in this case is the type $\mathit{Maybe}\ B'_{NM}$ and the mapping α_B can be implemented in terms of a mapping $\alpha'_B :: B_{PL} \rightarrow B'_{NM}$ like so:

$$\alpha_B :: B_{PL} \rightarrow B_{NM}$$
$$\alpha_B = \mathit{return} \circ \alpha'_B$$

Using the left inverse α_A° and α'_B, we define the operation on A_{NM} as follows:

$$f_{NM} :: A_{NM} \rightarrow B_{NM}$$
$$f_{NM} = \mathit{fmap}\ \alpha'_B \circ \mathit{fmap}\ f_{PL} \circ \alpha_A^\circ$$

This square commutes like so:

$$\begin{array}{ll} & f_{NM} \circ \alpha_A \\ \equiv & \{ \text{definition of } f_{NM} \} \\ & \mathit{fmap}\ \alpha'_B \circ \mathit{fmap}\ f_{PL} \circ \alpha_A^\circ \circ \alpha_A \\ \equiv & \{ \alpha_A^\circ \text{ is left inverse of } \alpha_A \} \\ & \mathit{fmap}\ \alpha'_B \circ \mathit{fmap}\ f_{PL} \circ \mathit{return} \\ \equiv & \{ \text{third monad law} \} \\ & \mathit{fmap}\ \alpha'_B \circ \mathit{return} \circ f_{PL} \end{array}$$

$$\begin{array}{ll} & \alpha_B \circ f_{PL} \\ \equiv & \{ \text{definition of } \alpha_B \} \\ & \mathit{return} \circ \alpha'_B \circ f_{PL} \\ \equiv & \{ \text{third monad law} \} \\ & \mathit{fmap}\ \alpha'_B \circ \mathit{return} \circ f_{PL} \end{array}$$

We can use this result to instantiate the commutative diagram of Figure 2.2 for EXPTUTORDIAGRAM and UNTYPEDLAMBDA, shown in Figure 2.5. A_{PL} is defined to be the type *ExpTutorDiagram*, which essentially implements a graph. Each node has a top plug and any number of holes, which contain plugs. Edges connect plugs. That allows for a lot of flexibility in the way nodes can be connected. ExpressionTutor is language agnostic but we can only talk about soundness of a notional machine with respect to some language and some aspect of that language.

Here we use ExpressionTutor as a notional machine focused on evaluation but ExpressionTutor can also be used, with small modifications, to focus on other aspects of programming languages. In Section 2.5 we will show how to use it to focus on types and in Section 4.4 we will use it to focus on parsing. Using our definition, one notional machine focuses on one aspect (corresponding to one function) so ExpressionTutor is what we call a family of notional machines, that we have to “instantiate” for a given aspect of focus and a given programming

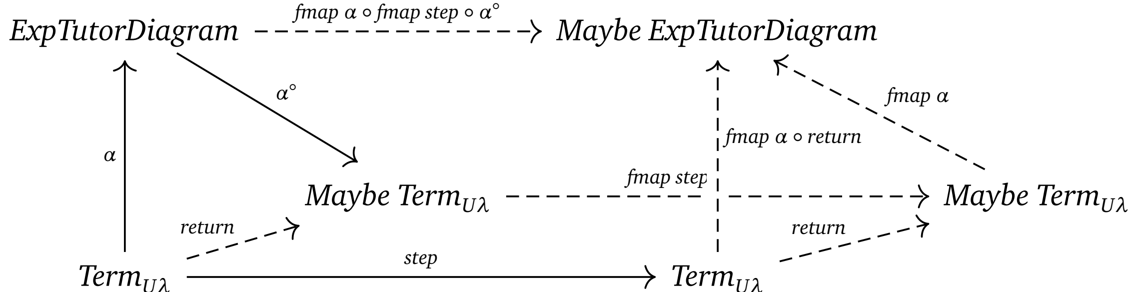


Figure 2.5. Instantiation of the commutative diagram in Figure 2.2 for the notional machine ExpTutorDiagram

language. The language considered here is again UNTYPEDLAMBDA (denoted again by the type $Term_{U\lambda}$) with f_{PL} again equal to *step*.

The construction of a mapping $\alpha :: Term_{U\lambda} \rightarrow ExpTutorDiagram$ is straight forward because a term $t :: Term_{U\lambda}$ forms a tree and from it we can always construct a corresponding ExpressionTutor diagram $d :: ExpTutorDiagram$ (a graph). For each possible term in $Term_{U\lambda}$, we need to define a pattern for the content of the corresponding *ExpTutorDiagram* node which will help the student identify the kind of node. The construction of the left inverse mapping $\alpha^{\circ} :: ExpTutorDiagram \rightarrow Maybe\ Term_{U\lambda}$ requires more care. We need to make sure that the diagram forms a proper tree and that the pattern formed by the contents of each *ExpTutorDiagram* node corresponds to a possible $Term_{U\lambda}$ node, besides making sure that they are connected in a way that indeed corresponds to a valid $Term_{U\lambda}$ tree. Using pattern synonyms [Pickering et al., 2016], we can make sure that the same patterns used to determine the contents of an *ExpTutorDiagram* node for a given $Term_{U\lambda}$ node (used in the implementation of α) are also used to implement the left inverse mapping α°.

In the next section, we construct another commutative diagram where f_{NM} is defined using f_{PL} and a left inverse mapping α°_{A}. To emphasize that point and simplify the diagrams, we will depict the left inverse in the diagram as a dotted line pointing from A_{NM} to A_{PL} (even though α°_{A} is of type $A_{NM} \rightarrow Maybe\ A_{PL}$ and not $A_{NM} \rightarrow A_{PL}$) and omit the path via $Maybe\ A_{PL}$ as shown in Figure 2.6.

We call these monomorphic notional machines because there is a monomorphism (injective homomorphism) between the notional machine and the aspect of the programming language it focuses on. This is the case here by design, to allow students to make mistakes by constructing wrong diagrams that don't correspond

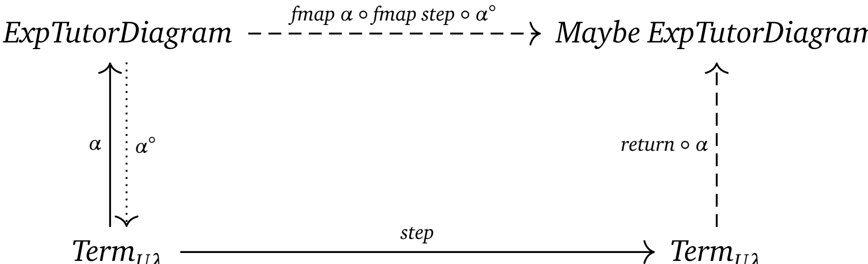


Figure 2.6. Simplified version of the commutative diagram for the notional machine ExpTutorDiagram shown in Figure 2.5

to programs. In general, this will be the case whenever there are values of A_{NM} (the abstract syntax of the notional machine) that have no correspondence in the abstract representation of the language (A_{PL}). That's often the case in memory diagrams [Holliday and Luginbuhl, 2004; Dalton and Kreahling, 2010; Dragon and Dickson, 2016] (notional machines used to show the relationship between programs and memory) because they typically allow for the representation of memory states that cannot be produced by legal programs. We show an example of such a notional machine in the next section.

2.4 A Notional Machine to Reason About State

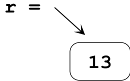
A common use of notional machines is in the context of reasoning about state[6]. An example of the use of a visual notation to represent state can be found in one of the most widely used programming language foundations textbooks, Pierce's Types and Programming Languages (TAPL) [Pierce, 2002, p. 155]. In Chapter 13 ("References"), the book extends the simply typed lambda-calculus with references (a language we will refer to as TypedLambdaRef[7]). It explains references and aliasing by introducing a visual notation to highlight the difference between a *reference* and the *cell* in the store that is pointed to by that reference. We will refer to this notation, which we will develop into a notional machine, as TAPLMemoryDiagram. In this notation, references are represented as arrows and cells are represented as rounded rectangles containing the representation of

[6] 29 of the 57 notional machines presented on https://notionalmachines.github.io/notional-machines.html at the time of this writing refer to the concept of *variable* or *array element*.

[7] The syntax and reduction rules for TypedLambdaRef are reproduced in Appendix A.3.

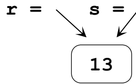
the value contained in the cell. Before designing the notional machine, we need to see the context in which this notation is used in the book.

The book first uses this notation to explain the effect of creating a reference. It shows that when we reduce the term `ref 13` we obtain a reference (a store location) to a store cell containing `13`. The book then represents the result of binding the name `r` to such a reference with the following diagram:



In the book, this operation is written as `r = ref 13`, but as we will see in the next Section, this form of name binding (`name = term`) exists only in a REPL-like context which is not part of the language.

The book continues explaining that we can "make a copy of `r`" by binding its value to another variable `s` (with `s = r`) and shows the resulting diagram:



The book then explains that one can verify that both names refer to the same cell by *assigning* a new value to `s` and reading this value using `r` (for example, the term `s := 82; !r` would evaluate to `82`). Right after, the book suggests to the reader an exercise to "draw a similar diagram showing the effects of evaluating the expressions `a = {ref 0, ref 0}` and `b = (λx:Ref Nat.{x, x}) (ref 0)`." Although we understand informally the use of this diagram in this context, how can we know what a correct diagram would be in general for any given program? This is what we aim to achieve by designing a notional machine based on this notation.

2.4.1 Designing a Notional Machine

Let's see how we would turn that kind of diagram into a sound notional machine. We want to construct a commutative diagram where A_{PL} is an abstract representation of the state of a TYPEDLAMBDAREF program execution, A_{NM} is an abstract representation of the diagram presented in the book, and f_{PL} is an operation that affects the state of the store during program execution.

In a first attempt, let's choose f_{PL} to be an evaluation step and A_{PL} to be modeled as close as possible to the presentation of a TYPEDLAMBDAREF program as described in the book. In that case, A_{PL} is the program's abstract syntax tree together with a *store*, a mapping from a location (a reference) to a value.

Problem: Beyond the Language

The first challenge is that the name-binding mechanism used in the examples above (written as `name = term`) exists only in a REPL-like context in the book used for the convenience of referring to terms by name. It is actually not part of the language (TypedLambdaRef) so it is not present in this representation of A_{PL} and as a result it cannot be mapped to A_{NM} (the notional machine). We will avoid this problem by avoiding this name-binding notation entirely and writing corresponding examples fully in the language. The only mechanism actually in the language to bind names is by applying a lambda to a term. Let's see how we can write a term to express the behavior described in the example the book uses to introduce the diagram (shown earlier), where we:

1. Bind `r` to the result of evaluating `ref 13`
2. Bind `s` to the result of evaluating `r`
3. Assign the new value `82` to `s`
4. Read this new value using `r`

Using only the constructs in the language, we express this with the following term:

```
(λr:Ref Nat.(λs:Ref Nat.s := 82; !r) r) (ref 13)
```

Problem: Direct Substitution

The problem now is that if we model A_{PL} and evaluation as described in the book, the result of reducing a term $(\lambda x.t_1)\ t_2$ is the term obtained by replacing all free occurrences of `x` in t_1 by t_2 (modulo alpha-conversion), so we don't actually keep track of name binding information. What we have in A_{PL} at each step is an abstract syntax tree and a store. But we have no information about which names are bound to which values, because the names were already substituted in the abstract syntax tree. We need to change A_{PL} and `step` to capture this information, keeping not only a store but an explicit name environment that maps names to values, and only substituting the corresponding value when we evaluate a variable. Like the definition of application in terms of substitution, we have to be careful to avoid variable capture by generating fresh names when needed.

Strictly speaking, we could have kept A_{PL} as just a term and a store (without the name environment). In fact, that's enough to do Exercise 13.1.1, for example,

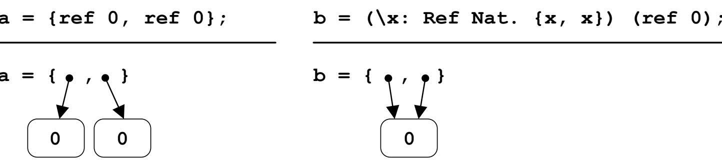


Figure 2.7. TAPLMemoryDiagram for TypedLambdaRef for TAPL Exercise 13.1.1

whose solution is shown in Figure 2.7. But the absence of the name environment makes it less suitable to talk about aliasing, because even though a term may contain, at any given point during evaluation, multiple occurrences of the same location (represented by multiple arrows pointing to the same store cell), it is not possible to know if these locations correspond to different names, and one may need to trace several reduction steps back to find out when a name was substituted by a location.

2.4.2 Illustrative Example

Figure 2.8 shows two variations of the notional machine being used to explain the evaluation of the term we had described before. It shows the state after each reduction step, on the left without an explicit name environment and on the right with an explicit name environment. Between each step, a line with the name of the applied reduction rule is shown. Notice that the representation with a name environment requires extra name lookup steps.

In both variations, the representation of the program (the term) being evaluated appears first (with gray background). Each term is actually an abstract syntax tree, which we represent here, like in the book, with a linearized textual form. Location terms are represented as arrows starting from where they appear in the abstract syntax tree and ending in the store cell they refer to.

The naming environment is shown as a table from variable names to terms. Store cells also contain terms. This means the textual representation of terms that appear both inside name environments and inside cells may also contain arrows to other cells.

2.4.3 Commutative Diagram

Similar to the abstract representation of the program execution, the abstract representation of the notional machine contains three parts: one for the program being evaluated, one for the name environment, and one for the store.

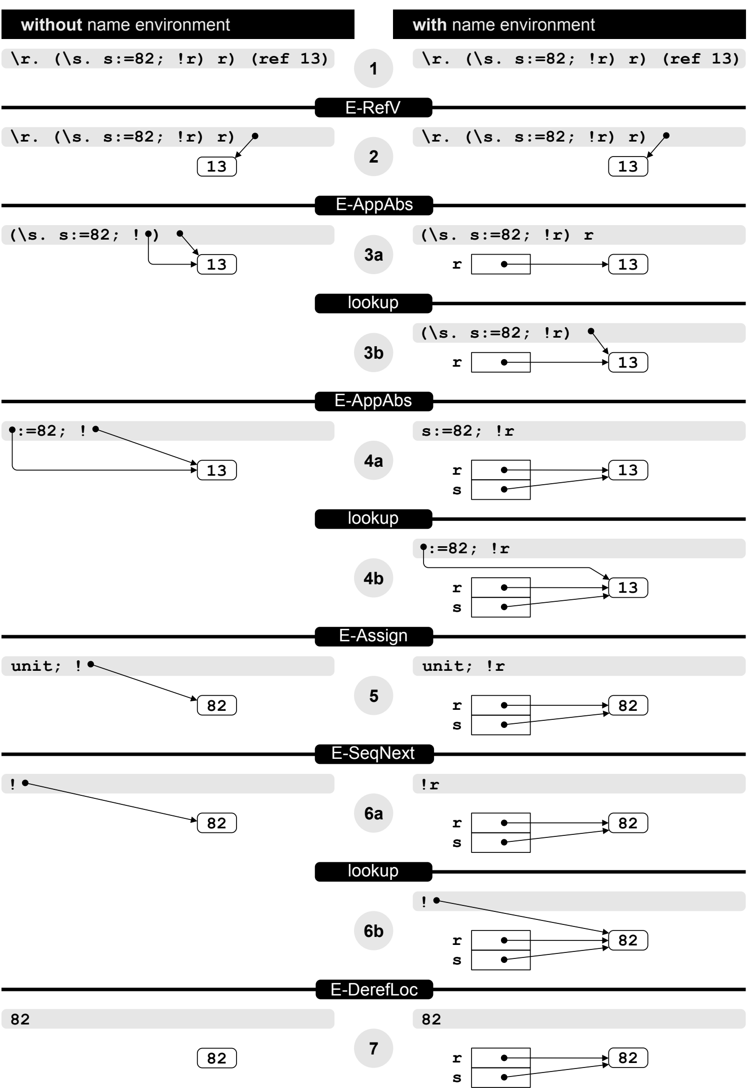


Figure 2.8. Trace of `(λr:Ref Nat.(λs:Ref Nat.s := 82; !r) r) (ref 13)` in TAPLMemoryDiagram for TypedLambdaRef

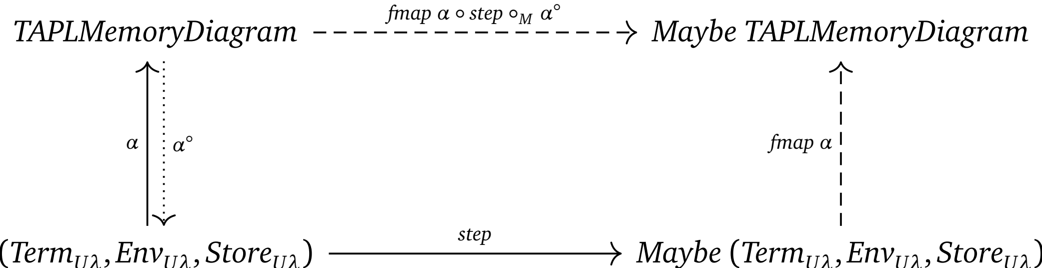


Figure 2.9. Instantiation of the commutative diagram in Figure 2.2 for the notional machine TAPLMemoryDiagram and the programming language TypedLambdaRef.

$$
\begin{array}{l}
\textbf{data}\ \mathit{TAPLMemoryDiagram}\ l = \mathit{TAPLMemoryDiagram}\ \{\\
\quad \mathit{memDiaTerm} :: \mathit{DTerm}\ l,\\
\quad \mathit{memDiaNameEnv} :: \mathit{Map}\ \mathit{Name}\ (\mathit{DTerm}\ l),\\
\quad \mathit{memDiaStore} :: \mathit{Map}\ (\mathit{DLocation}\ l)\ (\mathit{DTerm}\ l)\}
\end{array}
$$

The type *DLocation* corresponds to arrow destinations (arrow endpoints). A term is represented as a rose tree of *String*s augmented with a case for location.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{DTerm}\ l & = & \mathit{Leaf}\ \mathit{String}\\
 & \mid & \mathit{Branch}\ [\mathit{DTerm}\ l]\\
 & \mid & \mathit{TLoc}\ (\mathit{DLocation}\ l)
\end{array}
$$

The concrete representation of a *DTerm* can be in linearized text form or as a tree akin to that shown in Section 2.1. The representation of the nodes in a *DTerm* tree that are *TLoc* are shown as arrow starting points. These arrows end in the cell corresponding to the *DLocation* in each *TLoc*. The concrete representation of the store relates the visual position of each cell with the *DLocation* of each cell. That leads to the commutative diagram in Figure 2.9, where we use the symbol $\circ_M$ to denote monadic function composition (the fish operator <=< in Haskell).

The process of reworking A_{PL} and f_{PL} to expose an explicit name environment shows that if a given choice of types (A_{PL}) and functions (f_{PL}) doesn't allow for the construction of the commutative diagram for a notional machine, that doesn't mean the notional machine is necessarily unsound. The soundness of a given

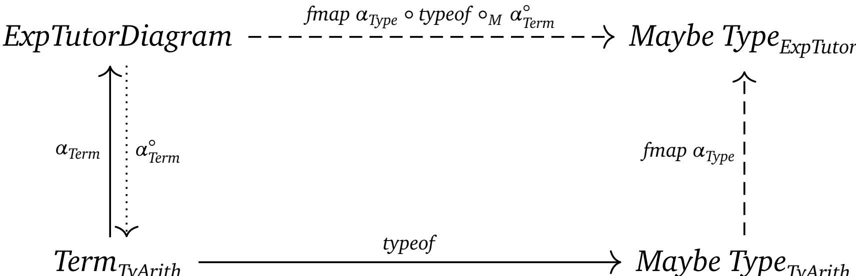


Figure 2.10. First attempt at instantiating the commutative diagram in Figure 2.2 for a notional machine that focuses on type-checking using the programming language TypedArith.

notional machine is predicated on the existence of an A_{PL} and f_{PL} compatible with the semantics of the language, but the construction of the specific types and functions that make the diagram commute may not be trivial.

2.5 A Notional Machine to Reason About Types

So far we have seen examples of commutative diagrams where f_{PL} is *step* (a function that performs a reduction step) but in principle, f_{PL} could be any operation on A_{PL} that is the focus of a given notional machine. Let's look at an example of notional machine where we do not focus on evaluating but on typing an expression. The language this notional machine focuses on is TypedArith[8], a language of typed arithmetic expression, which is the simplest typed language introduced in TAPL [Pierce, 2002, p. 91]. We will try two approaches.

In the first approach, represented in the diagram in Figure 2.10, the data type used for the notional machine (A_{NM}) is *ExpTutorDiagram*, used in Section 2.3. We represent a program in TypedArith with the type $Term_{TyArith}$ and the operation we focus on is $typeof : Term_{TyArith} \rightarrow Maybe\ Type_{TyArith}$, a function that gives the type of a term (for simplicity we use a *Maybe* here to capture the cases where a term is not well-typed). As in Section 2.3, the abstraction function has a left inverse, here α_{Term} and α°_{Term} respectively, which we use to produce f_{NM}. The notional machine operation f_{NM} produces a notional machine-level representation of maybe the type of a term.

[8] The syntax and typing rules for TypedArith are reproduced in Appendix A.2.

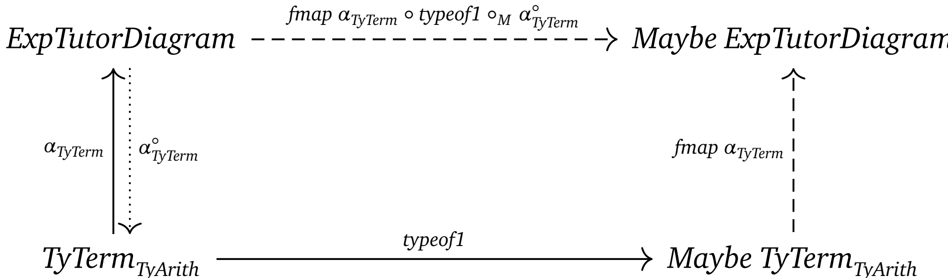


Figure 2.11. Second attempt at instantiating the commutative diagram in Figure 2.2 for a notional machine that focuses on type-checking using the language TypedArith. The notional machine now exposes the inner workings of the typing algorithm.

As is, a student may benefit from the notional machine's representation of the program's abstract syntax tree and that may be helpful to reason about typing but the notional machine doesn't expose to the student the inner workings of the process of typing a term.

The second approach, represented in the diagram in Figure 2.11, tackles this issue by enriching the notional machine in a way that allows it to go step-by-step through the typing algorithm. The idea is that f_{NM} now doesn't produce a type but gradually labels each subtree with its type as part of the process of typing a term. For this, *ExpTreeDiagram* has to be augmented so that each node may have an associated type label. For convenience, we still want to write f_{NM} in terms of f_{PL}. The key insight that enables this is to change f_{PL} from *typeof* to *typeof1*. The difference between *typeof* and *typeof1* is akin to the difference between big-step and small-step semantics: *typeof1* applies a single typing rule at a time. As a result, we have to augment our representation of a program by bundling each term with a possible type (captured in type $TyTerm_{TyArith}$). The abstraction function and its left inverse are updated accordingly. The resulting diagram for the notional machine is shown in Figure 2.12.

Interestingly, given an expression e, once we label all nodes in the ExpressionTutor diagram of e with their types, the depiction of the resulting diagram is similar to the typing derivation tree of e.

Note that types are themselves trees but here we're representing them in a simplified form as textual labels because the primary goal of ExpressionTutor is

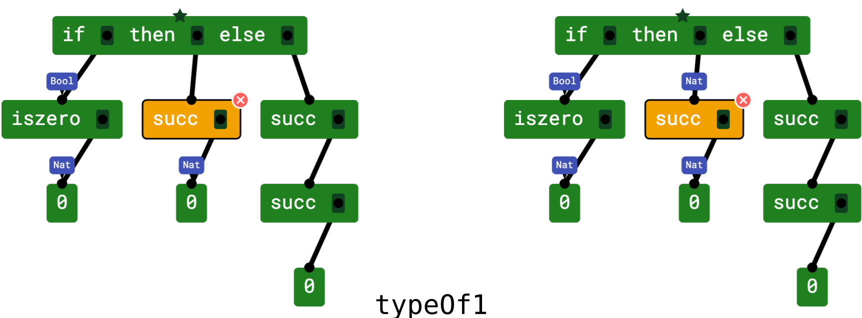


Figure 2.12. One step in the notional machine ExpTutorDiagram as it types the term `if iszero 0 then succ 0 else succ succ 0` in the language TypedArith.

to represent the structure of terms, not the structure of types. Representing the structure of types remains future work.

2.6 Conclusion

In this chapter, we have defined sound notional machines and shown how we can use the definition to devise a methodology to design notional machines that are sound by construction. We used the methodology to design different notional machines that focus on various aspects of multiple programming languages. In the next chapter, we will use the definition to devise a methodology for analyzing existing notional machines.

Chapter 3

Analyzing Existing Notional Machines

So far we have seen examples where we constructed f_{NM} using f_{PL} and functions that convert between A_{PL} and A_{NM}. Now let's look at two notional machines where that is not the case. Here an informal A_{NM} and α are given together with a description of f_{NM} completely in terms of A_{NM}. The idea is to use these descriptions to construct a commutative diagram that relates the notional machine and the corresponding programming language and in the process uncover inconsistencies in the notional machine as well as suggest improvements that eliminate those inconsistencies.

We will show two kinds of problems that may arise during the construction of the commutative diagram: (1) it may not be possible to construct a mapping α from A_{PL} to A_{NM}, because a certain construct in A_{PL} cannot be expressed in A_{NM} (i.e. α is not total), which we show in Section 3.1; (2) even though we may be able construct the types of the commutative diagram and connect them with total functions, there may be values flowing between these functions for which the diagram doesn't commute, which we show in Section 3.2.

Section	Notional Machine	Programming Language	Focus
3.1	REDUCT	UNTYPEDLAMBDA	Evaluation
3.2	ALLIGATOR	UNTYPEDLAMBDA	Evaluation

Table 3.1. The two notional machines used in this chapter, both described here focusing on evaluation of untyped lambda calculus terms. In the original paper, Reduct focuses on JavaScript. For simplicity, here we use untyped lambda calculus.

3.1 Reduct

Notional machines can serve as a basis for educational games. One such example is Reduct, an online game to teach “core programming concepts which include functions, Booleans, equality, conditionals, and mapping functions over sets” Arawjo et al. [2017]. Reduct aims to represent a subset of JavaScript ES2015. The gameplay of Reduct tightly interleaves program construction and program evaluation. Each level of the game has a goal: to reduce terms to an expected value. Within a level, at any point in time, the canvas contains a given number of independent game pieces that correspond to terms in JavaScript. The player clicks and drops these game pieces on each other both to compose them and to "reduce" them (reduce the terms they correspond to).

Like other notional machines, REDUCT employs metaphors to allow learners to reuse their understanding of the real world when learning the semantics of the programming language. Table 3.2 shows some of REDUCT’s metaphors[1]. Star, square, and triangle *shapes* represent the literal String values `"star"`, `"square"`, and `"rectangle"`. A *key* represents the Boolean value `true`, and a *broken key* represents `false`. A *lock* that protects a term represents a conditional operator. The lock’s keyhole takes on a condition (a term that produces a key). The term protected by the lock corresponds to the term to evaluate if the condition holds; the value produced if the condition does not hold, `null`, is not visually represented. A *reflecting glass*, with space for a term on either side, represents the equality operator. A *metal plate* represents a lambda abstraction: the circular hole on its left represents the variable binding, and the other hole on its right holds the lambda’s body. A *pipe* that sticks out of the canvas represent variable use. A value dropped in the metal plate’s circular hole materializes in each of the pipes connected to the plate.

While a metal plate with a hole and a pipe are two separate constructs in some levels of the Reduct game, most often they are inseparably connected. We will refer to this combination of one or more pipes connected to a plate with a hole as a *HolePipe*.

[1]Reduct introduces the idea of “concreteness fading”: while progressing through the levels of the game, the pieces a player gets to use become gradually more abstract. At the start, at the most concrete end of the spectrum, Reduct uses the metaphors described above. In the end, at the most abstract end of the spectrum, it uses blocks that essentially contain JavaScript code.

	Programming Language *JavaScript terms*	Notional Machine *Reduct pieces*
Literal value	`"star"`, `"triangle"`, and `"square"`	, , and
Boolean	`true` or `false`	or
Conditional	`x ? y : null`	
Comparison	`x == y`	
Variable	`x`	
Abstraction	`x => t`	
Variations on Abstraction	`x => x`	λx
	`x => [t1, t2]`	
	`x => [x, x]`	λx
	`x => [x, x, x]`	
	`x => x == x`	λx ==

Table 3.2. Metaphors used in Reduct

3.1.1 Illustrative Example

Figure 3.1 shows the step-by-step solution of level 17 of the Reduct game[2]. In Step 1, the canvas contains three independent pieces: a HolePipe, a star, and a reflecting glass. The goal of this level, shown in the top left corner, is to produce a key. To make progress, the player should grab the star and drop it into the HolePipe's hole. This results in the HolePipe and the star disappearing, and two stars (one for each pipe) appearing. That step is supposed to correspond to the application of a lambda to a term. This correspondence is not accurate, as we will show. In steps 2 and 3, the player should plug the stars into the holes around the reflecting glass, which triggers the evaluation of the comparison and produces a key. That step corresponds to reducing the term `"star" == "star"` to obtain `true`.

3.1.2 Commutative Diagram

The first step is to define A_{NM}, A_{PL}, and a mapping α from A_{PL} to A_{NM}. A_{PL} should be a type representing the terms of a subset of JavaScript as described in the paper, but for convenience, we will focus on a subset of terms that is effectively equivalent to the untyped lambda-calculus. Because of that, we can define A_{PL} to be $Term_{U\lambda}$. Let's define A_{NM} to be *ReductTerm*, a type representing the constructs in REDUCT. We will consider just the constructs that correspond (as closely as possible) to the terms represented with $Term_{U\lambda}$. Our first construct to consider is the HolePipe, which corresponds to a lambda abstraction.

The first thing to notice is that a HolePipe can contain multiple pipes next to each other, as shown in Figure 3.1. The effect of these pipes is simply to create more occurrences of the term that is dropped in the hole. This construct has no direct equivalent in the lambda-calculus, because placing a term in front of another means application in lambda-calculus and not "multiple returns". Perhaps pipes next to each other could correspond to some kind of list term on the language level so we would have to augment the language to add a term for that. But we don't need to worry about this mismatch. In itself, this mismatch is not a problem. It is still possible to have a commutative diagram if there are constructs in A_{NM} that cannot be directly mapped to A_{PL}. What is really missing in Reduct is a construct corresponding to function application. The intention of the authors was to represent function application by dropping a term into the hole of a HolePipe construct. When a term is dropped in the hole of a HolePiple, the HolePiple and the dropped term are replaced by as many copies of the dropped term as

[2] `https://www.therottingcartridge.com/games/programming/?level=17`

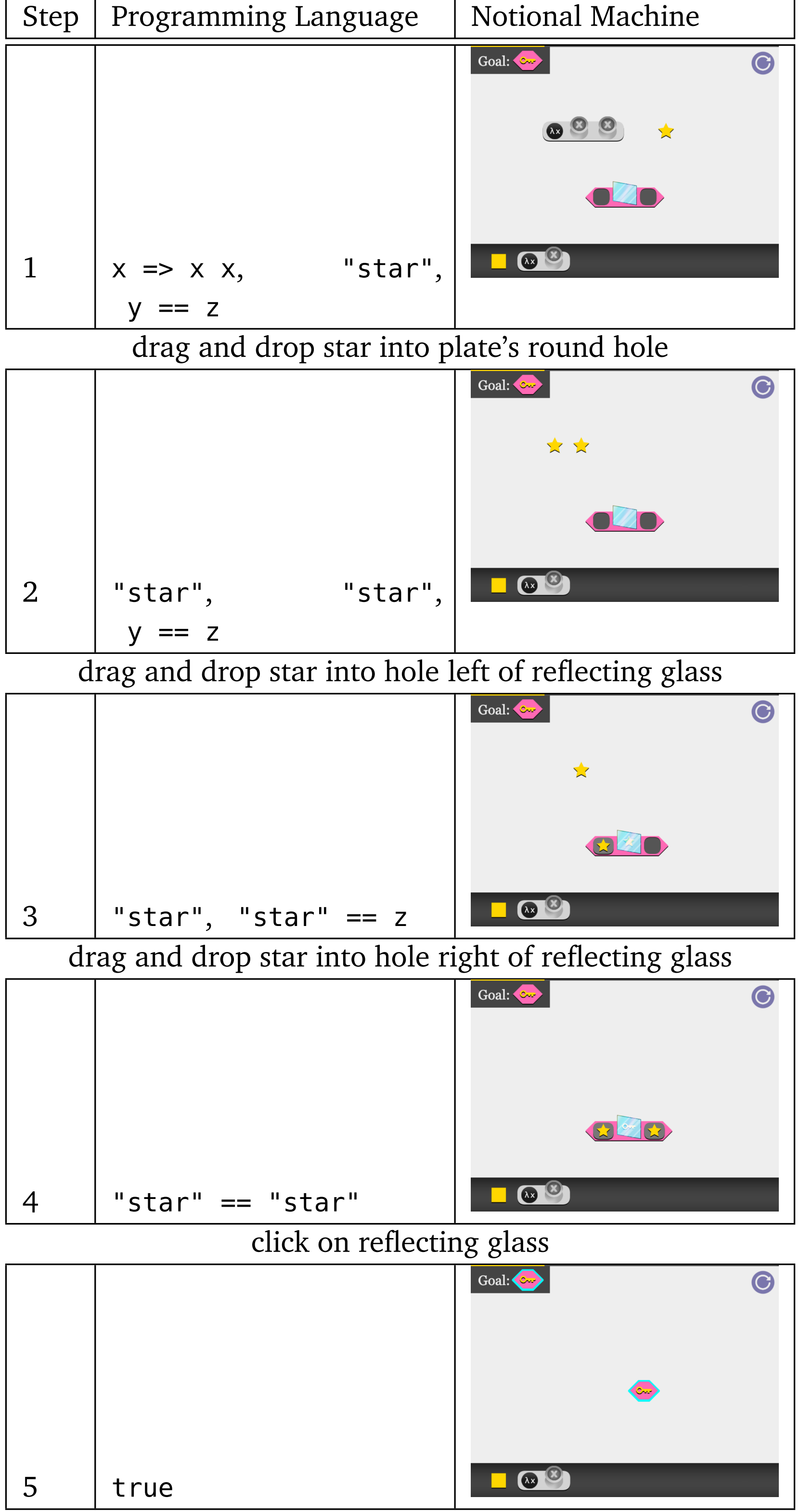


Figure 3.1. Evaluation in programming language (left) and Reduct notional machine (right).

there are pipes in the HolePipe. That would correspond to application, but more precisely it corresponds to the operational behavior of an application, not to the construction of an application term. Constructing an application term is not the same as applying it immediately.

Without an application term, it is impossible to write several kinds of programs. The Y-combinator, for example, which is used to construct recursive programs, contains application terms, and an essential part of its behavior is that, during execution, the terms that are part of these application terms will be substituted to different terms as the recursion unfolds. In general, there are several kinds of programs that contain application terms $t_1\ t_2$ where t_1 and t_2 cannot be statically known but depend on the runtime behavior of the program.

Notably, this mismatch between the notional machine and the programming language is not evident because the only "programs" one can write using Reduct are the ones made out of the building blocks provided in each level of the game. Effectively, the gameplay is a form of "puzzle solving" that corresponds to a mix of *constructing* and *reducing* terms. In a programming language, however, one does not modify the program during its execution.

To solve this mismatch, we need to not only add a construct in Reduct that corresponds to application, but we need also to adapt the way the player interacts with the game to "solve the puzzle". Adding a construct in Reduct is simple. More challenging is adapting the gameplay. Part of the challenge is that once programs are constructed, the only way a player should be able to interact with it to produce a given final result is by providing inputs. At the same time, we don't want to lose the stepwise reduction of terms triggered by the player, which can be very instructive. For this, there should be a way to distinguish between the moment when a program is built and the moment the program is run. This distinction could be directly controlled by the player, who could explicitly say when the program is ready. If a program doesn't depend on inputs, it can be run directly by clicking on the term to trigger reduction steps (similar to the current behavior). Programs that depend on inputs could, for simplicity, be expressed as terms where the root is a lambda abstraction. To run these programs, the player would drop a term into the outermost lambda and subsequently click to trigger the next reduction steps. An important difference from the current gameplay is that once a term is being executed it cannot be changed and a "reset" button would be needed to show the initial state of the program in case the desired final term is not reached.

In essence, it is not possible to construct a mapping α from $Term_{U\lambda}$ to an abstract representation of REDUCT because a lambda application node doesn't have an adequate correspondence in REDUCT.

3.2 Alligator Eggs

Alligator Eggs[3] is a game conceived by Bret Victor to introduce the lambda-calculus in a playful way. It is essentially a notional machine for the untyped lambda-calculus. The game has three kinds of pieces and is guided by three rules.

Pieces The pieces are *hungry alligators*, *old alligators*, and *eggs*. Old alligators are white, while hungry alligators and eggs are colored with colors other than white. The pieces are placed in a plane and their relative position with respect to each other determines their relationship. All pieces placed under an alligator are said to be guarded by that alligator. An alligator together with the pieces that may be guarded by it form a family. Families placed to the right of another family may be eaten by the guardian of the family on the left, depending on the applicability of the gameplay rules. Every egg must be guarded by an alligator with the same color (this must be a hungry alligator because eggs cannot be white).

Rules There are three rules that determine what we can call the “evolution of families" over time: the *old age rule*, the *color rule*, and the *eating rule*.

- Old age rule: If an old alligator is guarding only one egg or one family (which itself may be composed of multiple families), then the old alligator dies and is removed.
- Eating rule: If there is a family guarded by a hungry alligator in the plane and there is a family or egg to its right, then the hungry alligator eats the entire family (or egg) to its right and the pieces of the eaten family are removed. The alligator that ate the pieces dies and the eggs that were guarded by this alligator and that have the same color of this alligator are hatched and are replaced by a copy of what was eaten by the alligator.
- Color rule: Before a hungry alligator *A* can eat a family *B*, if a color appears both in *A*'s proteges and in *B*, then that color is changed in one of the families to another color different from the colors already present in these families.

Gameplay A few suggestions of gameplay are provided. An option would be to build a series of puzzles that challenge the player to find out, given a set of families organized in the plane, what should be the color of some of the pieces

[3] http://worrydream.com/AlligatorEggs/

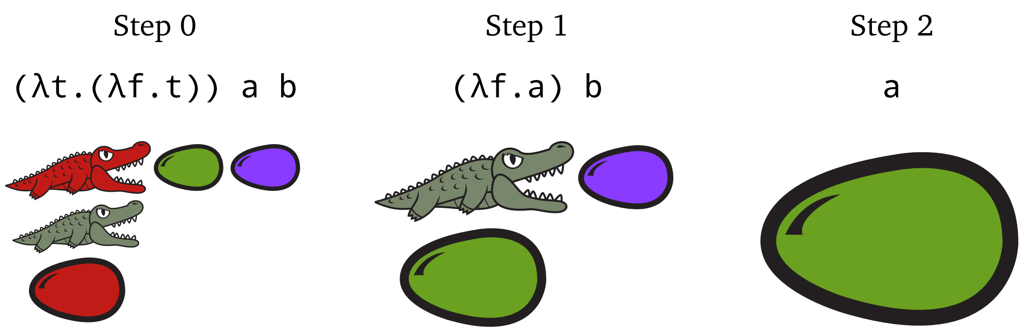


Figure 3.2. Evaluation of `(λt.(λf.t)) a b` in the untyped lambda calculus (top) and Alligator notional machine (bottom).

for the families to evolve into another given family. Similarly, the player could be asked to devise a family that when fed X produces Y. These puzzles could be embedded into a board game, where the player needs to solve puzzles to make progress.

According to their description, the way ALLIGATOR relates to the untyped lambda-calculus is as follows: "A hungry alligator is a lambda abstraction, an old alligator is parentheses, and eggs are variables. The eating rule corresponds to beta-reduction. The color rule corresponds to (over-cautious) alpha-conversion. The old age rule says that if a pair of parentheses contains a single term, the parentheses can be removed". Although very close, this relation is not completely accurate. We will identify the limitations and propose solutions.

3.2.1 Illustrative Example

Figure 3.2 shows a representation of the evaluation of the lambda-calculus term `(λt.(λf.t)) a b` using the ALLIGATOR notional machine. Step 0 shows the original term, consisting of two alligators for the two lambdas, a red egg for the term `t`, and a green and purple egg for the terms `a` and `b` respectively. In the first step, the red alligator eats the green egg: the alligator disappears, and its red egg is replaced by a green egg. In the second step, the grey alligator eats the purple egg: the alligator disappears, and given that there was no gray egg below the alligator, the purple egg is consumed without leaving any trace. We are left with the green egg.

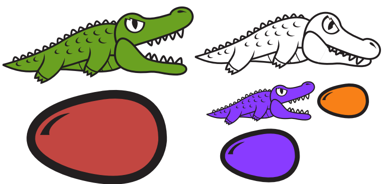


Figure 3.3. Old alligator in `(λa.y) ((λb.b) c)`

3.2.2 Commutative Diagram

To build a commutative diagram for ALLIGATOR, we need to build the abstract representation of the notional machine A_{NM}, which corresponds to the game pieces and the game board, the abstraction function $\alpha : Term_{U\lambda} \rightarrow A_{NM}$, and an f_{NM} function, which correspond to the rules that guide the evolution of alligator families. First, we analyse the game pieces to model A_{NM} and look more precisely at their correspondence with $Term_{U\lambda}$. Although A_{NM} and $Term_{U\lambda}$ are not isomorphic, that in itself doesn't prevent the construction of a commutative diagram:

- An egg corresponds to a variable use and its color corresponds to the variable name.
- A hungry alligator somewhat corresponds to a lambda abstraction with its color corresponding to the name of the variable introduced by the lambda (a variable definition) and the pieces guarded by the hungry alligator corresponding to the body of the lambda abstraction. But differently from a lambda abstraction, a hungry alligator doesn't have to be guarding any pieces, which has no direct correspondence with the lambda calculus because a lambda abstraction cannot have an empty body.
- An old alligator somewhat corresponds to parentheses but not exactly. The lambda abstraction in the term `(λt.λf.t) a b` requires parentheses because conventionally the body of a lambda abstraction extends as far to the right as possible, so without the parentheses its body would be `t a b` instead of `t`. However the corresponding alligator families shown in Figure 3.2 don't require an old alligator. On the other hand, if we want to represent the term `a (b c)`, then we need an old alligator. Figure 3.3 shows an example of a term that requires an old alligator. Like parentheses, old alligators are used to disambiguate an abstract syntax tree.

Now let's look at the kinds of terms in the untyped lambda-calculus. If hungry alligators are lambda abstractions and eggs are variables then what is an

application? Applications are formed by the placement of pieces on the game board. When an alligator family or egg (corresponding to a term t_1) is placed to the left of another family or egg (corresponding to a term t_2), then this corresponds to the term t_1 applied to t_2 (in lambda calculus represented as $t_1\ t_2$).

Notice that because every egg must be guarded by a hungry alligator with the same color, strictly speaking, an egg cannot appear all by itself. That corresponds to the fact that all values in the untyped lambda-calculus are lambda terms so a term cannot really have an unbound variable. Textbooks of course widely use examples with unbound variables but these are actually metavariables that stand for an arbitrary term. As a result, for convenience, we will consider an egg by itself as also forming a family.

We can then model an alligator family as the type *AlligatorFamily*, and a game board as just a list of alligator families.

```
data AlligatorFamily = HungryAlligator Color [AlligatorFamily]
                     | OldAlligator [AlligatorFamily]
                     | Egg Color
```

The abstraction function $\alpha :: Term_{U\lambda} \rightarrow [AlligatorFamily]$ relies on some function *nameToColor* that can map from a variable name to a color.

```
α :: Term_Uλ                 → [AlligatorFamily]
α  (Var name)                = [Egg (nameToColor name)]
α  (Lambda name e)           = [HungryAlligator (nameToColor name) (α e)]
α  (App e1 e2@(App _ _))     = α e1 ++ [OldAlligator (α e2)]
α  (App e1 e2)               = α e1 ++ α e2
```

Having covered the pieces of the game (the structure of terms), let's now turn to the evolution rules, which will constitute f_{NM}.

3.2.3 From Proof to Property-Based Testing

The commutativity of the diagrams presented in Chapter 2 was demonstrated using equational reasoning. Here instead, we implement the elements that constitute the commutative diagram and use property-based testing to test if the diagram commutes. This approach is less formal and it doesn't prove the notional machine correct, but it is lightweight and potentially more attractive to users that are not familiar with equational reasoning or mechanised proofs. We will see here that, despite its limitations, this approach can go a long way in revealing issues with a notional machine.

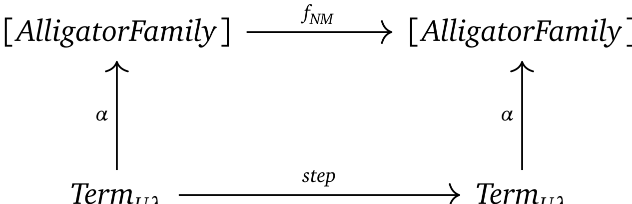


Figure 3.4. First attempt at instantiating the commutative diagram in Figure 2.2 for the notional machine Alligator.

The commutative diagram we would be aiming for is shown in Figure 3.4. With the property-based testing approach, a generator generates terms $t_i :: Term_{U\lambda}$ and checks that

$$(f_{NM} \circ \alpha)\ t_i \equiv (\alpha \circ step)\ t_i$$

de Bruijn Alligators

The first challenge is that we need to compare values of type [*AlligatorFamily*] that were produced using f_{NM} with values produced using *step*. As we have seen, the colors in *AlligatorFamily* correspond to variable names but the way *step* generates fresh names (which then are turned into colors) may be different from the way f_{NM} will generate fresh colors. In fact, the original description of ALLIGATOR anticipates the challenge of comparing alligator families. In the description of possible gameplays, they clarify that to compare alligator families we need to take into account that families with the same "color pattern" are equivalent. This can be achieved by using a *de Bruijn representation* [de Bruijn, 1972] of Alligators. We turn *AlligatorFamily* into *AlligatorFamilyF Color* and before comparing families we transform them into *AlligatorFamilyF Int* following the de Bruijn indexing scheme. The commutative diagram we are moving towards is shown in Figure 3.5.

Evaluation Strategy

With this setup in place, the next step is to implement f_{NM} in terms of the game rules. The eating rule (together with the color rule) somewhat corresponds to beta-reduction but under what evaluation strategy? The choice of evaluation strategy turns out to affect not only the eating rule but also the old age rule. According to the original description, any hungry alligator that has something to

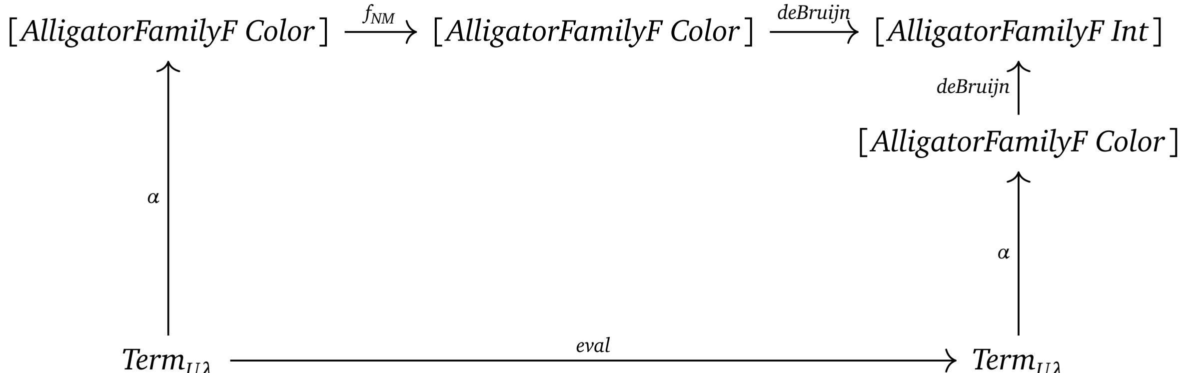


Figure 3.5. Second attempt at instantiating the commutative diagram in Figure 2.2 for the notional machine Alligator.

eat can eat and one of the original examples shows a hungry alligator eating an egg even when they are under another hungry alligator. That would correspond to a *full beta-reduction* evaluation strategy but we will stick to a call-by-value lambda-calculus interpreter so we will adapt the rules accordingly. The old age rule has to be augmented to trigger the evolution of an old alligator family that follows a topmost leftmost hungry alligator and families under a topmost leftmost old alligator. The eating rule should be triggered only for the topmost leftmost hungry alligator, unless it is followed by an old alligator (in which case the augmented old age rule applies).

The color rule plays an important role in the correct behavior of the eating rule as a correspondence to beta-reduction. That's because indeed "the color rule corresponds to (over-cautious) alpha-conversion", so it is responsible for avoiding variable capture.

With all the rules implemented, we can define a function *evolve* that applies them in sequence. We will then use *evolve* in the definition of f_{NM}.

```
evolve :: (Enum a, Eq a) => [AlligatorFamilyF a] -> [AlligatorFamilyF a]
evolve = applyRules [oldAgeRule, colorRule, eatingRule]
  where applyRules [] a                   = a
        applyRules (f : fs) a | f a == a  = applyRules fs a
                              | otherwise = f a
```

One application of *evolve* corresponds to one step in the notional machine

layer but that step doesn't correspond to a step in the programming language layer. For example, The main action of the old age rule (to remove old alligators) doesn't have a correspondence in the reduction of terms in UNTYPEDLAMBDA. In terms of simulation theory, in this case the simulation of the programming language by the notional machine is not lock-step. To adapt our property-based testing approach, instead of making f_{PL} equal to *step*, we will simply reduce the term all the way to a value (leading to the use of *eval* as f_{PL} in Figure 3.5) and correspondingly define f_{NM} to be the successive applications of *evolve* until we reach a fixpoint.

Problem: Substitution of Bound Variables

Now we have all the building blocks of the commutative diagram. We can put them together by running the property-based tests to try to uncover issues in the diagram and indeed we do. According to the eating rule, after eating, a hungry alligator dies and if she was guarding any eggs of the same color, each of those eggs hatches into what she ate. So the family corresponding to `(λa.(λa.a)) b` would evolve to `λa.b` instead of `λa.a`. This issue corresponds to a well-known pitfall in substitution: we cannot substitute *bound* occurrences of a variable, only the ones that are *free*. The solution is to refine the eating rule. When a hungry alligator with color c_i eats a family, the only eggs that should hatch are the ones with color c_i that are not already guarded by another alligator with that color.

In essence, we were able to construct the types and functions of the commutative diagram, but even though these functions were total, there were values of $Term_{U\lambda}$ for which the diagram didn't commute. We detected the issue using property-based testing and fixed the specification of ALLIGATOR and our implementation accordingly.

3.3 The Concrete Syntax of Alligators

The ALLIGATOR notional machine we have seen uses concrete images to represent alligators and eggs. Figure 3.6 illustrates how the concrete representation of a notional machine relates to the components of the commutative diagram we have seen so far. At its center, is the α function that maps from programming to notional machine concepts. However, on both sides, we have a concrete as well as an abstract representation. On the programming side, the *parse* function converts from the concrete code (a string) to the abstract representation (an abstract syntax tree). On the notional machine side, the *toDiagram* function maps

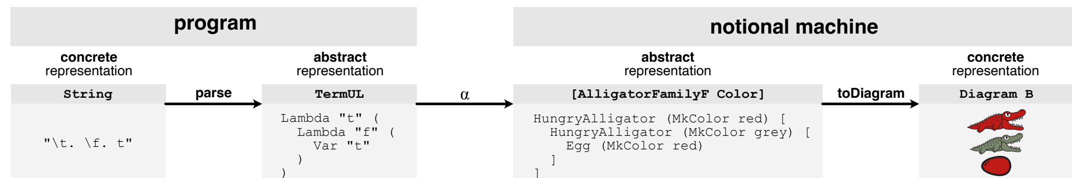


Figure 3.6. Both program and notional machine have abstract and concrete representations

from the abstract notional machine representation to a concrete representation, e.g., in the form of an actual diagram. In the case of ALLIGATOR, we use the *diagrams* library Yates and Yorgey [2015]; Yorgey [2012] to construct the concrete representation so *toDiagram* produces a value of type *Diagram B*, for a given backend *B* determining the output format (e.g. SVG).

The data structure that describes ALLIGATOR is its abstract syntax whereas the specific images used to depict the alligators and the eggs in the previous section are its concrete syntax. The Alligator Eggs web page also describes another concrete syntax that it calls “Schematic Form”. This concrete representation is suitable for working with the notional machine using pencil and paper. Figure 3.7 shows the Church numeral two (the term `λf.λx.f (f x)`), represented with images of alligators and eggs (a graphical representation), with the Schematic Form (taken directly from the website), and with an ASCII-art representation similar to the Schematic Form. Our implementation automatically generates both the graphical representation and the ASCII-art representation from the exact same abstract ALLIGATOR representation.

In the schematic representation, colors are presented with variable names. An alligator is drawn as a line ending with a < for a mouth, and is preceded by a variable name corresponding to its color. An old alligator is drawn with a line without a mouth. An egg is drawn just with the variable name corresponding to its color.

3.3.1 Designing a Concrete Representation

In principle, the abstract representation of a notional machine should contain all the information necessary for operating with the notional machine, whereas the concrete representation adds information that is inessential for the operation of the notional machine. In the case of ALLIGATOR, the number of teeth in the alligator’s mouth, for example, is inessential so this information is contained only in the concrete representation.

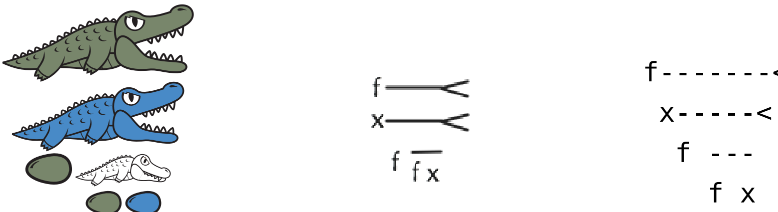


Figure 3.7. Different concrete representations of the same Alligator family in the Alligator notional machine.

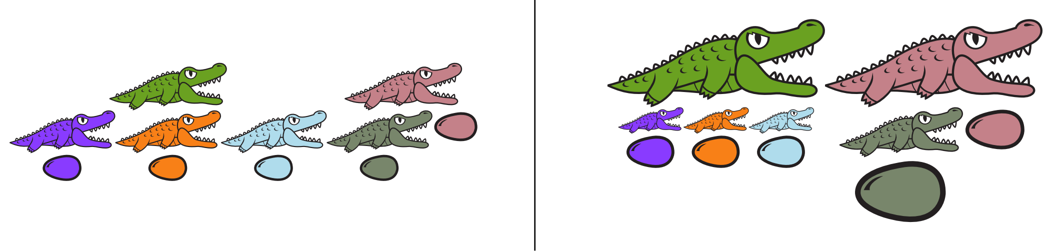


Figure 3.8. Confusion due to suboptimal concrete representation in the Alligator notional machine.

Good care should be taken when designing the concrete representation of a notional machine to avoid introducing misconceptions. For instance, in the graphical representation of ALLIGATOR let's focus on the *size* of the depiction of alligators and eggs. This information is not present in the abstract representation and indeed it does not seem to be important for the operation of the notional machine. The original description of ALLIGATOR doesn't prescribe a size for alligators and eggs. The examples shown on the Alligator Eggs web page are not consistent in terms of the size: sometimes they reduce the size of pieces that are guarded by other pieces and sometimes they do not. A closer look, though, reveals that in fact, the relative size of pieces is important because the relative position of alligators and eggs determines the relationship between them (pieces guarded by other pieces are placed under them and pieces that may eat other pieces are placed in front of them). Depending on the size of the pieces, it may not be obvious who is supposed to eat whom.

Figure 3.8 shows two concrete representations for the same abstract representation of a board of alligator families. On the left, all pieces have the same size. In that concrete representation, it looks like the light blue alligator should eat

the gray alligator[4]. But in fact, the light blue and the gray alligator are not "on the same level": the light blue alligator is being guarded by the green alligator, while the gray alligator is being guarded by the pink alligator. This confusion can be solved by scaling the sizes of alligators. In the concrete representation shown on the right of Figure 3.8, pieces are resized proportionally depending on their relationship with other pieces. The width of the pieces directly under an alligator adds up to 90% of the width of the alligator guarding them and at the same time the height of the pieces (alligators and eggs) on the same level should be the same. The effect in this case is that it is easier to see that the light blue alligator does not threaten the grey alligator. The relative sizes also help to show that the topmost alligator of each family is the one that can eat another family: the green alligator would first eat the pink alligator and its family.

Interestingly, looking at the schematic concrete representation (middle of Figure 3.7), we see that the author was aware of the importance of the relative size of pieces. In our ASCII-art representation (right of Figure 3.7), we not only made sure that the pieces guarded by an alligator "fit" under that alligator but also that the width of pieces guarded by an alligator is strictly smaller than the width of that alligator, so that the topmost alligator is further emphasized.

3.4 Conclusion

In this chapter, we have seen how we can analyze existing notional machines with respect to their soundness and, as a result, find inconsistencies and improve them. We also discussed the challenges of designing a concrete representation for a notional machine.

The analyses in this chapter were made simpler by our use of UNTYPEDLAMBDA as the programming language under the focus of the notional machines. In contrast, in the next chapter, we will use essentially the same technique to analyze and improve a notional machine that is focused on Java, a much larger language.

[4] The interpretation that the light blue alligator would eat the gray alligator and the interpretation that the orange alligator would eat the light blue alligator seem equally likely.

Chapter 4

A Family of Notional Machines for Expressions

Until now, we have seen how to design and analyze notional machines using as a basis the commutative diagram that relates the abstract representation of the notional machine with the abstraction that represents the aspect of the programming language under focus by the notional machine.

To this end, we have always used programming languages with well-known complete formalizations that were as small as needed to express the aspect of semantics we were interested in: when we focused on evaluation we used UNTYPEDLAMBDA, when we focused on references we used TYPEDLAMBDAREFREF, and for types we used TYPEDARITH. Even though these languages are used as core calculi for various widely used programming languages, they are not themselves widely used in industry or in education. Now we will instead use Java, a programming language widely used both in industry and in education.

There exist many formalizations of core aspects of Java. The most popular of these minimum core calculi for Java is Featherweight Java [Igarashi et al., 2001]. Typically, each core calculus is focused on a specific aspect of the language (e.g., Welterweight Java [Östlund and Wrigstad, 2010] focuses on imperative features and concurrency, FeatherTrait [Liquori and Spiwack, 2008] focuses on traits). They are small so proofs written with them are simpler. Here we want to focus

Section	Notional Machine	Programming Language	Focus
4.3	EXPTUTORDIAGRAM	JAVA	Typing
4.4	EXPTUTORDIAGRAM	JAVA	Parsing

Table 4.1. The variations of ExpressionTutor for Java shown in this chapter.

on expressions, but we want instead to consider the complete language up to Java 11. And rather than proving our construction, we will reason informally using as the basis for this reasoning the formal approach we have developed in the previous chapters.

On the side of the notional machine, we will again use ExpressionTutor, which we have introduced in Section 2.3 and then used later in Section 2.5, but here we will develop it further. When we introduced it, we characterized ExpressionTutor as a family of notional machines that can be instantiated for a given programming language and given aspect of that language. Before, we instantiated it for UNTYPEDLAMBDA focusing on program evaluation and TYPEDARITH focusing on types. In this chapter, we will use Java focusing first on types and then on parsing, as summarised in Table 4.1.

The use of the commutative diagram in the development of ExpressionTutor has a direct impact in shaping the implementation of the components of the tool. As we dive deeper, we will also discuss some facets of this implementation that are important in making it usable in practice as an educational tool.

We start by motivating why we want to focus on expressions (Section 4.1). Then, we define which Java constructs are expressions (Section 4.2). We are then ready to map from Java expression constructs to their diagrams. We put it all together showing how we can follow the principles we've seen so far to develop and improve a tool for the notional machine (the ExpressionTutor platform) to help teach and assess students about type-checking (Section 4.3) and parsing (Section 4.4) Java expressions. An extra benefit of this approach is that an implementation that follows the commutative diagram essentially results in a tool that can generate student activities about the focus of the notional machine and assess the students' solutions (Section 4.5).

4.1 Why Focus on Expressions?

Expressions[1] are syntactic phrases that are constructed compositionally. They all evaluate to values, and in statically typed languages they all have a type. Because they are built compositionally, we can understand a bigger expression by decomposing it into its smaller components[2]. We can reason about its type

[1] Authors sometimes use the words *term* and *expression* interchangeably. Other times, they use the word term to refer to expressions that produce values and the word expression in a more general sense [Pierce, 2002], standing also for phrases in other syntactic categories, including type expressions and kind expressions. Here we use expression to refer to syntactic phrases that produce values.

[2] In impure languages like Java, side-effects complicate this reasoning.

and its value by reasoning about the types and values of its subexpressions. This recursive view of expressions is a prime example of decomposition. It allows students to learn to evaluate or type expressions in a general and systematic way [Marceau et al., 2011].

In programming languages that are considered predominantly functional, expressions are the main building blocks of programs. In programming languages like Java, which are not predominantly functional, expressions seem to play a less important role, and this is often also reflected in teaching. Indeed, in 2008 a study carried out a Delphi process among experts to identify important and difficult concepts in introductory programming [Goldman et al., 2008]. Expressions only appear among topics such as "construct/evaluate boolean expressions" and "writing expressions for conditionals", both ranked as very important but moderately difficult. There is no mention of the concept of expressions being treated in a general form, instead of the narrow view of logic and arithmetic. This impression is also reflected in textbooks. Chiodini et al. [2022] systematically analyzed the contents of current Java textbooks to characterize how they present expressions and found that expressions are **neglected** in Java programming textbooks, which do not introduce expressions as a central and general concept.

But contrary to this impression, Chiodini et al. [2022] show that expressions are in fact **prevalent** in Java code written by students. The result comes from an empirical analysis of the use of expressions in Java programs written by novices performed using the Blackbox dataset, the largest repository of Java code written by students. Not only this but the authors also argue that expressions are **essential** in Java, by analyzing the grammar of the language and showing how small is the subset of constructs in the language that don't use expressions. They also elaborate on how expressions, statements, and definitions in Java are fundamentally connected to each other.

The general, prevalent, and essential nature of expressions makes them a prime candidate to be used as the focus of a notional machine.

4.2 Expressions in Java

The Java Language Specification[3] (JLS) [**?**] contains both a formal specification of the language's concrete syntax, as well as an informal specification of the language semantics. The language specification categorizes expressions into the following six syntactic forms:

[3] The version we are considering here is Java 11 (a Long-Term Support version), excluding modules and annotations.

(1) expression names

(2) primary expressions

(3) unary operator expressions

(4) binary operator expressions

(5) ternary operator expressions

(6) lambda expressions

But to be able to define an abstraction function that maps from expressions into the notional machine we need to define exactly which *constructs* of the language we are considering as expressions, and we need to define their structure.

4.2.1 Grammar is Not Enough to Identify Constructs

The grammar productions that determine the concrete syntax of the language are not good candidates to be used as language constructs. A language construct may have multiple syntactic representations and thus it may correspond to multiple grammar productions (e.g., array instances may be created with or without array initializers, which affects whether or not they contain subexpressions denoting the array dimensions). A grammar production may also correspond to multiple language constructs when more contextual information is needed to determine the exact construct. For example, a simple name can be a local variable access or a field access depending on the context in which it occurs. A grammar production may even correspond to only part of a language construct, which allows for the reuse of a grammar production in the definition of different language constructs. Moreover, grammar productions are sometimes built with the purpose of enforcing associativity and precedence rules. In essence, the concrete syntax of a language is not the right level of abstraction to define its constructs. The level of abstraction that we are looking for is captured by the abstract syntax of a language.

The abstract syntax of Java is not defined in the language specification, so we will consider the one defined by Eclipse's Java Development Tools (JDT) [The Eclipse Foundation, 2022]. Although JDT is closely modeled after the language specification, it diverges a little from it, mostly for practical implementation reasons. For example, it represents deeply nested expressions of the form `L op R op R2 op R3`, where the same binary operator appears between all the operands, with one AST (Abstract Syntax Tree) node holding all the operands. The language constructs that we will consider to be expressions mostly correspond

to JDT's AST nodes that are subtype of `Expression`, with small modifications whenever we found aspects that diverge from the language specification.

4.2.2 Java Expression Constructs

The Java expression constructs are shown in Table 4.2. Each row names a language construct, refers to the main JLS section where it is discussed, and specifies its structure. We represent the structure of a construct with a grammar.

EBNF Symbols

The bold symbols follow the conventions of EBNF:

- **[** `a` **]** denotes that `a` is an optional part of the construct;
- **{** `a` **}** denotes the absence or presence of one or more occurrences of `a` in the construct;
- `a` **|** `b` denotes the presence of either `a` or `b` in the construct (grouped where needed with **(** ... **)**).

Java Tokens

The `colored tokens` are used to denote tokens of the Java language.

Subexpressions

The meta-variable e denotes a subexpression. Some constructs restrict one of their subexpressions to only *variables* (JLS 15.26), represented as e_{var}, which according to the specification can be "named variables" (e.g., local variables) or "computed variables" (e.g., field accesses and array accesses). In terms of the constructs defined in Table 4.2, these are:

- `Id` - Simple Variable Access;
- **[(** e **|** T_r **)**.**]**`Id` - Field Access;
- **[**T_r.**]**`super.Id` - Super Field Access;
- e`[`e`]` - Array Access.

e_{var} is used in the left-hand side of an Assignment, as an operand of a Postfix Expression, and as an operand of some Prefix Expressions (the Prefix Increment Expression and the Prefix Decrement Expression (JLS 15.15.[1-2])).

Table 4.2. Java expression constructs. The meta-variable e denotes subexpressions. The bold symbols follow the conventions of EBNF. The colored tokens denote tokens of the Java language. The remaining meta-variables are described in Table 4.3.

<table>
<tr><th>Group</th><th>Construct</th><th>Java Spec.</th><th>Structure</th></tr>
<tr><td>Class Instance Creation</td><td>Class Instance Creation</td><td>15.9</td><td>[e.]new [<T{,T}>]T_r ([e{,e}])[Block]</td></tr>
<tr><td>This</td><td>This Expression</td><td>15.8.3</td><td>[T_r.]this</td></tr>
<tr><td rowspan="3">Variable</td><td>Simple Variable Access</td><td>6.5.6.1</td><td>Id</td></tr>
<tr><td>Field Access</td><td>15.11</td><td>[(e | T_r).]Id</td></tr>
<tr><td>Super Field Access</td><td>15.11.2</td><td>[T_r.]super.Id</td></tr>
<tr><td rowspan="2">Method Invocation</td><td>Method Invocation</td><td>15.12</td><td>[(e | T_r).][<T{,T}>]Id([e{,e}])</td></tr>
<tr><td>Super Method Invocation</td><td>15.12</td><td>[T_r.]super.[<T{,T}>]Id([e{,e}])</td></tr>
<tr><td rowspan="2">Array</td><td>Array Access</td><td>15.10.3</td><td>e[e]</td></tr>
<tr><td>Array Instance Creation</td><td>15.10.1</td><td>new T[<T{,T}>][e]{[e]}{[]}
new T[<T{,T}>][]{[]}ArrayInit</td></tr>
<tr><td rowspan="2">Type Comparison and Cast</td><td>Type Comparison</td><td>15.20.2</td><td>e instanceof T</td></tr>
<tr><td>Cast Expression</td><td>15.16</td><td>(T)e</td></tr>
<tr><td>Lambda</td><td>Lambda</td><td>15.27</td><td>Params -> (Block | e)</td></tr>
<tr><td rowspan="3">Method Reference</td><td>Constructor Reference</td><td>15.13</td><td>T_r::[<T{,T}>]new</td></tr>
<tr><td>Method Reference</td><td>15.13</td><td>(e | T_r)::[<T{,T}>]Id</td></tr>
<tr><td>Super Method Reference</td><td>15.13</td><td>[T_r.]super::[<T{,T}>]Id</td></tr>
<tr><td rowspan="5">Operator</td><td>Conditional Expression</td><td>15.25</td><td>e ? e : e</td></tr>
<tr><td>Assignment</td><td>15.26</td><td>e_{var} AssignOp e</td></tr>
<tr><td>Postfix Expression</td><td>15.14.2</td><td>e_{var} PostfixOp</td></tr>
<tr><td>Prefix Expression</td><td>15.15.1</td><td>PrefixOp (e | e_{var})</td></tr>
<tr><td>Infix Expression</td><td>15.18.2</td><td>e InfixOp e</td></tr>
<tr><td rowspan="6">Literal</td><td>Boolean Literal</td><td>15.8.1</td><td>true | false</td></tr>
<tr><td>Character Literal</td><td>15.8.1</td><td>CharacterLiteral</td></tr>
<tr><td>Null Literal</td><td>15.8.1</td><td>null</td></tr>
<tr><td>Number Literal</td><td>15.8.1</td><td>IntegerLiteral | FloatingPointLiteral</td></tr>
<tr><td>String Literal</td><td>15.8.1</td><td>StringLiteral</td></tr>
<tr><td>Class Literal</td><td>15.8.2</td><td>(T | void).class</td></tr>
</table>

Table 4.3. Meaning of `MetaVariable`s used in Table 4.2 with reference to relevant section(s) of the Java Language Specification.

`MetaVariable`	Meaning	Java Spec.
`T`	Any Type	4.1
T_r	Reference Type	4.3
`Block`	Code Block	4.2
`ArrayInit`	Array Initializer	10.6
`Id`	Identifier	3.8
`Params`	Lambda Parameters	15.27.1
`AssignOp`	Assignment Operator	15.26
`PostfixOp`	Postfix Operator	15.14
`PrefixOp`	Unary Operator (except cast)	15.15
`InfixOp`	Binary Operator (except `instanceof`)	15.[17-24]
`IntegerLiteral`	Integer Literal	3.10.1
`FloatingPointLiteral`	Floating-Point Literal	3.10.2
`CharacterLiteral`	Character Literal	3.10.4
`StringLiteral`	String Literal	3.10.5

Auxiliary Productions

The remaining `MetaVariables` (described in Table 4.3) are auxiliary grammar productions, mostly corresponding to productions in the JLS grammar with some simplifications.

Some constructs in this list compound various parts of the language as described in the JLS. In particular, Simple Variable Access may be an access to a local variable or a parameter. A Field Access may be an access to an instance variable, a class variable, or an enum constant. Another example is Class Instance Creation, which may be the creation of a class instance, an anonymous class instance, or even a qualified class instance.

Notice that array initializers (JLS 10.6) are not expressions. Even though they are used to instantiate arrays (in a field or local variable declaration, or as part of an Array Instance Creation expression), they cannot by themselves be evaluated to produce a reference to an array instance. Thus, they cannot be used wherever a value of an array type is expected. We also do not consider parenthesized expression (JLS 15.8.5) as a separate expression construct because they only affect the order of evaluation[4].

[4] Except for a corner case whereby -2147483648 and -9223372036854775808L are legal but -(2147483648) and -(9223372036854775808L) are illegal because those two decimal literals are allowed only as an operand of the unary minus operator.

4.3 ExpressionTutor for Java: A Typing Activity

We first introduced ExpressionTutor[5] in Section 2.3. There, we brought up ExpressionTutor as an example of a monomorphic notional machine because it allows students to make mistakes when constructing expression trees. In Section 2.5, we showed how we can use ExpressionTutor as a notional machine to focus on types by augmenting the diagram with a type label for each node. Now, we will start also with typing, but instead of TYPEDARITH we will use Java.

Let's resort back to the commutative diagram in Figure 2.2 and to our description of the design of a notional machine to reason about types, in Section 2.5. There we described the notional machine EXPTUTORDIAGRAM focused on typing expressions in the TYPEDARITH language, whose concrete representation is depicted in Figure 2.12.

Here, we want to keep the same notional machine EXPTUTORDIAGRAM, also augmented with type labels, but now the programming language is Java and we need to restrict ourselves to build diagrams for the subset of constructs in Java that are expressions. Figure 4.1 shows an instantiation of the diagram in Figure 2.2 for typing Java expressions. In addition to being an instantiation of the diagram in Figure 2.2, Figure 4.1 also represents effectively a conceptual view of the architecture of the tool. Next, we describe the components of the diagram in more detail.

4.3.1 Programming Language Layer With JDT

Before, the language (TYPEDARITH) was small and we implemented all the components of the commutative diagram in Haskell. Now, the language is much larger so we use the compiler infrastructure provided by JDT (Java Development Tools) [The Eclipse Foundation, 2022], which is mature and widely used.

Figure 4.1 shows a simplified representation of JDT being used to implement the programming language layer. After parsing the code, JDT keeps an AST representation of the program without information about types. Once we resolve the type bindings, we have essentially a typed AST, which we represent for simplicity as a tuple $(AST, TypeBinding)$.

4.3.2 From Java Expressions to ExpressionTutor Diagrams

ExpressionTutor provides instructors with various features to create activities from source code. The source code can be given by the instructor, the case we

[5] expressiontutor.org

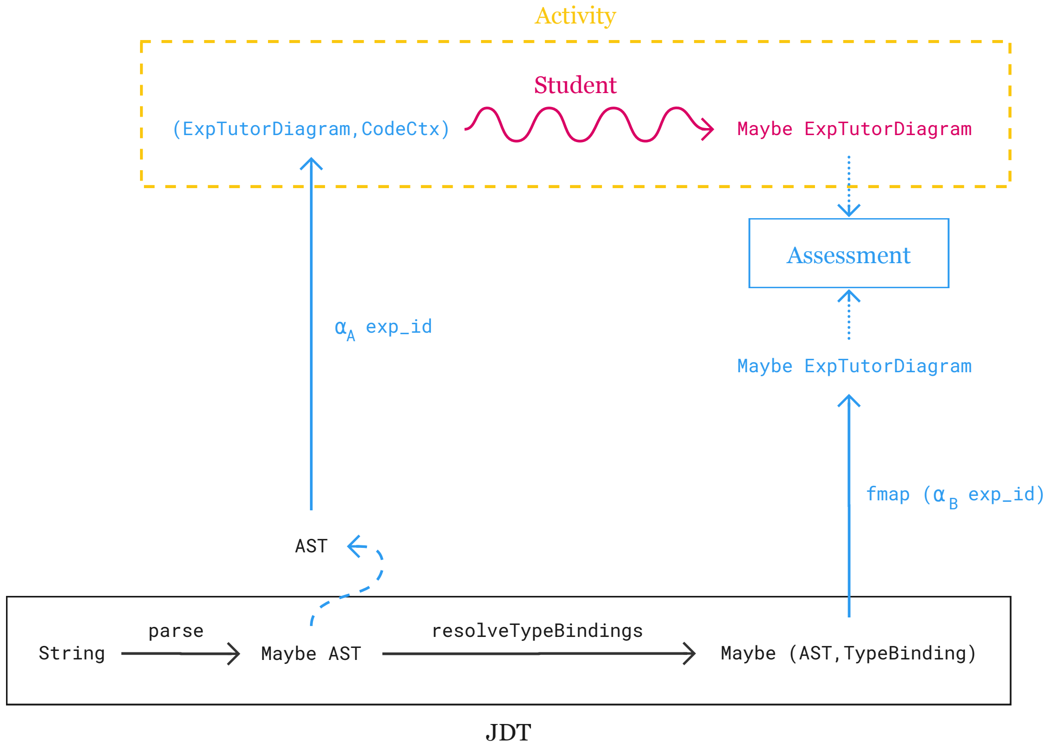


Figure 4.1. Instantiation of the diagram in Figure 2.2 for ExpressionTutor focused on typing Java expressions. The diagram serves simultaneously as a conceptual view of the tool's architecture.

consider here, or even come from students' code submitted to GitHub, which we will explain in Section 4.5.

If the code doesn't parse, an error is raised to the instructor, otherwise, we can safely unwrap the *AST* from its *Maybe*. The instructor is shown the expressions in the code they provided so they can select the one they want to use in the activity. In the Figure, the information that is used to identify the expression is represented as exp_{id} and the abstraction functions (α_A and α_B) are curried. The function α_A generates the ExpressionTutor diagram for the selected expression and the function α_B does the same, except that it also uses the type binding information to fill in the type labels in the diagram. The result of α_B is used to assess the student's answer, as we'll discuss in Section 4.5.

Ideally, the nodes in the AST provided by JDT would correspond to the expression constructs in Table 4.2, and in most cases they do, but it's common for production compilers to have optimizations that diverge from the conceptual AST.

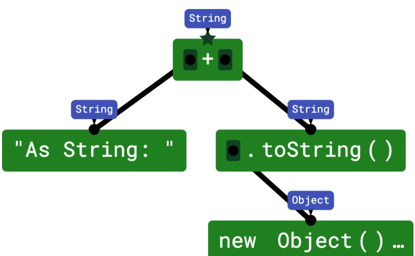


Figure 4.2. ExpressionTutor diagram for the expression `"As String: " + new Object(){ int m() { return 1 + 2; } }.toString()` whose AST contains descendants that are not sub-expressions.

For example, JDT represents consecutive applications of the same operator (e.g. `1 + 2 + 3`) as a single node with a list of operands, instead of a subtree with one node for each application of the operator. The abstraction functions have to account for that.

Non-Expressions Inside Expressions

More tricky are the cases of descendant nodes of an expression node that are not expressions. We identified three cases where this happens:

(i) Class Instance Creation expressions containing an Anonymous Class Declaration;

(ii) Lambda expressions containing a body (i.e., `Block` as a child);

(iii) Array Initializers, that are not expressions, as explained in Section 4.2.

These are represented with ellipsis (`...`) in their parent node. Figure 4.2 shows an example. Notice that it's possible for an expression node c to be a descendant of another expression node p in the AST but not be a sub-expression of p. As an example, in the expression in the Figure, `1 + 2` is an expression but not a sub-expression of the expression containing the string concatenation, and therefore would be represented with another diagram.

ExpressionTutor provides a reference page[6] with examples of ExpressionTutor diagrams for each expression construct in Java, up to Java 11, corresponding to the constructs in Table 4.2. Figure 4.3 contains an excerpt from the reference page showing an example of a diagram for a Qualified Class Instance Creation expression.

[6] expressiontutor.org/language/Java/expressionReference

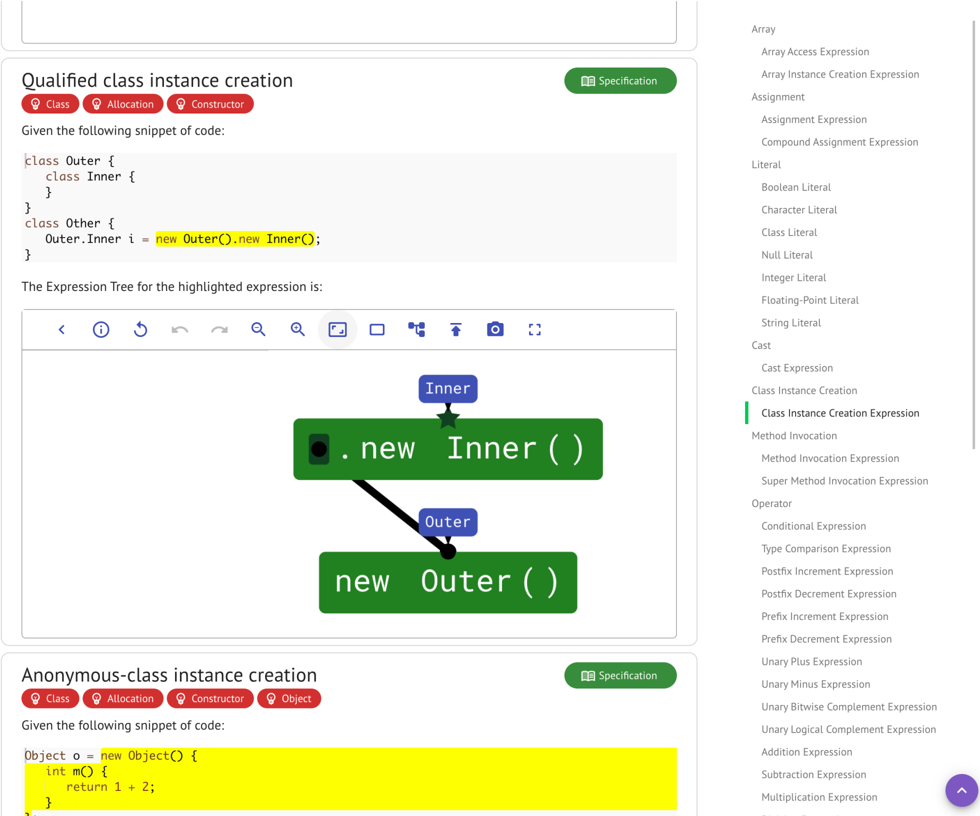


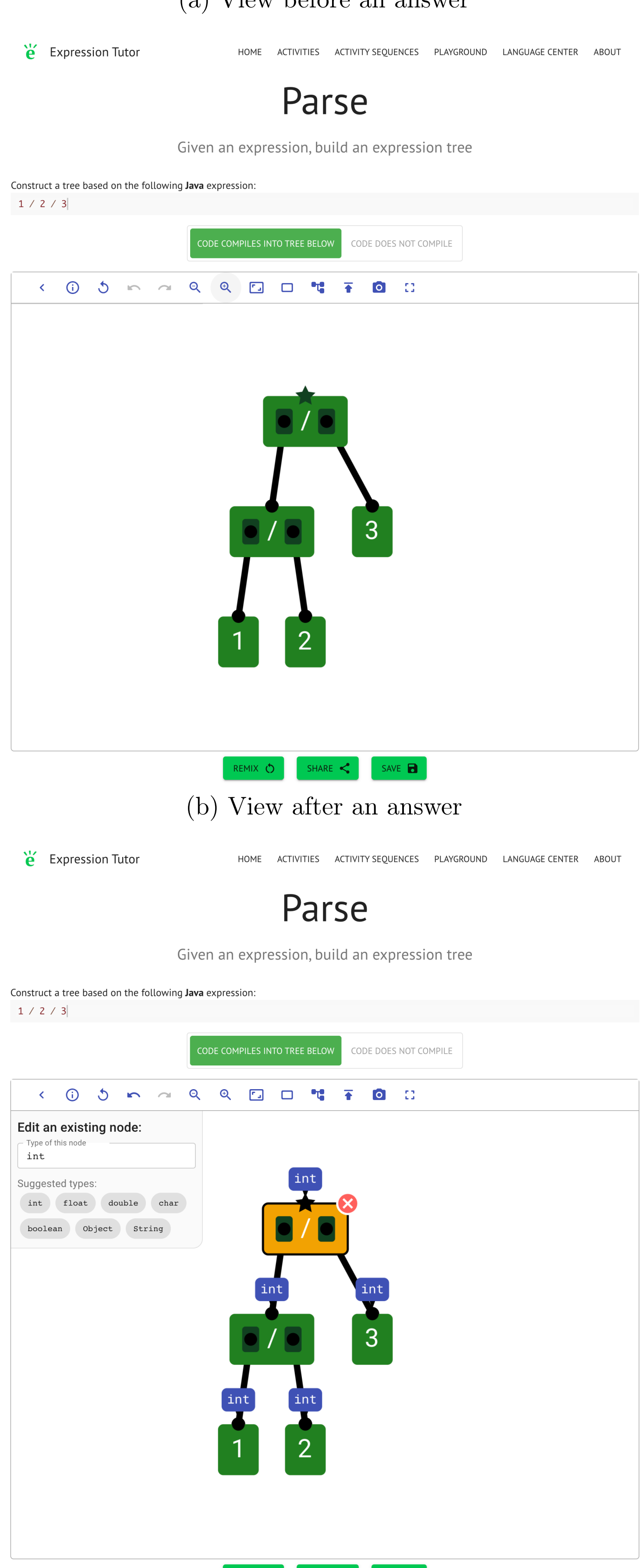
Figure 4.3. Excerpt of the ExpressionTutor reference page for Java, which contains examples of diagrams for each expression construct in Java, up to Java 11.

4.3.3 Notional Machine Layer as a Student Activity

An important aspect highlighted in Figure 4.1 is the view of the notional machine layer as an educational activity to be performed by the student. In this view, f_{NM} is not a function we implement but a "function" executed by the student. So $a_{NM} :: A_{NM}$ (in the Figure $A_{NM} \equiv (\mathit{ExpTutorDiagram}, \mathit{CodeCtx})$) is the input given to the student, $b'_{NM} :: B_{NM}$ (in the Figure $B_{NM} \equiv \mathit{Maybe\ ExpTutorDiagram}$) is the output produced by the student, and $b_{NM} :: B_{NM}$, produced by α_B, is the expected output (the correct answer), which may not be equal to the output produced by the student (b'_{NM}). The values b_{NM} and b'_{NM} can then be compared in the assessment module (see Section 4.5).

Figure 4.4 shows the interface that is currently shown to a student for a typing activity about the expression `1 / 2 / 3`, with what the student sees before and after (or during) answering. While doing the activity, a selected node is shown

Figure 4.4. Current interface for a typing activity about the expression `1 / 2 / 3`.



(a) View before an answer

(b) View after an answer

highlighted in orange with a removal icon in the top-right corner of the node, and a panel, used to select or write the type for that node, is revealed to the left of the diagram. There are currently various usability issues that are being improved. One of them is the title of all activity pages that currently always shows “Parse”, whereas for a typing activity, it should be “Types”, for example. Figure 4.5 shows a prototype for some simple improvements.

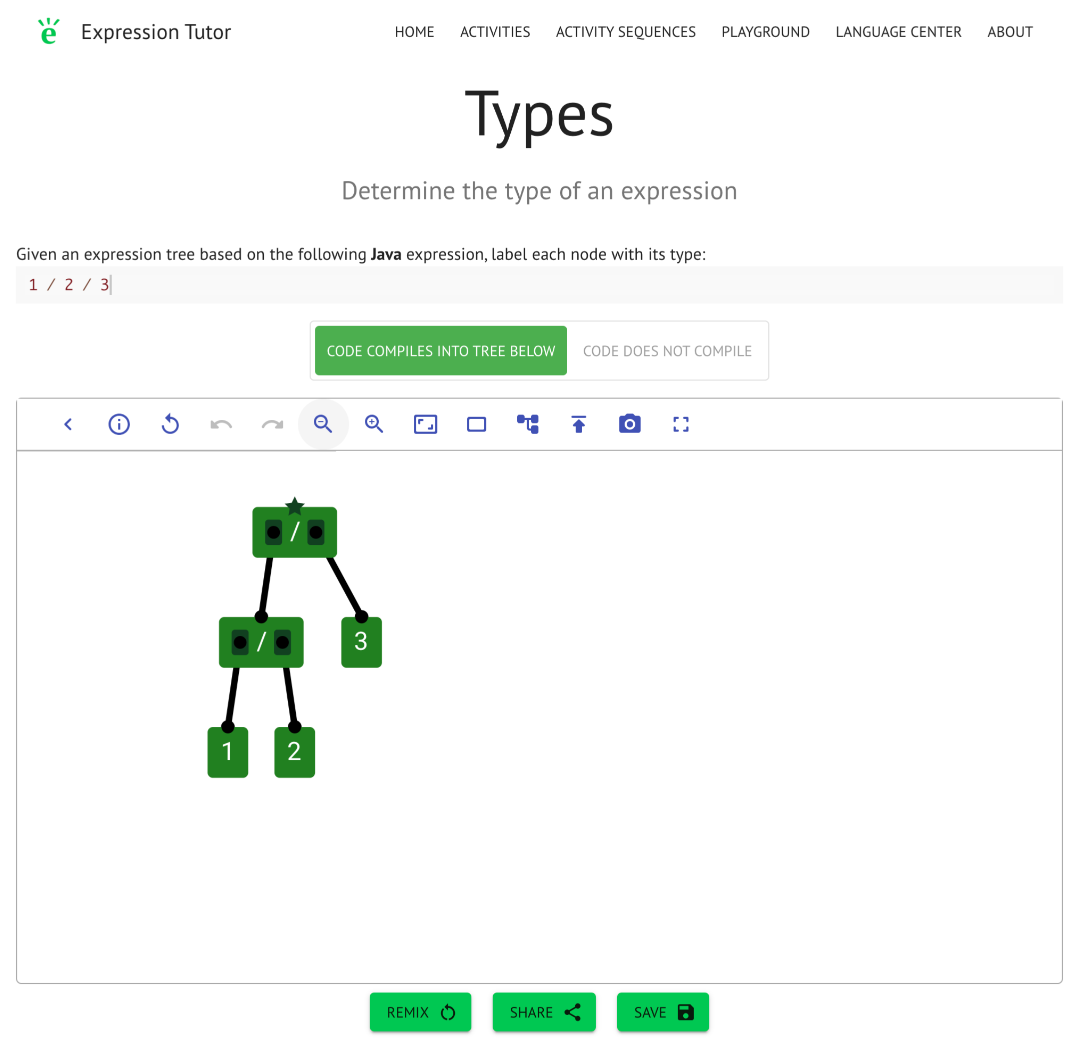


Figure 4.5. Improved interface for a typing activity about the expression `1 / 2 / 3`.

In terms of soundness, for the diagram in Figure 4.1 to work, we need to make sure that:

1. The types of b_{NM} and b'_{NM} are the same.

   This seems straightforward but it may not be. For example, type-checking may fail so B_{PL} is represented with a *Maybe* and therefore the types of b'_{NM} and b_{NM} should also be represented with a *Maybe* (or equivalent type), but a previous version of ExpressionTutor didn't have the toggle "CODE COMPILES INTO TREE BELOW"/"CODE DOES NOT COMPILE" above the diagram so it didn't allow for the solution to be a type error. This is an improvement resulting from our analysis. Although the current interface allows for the solution to be a type error, there is no way to express what the error is. Improvements to the platform are being considered to express that.

2. There is enough information in a_{NM} to produce a b'_{NM} that is equal to b_{NM}.

   This is, of course, necessary to make the notional machine sound. Although we produce $b_{NM} :: B_{NM}$ from JDT, because we are not implementing f_{NM}, one has to be very careful with the activity input given to the student ($a_{NM} :: A_{NM}$). Let's see the concrete challenges we face here.

Soundness Issue: Insufficient Information in Activity Input

Typing expressions in TypedArith is simple because the type of a term only depends on identifying the term itself and the type of its subterms (see the typing rules in Appendix A.2). A language like simply-typed lambda calculus, for example, is a little more complex because typing a term requires maintaining a typing map, a mapping from variables to types. In the case of Java, we need not only a typing map, for the types of parameters, local variables, and **`this`**, but we also need to look up the types of fields and methods. That information is not present in *ExpTutorDiagram*, so we need to augment A_{NM} with more information.

Partial fix: typing map We can add a typing map to the diagram. This would help with the types of parameters, local variables, and **`this`**, but it would not help with looking up the types of fields or methods, which is a more elaborate process. It would also make the notation heavier. Another important point is that the typing map is not one static table for the entire expression being typed, but each sub-expression has its typing map with names potentially being added to it as the expression is being typed.

Fix: code context Another possibility is to add to the activity the code context of the expression. This may be simply the code surrounding the expression, if it

contains information about the types of all the variables and methods used in the expression. Figure 4.6 shows an example of what an activity could look like in that case. If that's not the case, the instructor could explicitly communicate to the students where to find information about these declarations. This could be in the form of Java doc documentation, for example.

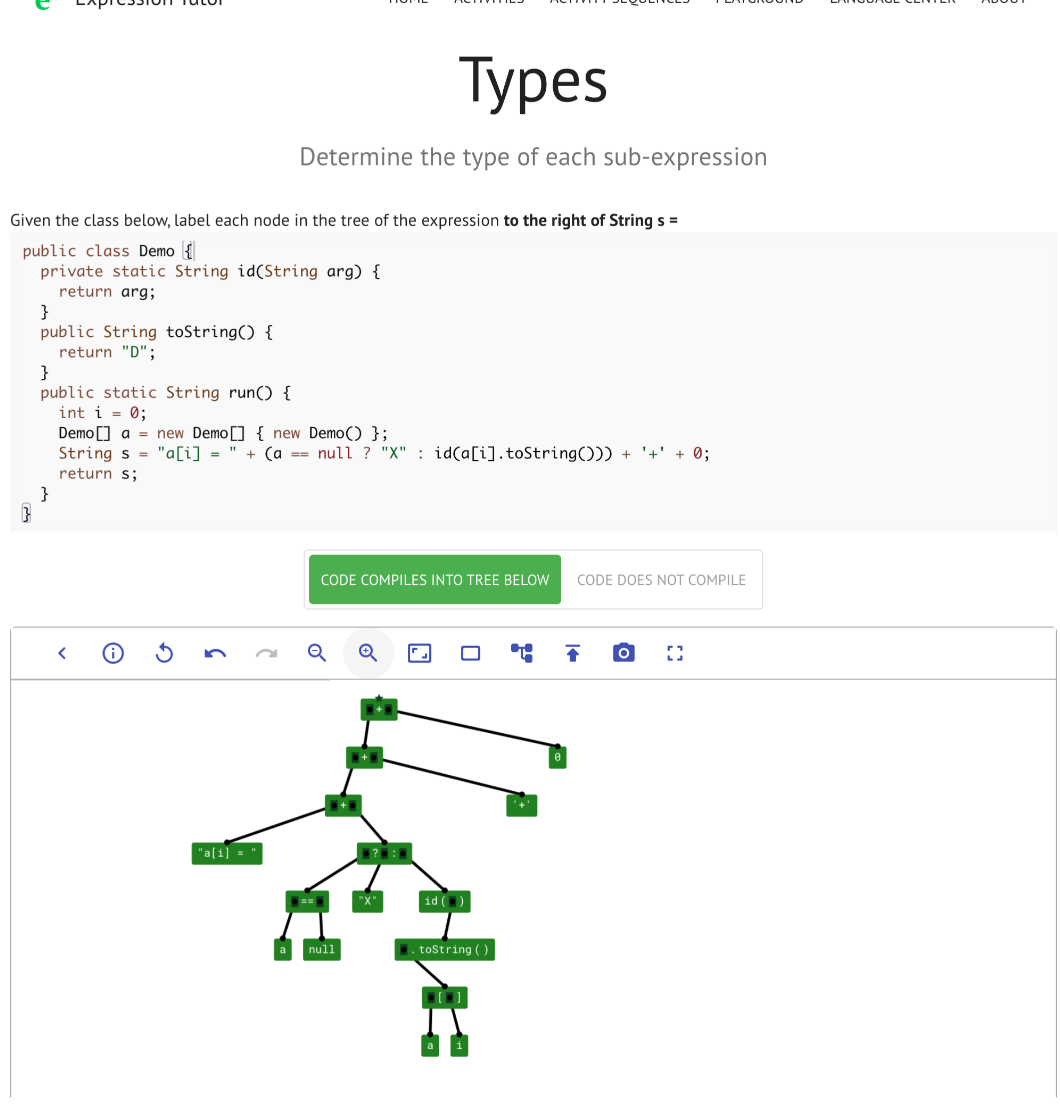


Figure 4.6. Typing activity showing to the student the code context of the expression.

4.3.4 Discussion

It is important to consider, from an educational point of view, what we want the students to learn at any given point in a course. Activities may be created with expressions that require different levels of complexity to type. That will help to determine the amount of information and kind of information the instructor may want to present together with the diagram. For example, it is possible to determine the type of some expressions without a typing map. For other expressions, a typing map may be required but lookups of fields and methods may not be necessary. Notice that typing a term may require knowledge about how to type programs in general, as could be the case with a term containing an anonymous inner class, for example.

Ultimately, we want the students to be able to determine the type of an expression in practice, which means we want them to be able to identify which constructs require type lookup and where and how to look up those types.

The soundness issue we identified and the suggested fixes illustrate an important point: the concrete representation of A_{NM} doesn't necessarily have to contain diagrammatic concrete representation of the information necessary to do the notional machine activity (i.e., to obtain a value of type B_{NM}). In the case of this typing activity, an instructor may decide to be more or less explicit about the information needed to type the expression and may require from the student the ability to identify the information needed to type the expression.

In general, it is for the instructor to opt for a lighter-weight representation for a notional machine, omitting some information from it and providing proxy sources for that information or providing other information that can be used by the student to synthesize the information needed. In this case, the commutative diagram helps the instructor to identify what information is needed in the notional machine layer and what simplifications are being made to the notional machine, ultimately giving the instructor more confidence in the correctness of these simplifications.

4.4 ExpressionTutor for Java: A Parsing Activity

In the previous Section, we have seen ExpressionTutor for typing. There, the input given to the student is already a tree, which the student has to label with types. But what about constructing a tree in the first place? In fact, we first introduced ExpressionTutor[7] in Section 2.3, as an example of monomorphic notional machine because it allows students to make mistakes when constructing expression trees.

[7] expressiontutor.org

The construction of expression trees from source code is an activity about parsing. Figure 4.7 shows an instantiation of the commutative diagram in Figure 2.2 for ExpressionTutor focused on parsing Java expressions. Like in the previous Section, this diagram is also a conceptual slice of the tool's architecture for this activity.

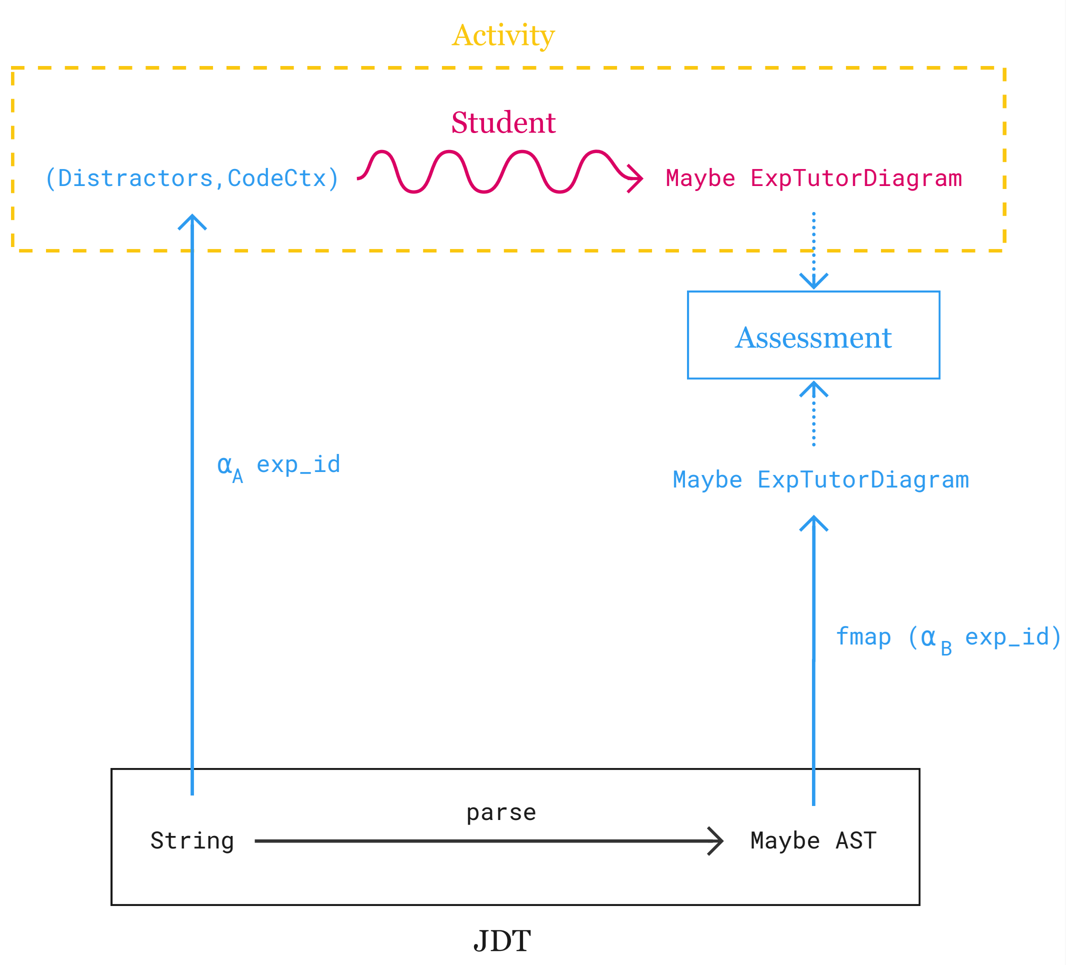


Figure 4.7. Instantiation of the diagram in Figure 2.2 for ExpressionTutor focused on parsing Java expressions. The diagram is also a conceptual view of the tool's architecture.

4.4.1 Commutative Diagram for Parsing

Like in the typing activity, the programming language layer is implemented with JDT and as in the typing activity, parsing may fail so we need a *Maybe* type (or equivalent) both in B_{PL} and B_{NM}. Being able to express in the notional machine

that parsing an expression may fail is important not only from a theoretical point of view but also because novices may struggle to identify whether a piece of code would parse or not. The curated inventory of programming language misconceptions that we have collected as part of prior work [Chiodini et al., 2021] documents 28 misconceptions about expressions. Out of those, 5 refer to expressions that the students believe to be illegal but that are actually legal. Expression trees may be a good medium for the students to communicate their misconceptions and for the instructors to identify them and explain them to the students. The use of expression trees to detect programming language misconceptions seems to be a promising direction for future work (see Chapter 7).

Like in the case of type errors, the interface allows for the solution to be "Code does not compile", but it is not yet possible to express whether the problem is a parse error or a type error and what exactly is the problem and where it is (corresponding to a type that contains more information than *Maybe*). These improvements are also being considered for the platform (see Chapter 7).

In the previous chapters, we have not constructed any commutative diagram for a notional machine that focuses on parsing. An interesting property for the case of parsing is that, in principle, the input in the notional machine layer could be the same as the input in the programming language layer: a String of the program to be parsed. However, that's not what we have, as you can see in Figure 4.7, so let's unpack the reasons that led to the current design.

Code Context is Needed

In a language like TypedArith or UntypedLambda, the entire program is an expression so A_{PL} and A_{NM} could both be Strings of the program to be parsed. In a language like Java, the instructor must select from the program the expression for which the student will construct the tree.

It may be tempting then to restrict the input of the notional machine layer to be a String containing only the expression for which we want to construct the diagram, but in general, the String of an expression is not enough to correctly determine the AST of that expression. For example, an identifier to the left of the dot in a method invocation may or may not be an expression. An identifier to the left of the dot in an instance method invocation (e.g. `o` in `o.m(...)`) is an expression (the use of a variable that refers to an object that is the target of the method invocation). However, if the identifier to the left of the dot is a class name (e.g. `Math` in `Math.max(...)`), then we are dealing with a static method invocation and this identifier is not an expression. The problem here is analogous to the problem in the typing activity, where the student needs to know the code

context to be able to do the activity. Besides the code context, the input given to the student (A_{NM}) also contains distractor nodes.

4.4.2 Distractor Nodes

In terms of the user experience of constructing the tree, once given the nodes, the user can freely connect them to each other by clicking and dragging connectors between nodes and the holes inside other nodes. So the only thing missing is really to construct the nodes. To that end, we could provide the user with:

(i) A language to construct individual nodes, which is currently what we use internally as instructors. But for students, that would require them to learn yet another language: a meta-level language used to construct nodes.

(ii) An editor that allows one to interactively construct a node. That wouldn't require the students to learn a new language but still requires them to reason in a meta level about the programming language, something that may not be educationally adequate in many courses.

To simplify the process, we have instead opted for giving the students a set of nodes that can be used to construct the tree. This means that the input type is not *String* but (*String*, *Set Node*). Another benefit of this approach is that it facilitates the correction of the submissions making it more amenable to automation.

The question that arises at this point is: "how can we determine these nodes in a way that allows the students to still express the mistakes they would make if they could freely construct a tree?". For that, we resorted to analyzing questions in previous years' exams that asked the students to draw expression trees. The goal is to identify patterns of mistakes made by students when drawing trees and use those patterns to generate the nodes we give to the students.

Students' Mistakes in Paper-Based Expression Trees

We analyzed two questions, one from a midterm exam and one from a final exam, both from the course "Programming Fundamentals 2" (PF2), a second-year bachelor course that teaches an introduction to object-oriented programming offered in the Spring semester of 2022. The questions are shown in Appendix B. Both questions present a Java class and ask the student to draw the expression tree for a given expression.

The first question asks: "For the expression to the right of the equals sign (=) in the run method, draw the expression tree [...]". The method run contains only

one equals sign in the line:

```
String s = publish(make("this"), make("that"));
```

The second question asks: “Draw the expression tree of the expression to the right of `String s =` ”. The class contains only one such fragment of code in the line:

```
String s = "a[i] = " + (a == null ? "X" : id(a[i].toString())) + '+' + 0;
```

We randomly selected one third of the exams from each of the two questions (the test set) to analyze the students’ answers and categorized each occurrence of a wrong node. The idea is to generate nodes using those categories as reference and evaluate our node generation strategy by comparing the generated nodes with the nodes in the remaining two thirds of the exams (the training set). Table 4.4 contains a short description of each category and two examples of mistakes in that category, the first example taken from the midterm exam and the second one from the final exam (except for the category **CharAsString**, which only happened in the final exam). Each example shows the content of a node with holes represented as #. We describe the categories we identified in more detail below:

Inline Node containing token that should be in a child node. Inlining can happen on multiple levels, not only of a leaf into a branch but also of branches inside other branches. That’s especially the case for binary operators when the child and the parent nodes are the same operator.

Extract Node containing token that should be in a parent node. A typical example is method names, which are not themselves expressions but are sometimes extracted into a separate node. Another example is conditional expression node split into two nodes.

EvaluatedExp Node containing the result of evaluating an expression. More precisely, in terms of operational semantics, for an expression t, these wrong nodes contain tokens that are part of terms that are present in the derivation tree resulting from the evaluation of t[8].

OtherCode Node containing tokens that are not part of the expression tree and not used anywhere in the evaluation of the expression. Typically, this

[8] In Java, the evaluation of a sub-expression may involve the execution of statements. So in those situations, some wrong nodes with non-expression tokens may be classified in this category.

happens when the expression of which we're drawing the tree appears in a line where there are tokens that don't form expressions. For example, in variable initializations or return statements.

NoParenMethodCall Method call without parenthesis. This is not a severe problem if the student has the correct high-level understanding of the structure of the tree. One may argue that it's just a different notation.

CharAsString Single character delimited by double quotes instead of single quotes. This may indicate the student doesn't understand the difference between a Char and a String.

CodeInString Holes inside a String. This problem happens when there's a String that contains a text that looks like source code. Although it could be considered a special case of **Extract**, this problem is of a different nature because the content of the String is not of tokens in the language.

MissingQuotes String literal without quotes. Different then **NoParenMethodCall**, this problem may not be a case of simply overlooking notation but, like **CodeInString**, it may indicate a more fundamental problem of confusing Strings with code in the language.

MissingHoles Node with missing holes. Even though the students were instructed in all expression tree exercises during the course to be explicit about where the holes are in each node, some students drew branch nodes without drawing the holes. Like **NoParenMethodCall**, one may argue this is not a mistake but simply a different notation.

Misc This category is a catch-all for mistakes that don't fit in any other category.

We classify each wrong node with only one category, so if a node presents more than one category of mistake we choose one that seems more clear from the context. Table 4.5 shows the number of wrong nodes we found in each category (occurrences) and the percentage this number represents of the total number of wrong nodes. The table includes the mistakes of both exams.

Although our original intention when analyzing these exams was to ground the generation of distractor nodes, a lot of insight can be gained from the analysis of expression trees drawn by students in exams. On that direction, Chapter 7 suggests directions for future work.

Table 4.4. Categories of mistakes made by students when drawing expression trees in paper-based exams.

Category	Description	Examples
Inline	Node containing token that should be in a child node	`make("this")` `#+#+#+#`
Extract	Node containing token that should be in a parent node	`make` `#:#`
EvaluatedExp	Node containing the result of evaluating an expression	`"made" + #` `"D"`
OtherCode	Node containing tokens that are not part of the expression tree	`String s =` `run()`
NoParenMethodCall	Method call without parenthesis	`make#` `id#`
CharAsString	Single character delimited by double quotes instead of single quotes	`"+"`
CodeInString	Holes inside a String	`#=#` `"a[#] = "`
MissingQuotes	String literal without quotes	`make(this)` `a[i] =`
MissingHoles	Node with missing holes	`a[]` `+`
Misc	Mistakes that we were not able to categorize	`string#` `a=null(#,#)`

Distractor Generation

Using these patterns of mistakes, we set out to generate a set of *distractor nodes* for any given expression by modifying the nodes from the correct expression tree. The idea is that the set of nodes given to the students to construct the tree should consist of the nodes in the correct answer, generated by JDT, augmented with the distractor nodes.

An important constraint is that we want to be able to allow the students to make the most commonly found mistakes but we want to present the students with a relatively small set of nodes they can use to construct the tree. To that end, we have targeted a subset of the categories and developed various heuristics to generate distractor nodes in these categories:

Inline (a) the content of each child node is inlined into its parents, but not

Table 4.5. Categories of mistakes and number of occurrences of mistakes in each category found in two exam questions about expression trees.

Category	**Occurrences**	**Percentage**
Inline	102	37.09%
Extract	41	14.90%
EvaluatedExp	33	12.00%
OtherCode	22	8.00%
Misc	19	6.91%
NoParenMethodCall	17	6.18%
CodeInString	15	5.45%
MissingQuotes	9	3.27%
MissingHoles	9	3.27%
CharAsString	8	2.91%
Total	275	100.00%

recursively;

(b) a fully inlined node is created for every subtree of depth three or less;

(c) for every node, inline its children if they have the same content.

Extract (a) for every name that appears in a node and is not itself an expression, generate a node with that name and a node with the content of the original node replacing the name by a hole;

(b) for every ternary operator node we generate two binary operator nodes (e.g. the node `#?#:#` gives rise to `#?#` and `#:#`).

MissingQuotes (a) for every leaf node with content surrounded by single quotes, double quotes, or back quotes, generate a node containing the content between the quotes and another node with the hole surrounded by the quotes.

The remaining categories are not included in the distractor generation: **EvaluatedExp** is not included because we are restricting ourselves to static analysis, at this point; **OtherCode** is not included because our analysis is currently generating only wrong nodes that contain tokens that are part of the expression; **Misc** doesn't seem to follow any noticeable pattern; **NoParenMethodCall** and **MissingHoles** could be the result of imprecise (or different) notation by the student, which we

want to simply avoid in our tool; **CodeInString** would require analysis of the String's content; and **CharAsString** happened with low frequency.

Figure 4.8 shows the set of distractor nodes generated for the expression `"a[i] = " + (a == null ? "X" : id(a[i].toString())) + '+' + 0` using the heuristics described above.

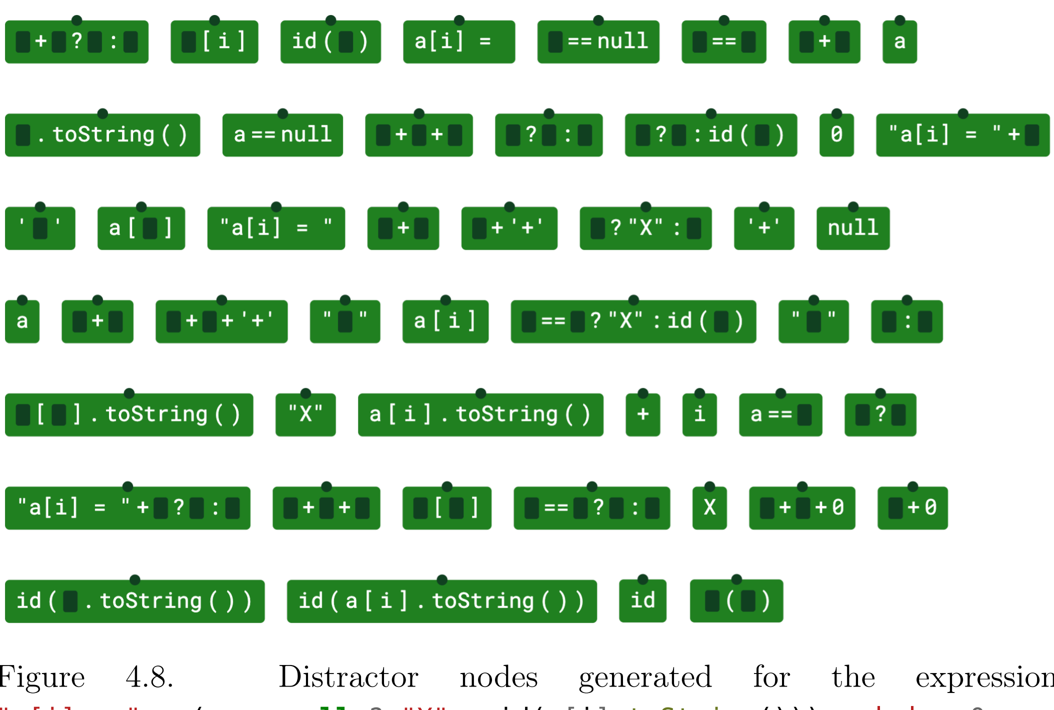


Figure 4.8. Distractor nodes generated for the expression `"a[i] = " + (a == null ? "X" : id(a[i].toString())) + '+' + 0`

Evaluation of Distractor Nodes

We want to evaluate how "real" our distractor nodes are: how do they compare to the wrong nodes created by students on paper in the exams we analyzed previously. Although we were able to generate only 47.27% of the wrong nodes created by the students on paper, if we consider only the categories that we are actually trying to generate, 73.68% of the wrong nodes in those categories were generated. From these, if we consider only the nodes in the test set, we were able to generate 74.44% of the nodes.

Our strategy to generate distractor nodes is quite primitive and there is a lot of room for improvement. The first thing would be to include other categories of mistakes. The primary candidates would be: **EvaluatedExp** could be targeted with partial evaluation techniques, for example; **OtherCode** could be targeted by analyzing the whole AST of the program and creating nodes with tokens from

the statement containing the expression (e.g. return statements and variable initialization). **CodeInString** could also be targeted, at least in a partial way, by some simple heuristics, for example by looking into the String for names that were declared in the program.

We could also analyze more written exams, which could lead to uncovering more mistakes so the proportions of categories of mistakes could be different. Or the very classification of mistakes could be improved leading to a different clustering of mistakes.

It is also possible that with a different user interface (for example with a toolbar of distractor nodes augmented with a search feature), we could relax the restrictions on the number of generated distractors allowing for the generation of more or a bigger variety of distractors.

Ultimately, it is not clear how foundational is the problem of generating distractor nodes. For a certain audience, creating their own nodes would not only be feasible but also have an educational value. We should lower the bar for the user to create their own nodes and investigate the impact of that both in the usability of the tool and educationally in the mistakes the students make when constructing expression trees.

On the other hand, the investigation into patterns of mistakes, that we used as the basis for the generation of distractor nodes may be of greater value. They may be symptoms of general patterns of mistakes in the understanding of the syntax or semantics of a programming language: a programming language misconception, as we define in previous work [Chiodini et al., 2021]. If that's the case, the generation of distractors based on these misconceptions may indeed be of more foundational value. We suggest work in this direction in Chapter 7.

4.5 Automatic Generation and Assessment of Activities

We have shown the typing and parsing activities while describing their respective commutative diagrams, shown in Figures 4.1 and 4.7, which also double as simplified views of the architecture of the tool. These diagrams show an assessment module, that we have alluded to before and will describe in more detail here together with the automatic generation of activities. The key insight is that, by implementing the components of the commutative diagram that describes the notional machine, we essentially get the key components of a backend that can automatically generate and assess activities.

4.5.1 Single Activity Generation

The input for activities can be created manually by the instructor but that is time-consuming and error-prone. Using JDT, we can automatically generate activities which the instructor can then review and configure.

The instructor provides one or more Java classes, we automatically identify the expressions and, once the instructor selects an expression, we generate the activity. This includes not only the input for these activities but also the distractor nodes and the correct answer, which we store and use to automatically correct the activities and provide feedback to the students. The instructor reviews the activity and configures various parameters such as the distractor nodes, whether the students can add type labels or not, whether they can receive automated feedback or not, and many others.

This kind of automation is useful when the instructor wants to create an activity for a specific expression, for example, to use in a quiz or exam. But we can also automate the generation of personalized activities for each student.

4.5.2 Personalized Activity Generation

Students often write code they don't really understand. The code may be pieced together from code fragments found online or from other students. With the growing capabilities and prominence of large language models (LLMs), that generate code based on user prompts, this issue becomes increasingly significant. It's never been easier to produce code. But there's no guarantee that the code produced by these models is correct so reasoning about code becomes even more important. With ExpressionTutor, we can automatically generate activities for the expressions in the student's own code submitted to GitHub as part of their assignments.

For every assignment, the instructor has to define the kind of expression that should appear in the generated activities. When the students submit their code, we automatically identify these expressions in their code and generate activities for them. Each student then receives a link to their personalized activity in a GitHub issue. Once the activity is completed, the student receives feedback generated by the assessment module.

4.5.3 Automatic Assessment of Activities

Automatically generating personalized activities is very useful, but to make it really scalable we also need to be able to automatically assess the students'

solutions to these activities. That's the purpose of the assessment module, that is depicted in Figures 4.1 and 4.7. The information produced by this module is used to automatically generate (1) feedback to each student about their solution and (2) reports to instructors to help them make sense of the students' solutions both in an individual level (per student) and in an aggregate level (for a set of students).

The core function of this module is to compare two *ExpTutorDiagram* data structures: one produced by the student when answering the activity and the other produced automatically when the activity was created, representing the correct answer. The *ExpTutorDiagram* data structure is a graph, as we explained in Section 2.3, in order to allow students more freedom to express mistakes. So in principle, we could use a graph isomorphism algorithm to compare them. But the graph formed by the data structure is quite particular because the connections from a node to a hole are restricted by the holes that exist in each node and the specific hole a node is connected to really matters. So we get better results by comparing the diagrams using a tree comparison algorithm whenever the diagram produced by a student is a tree.

Leveraging Tree-edit Distance to Compare Expression Trees

We compare trees mostly using the tree-edit distance algorithm by Zhang and Shasha [1989]. But instead of producing only a number (the edit distance), we want to know which nodes are in one tree but not the other and which nodes are in both trees, taking into account where they appear in the tree, of course. So we need to find a correspondence between nodes that takes into account the node and where it shows up in the tree. For this, we explore the way the tree-edit distance is calculated.

The distance is obtained by first producing a list of operations that when applied to one tree will produce the other. These operations can be (1) insert a node, (2) remove a node, or (3) change a node into another (these nodes may or may not be equal to each other, for some definition of equality). A cost function is then applied to these operations to produce a total cost, which is the distance (the distance can be measured also in operations simply by tuning the cost function appropriately). The algorithm produces a list of operations that minimizes the total cost.

The idea is to leverage the list of operations produced by the tree-edit distance algorithm to find which nodes in the student's submission have equal corresponding nodes in the reference solution (these would be correct nodes) and, from that, also determine which nodes are wrong in the student submission (either

because they are not present in the reference solution or because they correspond to a different node in the reference solution) and which nodes from the reference solution are missing in the student submission.

This approach is not without its limitations. The resulting tree comparison is at times different from what one might expect or intuitively do when trying to compare trees. That's essentially because the operations that the algorithm uses to turn one tree into the other don't include, for example, node swapping or other kinds of edge manipulation that would correspond to moving around nodes in the tree, which is something that one could prefer to do when comparing trees.

Even when the diagram is not a tree, we can still collect useful information by analyzing the structure of the diagram and identifying the reasons why the diagram is not well-formed. In fact, we collect various other metrics about the diagrams that are easy to calculate. All this information is used to produce the feedback that is shown to students after they finish an activity and the assessment report that is shown to instructors, which aggregates information from multiple submissions.

Feedback to Students

When creating an activity, an instructor can enable or disable the feedback functionality. Having the feedback disabled is useful when the activity is part of an exam, for example. When the feedback is enabled, the student can obtain feedback once the activity is completed.

Figure 4.9 shows an example of feedback generated for a student about the student's answer to an activity that was automatically generated based on the student's code. On the left, there is a list of incorrect and correct aspects of the submission with a little explanation text underneath each aspect. These aspects include well-formedness aspects of the diagram, for example, aspects that are needed for it to form a tree, and tree-comparison aspects. On the right, the students submitted answer is shown with nodes and edges that are correct in green, nodes and edges that are wrong in red, and nodes that are not in the tree in grey. For tree comparison, we consider the the biggest connected component. This feedback does not give away the answer but gives a direction of what can be improved.

Notice that all the incorrect nodes in this student's submission are instances of the same mistake: representing method names as expressions, clearly the symptom of a misconception.

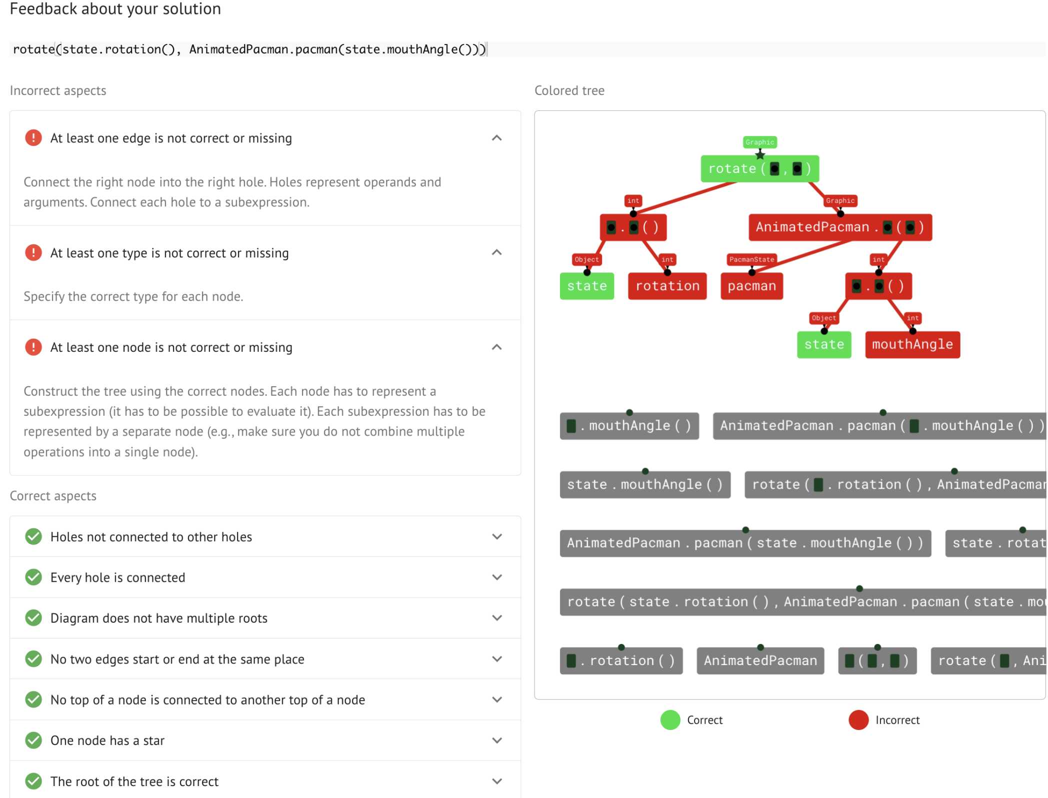


Figure 4.9. Feedback automatically generated for a student's answer to an automatically generated activity based on the student's code.

Aggregated Assessment to Instructors

The instructor has access to a dashboard where they can see reports about the students' submissions. These reports vary depending on whether the submissions being analyzed are all answers to the same activity, as is the case in a quiz for example, or if they are answers to personalized activities.

Figure 4.10 and Figure 4.11 show two reports generated about the answers to a quiz. The first report (Figure 4.10) contains the reference solution and a small sample of the submissions selected using stratified sampling. The idea is to show a sample of submissions from different grade brackets. The second report (Figure 4.11) contains submissions grouped by the incorrect nodes they contain. The groups are sorted showing incorrect nodes that appeared more frequently at the top. The nodes are shown with a notation that replaces holes by #.

Figure 4.12 shows a report generated about answers to personalized activities generated using the students' solutions to one of the labs. On top, a table view

showing various metrics collected about each submission, and on the bottom, a view of the same page but with one of the rows expanded to reveal a side-by-side comparison between a student's submission (on the left) and the reference solution (on the right). The nodes are colored according to the tree comparison based on tree-edit distance as described before. So red nodes on the left tree are wrong in the student's submission and red nodes on the right tree are nodes that exist in the reference solution but are missing from the student's submission.

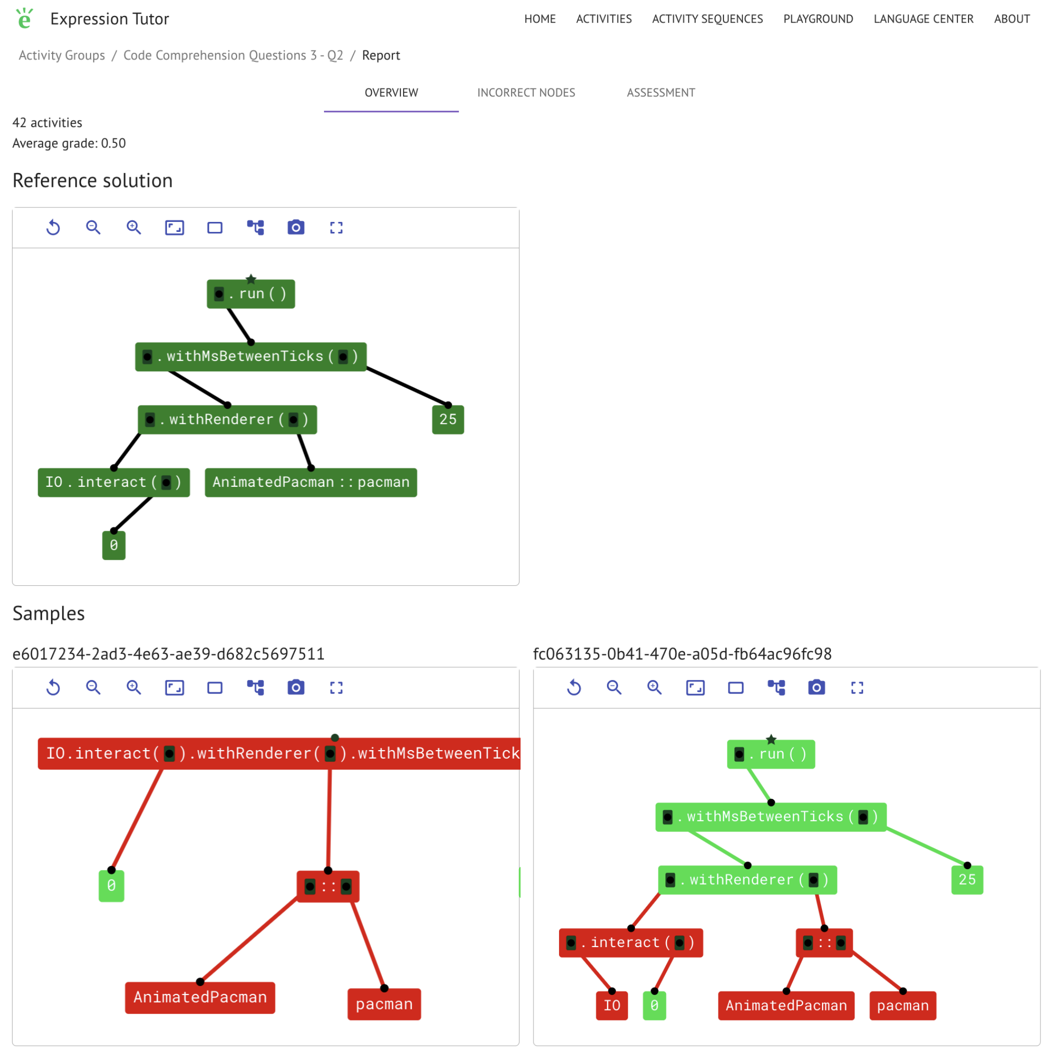


Figure 4.10. Overview report shown for answers to an activity used in a quiz.

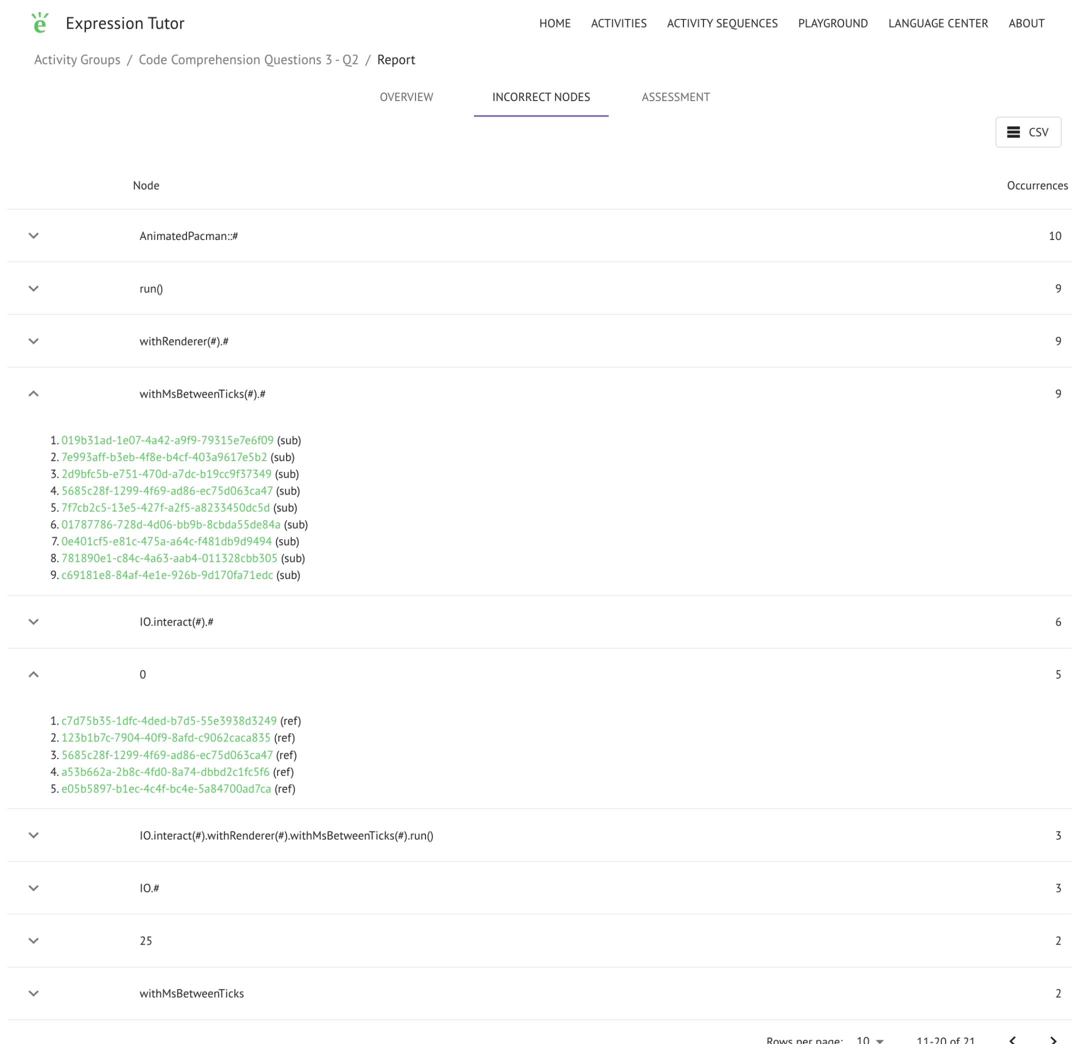


Figure 4.11. Incorrect nodes report for answers to an activity used in a quiz. Answers are grouped by wrong nodes.

Figure 4.12. Assessment report shown to an instructor about a set of student submissions.

(a) Table view showing various metrics collected about each submission.

Activity Groups / Lab 08 A / Report

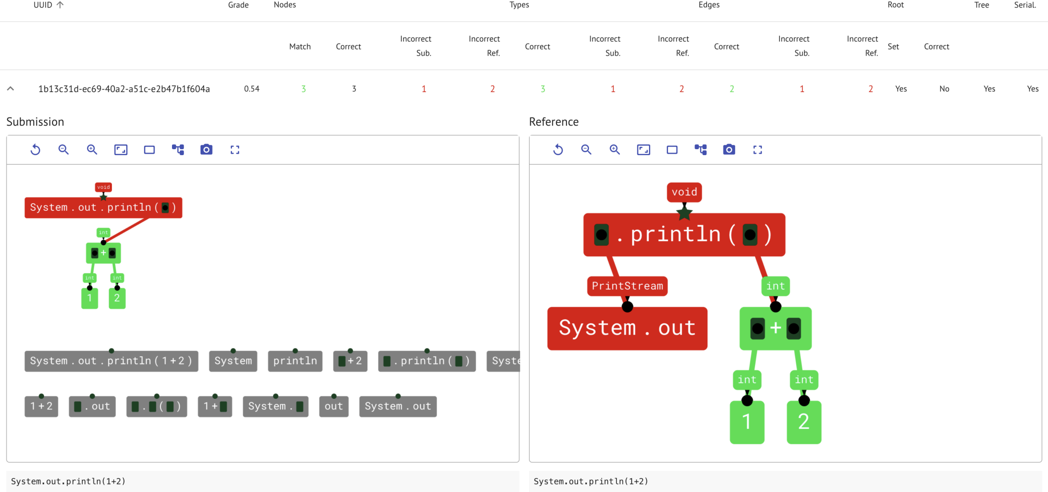
OVERVIEW INCORRECT NODES ASSESSMENT

JSON CSV

UUID	Grade	Nodes				Types			Edges			Root		Tree	Serial.
		Match	Correct	Incorrect Sub.	Incorrect Ref.	Correct	Incorrect Sub.	Incorrect Ref.	Correct	Incorrect Sub.	Incorrect Ref.	Set	Correct		
4b638bdb-677b-4743-97d5-76deec28b4ec	0.47	5	5	2	1	0	0	6	3	3	2	No	No	Yes	Yes
facbe6ef-c7d7-4ff2-b1ad-2dbf7619c5f4	0.29	2	2	2	1	0	0	3	0	3	2	Yes	No	Yes	Yes
e0cfbb59-3da3-48de-a22c-89c324ca4893	1.00	3	3	0	0	3	0	0	2	0	0	Yes	Yes	Yes	Yes
1b6f1c4a-cdb0-4bdc-b05a-b01a0c0ffa0f	0.19	3	3	6	3	1	8	5	0	8	5	Yes	Yes	Yes	Yes
d8eca99e-5e68-4637-902c-645819f39e12	1.00	5	5	0	0	5	0	0	4	0	0	Yes	Yes	Yes	Yes
06534f5b-d9c9-4991-891b-67cb510a2bef	1.00	3	3	0	0	3	0	0	2	0	0	Yes	Yes	Yes	Yes
de267ad4-e48e-4053-ae14-c02265e4fdcf	1.00	1	1	0	0	1	0	0	0	0	0	Yes	Yes	Yes	Yes
1b0e5b09-b4f0-4530-af65-dc176252e6f3	0.81	6	6	0	0	2	1	4	5	0	0	Yes	Yes	Yes	Yes
beaadece-01a2-4078-8e80-fed5dd8510b9	0.75	3	3	0	0	0	0	3	2	0	0	Yes	Yes	Yes	Yes

Rows per page: 10 1-9 of 9

(b) View of one of the rows expanded to reveal a side-by-side comparison between a student's submission and the reference solution. Red nodes on the left are wrong in the student's submission and red nodes on the right are nodes from the reference solution that are missing in the student's submission.



4.6 Conclusion

In this Chapter, we have used the theoretical framework we have proposed to reason about notional machines in practice in the analysis and improvement of ExpressionTutor, a family of notional machines that can be used to focus on different aspects of expressions, such as parsing and type-checking.

ExpressionTutor is language agnostic but here we use Java as a language under focus. That stands in contrast to the analysis in Chapter 3, which used notional machines focused on untyped lambda calculus, a language of mostly theoretical use. By focusing on Java, we show the applicability and scalability of our approach to "real world" scenarios.

In addition to the analytical benefit, the commutative diagram that we use to reason about ExpressionTutor maps directly to components of the real tool. In a real use by a student, the function that describes the operation in the notional machine level (f_{NM}), is enacted by the student. Combining this fact with an implementation of the components of the commutative diagram leads to a tool that can automate the generation and assessment of educational activities centered on the notional machine.

Finally, to further ground the development of the tool, we have analyzed students' answers to exam questions that required them to draw expression trees on paper. This analysis had the original purpose of informing practical aspects of the design and implementation of the tool, but may have the potential to be used in other educational contexts, which we discuss further as part of suggestions for future work in Chapter 7.

Chapter 5

Notional Machines as Assessment Instruments

Until now, we have used the commutative diagram that defines the soundness of notional machines as the basis for the design and analysis of notional machines. But we also would like to investigate actual uses of notional machines and using essentially the same reasoning framework can help us in this investigation.

We can separate the uses of notional machines into two categories: teaching and assessment. In the teaching context, a notional machine would typically be used in instructional materials or in class (they could equivalently be used by students when studying on their own). Although it is essential to evaluate the effectiveness of a notional machine for teaching, we will instead focus on uses of notional machines for assessment. In the assessment context, a notional machine is an *assessment instrument*. As such, it would typically be used by students during an assessment activity and the instructor would then use the students' answers to assess their understanding of the aspect of the programming language under focus by the notional machine. For example, students could be assessed in an exam with a question that asks them to, given a program, draw a memory diagram that describes the state of the program at a given point in the execution.

The appeal of assessing students using notional machine questions compared to traditional question formats is that, when compared with multiple-choice questions, notional machine questions have a higher information density and, when compared to open questions, notional machine questions are more precise and less ambiguous. Another advantage, when compared to open questions, is that notional machine questions are more amenable to automation.

The use of notional machines as assessment instruments relies on the assumption that students' usage of a notional machine reflects their understanding of the

aspect of the programming language under focus by that notional machine. More precisely, instructors assume that usage patterns of a notional machine correspond to patterns in the student's understanding of the aspect of the programming language under focus by that notional machine. In fact, we interviewed instructors and identified evidence supporting that indeed this assumption is made in practice. These interviews are described in Section 5.1.

Under this assumption, the effectiveness of a notional machine as an assessment instrument relies on there being a correspondence between the notional machine and the aspect of the programming language under focus by that notional machine. This correspondence is precisely what the soundness condition describes. So, we use the soundness condition to devise a methodology to design experiments that can evaluate the effectiveness of a given notional machine as an assessment instrument. This methodology is described in Section 5.2. We then instantiate this methodology to design a pilot study to evaluate ExpressionTutor. This pilot study is described and its results are analyzed in Section 5.3. A final discussion is provided in Section 5.4.

5.1 The Instructor's Perspective

We performed semi-structured interviews with three instructors with the goal of investigating how instructors use ExpressionTutor as an assessment instrument. The instructors were first given a text explaining ExpressionTutor as an instrument to assess the students' understanding of the structure of expressions. They were then given ten ExpressionTutor diagrams submitted by students for the same parsing activity. Their task was to grade each diagram according to how well the student that submitted the diagram demonstrated to understand the structure of the expression in the activity. While grading, they were asked to think aloud and explain their reasoning.

The instructors developed various grading schemes. Although one of the instructors said that it would be better for grading if students had to submit explanations together with the trees, for all the instructors, the presence or absence of specific patterns in the diagram had a measurable impact on the grade. The instructors justified these choices typically by mentioning what they believe the presence or absence of those patterns meant in terms of the student's understanding of the structure of the expression. For example, Instructor 1 said when looking at one of the trees: "they are trying to express something... I would say it's not completely wrong". When grading a solution of a typing activity where some nodes were not labeled with types, Instructor 2 said that if the non-labeled

nodes mean the student thinks those nodes don't have types this would mean a worse understanding of expressions. The same instructor said about the presence of a return node in one of the solutions: "it's clear they confuse expression and statement".

These interviews provide evidence for our assumption that instructors assume that usage patterns of a notional machine correspond to patterns in the students' understanding of the aspect of the programming language under focus by that notional machine.

5.2 Experiment Design Methodology

This experiment design methodology relies on the relationship between a notional machine and the aspect of the programming language under its focus described by the soundness condition.

The idea is that if a notional machine *NM* focused on an aspect f of a language *PL* is used in an assessment activity, then what we're assessing is the students' knowledge about f. So we can devise questions about f using the notional machine *NM* and compare them with "ground truth" questions about f that do not use *NM*. In the context of a course, students should be assessed according to clear learning goals so one should select an f that is aligned with the learning goals of the course.

Let's describe the methodology in more detail. Given a notional machine *NM* described by $(A_{NM}, B_{NM}, f_{NM} :: A_{NM} \rightarrow B_{NM})$ and an aspect of a programming language *PL* described by $(A_{PL}, B_{PL}, f_{PL} :: A_{PL} \rightarrow B_{PL})$ under focus by this notional machine, the steps of the experiment design are as follows:

Design ground truth questions Devise a set of questions Q_{PL} such that each question $q_{i_{PL}} \in Q_{PL}$ should be formulated in terms of an input $v_{i_{PL}} :: A_{PL}$ and the operation $f_{PL} :: A_{PL} \rightarrow B_{PL}$, such that the correct answer can be obtained with information present in $f_{PL}\ (v_{i_{PL}}) :: B_{PL}$.

These questions should be "ground truth" in the sense that they should be considered more likely (or at least as likely) to be answered correctly if the students understood the aspect of the programming language under focus. For these questions to have this effect, not only the questions themselves have to be carefully designed but so must be the educational intervention into which they are embedded, i.e. the context in which the questions are asked. One such context could be, for example, that of Mastery Checks [Bloom, 1968; Guskey, 2010; Wrigstad and Castegren, 2019].

Design notional machine questions Devise a set of corresponding questions Q_{NM}, in the notional machine space. Each question $q_{i_{NM}} \in Q_{NM}$ should be formulated in terms of an input $v_{i_{NM}}$ obtained from the input $v_{i_{PL}}$ to the corresponding ground truth question $(v_{i_{NM}} = \alpha_A\ (v_{i_{PL}}) :: A_{NM})$. The correct answer to this question, resulting from operating the notional machine, would be given by $f_{NM}\ (v_{i_{NM}}) :: B_{NM}$. The idea of this setup is that each notional machine question should assess the same conceptual knowledge required to answer the corresponding ground truth question:
$f_{NM}\ (v_{i_{NM}}) \equiv \alpha_B\ (f_{PL}\ (v_{i_{PL}}))$.

Compare and analyze answers Compare the answers to the ground truth questions with the answers to the notional machine questions. The expectation is that a student would answer the notional machine questions correctly if and only if they would answer the ground truth questions correctly.

This comparison may be challenging because these two types of questions may not require exactly the same information to answer. Although we would like the information contained in both answers to be the same, a ground truth question would likely have smaller granularity, eliciting only part of the information captured by f_{PL}. If that's the case, we can describe this "part of" information by a *query* on the abstract representation of the notional machine or a *pattern* matching part of this abstract representation.

In the remainder of this chapter, we demonstrate how this methodology can be applied by designing a pilot study that evaluates the effectiveness of ExpressionTutor, (we introduced ExpressionTutor in Section 2.3, used it later in Section 2.5, and expanded it further in Chapter 4) as an assessment instrument.

5.3 Pilot Study Design

We designed and ran a pilot study in the context of a second-semester university course that uses Java as a programming language to evaluate the effectiveness of ExpressionTutor as an assessment instrument.

As we have seen in Section 2.3, ExpressionTutor is actually a family of notional machines, that can be instantiated for different programming languages and different aspects of these programming languages. In this study, we focus on the instantiations of ExpressionTutor for Java focusing on parsing.

5.3.1 Designing Ground Truth and Notional Machine Questions

The pilot study was designed as a quiz administered in the Programming Fundamentals 2 (PF2) course, taught in the second semester of the first year. In the next section, we describe the course context in more detail. After that, we describe the structure of the quiz and subsequently how its content corresponds to the ground truth and notional machine questions.

Course Context

The quiz was administered during the Programming Fundamentals 2 (PF2) course, an introduction to object-oriented programming in Java. The course is taught in the second semester of the first year of the bachelor's degree in computer science. It follows Programming Fundamentals 1 (PF1), which is taught using the book How to Design Programs (HtDP) by Felleisen et al. [2018] that follows a sequence of sublanguages of Racket. PF2 picks up where PF1 left off, starting with a subset of Java that is as close as possible to the subset of Racket used in PF1. This approach builds upon our experiences in bridging from functional to object-oriented programming, described in previous work [Santos et al., 2019], and takes it even further. It also has some commonalities with the approaches described by Matthias Felleisen et al. [2012]; Gray and Flatt [2003].

Knowledge about the programming language The quiz was administered in the second week of the course. At that point in the course, the students had seen only static method calls, static method definitions containing only a single `return` statement (and optionally assertions), arithmetic operators, and binary operators. The programs written in class and in the assignments were all using JTamaro, a library based on PyTamaro [Chiodini et al., 2023], which is designed to help teaching programming by focusing on composition and using graphics as a medium. The library is inspired by the image teachpack from How to Design Programs (HtDP) [Felleisen et al., 2018] and the Haskell *diagrams* library [Yates and Yorgey, 2015; Yorgey, 2012].

Knowledge about the notional machine In the lecture before the quiz, the students had been introduced to expression trees. They were shown how expressions form trees and how we can traverse these trees to evaluate the expression. In that lecture, the students also did one small in-class exercise to practice. The explanations and the exercise were done on paper, not using ExpressionTutor. The students did the quiz in the subsequent lecture, which happened two days

later. Before doing the quiz, the students were shown how to use ExpressionTutor to click together expression trees. They were also given an exercise to do using ExpressionTutor so they could practice before the quiz.

Quiz Structure

The quiz consists of multiple questions administered on Moodle. Figure 5.1 shows part of a Moodle page with a question. Each question in the quiz contains three parts:

1. A code context, which contains a fragment of Java source code.

2. An ExpressionTutor activity question, which effectively consists of four parts:

   (a) A link to an ExpressionTutor activity to be completed on our platform;

   (b) A piece of text that identifies the notional machine input (i.e., the expression for which the student should provide a tree) either giving a line number, as shown in Figure 5.1, or repeating the source code of the expression in the text of the question.

   (c) A text that explains which activity should be performed and a set of reminders about how to operate with the notional machine;

   (d) Instructions on how to save their answer. This is necessary because the ExpressionTutor platform has no Moodle integration so the student must explicitly click "Save" on the platform and copy the given URL back into the quiz.

3. Several multiple-choice items. In Figure 5.1, only two multiple-choice items are shown.

   (a) A stem;

   (b) A number of options (from which only one can be chosen). To alleviate the problem of guessing and unserious attempts, we communicated clearly that the quizzes would not be graded and we added an "I don't know" option to every item (on top of the usual possibility of not answering). In addition to that, every question also had a "Code Does Not Compile" option because, as shown in Chapter 4, parsing and typing activities can fail. We decided to add this option to every question also because it would be consistent with future quizzes.

(c) A free-text field to write a brief explanation for the answer. The inclusion of this field follows the suggestions of Chiodini and Hauswirth [2021], which shows the risks of adopting answers to multiple-choice questions as a pedagogical instrument without taking into account student's explanations for their answers. In Section 5.3.2 we explain how exactly these explanation, were used.

In Figure 5.1 the ExpressionTutor question is presented before the set of multiple-choice items but, to mitigate possible learning effects, we swapped this order in half of the questions. Note that irrespective of the order in which the questions were presented, we cannot guarantee that the students completed them in this order.

Quiz Content

Notional machine questions

In terms of the methodology presented in Section 5.2, each ExpressionTutor question corresponds to a notional machine question $q_{i_{NM}} \in Q_{NM}$. The operation $f_{NM} :: A_{NM} \rightarrow B_{NM}$ is parsing and the input $v_{i_{NM}} :: A_{NM}$ is given by:

(1) the code context;

(2) the information that identifies the expression in the code context (the line number or the text of the expression);

(3) a set of distractor nodes.

The correct answer is given by $f_{NM}\ (v_{i_{NM}}) :: B_{NM}$. For example, Figure 5.3 contains the input and expected solution for the ExpressionTutor question shown in Figure 5.1.

Ground truth questions

On the side of the ground truth questions, a question $q_{i_{PL}} \in Q_{PL}$ actually correspond, here to a multiple-choice item of a quiz question. As we have seen in Section 4.4.1, the inputs $v_{i_{PL}}$ and $v_{i_{NM}}$ for a parsing activity can conceptually be considered the same: the text of the expression to be parsed and the code context, although in ExpressionTutor we augment $v_{i_{NM}}$ with distractor nodes.

The stem of each multiple-choice item $q_{i_{PL}}$ was formulated such that answering with one of the options corresponds to a pattern on the tree of the corresponding notional machine question $q_{i_{NM}}$. Most multiple-choice items have effectivelly two options to choose from, in which case, either the correct answer to the multiple-choice item corresponds to the presence of a pattern in the tree and the wrong

Figure 5.1. Beginning of the page in the Moodle quiz as shown to students containing an ExpressionTutor question for a parsing activity.

Code context

Consider the following Java source code:

```
1 class C {
2     public static int prod(int n) {
3         return 1 * n;
4     }
5     public static int sum() {
6         return prod(2) + 3;
7     }
8 }
```

ExpressionTutor question

Open this ExpressionTutor activity.

Create the tree corresponding to the expression in line 6.

If the piece of code is not an expression or the code does not compile, select the "Code Does Not Compile" button instead.

To construct the tree, do the following:

- Connect together the right nodes to form a tree (all the nodes you might need are already there).
- **Mark the root of the tree** with a star by double-clicking on that node.

To submit your answer, click the **Save** button at the bottom of the page, copy the URL shown in the dialog that will open, and paste it below.

Answer:

Multiple-choice question items

Answer the following multiple-choice questions **with respect to the expression in line 6**:

Question A.1: Is `return` part of the expression?

○ Yes
○ No
○ Code Does Not Compile
○ I don't know

Explain your reasoning:

Question A.2: Is `;` part of the expression?

○ Yes
○ No
○ Code Does Not Compile
○ I don't know

Explain your reasoning:

answer corresponds to the absence of the pattern or vice versa (the correct answer corresponds to the absence of the pattern and the wrong answer corresponds to the presence of the pattern). For example, consider the multiple-choice item A.1 shown in Figure 5.1, whose stem reads: "Is `return` part of the expression?". The option "Yes" (a wrong answer) corresponds to the presence of a `return` node in the tree, and the option "No" (the correct answer) corresponds to the absence of a `return` node in the tree. This multiple-choice item is testing if the student understands that the `return` keyword is not part of the expression and the idea is that this knowledge can be assessed both by the answer to this multiple-choice item and by the presence or absence of a `return` node in the tree.

Notice that this item has actually four options to choose from and in Section 5.3.3 we will see how we deal with this. We also need to incorporate in the answers the information about student's explanations, which we discuss in the next Section.

The content of the questions (the choice of the specific information we were trying to elicit from the students) was heavily influenced by our previous work on programming language misconceptions [Chiodini et al., 2021]. Our intention was to try to ask questions that could reveal that a student has a previously known programming language misconception. In fact, we believe that it could be perhaps possible to use notional machines to help detect programming language misconceptions but that remains to be studied in future work (Chapter 7) and is not the focus of this research.

For understandability and reproducibility, the entire content of the quiz is shown together with the analysis of the results in Sections 5.3.4 and 5.3.5.

5.3.2 Analyzing Explanations

We want to compare the answers to the notional machine questions with the answers to the ground truth questions. Each ground truth question is made of a multiple-choice item and a textual explanation of the answer. The reliability of this ground truth instrument depends heavily on our analysis of the explanations provided by the students. Following the recommendations of Chiodini and Hauswirth [2021], we classified the explanations of the students into four categories[1]:

Expl-Correct The explanation shows with enough strength that the student has the correct understanding required to solve the question.

[1] We used the same categories as Chiodini and Hauswirth [2021] but the conditions for each category here diverge a little from the ones the authors use.

Expl-Wrong The explanation shows with enough strength that the student has the wrong understanding about the question posed and thus cannot properly answer the question.

Expl-Imprecise The explanation is insufficient by itself to support a correct answer.

Expl-Missing No explanation is provided or the text provided is not an explanation.

Let's see some examples of explanations that fall into each category. Table 5.1 shows answers and explanations given by students for the multiple-choice item "Is `return` part of the expression?" classified with each of the categories.

Category	Answer	Explanation
Expl-Correct	No	return is just a command for the method
	No	Return tells the compiler which expression need to return in that method, its not part of the inner expression
	No	return is a statement
Expl-Wrong	Yes	we require a return statement without the return part we can not have the code
	Yes	return is part of the method, and therefore is a part of the expression
	Yes	Return is a function that says to the program to stop running and return a value.
Expl-Imprecise	No	Because we just construct a tree, we dont need exactly output
	No	the expression is what the code should execute
	No	If I remember correctly, which I might not :D , return is not an expression but a method and/or a statement.
Expl-Missing	No	have no clue
	Yes	i think s, but i am not so sure

Table 5.1. Examples of answers and explanations given by students for the question "Is `return` an expression?" classified with each category.

Classifying these explanations is not always straightforward. We briefly discuss

some of the challenges we faced when classifying the explanations given by students to their answers.

Explanation That Contradicts the Answer

Sometimes, a student chooses an answer that is correct but provides an explanation that is wrong or vice versa. For example, Table 5.2 shows examples of correct answers justified with wrong explanations. These answers are particularly worrisome because not only they are factually wrong (`return` is not a call, a method, or a function) but even if they were factually correct (if `return` was a call, a method, or a function) that would be a reason for `return` to actually be part of the expression, contradicting the answer.

Category	**Answer**	**Explanation**
Expl-Wrong	No	Return is a call
	No	is a method in Java
	No	return is a function

Table 5.2. Examples of answers and explanations given by students for the question "Is `return` an expression?" showing that a wrong explanation can be given for a correct answer.

When the explanation contradicts the answer, we can more confidently classify it as *Expl-Wrong* but when it is not an explanation for the answer or the explanation is unclear or substantially unrelated to the answer then we classify it as *Expl-Imprecise*. Table 5.1 contains some examples.

Same Explanation for Opposite Answers

It is also possible that the same explanation is given both for a correct and a wrong answer. For example, Table 5.3 shows the same explanation given for both the correct answer and the wrong answer. To resolve this ambiguity, we don't classify the text of the explanation by itself but as a text that justifies the answer. For example, the explanation "return is a statement" for the answer "No" reads as "Is return part of the expression? No because return is a statement" and is classified as *Expl-Correct*. While the explanation "It's a return statement" for the answer "Yes" reads as "Is return part of the expression? Yes because it's a return statement", meaning that statements are expressions, so the explanation is classified as *Expl-Wrong*. Of course, it is also possible that the student simply

made a mistake selecting the wrong option but there's not enough information in the explanation to determine that.

Category	Answer	Explanation
Expl-Correct	No	it is statement
Expl-Wrong	Yes	It's a return statement

Table 5.3. Examples of answers and explanations given by students for the question "Is `return` an expression?" showing that the same explanation can be given for both a correct and a wrong answer.

A Different Interpretation of the Question

With this approach, it is much more likely to classify with explanation *Expl-Wrong* a submission with a correct answer than it is to classify with explanation *Expl-Correct* a submission with a wrong answer. It is nevertheless possible to have a submission with a wrong answer and an explanation *Expl-Correct*. For example, for the question "Is `1 + 2 + 3` equivalent to `(1) + (2) + (3)`?", a student answered "No" and explained with "It is equivalent to (1 + 2) + 3". As we will discuss in Section 5.3.3, although this is a wrong answer for behavioral equivalence, it is a correct answer for structural equivalence.

Using Explanations to Filter Answers

Because of all this variability in the relationship between answers and explanations, we are conservative and consider in the results reported in the next section only (1) correct answers that come with explanations classified as *Expl-Correct* and (2) wrong answers that come with explanations classified as *Expl-Wrong*. The other answers were filtered out.

5.3.3 Results: Compare and Analyze Answers

The course had 89 students registered for the final exam. The students were asked at the beginning of the semester for permission to have their quizzes analyzed as part of a research study and 67 gave their consent[2]. Of the 67 students who gave their consent, 54 students participated in the quiz.

[2]The protocol for this study was approved by the university's ethics committee.

The students had at most 45 minutes to answer the questions. Answers submitted after that time were not considered. To select higher-quality answers, we not only used the explanations given by students as explained before but also excluded results with ExpressionTutor diagrams that didn't form a tree. Because the students had to explicitly click save on the ExpressionTutor platform and copy the given URL back into the quiz, there were also some results that had to be discarded because the student copied the wrong URL (a URL with an expression tree solution of one question submitted to another question).

The results vary considerably between multiple-choice items so we report the results of each multiple-choice item individually. For each question, we show a Figure (e.g. Figure 5.3 for exercise A) containing the input to the question and the expected solution for the ExpressionTutor question, as we described before. Remember that part of this input, namely the code context and the text that identifies the expression of interest in the code context are both inputs to (1) the ExpressionTutor question and to (2) the multiple-choice items. For each multiple-choice item, we show a Figure (e.g. Figure 5.5 for exercise A.1) containing:

(a) The stem with the options that came with it and the correct answer.

(b) A description of the tree pattern that is expected to be associated with the wrong answer to that multiple-choice item followed by an example of an answer to the ExpressionTutor question submitted by a student containing a tree that has this pattern. The pattern could be, for example, the *presence* of certain nodes or a certain subtree but could also be the *absence* of certain nodes or subtrees.

(c) A table comparing

    (1) the correctness of answers to the multiple-choice item (**MC answer** rows) with

    (2) the correctness of answers to the ExpressionTutor question with respect to the tree pattern expected to be associated with that multiple-choice item (**Tree pattern** columns).

    There are four cases and for each case, we report the number of students in that case (between parenthesis) and the percentage this number represents of the number of students in the four cases. For example, in the cell in column "Tree pattern - correct" and in row "MC answer - correct" are students that submitted trees that don't contain the wrong pattern and gave the

ET Question	MC Item	Stem	Results
Figure 5.3	A.1	Is `return` part of the expression?	Figure 5.5
	A.2	Is `;` part of the expression?	Figure 5.6
	A.3	Does the expression contain `1 * n`?	Figure 5.7
	A.4	Does the expression contain `1 * 2`?	Figure 5.8
	A.5	Is 2 an expression?	Figure 5.9
Figure 5.10	B.1	How many steps is `1 + 2 + 3` evaluated in?	Figure 5.11
	B.2	Is `1 + 2 + 3` equivalent to `(1) + (2) + (3)`?	Figure 5.12
	B.3	To evaluate `1 + 2 + 3`, is `2 + 3` evaluated before adding 1?	Figure 5.13

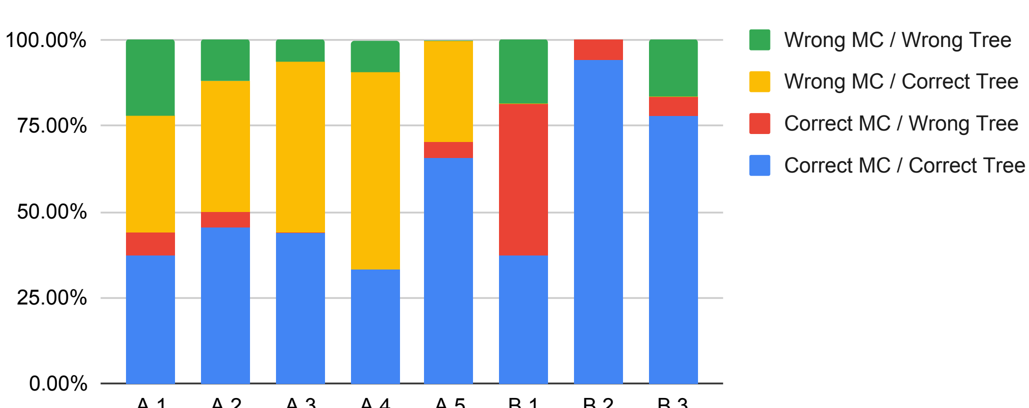
Table 5.4. For each multiple-choice (MC) item, a Figure with the corresponding ExpressionTutor (ET) question, its stem, and the Figure summarizing the comparison between the answers to that item and the corresponding expression trees.

correct answer to the multiple-choice item. Notice that the trees these students submitted may have other mistakes but they are not relevant for that comparison. The row "MC answer - wrong" doesn't include the answer "I don't know", which was included as an option to avoid guessing, and "Code does not compile", which was included in every question (as explained before).

(d) For completeness, we also report a list of exclusion criteria (previously described) and, for each criterion, the number of students (between parenthesis) and the percentage this number represents of the total number of students that participated in that quiz. Notice that some students may be in more than one criterion of this list.

Table 5.4 shows the stem of each multiple-choice item, a reference to the Figure containing the input to the corresponding ExpressionTutor (ET) question and the correct answer, a reference to the Figure that summarises the results of the comparison between the answers to that item and the answers to the ExpressionTutor question. To understand the stems in the multiple-choice items of question A, we need to know which expression we are referring to. This information was shown to the students as depicted in Figure 5.1, which shows the beginning of the Moodle page for the quiz. Before the multiple-choice items,

Figure 5.2. Correctness of answer to multiple-choice (MC) questions vs. correctness of answers to ExpressionTutor questions.



we state “Answer the following multiple-choice questions **with respect to the expression in line 6**”.

Result Overview

Figure 5.2 shows a summary of the values in the **Result** tables (listed in Table 5.4). The results of each table are shown here in one vertical bar, with the four cases stacked on top of each other.

The results we would expect are (1) correct answers to multiple-choice items whenever there are also correct answers to the corresponding ExpressionTutor questions (with respect to the tree pattern of the corresponding multiple-choice item), shown in blue, and (2) wrong answers to multiple-choice items whenever there are also wrong answers to the corresponding ExpressionTutor questions, shown in green. From the unexpected cases, the most notable was the substantial occurrence of wrong answers to multiple-choice items paired with correct answers to the corresponding ExpressionTutor questions, shown in yellow, in the multiple-choice items of question A.

These results are analyzed in depth in Sections 5.3.4 and 5.3.5.

Student Interviews

We also conducted unstructured interviews with a small number of students (5 students) who submitted answers to the ExpressionTutor questions in the form of trees that had patterns that were unexpected given their answers to the corresponding multiple-choice items. Differently from the quizzes, which

happened in the first three weeks of the course, these interviews happened at the end of the semester. By the end of the semester, the students have had much more practice with expression trees and have even been asked an exam question about it. In the interviews, the students were first not shown their answers but were simply asked again to answer the contradicting pair of questions (the ExpressionTutor question and the multiple-choice item that contradicted it). In most cases, the students were able to answer both correctly. They then were shown the answers they gave and asked to further clarify and explain their answers.

In the next Section, we dive deeper into the results of the quiz and analyze them in light of the insights we gained from the interviews.

5.3.4 Question A

Multiple-Choice Items A.1 and A.2

Motivation for the item From our analysis of previous years' exam questions about expression trees (see Section 4.4.2), we know that students often have difficulty identifying which tokens belong to an expression when non-expression tokens appear in the same line as the expression. The multiple-choice items A.1 (Figure 5.5) and A.2 (Figure 5.6) were designed to assess that.

Analysis of the results One of the problems with the setup of this question was that the instructions and information present on the Moodle page were not the same as the one present on the ExpressionTutor page, where they were drawing the diagram. Figure 5.4 shows the ExpressionTutor page as seen by students when they followed the link "this ExpressionTutor activity" shown in the Moodle page that was reproduced in Figure 5.1. On the ExpressionTutor page the student is asked to "Construct a tree based on the following Java expression" and shows the expression `prod(2) + 3`. On Moodle, the student is given the entire code of class `C` and asked to "Create the tree for the expression in line 6". It is difficult to say the extent to which this difference in instructions affected the students' answers but we have some evidence that it did:

(a) during the quiz, two students asked for clarification pointing out this difference and asking which expression should they draw the diagram for;

(b) one of the submitted explanations seems to refer to that, saying "this is a piece of code and is not an expression, it is not compiler without first method (prod) on line 3.". This student answered "Code coes not compile".

Figure 5.3. Information given to students and expected solution to Expression-Tutor activity in question A.

(a) Code context followed by instructions shown to the students.

```
1 class C {
2     public static int prod(int n) {
3         return 1 * n;
4     }
5     public static int sum() {
6         return prod(2) + 3;
7     }
8 }
```

Create the tree corresponding to the expression in line 6.

(b) Distractor nodes that are part of the initial state of the activity.

prod( )+3 · prod(2)+3 · prod( )+ · prod(2) · 1*n · ; · 2 · * · 1 · return ; · ( ) · prod · return · n · 3 · prod( ) · + · 1*2 · 1*2+3 · 1*n+3 · * + · +3

(c) Expected solution.

+ · prod( ) · 3 · 2

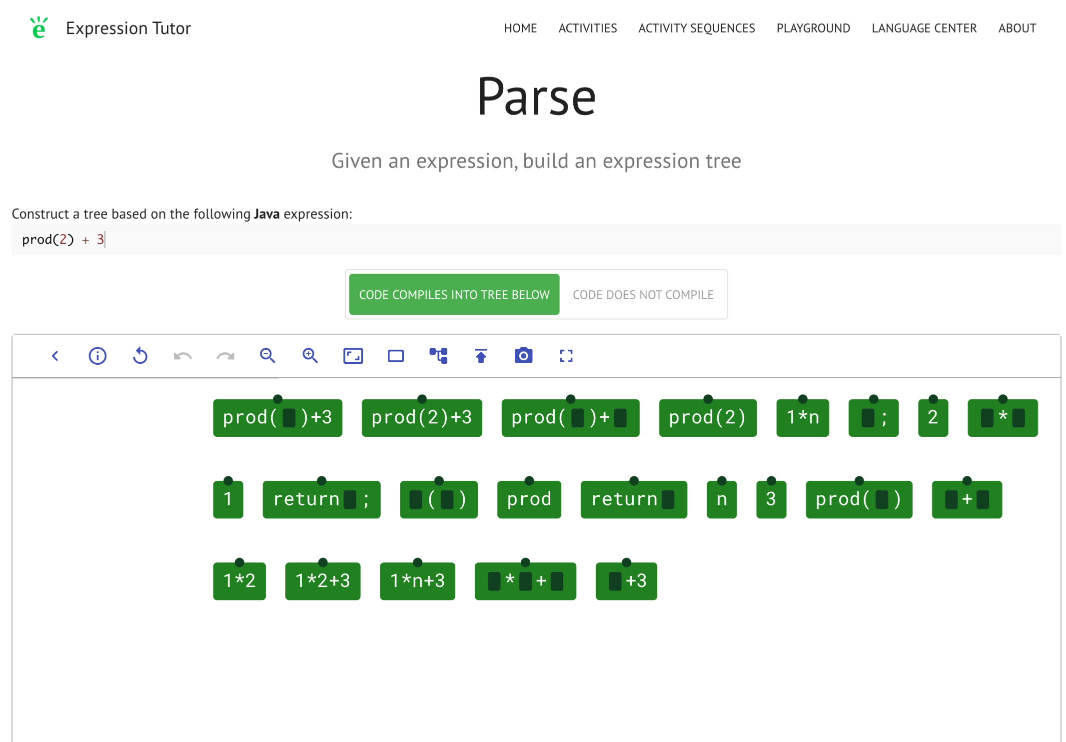


Figure 5.4. ExpressionTutor page as seen by students when coming from the Moodle question A.

(c) one of the students interviewed clarified that he thought **`return`** was part of the expression and the only reason for not including it in the tree was that the code shown on the ExpressionTutor page did not include it.

That could explain the high number of cases where the answer to multiple-choice item A.1 was wrong and the tree did not include the **`return`** token. The same problem could have affected the answers to multiple-choice item A.2, where the students were asked if the `;` token was part of the expression.

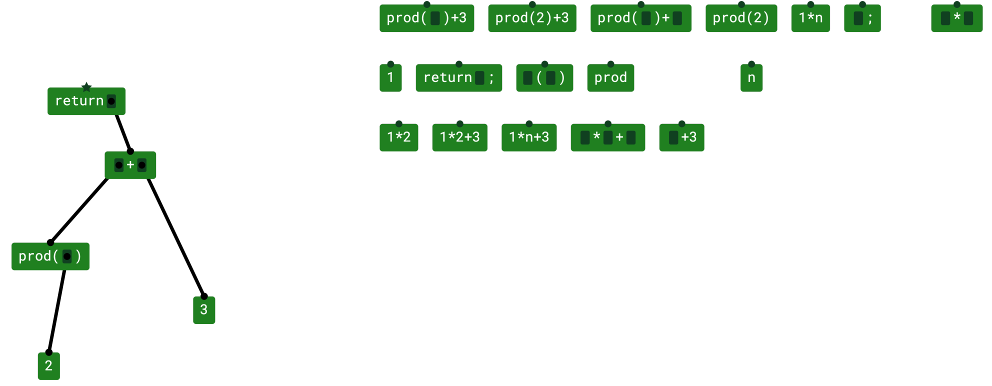
Figure 5.5. Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.1.

(a) Multiple-choice item A.1 and its correct answer.

Question A.1	Is `return` part of the expression?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	No

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a node containing return.



(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	6.2% (2)	37.5% (12)
	wrong	21.9% (7)	34.4% (11)

(d) Exclusion Results for multiple-choice item A.1

Exclusion Criteria	**Percent (Count)**
Answer is "Code Does Not Compile"	5.1% (2)
Answer is "I don't know"	2.6% (1)
Diagram is not a tree	7.7% (3)
Diagram of another question	2.6% (1)

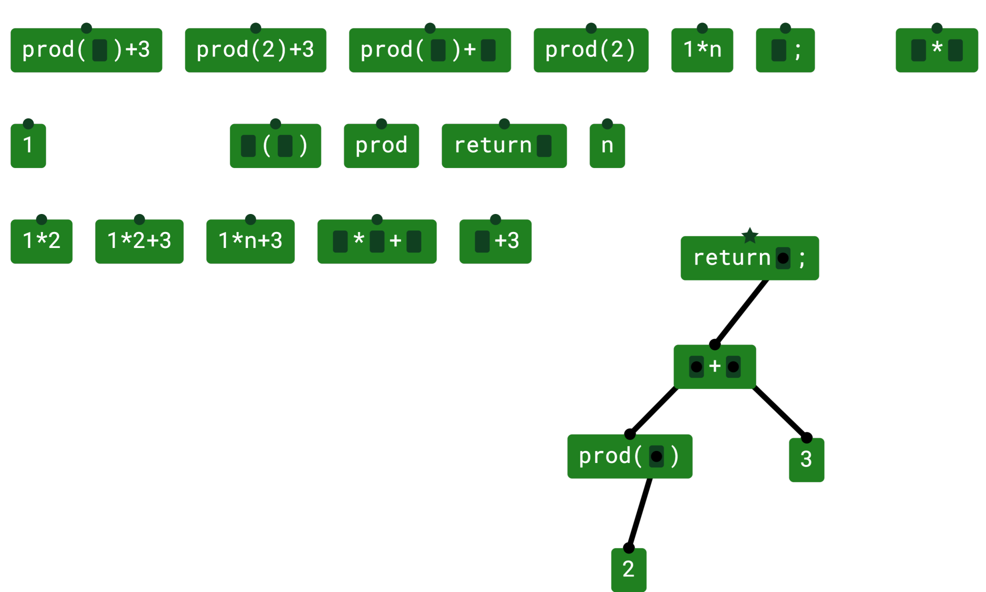
Figure 5.6. Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.2.

(a) Multiple-choice item A.2 and its correct answer.

Question A.2	Is ; is part of the expression?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	No

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a node containing semicolon.



(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	4.8% (2)	45.2% (19)
	wrong	11.9% (5)	38.1% (16)

(d) Exclusion Results for multiple-choice item A.2

Exclusion Criteria	Percent (Count)
Answer is "Code Does Not Compile"	2.1% (1)
Answer is "I don't know"	0.0% (0)
Diagram is not a tree	6.4% (3)
Diagram of another question	2.1% (1)

Multiple-Choice Items A.3 and A.4

Motivation for the item We also have evidence, from our analysis of previous years' exam questions about expression trees (see Section 4.4.2), that students confuse the static structure of expressions with what happens at runtime when they are evaluated. A symptom of that is the presence of tokens in the tree that are not part of the expression but that are part of other expressions that are evaluated when the first expression is evaluated. The multiple-choice items A.3 (Figure 5.7) and A.4 (Figure 5.8) were designed to assess that. To evaluate the expression of interest (the one in line 6: `prod(2) + 3`), the body of the method `prod`, which contains a multiplication, has to be evaluated. So we want to look for the presence of a node containing a multiplication token in the tree.

Analysis of the results Because both questions are essentially trying to detect the same underlying misconception, we aggregated the results for both questions in Table 5.5. In this table, results appear in the row "MC answer - correct" if the student answered correctly to both questions (meaning that the student doesn't appear to have the misconception) and in the row "MC answer - wrong" if the student answered incorrectly to at least one of the questions (meaning that the student appears to have the misconception).

Table 5.5. Aggregated results for multiple-choice items A.3 (shown and Figure 5.7) and A.4 (shown and Figure 5.8).

		Tree pattern	
		wrong	correct
MC answers	correct	0.0% (0)	23.8% (10)
	wrong	11.9% (5)	64.3% (27)

We see that whenever the tree was wrong, the student answered incorrectly to at least one of the questions. But a correct tree did not always imply a correct answer to the questions: A surprising number of students drew the tree without a node containing multiplication but answered Yes to whether the expression contains `1 * n` or `1 * 2`. Students justified their answers with various explanations that refer in one way or another to what is essentially the runtime behavior of the program. For example: "Yes, because it first evaluates prod(2) which contains 1 * n and then adds 3"; "because it is contained in prod(n)"; "Yes, because it substitutes n in 1 * n by 2 and, therefore, we have 1 * 2".

One of the students interviewed had answered Yes to one of the multiple-choice items and, differently from other interviewed students who were able to

answer the questions correctly during the interview, still said that the expression contained a multiplication. When asked for further clarification, the student said that it "contained" because it was "related". This interview was particularly revealing because during it there were often miscommunications that seemed to be related to limited knowledge of the natural language being used in the interview[3]. This is a valuable experience because it highlights that the sometimes limited knowledge of the natural language used in the course plays a big role in the results of the comparison between the answers to the multiple-choice items and the answers to the ExpressionTutor questions.

Further investigation with the students would be needed to better understand the high number of wrong multiple-choice answers that did not correspond to wrong expression trees.

[3]The interviews were conducted in English, which is neither the mother tongue of the interviewer nor the mother tongue of the student being interviewed, as it's the case for many students in the course.

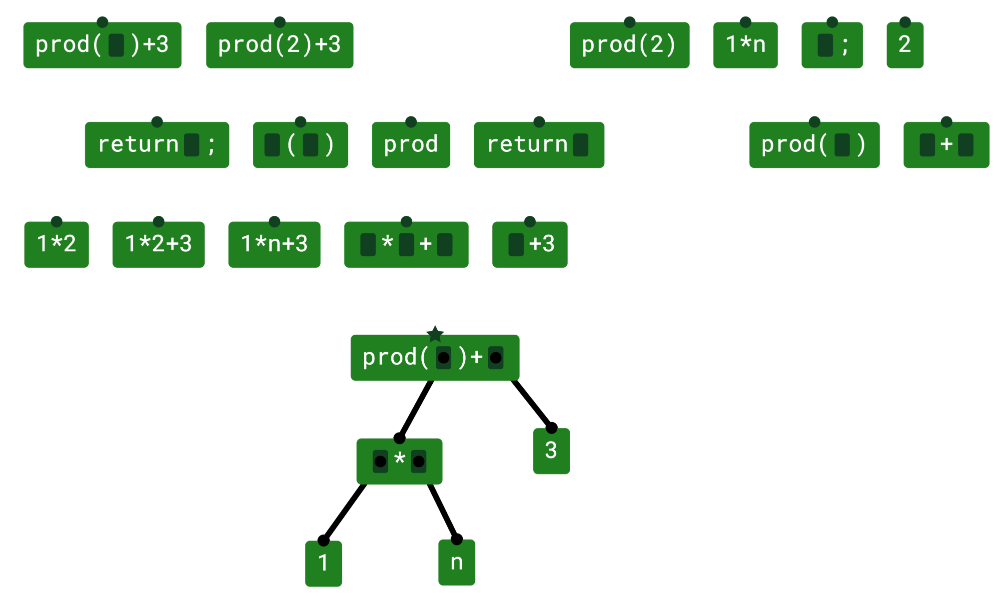
Figure 5.7. Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.3.

(a) Multiple-choice item A.3 and its correct answer.

Question A.3	Does the expression contain `1 * n`?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	No

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a node containing a multiplication.



(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	0.0% (0)	43.8% (14)
	wrong	6.2% (2)	50.0% (16)

(d) Exclusion Results for multiple-choice item A.3

Exclusion Criteria	Percent (Count)
Answer is "Code Does Not Compile"	2.4% (1)
Answer is "I don't know"	9.8% (4)
Diagram is not a tree	7.3% (3)
Diagram of another question	2.4% (1)

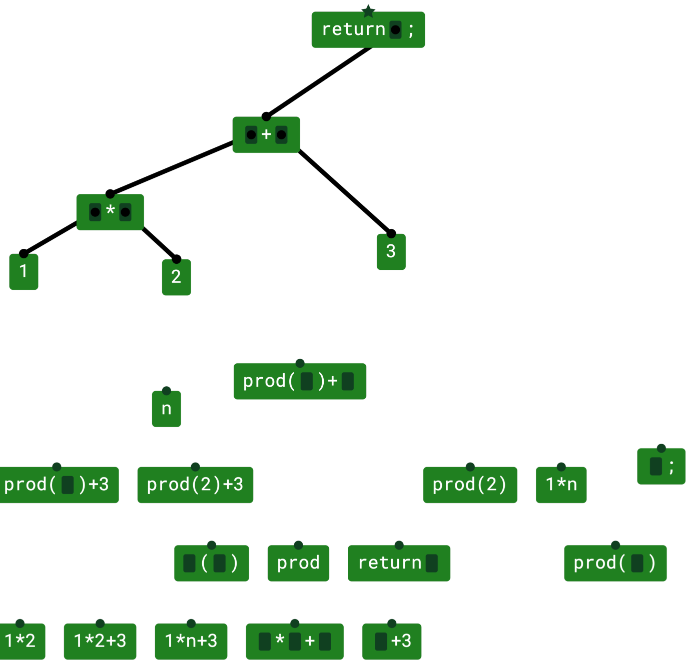
Figure 5.8. Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.4.

(a) Multiple-choice item A.4 and its correct answer.

Question A.4	Does the expression contain `1 * 2`?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	No

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a node containing a multiplication.



(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	0.0% (0)	33.3% (14)
	wrong	9.5% (4)	57.1% (24)

(d) Exclusion Results for multiple-choice item A.4

Exclusion Criteria	**Percent (Count)**
Answer is "Code Does Not Compile"	0.0% (0)
Answer is "I don't know"	2.1% (1)
Diagram is not a tree	6.4% (3)
Diagram of another question	2.1% (1)

Multiple-Choice Item A.5

Motivation for the item   Expressions can sometimes be a single token (for example a variable use or an integer literal) and these atoms can, of course, also be used as sub-expressions. It has been previously documented that students don't believe (or don't identify) these atoms to be expressions (documented by Chiodini et al. [2021] as NoAtomicExpression). We also have anecdotal evidence from previous years' exams that students sometimes draw expression trees with these atoms inlined. The multiple-choice item A.5 (Figure 5.9) was aimed at investigating the relationship between these two pieces of evidence.

Analysis of the results   We see that when the tree correctly contains a node with the number 2, in the majority of the cases (69%) the students answered that 2 is indeed an expression. The surprise here was the two students who answered the question correctly but still drew the tree with 2 inlined, for which we don't have a good explanation.

An important pattern can be seen in the students' explanations when they answered the multiple-choice item incorrectly (all associated with correct trees). In general, they do seem to understand that 2 is an atomic component that is part of the expression, which is perhaps why they were able to draw the tree correctly, but 2 cannot be an expression because expressions must be made of multiple parts. These are some examples of explanations for the wrong answer:

- "is just a value"
- "Its a value, an integer, therefore I dont think of it as being an expression"
- "2 is a value, its not an expression. Integer are natural number, double are all the numbers. This means that the value is 2 and the type could be both Int and Double"
- "2 is value and it can be maybe a piece of expression body"

That's in fact consistent with the NoAtomicExpression misconception.

If we connect back to the methodology, these results show that there is an incongruence in the information elicited by the multiple-choice item and the information required to draw the tree: to create a tree with a node containing 2 it seems that one just needs to identify 2 as a separate atom used to compose the expression, which doesn't require 2 to itself be called an expression.

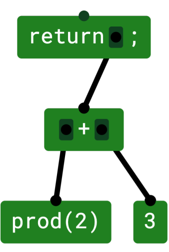
Figure 5.9. Comparison between answers to ExpressionTutor question A and answers to multiple-choice item A.5.

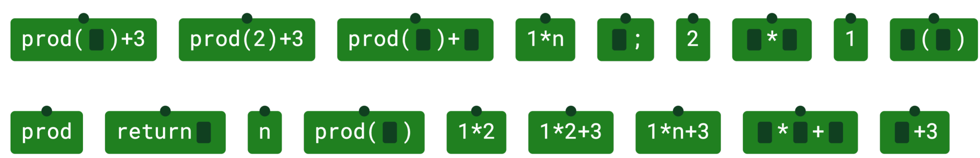
(a) Multiple-choice item A.5 and its correct answer.

Question A.5	Is 2 an expression?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	Yes

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a node with 2 inlined.





(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	4.5% (2)	65.9% (29)
	wrong	0.0% (0)	29.5% (13)

(d) Exclusion Results for multiple-choice item A.5

Exclusion Criteria	**Percent (Count)**
Answer is "Code Does Not Compile"	0.0% (0)
Answer is "I don't know"	0.0% (0)
Diagram is not a tree	6.2% (3)
Diagram of another question	2.1% (1)

5.3.5 Question B

Multiple-Choice Item B.1

Motivation for the item Figure 5.11 shows an example of a tree with the pattern we were aiming to identify with this multiple-choice item. We have also identified similar patterns in trees produced by students in exams, where binary operators applied to other binary-operator applications are depicted as a single node, especially when both operators are the same.

Analysis of the results We see that whenever the tree was correct, the student's answer to the multiple-choice question was also correct. However an incorrect tree was not associated with an incorrect answer to the multiple-choice question. In fact, the majority of the students (44.2%) actually answered the ExpressionTutor question incorrectly and the multiple-choice question correctly.

An explanation for that may be rooted in the formulation of this question, which doesn't appropriately follow the methodology we described. The multiple-choice question is formulated as "How many steps is `1 + 2 + 3` evaluated in?". But evaluation is not the operation that we're performing when building the tree. The operation on the ground truth question is different from the operation on the notional machine. We were aware of that when creating the question. This formulation came from our attempt to describe the fact that the correct tree has three levels without referring to the structure of the expression. However this indirect description led to a multiple-choice question that is not really asking for the same information as the notional machine question which may have skewed the results. Hypothetically, the structure of the expression could be flat and, at the same time, its evaluation could be in two steps.

Figure 5.10. Information given to students and expected solution to ExpressionTutor activity in question B.

(a) Code context followed by instructions shown to the students.

```
1 class C {
2     public static int m() {
3         return 1 + 2 + 3;
4     }
5 }
```

Create the tree corresponding to the expression 1 + 2 + 3 in line 3.

(b) Distractor nodes that are part of the initial state of the activity.

(c) Expected solution.

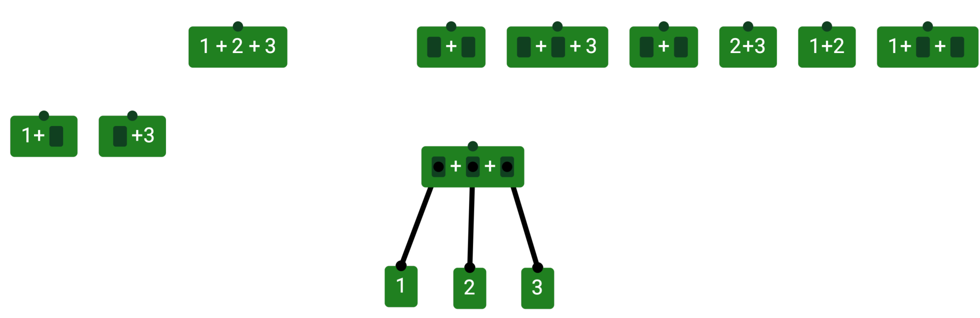
Figure 5.11. Comparison between answers to ExpressionTutor question B and answers to multiple-choice item B.1.

(a) Multiple-choice item B.1 and its correct answer.

Question B.1	How many steps is `1 + 2 + 3` evaluated in?
Options	A) 1 step B) 2 steps C) Code Does Not Compile D) I don't know
Correct answer	2 steps

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a node with two additions.



(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	44.2% (19)	37.2% (16)
	wrong	18.6% (8)	0.0% (0)

(d) Exclusion Results for multiple-choice item B.1

Exclusion Criteria	**Percent (Count)**
Answer is "Code Does Not Compile"	0.0% (0)
Answer is "I don't know"	0.0% (0)
Diagram is not a tree	2.3% (1)
Diagram of another question	0.0% (0)

Multiple-Choice Item B.2

Motivation for the item We know that some students inline operands in expressions like `1 + 2 + 3`. Our intention with the multiple-choice item B.2 (Figure 5.12) was to identify that. The idea was that if a student understands that `1 + 2 + 3` is equivalent to `(1) + (2) + (3)` then this student would know that `1`, `2`, and `3` are subexpressions and therefore would not inline them in the tree.

Analysis of the results There are several problems with this multiple-choice item. One problem is with the idea of "equivalence". Of course, `1 + 2 + 3` is equivalent to `(1) + (2) + (3)` in the sense that they reduce to the same value (behavior equivalence). But typically, when talking about associativity of binary operations, an instructor would say "`1 + 2 + 3` equivalent to `(1 + 2) + 3`" to mean that the operation is left-associative. Equivalence in this case means that the ASTs of both expressions are the same. That's a different kind of equivalence than the one asked about in this multiple-choice item. In fact, knowing that `1 + 2 + 3` is equivalent to `(1) + (2) + (3)` (in the sense that they reduce to the same value) doesn't tell us anything about the associativity of the operation. Interestingly one of the students answered "No" with explanation "It is equivalent to `(1 + 2) + 3`", revealing that this student interpreted the question as asking about associativity. This explanation was classified as *Expl-Correct* because when focusing on the idea of associativity, the student is correct and actually demonstrates more understanding than some of the students who answered "Yes". Here are some explanations of students who answered "Yes":

Category	**Answer**	**Explanation**
Expl-Imprecise	Yes	if we put the parenthes to identify each method it doesnt change nothing.
	Yes	because there arent another operations.
Expl-Wrong	Yes	yes theyre equivalent because there is no syntax errors.
	Yes	it is a commutative expression.

Table 5.6. Table of Answers and Explanations

Another problem, that is also highlighted by these examples, is that essentially this question is not as tightly related with a pattern in the tree as we would want. In other words, it is not really strictly following the methodology. Even considering the meaning of equivalence to be only behavioral equivalence, that

information doesn't directly correspond to a query/pattern on the expression tree. Knowing (or realizing) that `1 + 2 + 3` is equivalent to `(1) + (2) + (3)` could perhaps work as a hint or could help to trigger an insight in a student to produce a tree without the mistake (a tree without inlined numbers). But the information present in the answer to this question is not the same information present in the tree.

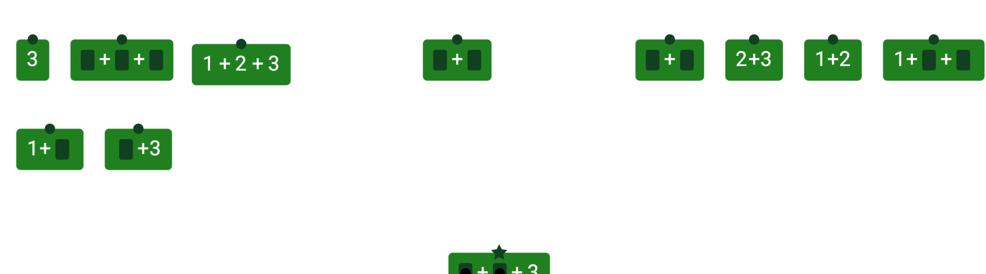
Figure 5.12. Comparison between answers to ExpressionTutor question B and answers to multiple-choice item B.2.

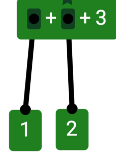
(a) Multiple-choice item B.2 and its correct answer.

Question B.2	Is `1 + 2 + 3` equivalent to `(1) + (2) + (3)`?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	Yes

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Absence* of a non-inlined numbers.





(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	5.9% (2)	94.1% (32)
	wrong	0.0% (0)	0.0% (0)

(d) Exclusion Results for multiple-choice item B.2

Exclusion Criteria	**Percent (Count)**
Answer is "Code Does Not Compile"	5.3% (2)
Answer is "I don't know"	2.6% (1)
Diagram is not a tree	2.6% (1)
Diagram of another question	0.0% (0)

Multiple-Choice Item B.3

Motivation for the item Our intention in this multiple-choice item was to detect another possible tree formulation for the same expression, shown in Figure 5.13. In this formulation, addition is depicted as a right-associative operation so a binary addition is applied to 1 and the result of applying addition to 2 and 3.

Analysis of the results In this question, we were more strict and did not include trees like the ones shown in Figure 5.11, with the binary operators flattened into a single level. That's because we wanted to clearly distinguish right-associative trees from left-associative trees.

Here the results are more clearly what we expected, with only one student answering the multiple-choice item correctly but producing the wrong tree.

Notice that, like the multiple-choice item B.1, this multiple-choice item does not strictly follow the methodology because the multiple-choice item also asks about the runtime behavior of the program. The student needs to not only be able to produce the correct expression tree but also understand how to evaluate them. This was something explained and worked on in class when expression trees were introduced but it's an additional piece of information that, strictly speaking, creates a mismatch in the information required to answer the two kinds of questions.

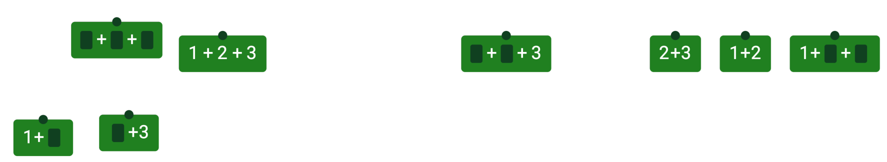
Figure 5.13. Comparison between answers to ExpressionTutor question B and answers to multiple-choice item B.3.

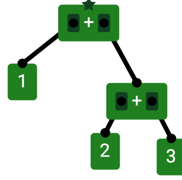
(a) Multiple-choice item B.3 and its correct answer.

Question B.3	To evaluate `1 + 2 + 3`, is `2 + 3` evaluated before adding `1`?
Options	A) Yes B) No C) Code Does Not Compile D) I don't know
Correct answer	No

(b) Submission with tree pattern expected to be associated with the wrong answer.

Wrong tree pattern: *Presence* of a subtree containing 2 + 3 (with children possibly inlined).





(c) Comparing correctness of multiple-choice item (MC) answers with correctness of tree. Showing percentage of submissions and the number of submissions in parentheses.

		Tree pattern	
		wrong	correct
MC answer	correct	5.5% (1)	77.8% (14)
	wrong	16.7% (3)	0.0% (0)

(d) Exclusion Results for multiple-choice item B.3

Exclusion Criteria	**Percent (Count)**
Answer is "Code Does Not Compile"	0.0% (0)
Answer is "I don't know"	4.1% (2)
Diagram is not a tree	2.0% (1)
Diagram of another question	0.0% (0)

5.4 Discussion

Notional machines are often used for assessment (i.e. as assessment instruments). The idea is to measure the student's knowledge about an aspect of a programming language by measuring the students' uses of a notional machine. We want to take advantage of the relationship between the notional machine and the programming language given by the soundness condition to design experiments to evaluate the effectiveness of a notional machine as an assessment instrument. These experiments consist of comparing the measurement made with the notional machine with a ground truth measurement made using another instrument. There are in principle several points of failure in this experimental design.

Discrepancy Between Notional Machine and Programming Language

It is possible that the notional machine being evaluated doesn't actually represent the aspect of the programming language under focus or does so inconsistently.

This issue is addressed by the guarantees provided by the soundness condition so the abstract representation of a sound notional machine is guaranteed to not have this issue. It is still possible that there are usability issues with its concrete representation though, which can't be guaranteed by the soundness condition.

Insufficient Training With the Notional Machine

It is possible that the students didn't learn how to map from the programming language into the notional machine or how to operate with the notional machine.

Indeed, our pilot study suffered from this issue. The quiz was administered very early in the course and at that time the students had little contact both with the notional machine and the programming language. In the interviews, conducted at the end of the semester, some students reported that they weren't sure about the trees they had built in the quiz and that after going through the semester they understood everything better. If we want to evaluate the effectiveness of a notional machine as an assessment instrument we need to make sure the students have proper training with the notional machine so they can properly express themselves with it.

Even with more training, we don't want to include in the analysis data that comes from guessing answers. In fact, we added the "I don't know" option to the multiple-choice items precisely with that purpose in mind but we didn't have anything similar for the answers to the ExpressionTutor questions. An improvement in the experiment design would have been to require the students to provide a

score of “degree of certainty” to each answer including the ExpressionTree answer. In its simplest form, this would correspond to an “I don’t know” option also for the notional machine questions.

Noisy Ground Truth Instrument

It is also possible that the ground truth instrument is badly designed and doesn’t actually measure correctly the students’ understanding of the aspect of the programming language under focus or simply that it measures something different than what the notional machine is measuring.

In our pilot study, we’ve also experienced this problem. On top of that, issues in understanding the content knowledge of the course may have been compounded by issues in understanding the natural language used in the course. This in fact raises questions about the reliability of the use of multiple-choice questions, even if augmented with explanations, as a ground truth instrument.

An improvement in the experiment design would have been to include more interviews to complement the information provided by the multiple-choice questions. Alternatively, one could replace the multiple-choice questions with Mastery Checks [Bloom, 1968; Guskey, 2010; Wrigstad and Castegren, 2019] as a ground truth instrument, although Mastery Checks are difficult to scale.

One could also consider trying alternative designs for the ground truth instrument. For example, instead of questions that correspond to the information present in tree patterns, one could try to design questions that correspond to the information present in the whole tree. For example, one could ask the students to parenthesize all sub-expressions of an expression or to spell out all sub-expressions of an expression. These question formats could come with their own challenges, which one would have to experimentally verify. Although they could work for expression trees, they wouldn’t work for other notional machines so Mastery Checks seem a more general solution.

Even though our own instantiation of the methodology fell short in several regards, the systematicity of the process may have helped to make these problems easier to identify. Our intention with the proposed experimental design methodology is to help to avoid some of these pitfalls by making the process more systematic and helping to design experiments that follow the structure of the commutative diagram that describes the relation between the notional machine and the aspect of the programming language, the very relation on which one is relying when using a notional machine as assessment instrument.

Chapter 6

Related Work

Most of the related work was discussed through the chapters in the context where they were mostly relevant. In this chapter, we discuss additional related work and reiterate some of the key references.

The idea of simulation or representing a program by means of another program is an old one and was first studied in detail in the 1970s by Milner [1971] and Hoare [1972]. Many of the notional machines we consider illustrate a reduction, stepping, or evaluation 'aspect' of a programming language. Wadler et al. [2020] describe how to relate such reduction systems with simulation, lock-step simulation, and bisimulation. The commutative diagram describing the desired property of a notional machine appears in many places in the literature and is a basic concept in Category Theory. Closer to our application is its use as "promotion condition" [Bird, 1984].

Where computing education researchers capture program behavior through notional machines, programming language researchers instead use semantics [Krishnamurthi and Fisler, 2019]. Our work can be seen as a rather standard approach to show the correctness of one kind of semantics of (part of) a programming language, most often the operational semantics, with respect to another semantics, often a reduction semantics. An example of such an approach has been described by Clements et al. [2001], whose Elaboration Theorem describes a property that is very similar to our soundness requirement. The lack of a formal approach to showing the soundness of notional machines is also noted by Pollock et al. [2019], who developed a formal approach to specifying correct program state visualization tools, based on an executable semantics of the programming language formulated in the K framework. In our research, we study a much broader collection of notional machines than just program state visualization tools, and we apply our approach to study the soundness of notional machines.

In practice, computing educators use a diverse set of notional machines [Fincher et al., 2020]. Some notional machines form the basis of automated tools. The BlueJ IDE, which features prominently in an introductory programming textbook [Kölling and Barnes, 2017], includes a graphical user interface to visualize objects, invoke methods, and inspect object state. PythonTutor [Guo, 2013], an embeddable web-based program visualization system, is used by hundreds of thousands of users to visualize the execution of code written in Python, Java, JavaScript, and other programming languages. UUhistle [Sorva and Sirkiä, 2010], a "visual program simulation" system, takes a different approach: instead of visualizing program executions, it requires students to perform the execution steps in a constrained interactive environment. Another example of visual program simulation is the Informa clicker system [Hauswirth and Adamoli, 2013], a software clicker tool where students answer questions by constructing various kinds of visual representations of programs. When developing such widely used tools, starting from a sound notional machine is essential.

Dickson et al. [2022] discuss the issues around developing and using a notional machine in class. They note, amongst others, "that a notional machine must by definition be correct, but a student's mental model of the notional machine often is not", and that "specifying a notional machine was more difficult than we thought it would be". Our work can help in developing a notional machine and pointing out flaws in it.

Our theoretical analysis of Java is based on the Java Language Specification (JLS) [**?**] and Eclipse's Java Development Tools (JDT) [The Eclipse Foundation, 2022] compiler. Another approach would be to use a well-known formally specified subset of Java, such as Featherweight Java [Igarashi et al., 2001] or other subsets based on it. An advantage of using formally specified languages is that the constructs that are expressions in those languages are unambiguously specified. On the other hand, because these approaches aim to develop a core calculus intended to investigate some aspect of the semantics of Java, they cover a subset of the language that is as small as possible to investigate that aspect. For example, Middleweight Java [Bierman et al., 2003] is a minimal imperative core calculus for Java and Welterweight Java [Östlund and Wrigstad, 2010] is a core calculus with imperative features and concurrency used to formalize ownership. Although we also cover a subset of Java (the subset of expressions), we aim to be complete in that subset, including constructs that wouldn't be considered all together in a core calculus. There does exist a formalization of the full Java semantics using the $\mathbb{K}$ framework [Bogdănaş, 2015], however, it only covers up to Java 1.4 whereas we cover all expression constructs up to Java 11.

Chapter 7

Future Work

This research leaves open several avenues for future work. Some of them were discussed or referred to in context throughout the work. We select some of them and discuss them a bit more in depth in this chapter. The main threads are: (1) use our approach with notional machines that describe data structures (Section 7.1); (2) use mechanized proofs to design and analyze notional machines (Section 7.2); (3) implement the remaining improvements to ExpressionTutor that we have identified using our theoretical analysis (Section 7.3); (4) use sound notional machines to detect programming language misconceptions (Section 7.5).

7.1 Notional Machines for Data Structures

Notional machines are not only used to aid in the understanding of aspects of programming languages but at times they can focus specifically on a data structure[1]. The soundness of these notional machines, with respect to the data structure they focus on, is as important as the soundness of notional machines that focus on an aspect of a programming language. In principle, our description of notional machines in terms of simulation is expressible enough to describe notional machines that focus on data structures, but demonstrating this is future work.

The idea is that the bottom of the commutative diagram would have the abstract representation of a data structure (given by the pair (A_{DS}, B_{DS}), where A_{DS} and B_{DS} may be the same depending on the situation), instead of the abstract

[1] In the website `https://notionalmachines.github.io/`, at least 6 notional machines focus on data structures. For example, the notional machine "Hash Set as Hanging Folders", that describes Hash Sets.

representation of a programming language (A_{PL}, B_{PL}). A function f_{DS} would then describe essentially the evolution of the data structure over time.

Additionally, this may be a great opportunity to investigate an important and subtle issue in the traditional way of approaching the teaching of data structures and algorithms. In classic algorithms textbooks [Cormen, 2009], the idea is to reason abstractly about data structures in a way that is independent of how they are actually implemented in a given program and programming language. To that effect, algorithms are often described using pseudocode. There is a subtle issue though with this approach: there are often hidden assumptions about a certain computational model and programming language semantics when one is representing algorithms with pseudocode. Even when the assumptions are explicitly stated in natural language (for example, Cormen [2009] state their "Pseudocode conventions" in p. 20), it's in the nature of being *pseudo* that pseudocode has neither a formal definition nor an implementation. By using our approach, one must first separate the concrete representation of a notional machine from its abstract representation and from the abstract representation of that which is the focus of the notional machine, and in that process can make these assumptions explicit. This may be a particularly useful way to investigate high-level abstract treatments of data structures because with this approach we can have both theoretical rigor, reasoning in terms of soundness and proofs, and produce practical results that can be used in the classroom, via the construction of notional machines.

7.2 Proven Sound by Construction

We have designed and analyzed several notional machines in the previous chapters. The confidence we have in our assessment of the soundness (or lack of soundness) of these notional machines varies with the degree of rigor of the technique that we have employed in each case. Informal reasoning systematically guided by our construction can be very useful in spotting mistakes but gives us little confidence in the overall soundness of a given notional machine. Implementing the notional machine and testing it and its relationship with the corresponding programming language increases our confidence. We can raise our confidence even further by writing this implementation in a language that allows us to equationally reason about its soundness or by writing this implementation using our sound-by-construction design methodology. But even then we can not be sure there are no mistakes in the implementation or in these manual proofs.

The next level would be to mechanize these proofs, for example writing these

implementations with Agda or Coq. With Agda, we could use agda2hs [Cockx et al., 2022] to reuse part of the artefacts we have produced in Haskell. agda2hs can translate a subset of Agda into human-readable Haskell. The idea is to implement the components of the commutative diagram in that subset and use full Agda to prove the soundness condition. The translation to Haskell erases dependent types and proofs and the resulting code can interact with the rest of the Haskell artifact.

Another possibility, which would be even closer to the artefacts we have produced, would be to use LiquidHaskell [Rondon et al., 2008; Vazou et al., 2013]. LiquidHaskell started as a way to embed refinement types into Haskell, but has evolved into a full-fledged theorem prover for Haskell programs [Vazou et al., 2017]. Refinement types decorate programs with SMT-decidable predicates used to verify various safety and correctness properties, such as array bounds checking. LiquidHaskell now can be used to write not only refinement types but also general theorems about the program. The proofs of these theorems can be written in an equational-reasoning style directly inside the program [Vazou et al., 2018], or even be automatically generated by LiquidHaskell.

7.3 Expression Tutor Improvements

In Chapter 4, we used our theoretical approach in practice to analyze and identify improvements to ExpressionTutor. Many of these improvements have been implemented, but others remain.

In Section 4.3.3, we have identified that the information in the activity input is insufficient to determine the correct answer. We have proposed a solution to this problem that consisted of adding to the activities more information about the cold context of the expression, which could be not only the code surrounding the expression but also references to external documentation about the code. The interface currently doesn't support that but a sketch of an improved interface is shown in Figure 4.6.

As we have discussed in the Chapter, parsing and typing may fail so notional machines that focus on these aspects should be able to express that. In ExpressionTutor, the interface of the parsing activity and the typing activity allow for the solution to be "Code does not compile", but it is not yet possible to express whether the problem is a parse error or a type error and what exactly is the problem and where it happens in the tree. These improvements are also being considered for the platform.

7.3.1 Generalized Automatic Assessment

In ExpressionTutor, we have used tree-edit distance as the basis for the comparison between the student submissions and the correct answer, which is the basis for the automated feedback to students and instructors. But in the same way the commutative diagram that describes a sound notional machine is general for any notional machine, it would be valuable to use a general approach to compare students' submissions with the correct answers for any notional machine.

We can frame this problem as a datatype-generic algorithm that, for any algebraic datatype T, computes the difference between two values of type T. The first approach to this problem was proposed by Lempsink et al. [2009]. It also leverages tree-edit distance but in the meta level, operating on the trees formed by values of algebraic datatypes. Recently, more efficient approaches have been proposed by Miraldo and Swierstra [2019] and Erdweg et al. [2021].

7.4 Experiment to Evaluate ExpressionTutor

In Chapter 5, we presented a methodology to design experiments to evaluate the effectiveness of notional machines as assessment instruments. We showed how this methodology can be used by applying it to the design of a pilot study to evaluate ExpressionTutor. In this pilot study, we identified various shortcomings that we can use to inform the design of a larger-scale experiment to evaluate ExpressionTutor as an assessment instrument.

Some of these shortcomings are related to the use of multiple-choice questions and the accompanying explanations as source of ground truth information about the students' knowledge. We may be able to mitigate some of these issues by improving the process of classification of explanations. Even with the effort we described in systematizing it, the classification of each explanation still depends on the interpretation of the person doing the classification. To improve the process, we could use multiple independent classifiers and incorporate into the results an inter-rater agreement score.

Other issues range from improvements to the ExpressionTutor activity page, to students' insufficient training with the notional machine, and even the design of the question stems and their relationship with the notional machine question. The discussion Section 5.4 provides a detailed description of these and other insights gathered during the pilot study that can be used to improve the design of a larger-scale study.

7.5 Misconception Detection

Misconceptions are an important topic of research in computer science education. In fact, the paper "Identifying student misconceptions of programming", by Kaczmarczyk et al. [2010], is the top-ranked paper in the SIGCSE "Top Ten Symposium Papers of All Time Award"[2]. However, many studies either use the word *misconception* without an explicit definition (e.g., Détienne [1997]; Holland et al. [1997]; Hristova et al. [2003]; Ragonis and Ben-Ari [2005]), or use a definition that is too broad (e.g., by Smith III et al. [1994], in the context of science and mathematics education: "student conceptions that produce a systematic pattern of errors"; or the one by Sorva [2013]: "understandings that are deficient or inadequate for many practical programming contexts").

In previous work [Chiodini et al., 2021], we have proposed a more focused definition of *programming language misconceptions*: "A programming language misconception is a statement that can be disproved by reasoning entirely based on the syntax and/or semantics of a programming language". The definition excludes strategic knowledge (knowledge about the programming *process*) and focuses on syntactic and semantic knowledge (the knowledge captured in the specification of a programming language). Notice that in the same way a notional machine can only be sound (or not) with respect to a specific programming language, a given programming language misconception is also tied to a specific programming language. By tying it to the syntax and semantics of a programming language, which is completely defined by its specification (be it formal or not) or by its implementation, one can decide with some certainty whether a student's statement is or is not a programming language misconception.

Nevertheless, we are still dependent on the student's ability to formulate such a statement or on an instructor's means to elicit this information. That's where notional machines may be valuable. A notional machine that faithfully represents a given aspect of the programming language under its focus may be the means for a student to express the way they believe this aspect works. In that way, a misconception can be seen as some wrong f'_{PL}, one who's behavior diverges from f_{PL} in some particular way. The assumption is that this wrong f'_{PL} corresponds to a wrong f'_{NM}. If we can identify the wrong f'_{NM} by the wrong value they produce, which may be the case especially for well-known misconceptions, we could then devise a misconception detector in the form of queries (or patterns) on those values in a way somewhat similar to what was described in Section 5.3.1. In the paper [Chiodini et al., 2021], we also present a curated inventory of programming

[2] `https://www.acm.org/media-center/2019/march/sigcse-top-10-papers`

language misconceptions, which may be a good starting set of misconceptions to include in such a detector. The inventory is kept up to date on a website[3]. Currently, the website also hints at the connection with notional machines when it classifies some programming language misconceptions as "expressible in" a notional machine[4]. The paper doesn't define what makes a misconception expressible in a notional machine. A definition in terms of a sound notional machine not only seems adequate but may coincide with the detectability of the misconception by the notional machine sketched here.

Another indication of the connection between this research and programming language misconceptions comes from our analysis of previous years' exam questions about expression trees, which we described in Section 4.4.2. We identified that 12% of wrong nodes were classified as **EvaledExp**. This category of wrong nodes corresponds exactly to the misconception InlineCallInExpressionTree[5]. described in the website.

[3] `https://progmiscon.org/`

[4] These misconceptions were previously referred to as being "About notional machine", as described in the paper.

[5] `https://progmiscon.org/misconceptions/Java/InlineCallInExpressionTree/`

Chapter 8

Conclusion

A notional machine is a pedagogic device to assist the understanding of some aspect of programs or programming under focus by the notional machine. They are popular in computer science education, commonly used both by instructors in their teaching practice as well as by researchers. In spite of their popularity, there exists currently no consideration of their soundness, which we can think of informally as a form of correctness or consistency of the notional machine with respect to the aspect of programs under focus by the notional machine. There is in fact no definition of what should be this relationship between a notional machine and the aspect under its focus.

In this research, we started by formally defining a notion of soundness for notional machines. The definition is based on the idea of simulation, widely used in many areas of computer science, from the analysis of state-transition systems and programming languages to proofs of correctness of data representations. For a formal definition, we need a formalization of (1) the notional machine, (2) the aspect of the programming language under focus by the notional machine, and (3) the relationship between them. To formalize (1) a notional machine, we distinguish between the concrete representation of the notional machine (typically visual) and its abstract representation, which we can make formal statements about. This distinction is akin to the distinction between the concrete and the abstract syntaxes of a programming language. The description of the notional machine also includes an operation on this abstract representation. As a formalization of (2) the aspect of the programming language under focus by the notional machine, we used well-known formalisms, such as operational semantics to describe the evaluation of programs, although other formalisms could be used. Soundness is demonstrated by the soundness condition, which formally describes (3) the relationship between the notional machine and the aspect of the

programming language under its focus. The soundness condition can be visually represented as a commutative diagram that relates the notional machine and the programming language and essentially guarantees that the notional machine is consistent with the aspect of the programming language under its focus.

The soundness condition and the corresponding commutative diagram were then used as a general framework to reason about notional machines and, as such, used as the basis to address other challenges related to notional machines.

One of these challenges is the design of notional machines. We have shown how we can derive from the definition of soundness two similar methods to design notional machines that are sound by construction. We call the resulting notional machines isomorphic and monomorphic notional machines, due to the nature of their relationship with the aspect under its focus.

Similarly, we can use the commutative diagram to analyze existing notional machines with respect to soundness, identifying issues and opportunities for improvement. In these analyses, we showed that one can benefit from reasoning in terms of soundness even without fully formalizing and writing proofs about the notional machine. For example, using just property-based testing, we were able to identify a subtle issue in Alligator Eggs (a notional machine focused on the evaluation of lambda calculus programs), even though the notional machine has a very simple, clear, and seemingly correct definition.

As we progressed in our work, we have instantiated the commutative diagram that represents the soundness condition many times, using different notional machines, programming languages (from small languages like the untyped lambda calculus to large languages like Java), aspects of those programming languages (from static aspects like parsing and type-checking to dynamic aspects like reduction and references), and degrees of formalism (from formal to informal). Table 8.1 summarizes these instantiations showing each of these dimensions.

In a bigger example with a bigger language (Java), we show how we used the same reasoning framework to improve the design of ExpressionTutor, a family of notional machines centered on expressions. We begin by arguing for the importance of focusing on expressions even in languages that are not predominantly functional. We then continue by establishing a concise and complete description of the subset of Java that contains only its expression constructs. We used this subset of Java in two instantiations of the commutative diagram, one focused on type-checking and the other on parsing. To further ground the design of ExpressionTutor, we also analyzed expression trees drawn by students in paper-based exams. An important insight that comes from the use of the commutative diagram in the development of ExpressionTutor is that, in general, the implementation of the components of the commutative diagram that defines

Table 8.1. Instantiations of the commutative diagram in Figure 2.2 that represents the soundness condition for notional machines. For each instantiation, we show the different dimensions we explored.

<table>
<tr><th>Level of formalism</th><th>Notional Machine</th><th>Programming Language</th><th>Focus</th></tr>
<tr><td rowspan="4">Equational reasoning proof</td><td>ExpTree</td><td rowspan="2">UntypedLambda</td><td rowspan="2">Reduction</td></tr>
<tr><td rowspan="2">ExpTutorDiagram</td></tr>
<tr><td>TypedArith</td><td>Typing</td></tr>
<tr><td>TAPLMemoryDiagram</td><td>TypedLambdaRef</td><td>References</td></tr>
<tr><td rowspan="2">Property-based testing</td><td>Reduct</td><td rowspan="2">UntypedLambda</td><td rowspan="2">Reduction</td></tr>
<tr><td>Alligator</td></tr>
<tr><td rowspan="2">Unit testing</td><td rowspan="2">ExpTutorDiagram</td><td rowspan="2">Java</td><td>Parsing</td></tr>
<tr><td>Typing</td></tr>
</table>

a notional machine yields the key components of an educational tool for that notional machine that can generate activities for students to practice the aspect under focus and automatically assess the students' solutions.

Finally, we used the soundness condition once again, but this time as the blueprint for the development of a methodology for the design of experiments that can evaluate the effectiveness of notional machines as assessment instruments. We demonstrated the methodology by designing a pilot study to evaluate the effectiveness of ExpressionTutor as an assessment instrument. We then analyzed the results of the study and discussed issues that could explain some of the unexpected results.

The main issue to keep in mind is that the effectiveness of a notional machine as an assessment instrument depends on the student properly learning how the aspect of the programming language is mapped into the notional machine space, which means a sufficiently precise practical understanding of the abstraction functions (the mapping between code and notional machine) and the mechanics of the notional machine. Learning this mapping is not cheap. It requires practice and, in a way, it equates to essentially learning a whole other language (the notional machine language) and a correspondence between these languages, although the notional machine language is arguably simpler, given it's focused on one or a small set of concepts. This of course raises an important question: is it really worth it? Wouldn't it be better to simply learn the programming language under focus? This is an important question but it's a question about the effectiveness of notional machines as a teaching instrument, something that

we have not addressed in this research so we can only speculate. Let's consider, for example, how would an instructor explain references and mutation, and how would they assess if a student learned it. Using just natural language and examples of program behavior to really understand concepts is challenging and the more challenging the more complex is the relationship between the aspect of the programming language (the concept) under focus and any noticeable program behavior. An alternative would be to express those ideas in a more precise formal or semi-formal language, for example, by defining the language being taught using another language previously known to the students. Another known alternative is to grow the language, defining new constructs in terms of previously defined constructs of the same language, something that comes with its own challenges.

The hope is that notional machines can be an alternative that is both simpler and have the necessary precision. If we can trust that they are consistent with the aspect of the programming language under focus, and they are presented to the students in a way that is sufficiently precise, at the discretion of the instructor, then even with a high cost, they can be a good investment. Another advantage of using notional machines is that they are focused purely on concepts, avoiding any specific programming language and one could say that's exactly what we want from students: we want them to learn general concepts and not details specific to a given language. On top of that, by focusing on concepts, notional machines are hopefully more reusable and this reusability could help to amortize its cost. Investigating the use of the same notional machine across different courses and different programming languages would be an interesting direction to explore.

Ultimately, we hope that a more principled approach to reasoning about notional machines, such as the one we propose here, can contribute to higher quality notional machines and ultimately to an improvement of the effectiveness of teaching and assessing students about programming and programming languages.

Appendix A

Programming Language Definitions

The languages used in Chapters 2 and 3 are defined in this appendix using operational semantics. This presentation mostly follows the book “Types and Programming Language” by Pierce [2002] with minor changes. In particular, italics are used for metavariables and the axioms in the reduction (evaluation) rules and typing rules are shown with explicitly empty premises.

A.1 UntypedLambda

Figure A.1 shows the syntax and evaluation rules for the untyped lambda calculus by Church [1936, 1941], that we have referred to as UNTYPEDLAMBDA. The presentation here is taken from Pierce [2002]. The book contains a good explanation of the pitfalls of the substitution operation in the Rule E-APPABS, which was the source of the problem found in ALLIGATOR (Section 3.2). This language is used in Sections 2.1, 2.3, 3.1, and 3.2 .

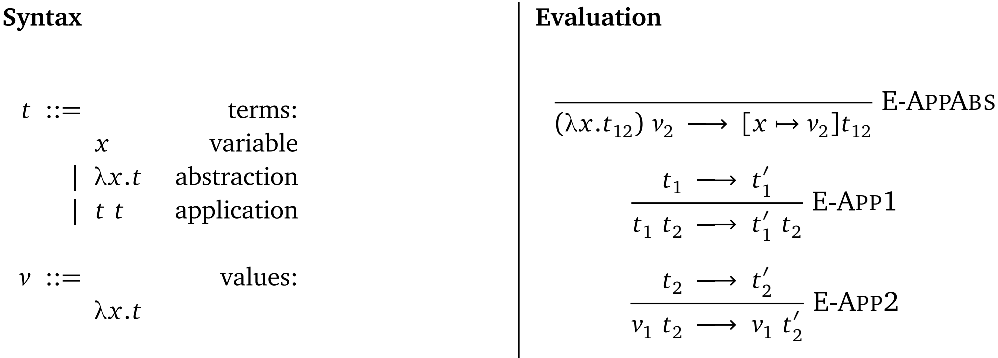


Figure A.1. The untyped lambda calculus (UntypedLambda).

A.2 TypedArith

We define here the language TYPEDARITH, used in Section 2.5. Figure A.2 shows its syntax and evaluation (reduction) rules and Figure A.3 shows its typing rules. The presentation here is taken from Pierce [2002], but here all the rules are shown together. The appeal of using this language to present a notional machine focused on the types is its simplicity. Terms don't require type annotations and the typing rules don't require a type environment. In fact, Pierce uses it as the simplest example of a typed language when introducing type safety.

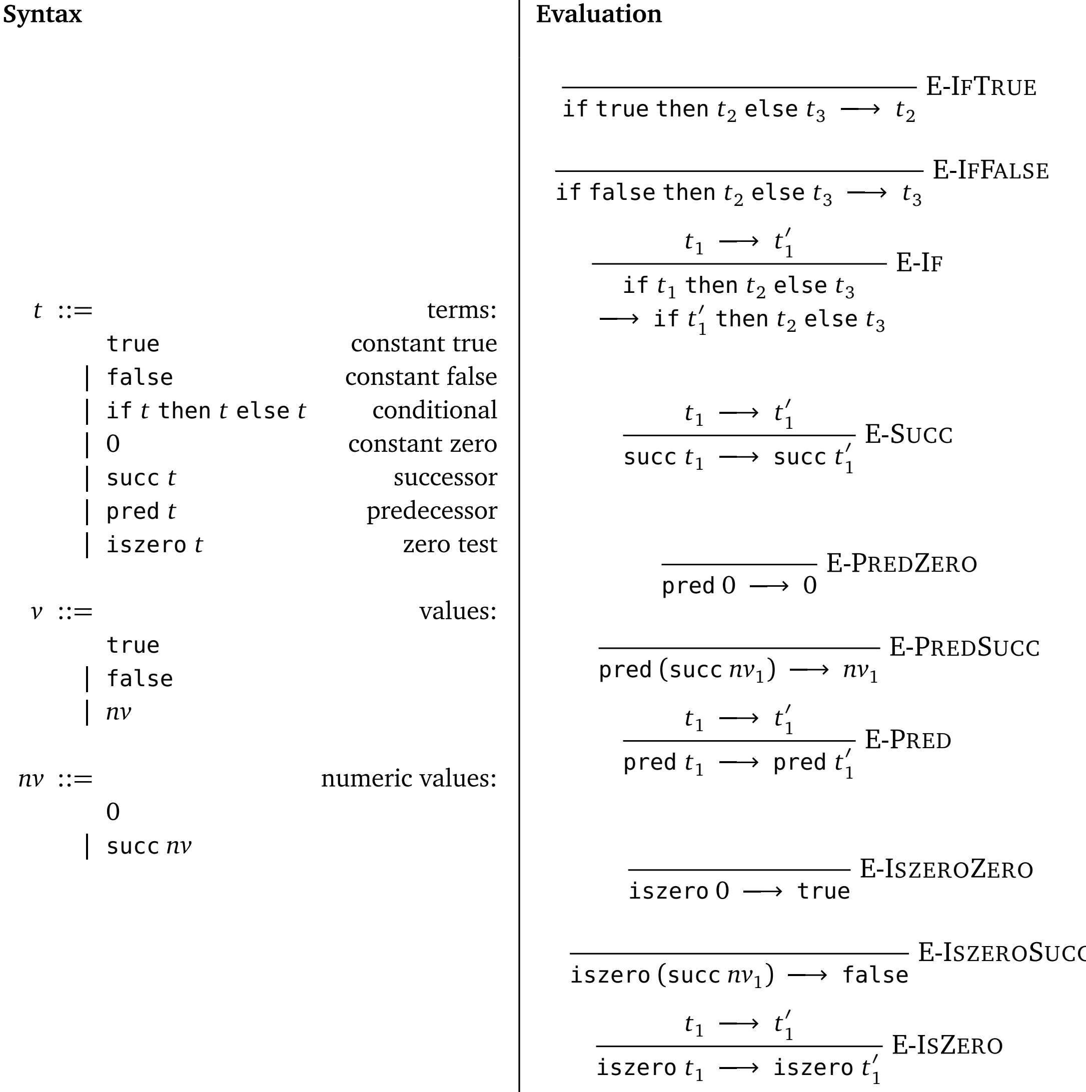


Figure A.2. Syntax and reduction rules of the TypedArith language.

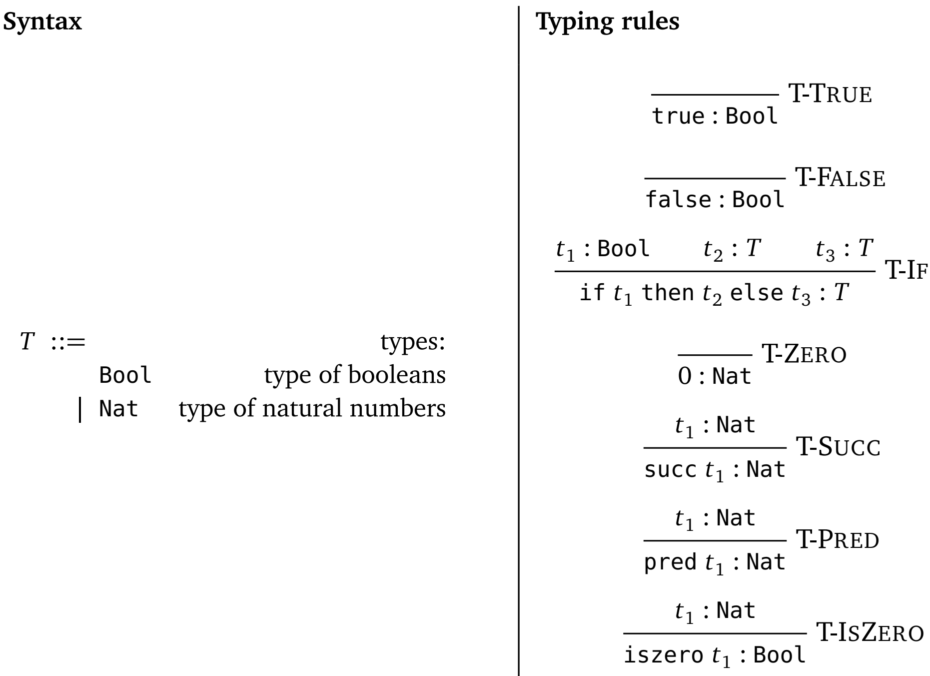


Figure A.3. Syntax of types and typing rules of the TypedArith language.

A.3 TypedLambdaRef

In Section 2.4, we showed the language TypedLambdaRef, used to design a notional machine that focuses on references. This language is composed of the simply-typed lambda calculus, the TypedArith language, tuples, the `Unit` type, sequencing, and references. The simply-typed lambda calculus and each of these extensions is presented by Pierce [2002] before introducing references in Chapter 13, which contains the memory diagram notation we refer to in Section 2.4. Our goal is again simplicity and this is the simplest language we need for the examples in the book that use the diagram. Figure A.4 shows its syntax and evaluation (reduction) rules. We show only the reduction rules for sequencing, references, and tuples because the rules for the rest of the language would be similar to what we showed before, except for the store then needs to be threaded through all the rules. In fact, the notation here is denser than the previous languages because the store, which is only manipulated in rules E-RefV, E-DerefLoc, and E-Assign, needs to be carried over through all the other rules. Although that is a typed language, we don't present its typing rules because the notional machine in Section 2.4 is focused only on its runtime behavior and not its types.

Syntax

t ::=		terms:
	x	variable
|	$\lambda x : T.t$	abstraction
|	$t\ t$	application
|	`true`	boolean true
|	`false`	boolean false
|	`0`	zero
|	`succ` t	successor
|	`pred` t	predecessor
|	`iszero` t	zero test
|	`unit`	unit constant
|	$t;\ t$	sequence
|	`ref` t	reference creation
|	`!`t	dereference
|	$t\ := t$	assignment
|	l	location
|	$\{t_i{}^{i\in 1..n}\}$	tuple
|	$t.i$	projection
v ::=		values:
	$\lambda x : T.t$	
|	`true`	
|	`false`	
|	`0`	
|	`succ` v	
|	`unit`	
|	l	
|	$\{v_i{}^{i\in 1..n}\}$	
T ::=		types:
	$T \to T$	function type
|	`Bool`	boolean type
|	`Nat`	natural number type
|	`Unit`	unit type
|	`Ref` T	reference type
|	$\{T_i{}^{i\in 1..n}\}$	tuple type
μ ::=		store:
	$\emptyset$	empty store
|	$\mu, l \mapsto v$	location binding

Evaluation

$$\frac{t_1|\mu \longrightarrow t_1'|\mu'}{t_1;\ t_2|\mu \longrightarrow t_1';\ t_2|\mu'} \text{ E-SEQ}$$

$$\frac{}{\texttt{unit};\ t_2|\mu \longrightarrow t_2|\mu} \text{ E-SEQNEXT}$$

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ v_1|\mu \longrightarrow l|(\mu, l \mapsto v_1)} \text{ E-REFV}$$

$$\frac{t_1|\mu \longrightarrow t_1'|\mu'}{\texttt{ref}\ t_1|\mu \longrightarrow \texttt{ref}\ t_1'|\mu'} \text{ E-REF}$$

$$\frac{\mu(l) = v}{!l|\mu \longrightarrow v|\mu} \text{ E-DEREFLOC}$$

$$\frac{t_1|\mu \longrightarrow t_1'|\mu'}{!t_1|\mu \longrightarrow !t_1'|\mu'} \text{ E-DEREF}$$

$$\frac{}{l\ := v_2|\mu \longrightarrow \texttt{unit}|[l \mapsto v_2]\mu} \text{ E-ASSIGN}$$

$$\frac{t_1|\mu \longrightarrow t_1'|\mu'}{t_1\ := t_2|\mu \longrightarrow t_1'\ := t_2|\mu'} \text{ E-ASSIGN1}$$

$$\frac{t_2|\mu \longrightarrow t_2'|\mu'}{v_1\ := t_2|\mu \longrightarrow v_1\ := t_2'|\mu'} \text{ E-ASSIGN2}$$

$$\frac{}{\{v_i{}^{i\in 1..n}\}.j|\mu \longrightarrow v_j|\mu} \text{ E-PROJTUPLE}$$

$$\frac{t_1|\mu \longrightarrow t_1'|\mu'}{t_1.i|\mu \longrightarrow t_1'.i|\mu'} \text{ E-PROJ}$$

$$\frac{t_j|\mu \longrightarrow t_j'|\mu'}{\{v_i{}^{i\in 1..j-1}, t_j, t_k{}^{k\in j+1..n}\}|\mu \longrightarrow \{v_i{}^{i\in 1..j-1}, t_j', t_k{}^{k\in j+1..n}\}|\mu'} \text{ E-TUPLE}$$

Figure A.4. TypedLambdaRef: Syntax and Evaluation

Appendix B

Exam Questions

In the analysis of students' mistakes in answers to paper-based questions about expression trees (see Section 4.4.2), we analyzed two questions from different written exams, both from the course "Programming Fundamentals 2" (PF2), a second-year bachelor course that teaches an introduction to object-oriented programming. The next two sections show the questions as they appeared in the exams. The first appeared in the midterm exam and the second in the final exam, both from the Spring semester of 2022.

B.1 Midterm Exam Question

Expression Tree and Execution Trace (12 Points)

```
public class Maker {

  public String publish(String a, String b) {
    System.out.println("pub");
    System.out.println(a);
    return "done";
  }

  public String make(String thing) {
    System.out.println("making");
    return "made " + thing;
  }

  public void run() {
    String s = publish(make("this"), make("that"));
    System.out.println("complete");
  }

}
```

(a) (7 points) For the expression to the right of the equals sign (=) in the run method, draw the expression tree (nodes, edges, **and a star on the root node**). For each node, including the root, draw its type, and draw its value.

(b) (5 points) What gets printed on the console after executing the run method?

B.2 Final Exam Question

Expression Tree (16 Points)

Given this class:

```
public class Demo {

  private static String id(String arg) {
    return arg;
  }

  public String toString() {
    return "D";
  }

  public static String run() {
    int i = 0;
    Demo[] a = new Demo[] { new Demo() };
    String s = "a[i] = " + (a==null ? "X" : id(a[i].toString())) + '+' + 0;
    return s;
  }
}
```

Draw the expression tree of the expression to the right of `String s =`.

Indicate which node is the **root** (with a star).

For each node indicate its **type**. As the type of the value `null`, write `NULL`.

For each node indicate its **value**. Use `@1`, `@2`, `@3`, ... to represent reference values (like we did in class), except for the null reference, which you represent as the `null` literal, and for strings, which you represent as a `String` literal.

Bibliography

Arawjo, I., Wang, C.-Y., Myers, A. C., Andersen, E. and Guimbretière, F. [2017]. Teaching Programming with Gamified Semantics, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ACM, Denver Colorado USA, pp. 4911–4923.

Bierman, G. M., Parkinson, M. J. and Pitts, A. M. [2003]. MJ: An imperative core calculus for Java and Java with effects, *Technical Report UCAM-CL-TR-563*, University of Cambridge, Computer Laboratory.

Bird, R. S. [1984]. The promotion and accumulation strategies in transformational programming, *ACM Transactions on Programming Languages and Systems* **6**(4): 487–504.

Bird, R. S. [1989]. Algebraic identities for program calculation, *The Computer Journal* **32**(2): 122–126.

Bloom, B. S. [1968]. Learning for Mastery. Instruction and Curriculum. Regional Education Laboratory for the Carolinas and Virginia, Topical Papers and Reprints, Number 1., *Evaluation Comment* **1**(2).

Bogdănaş, D. [2015]. *A Complete Semantics for Java*, PhD thesis, Alexandru Ioan Cuza University of Iaşi.

Chiodini, L. and Hauswirth, M. [2021]. Wrong Answers for Wrong Reasons: The Risks of Ad Hoc Instruments, *21st Koli Calling International Conference on Computing Education Research*, ACM, Joensuu Finland, pp. 1–11.

Chiodini, L., Moreno Santos, I., Gallidabino, A., Tafliovich, A., Santos, A. L. and Hauswirth, M. [2021]. A Curated Inventory of Programming Language Misconceptions, *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, pp. 380–386.

Chiodini, L., Moreno Santos, I. and Hauswirth, M. [2022]. Expressions in Java: Essential, Prevalent, Neglected?, *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E 2022, Association for Computing Machinery, New York, NY, USA, pp. 41–51.

Chiodini, L., Sorva, J. and Hauswirth, M. [2023]. Teaching Programming with Graphics: Pitfalls and a Solution, *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E 2023, Association for Computing Machinery, New York, NY, USA.

Church, A. [1936]. An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics* **58**(2): 345–363.

Church, A. [1941]. *The Calculi of Lambda-Conversion*, number 6, Princeton University Press.

Clements, J., Flatt, M. and Felleisen, M. [2001]. Modeling an Algebraic Stepper, *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, Springer-Verlag, Berlin, Heidelberg, pp. 320–334.

Cockx, J., Melkonian, O., Escot, L., Chapman, J. and Norell, U. [2022]. Reasonable Agda is correct Haskell: Writing verified Haskell using agda2hs, *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, Haskell 2022, Association for Computing Machinery, New York, NY, USA, pp. 108–122.

Cormen, T. H. (ed.) [2009]. *Introduction to Algorithms*, 3rd ed edn, MIT Press, Cambridge, Mass.

Dalton, A. R. and Kreahling, W. [2010]. Automated construction of memory diagrams for program comprehension, *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, Association for Computing Machinery, New York, NY, USA, pp. 1–6.

de Bruijn, N. G. [1972]. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indagationes Mathematicae (Proceedings)* **75**(5): 381–392.

Détienne, F. [1997]. Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams, *Interacting with Computers* **9**(1): 47–72.

Dickson, P. E., Richards, T. and Becker, B. A. [2022]. Experiences Implementing and Utilizing a Notional Machine in the Classroom, *Proceedings of the 53rd ACM Technical Symposium V.1 on Computer Science Education*, SIGCSE 2022, Association for Computing Machinery, New York, NY, USA, pp. 850–856.

Dragon, T. and Dickson, P. E. [2016]. Memory Diagrams: A Consistant Approach Across Concepts and Languages, *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, Association for Computing Machinery, New York, NY, USA, pp. 546–551.

Du Boulay, B. [1986]. Some Difficulties of Learning to Program, *Journal of Educational Computing Research* **2**(1): 57–73.

Du Boulay, B. and O'Shea, T. [1976]. How to Work the LOGO Machine, *Technical Report 4*, Department of Artificial Intelligence, University of Edinburgh.

Erdweg, S., Szabó, T. and Pacak, A. [2021]. Concise, type-safe, and efficient structural diffing, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, Association for Computing Machinery, New York, NY, USA, pp. 406–419.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. [2018]. *How to Design Programs, Second Edition: An Introduction to Programming and Computing*, MIT Press.

Feynman, R. [1985]. *Surely You're Joking, Mr. Feynman!*, W. W. Norton.

Fincher, S., Jeuring, J., Miller, C. S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J. L. and Petersen, A. [2020]. Notional Machines in Computing Education: The Education of Attention, *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, pp. 21–50.

Gibbons, J. [2002]. Calculating Functional Programs, *in* R. Backhouse, R. Crole and J. Gibbons (eds), *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction: International Summer School and Workshop Oxford, UK, April 10–14, 2000 Revised Lectures*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 151–203.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C. and Zilles, C. [2008]. Identifying Important and Difficult Concepts in Introductory

Computing Courses Using a Delphi Process, *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, ACM, New York, NY, USA, pp. 256–260.

Gray, K. E. and Flatt, M. [2003]. ProfessorJ: A Gradual Introduction to Java Through Language Levels, *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, ACM, New York, NY, USA, pp. 170–177.

Guo, P. J. [2013]. Online python tutor: Embeddable web-based program visualization for cs education, *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*, ACM Press, Denver, Colorado, USA, p. 579.

Guskey, T. [2010]. Lessons of Mastery Learning, *Educational leadership: journal of the Department of Supervision and Curriculum Development, N.E.A* **68**: 52–57.

Hauswirth, M. and Adamoli, A. [2013]. Teaching Java programming with the Informa clicker system, *Science of Computer Programming* **78**(5): 499–520.

Hoare, C. A. [1972]. Proof of correctness of data representations, *Acta Informatica* **1**(4): 271–281.

Holland, S., Griffiths, R. and Woodman, M. [1997]. Avoiding Object Misconceptions, *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, ACM, New York, NY, USA, pp. 131–134.

Holliday, M. A. and Luginbuhl, D. [2004]. CS1 assessment using memory diagrams, *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, Association for Computing Machinery, New York, NY, USA, pp. 200–204.

Hristova, M., Misra, A., Rutter, M. and Mercuri, R. [2003]. Identifying and correcting Java programming errors for introductory computer science students, *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, Association for Computing Machinery, New York, NY, USA, pp. 153–156.

Igarashi, A., Pierce, B. C. and Wadler, P. [2001]. Featherweight Java: A minimal core calculus for Java and GJ, *ACM Transactions on Programming Languages and Systems* **23**(3): 396–450.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P. and Herman, G. L. [2010]. Identifying student misconceptions of programming, *Proceedings of the 41st ACM Technical Symposium on Computer Science Education - SIGCSE '10*, ACM Press, Milwaukee, Wisconsin, USA, p. 107.

Kölling, M. and Barnes, D. [2017]. *Objects First With Java: A Practical Introduction Using BlueJ*, 6th edn, Pearson.

Krishnamurthi, S. and Fisler, K. [2019]. Programming Paradigms and Beyond, *in* A. V. Robins and S. A. Fincher (eds), *The Cambridge Handbook of Computing Education Research*, Cambridge Handbooks in Psychology, Cambridge University Press, Cambridge, pp. 377–413.

Lempsink, E., Leather, S. and Löh, A. [2009]. Type-safe diff for families of datatypes, *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, WGP '09, Association for Computing Machinery, New York, NY, USA, pp. 61–72.

Liquori, L. and Spiwack, A. [2008]. FeatherTrait: A modest extension of Featherweight Java, *ACM Transactions on Programming Languages and Systems* **30**(2): 11:1–11:32.

Marceau, G., Fisler, K. and Krishnamurthi, S. [2011]. Do values grow on trees?: Expression integrity in functional programming, *Proceedings of the Seventh International Workshop on Computing Education Research - ICER '11*, ACM Press, Providence, Rhode Island, USA, p. 39.

Matthias Felleisen, Matthew Flatt, Robert Bruce Findler, Kathryn E. Gray, Shriram Krishnamurthi and Viera K. Proulx [2012]. *How to Design Classes: Data: Structure and Organization*.

Milner, R. [1971]. An algebraic definition of simulation between programs, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI'71, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 481–489.

Miraldo, V. C. and Swierstra, W. [2019]. An efficient algorithm for type-safe structural diffing, *Proceedings of the ACM on Programming Languages* **3**(ICFP): 113:1–113:29.

Östlund, J. and Wrigstad, T. [2010]. Welterweight Java, *in* J. Vitek (ed.), *Objects, Models, Components, Patterns*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 97–116.

Pickering, M., Érdi, G., Peyton Jones, S. and Eisenberg, R. A. [2016]. Pattern Synonyms, *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, ACM, New York, NY, USA, pp. 80–91.

Pierce, B. C. [2002]. *Types and Programming Languages*, MIT Press, Cambridge, Mass.

Pollock, J., Roesch, J., Woos, D. and Tatlock, Z. [2019]. Theia: Automatically generating correct program state visualizations, *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, SPLASH-E 2019, Association for Computing Machinery, New York, NY, USA, pp. 46–56.

Ragonis, N. and Ben-Ari, M. [2005]. A long-term investigation of the comprehension of OOP concepts by novices, *Computer Science Education* **15**(3): 203–221.

Robins, A., Rountree, J. and Rountree, N. [2003]. Learning and Teaching Programming: A Review and Discussion, *Computer Science Education* **13**(2): 137–172.

Rondon, P. M., Kawaguci, M. and Jhala, R. [2008]. Liquid types, *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, Association for Computing Machinery, New York, NY, USA, pp. 159–169.

Santos, I. M., Hauswirth, M. and Nystrom, N. [2019]. Experiences in bridging from functional to object-oriented programming, *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E - SPLASH-E 2019*, ACM Press, Athens, Greece, pp. 36–40.

Smith III, J. P., diSessa, A. A. and Roschelle, J. [1994]. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition, *Journal of the Learning Sciences* **3**(2): 115–163.

Sorva, J. [2013]. Notional machines and introductory programming education, *ACM Transactions on Computing Education* **13**(2): 1–31.

Sorva, J., Lönnberg, J. and Malmi, L. [2013]. Students' ways of experiencing visual program simulation, *Computer Science Education* **23**(3): 207–238.

Sorva, J. and Sirkiä, T. [2010]. UUhistle: A software tool for visual program simulation, *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, ACM Press, Berlin, Germany, pp. 49–54.

The Eclipse Foundation, J. [2022]. JDT Core Component, https://www.eclipse.org/jdt/core/index.php.

Vazou, N., Breitner, J., Kunkel, R., Van Horn, D. and Hutton, G. [2018]. Theorem proving for all: Equational reasoning in liquid Haskell (functional pearl), *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, Association for Computing Machinery, New York, NY, USA, pp. 132–144.

Vazou, N., Rondon, P. M. and Jhala, R. [2013]. Abstract Refinement Types, *in* M. Felleisen and P. Gardner (eds), *Programming Languages and Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 209–228.

Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R. G., Newton, R. R., Wadler, P. and Jhala, R. [2017]. Refinement reflection: Complete verification with SMT, *Proceedings of the ACM on Programming Languages* **2**(POPL): 53:1–53:31.

Wadler, P., Kokke, W. and Siek, J. G. [2020]. *Programming Language Foundations in Agda*.

Wrigstad, T. and Castegren, E. [2019]. Mastery Learning-Like Teaching with Achievements, *CoRR* **abs/1906.03510**.

Yates, R. and Yorgey, B. A. [2015]. Diagrams: A functional EDSL for vector graphics, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, FARM 2015, Association for Computing Machinery, New York, NY, USA, pp. 4–5.

Yorgey, B. A. [2012]. Monoids: Theme and Variations (Functional Pearl), *ACM SIGPLAN Notices* **47**(12): 105–116.

Zhang, K. and Shasha, D. [1989]. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems, *SIAM J. Comput.* **18**: 1245–1262.