Programming Languages

Computer scientists often need to learn new programming languages quickly. The best way to prepare is to understand the foundational principles that underlie even the most complicated industrial languages.

This text for an undergraduate programming-languages course distills great languages and their design principles down to easy-to-learn "bridge" languages implemented by interpreters whose key parts are explained in the text. The book goes deep into the roots of both functional and object-oriented programming, and it shows how types and modules, including generics/polymorphism, contribute to effective programming.

The book is not just about programming languages; it is also about programming. Through concepts, examples, and more than 300 practice exercises that exploit the interpreters, students learn not only what programming-language features are common but also how to do things with them. Substantial implementation projects include Milner's type inference, both copying and mark-and-sweep garbage collection, and arithmetic on arbitrary-precision integers.

Norman Ramsey is Associate Professor of Computer Science at Tufts University. Since earning his PhD at Princeton, he has worked in industry and has taught programming languages, advanced functional programming, programming language implementation, and technical writing at Purdue, the University of Virginia, and Harvard as well as Tufts. He has received Tufts's Lerman-Neubauer Prize, awarded annually to one outstanding undergraduate teacher. He has also been a Hertz Fellow and an Alfred P. Sloan Research Fellow. His implementation credits include a code generator for the Standard ML of New Jersey compiler and another for the Glasgow Haskell Compiler.

PROGRAMMING LANGUAGES
Build, Prove, and Compare

Norman Ramsey
Tufts University, Massachusetts

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/highereducation/isbn/9781107180185
DOI: 10.1017/9781316841396

First published 2023

A catalogue record for this publication is available from the British Library.

ISBN 978-1-107-18018-5 Hardback

Additional resources for this publication at www.cambridge.org/ramsey

To Cory, who also knows joy in creation

CONTENTS

Preface

This textbook, suitable for an undergraduate or master's-level course in programming languages, is about great language-design ideas, how to describe them precisely, and how to use them effectively. The ideas revolve around functions, types, modules, and objects. They are described using formal semantics and type theory, and their use is illustrated through programming examples and exercises.

The ideas, descriptive techniques, and examples are conveyed by means of *bridge languages*. A bridge language models a real programming language, but it is small enough to describe formally and to learn in a week or two, yet big enough to write interesting programs in. The bridge languages in this book model Algol, Scheme, ML, CLU, and Smalltalk, and they are related to many modern descendants, including C, C++, OCaml, Haskell, Java, JavaScript, Python, Ruby, and Rust.

Each bridge language is supported by an interpreter, which runs all the examples and supports programming exercises. The interpreters, which are presented in depth in an online Supplement, are carefully crafted and documented. They can be used not only for exercises but also to implement students' own language-design ideas.

The book develops these concepts:

- Abstract syntax and operational semantics
- Definitional interpreters
- Algebraic laws and equational reasoning
- Garbage collection
- Symbolic computing and functional programming
- Parametric polymorphism
- Monomorphic and polymorphic type systems
- Type inference
- Algebraic data types and pattern matching
- Data abstraction using abstract types and modules
- Data abstraction using objects and classes

The concepts are supported by the bridge languages as shown in the Introduction (Table I.2, page 5), which also explains each bridge language in greater detail (pages 3 to 7).

The book calls for skills in both programming and proof:

- As prerequisites, learners should have the first-year, two-semester programming sequence, including data structures, plus discrete mathematics.

- To extend and modify the implementations in Chapters 1 to 4, a learner needs to be able to read and modify C code; the necessary skills have to be learned elsewhere. C is used because it is the simplest way to express programs that work extensively with pointers and memory, which is the topic of Chapter 4.

To extend and modify the implementations in Chapters 5 to 10, a learner needs to be able to read and modify Standard ML code; the necessary skills are developed in Chapters 2, 5, and 8. Standard ML is used because it is a simple, powerful language that is ideally suited to writing interpreters.

- To prove the simpler theorems, a learner needs to be able to substitute equals for equals and to fill in templates of logical reasoning. To prove the more interesting theorems, a learner needs to be able to write a proof by induction.

Designing a course to use this book

Some books capture a single course, and when using such a book, your only choice is to start at the beginning and go as far as you can. But in programming languages, instructors have many good choices, and a book shouldn't make them all for you. This book is designed so you can choose what to teach and what to emphasize while retaining a coherent point of view: how programming languages can be used effectively in practice. If you're relatively new to teaching programming languages and are not sure what to choose, you can't go wrong with a course on functions, types, and objects (Chapters 1, 2, 6, 7, and 10). If you have more experience, consider the ideas below.

Programming Languages: Build, Prove, and Compare gives you interesting, powerful programming languages that share a common syntax, a common theoretical framework, and a common implementation framework. These frameworks support programming practice in the bridge languages, implementation and extension of the bridge languages, and formal reasoning about the bridge languages. The design of your course will depend on how you wish to balance these elements.

- To unlock the full potential of the subject, combine programming practice with theoretical study and work on interpreters. If your students have only two semesters of programming experience and no functional programming, you can focus on the core foundations in Chapters 1 to 3: operational semantics, functional programming, and control operators. You can supplement that work with one of two foundational tracks: If your students are comfortable with C and pointers, they can implement continuation primitives in μScheme+, and they can implement garbage collectors. Or if they can make a transition from μScheme to Standard ML, with help from Chapter 8, they can implement type checkers and possibly type inference.

  If your students have an additional semester of programming experience or if they have already been exposed to functional programming, your course can advance into types, modules, and objects. When I teach a course like this, it begins with four homework assignments that span an introduction to the framework, operational semantics, recursive functions, and higher-order functions. After completing these assignments, my students learn Standard ML, in which they implement first a type checker, then type inference. This schedule leaves a week for programming with modules and data abstraction, a couple of weeks for Smalltalk, and a bit of time for the lambda calculus.

  A colleague whose students are similarly experienced begins with Impcore and μScheme, transitions to Standard ML to work on type systems and type inference, then returns to the bridge languages to explore μSmalltalk, μProlog, and garbage collection.

  If your students have seen interpreters and are comfortable with proof by induction, your course can move much more quickly through the foundational material, creating room for other topics. When I taught a course like this, it explored everything in my other class, then added garbage collection, denotational semantics, and logic programming.

- A second design strategy tilts your class toward programming practice, either de-emphasizing or eliminating theory. To introduce programming practice in diverse languages, *Build, Prove, and Compare* occupies a sweet spot between two extremes. One extreme "covers" N languages in N weeks. This extreme is great for exposure, but not for depth—when students must work with real implementations of real languages, a week or even two may be enough to motivate them, but it's not enough to build proficiency.

  The other extreme goes into full languages narrowly but deeply. Students typically use a couple of popular languages, and overheads are high: each language has its own implementation conventions, and students must manage the gratuitous details and differences that popular languages make inevitable.

  Build, Prove, and Compare offers both breadth and depth, without the overhead. If you want to focus on programming practice, you can aim for "four languages in ten weeks": μScheme, μML, Molecule, and μSmalltalk. You can bring your students up to speed on the common syntactic, semantic, and implementation frameworks using Impcore, and that knowledge will support them through to the next four languages. If you have a couple of extra weeks, you can deepen your students' experience by having them work with the interpreters.

- A third design strategy tilts your class toward applied theory. *Build, Prove, and Compare* is not suitable for a class in pure theory—the bridge languages are too big, the reasoning is informal, and the classic results are missing. But it is suitable for a course that is primarily about using formal notation to explain precisely what is going on in whole programming languages, reinforced by experience implementing that notation. Your students can do metatheory with Impcore, Typed Impcore, Typed μScheme, and nano-ML; equational reasoning with μScheme; and type systems with Typed Impcore, Typed μScheme, nano-ML, μML, and Molecule. They can compare how universally quantified types are used in three different designs (Typed μScheme, nano-ML/μML, and Molecule).

- What about a course in interpreters? If you are interested in *definitional* interpreters, *Build, Prove, and Compare* presents many well-crafted examples. And the online Supplement presents a powerful infrastructure that your students can use to build more definitional interpreters (Appendices F to I). But apart from this infrastructure, the book does not discuss what a definitional interpreter is or how to design one. For a course on interpreters, you would probably want an additional book.

All of these potential designs are well supported by the exercises (345 in total), which fall into three big categories. For insight into how to use programming languages effectively, there are programming exercises that use the bridge languages. For insight into the workings of the languages themselves, as well as the formalism that describes them, there are programming exercises that extend or modify the interpreters. And for insight into formal description and proof, there are theory exercises. Model solutions for some of the more challenging exercises are available to instructors.

A few exercises are simple enough and easy enough that your students can work on them for 10 to 20 minutes in class. But most are intended as homework.

- To introduce a new language like Impcore, μScheme, μML, Molecule, or μSmalltalk, think about assigning from a half dozen to a full dozen programming exercises, most easy, some of medium difficulty.
- To introduce proof technique, think about assigning around a half dozen proof problems, maybe one or two involving some form of induction (some metatheory, or perhaps an algebraic law involving lists).
- To develop a deep understanding of a single topic, assign one exercise or a group of related exercises aimed at that topic. Such exercises are provided for continuations, garbage collection, type checking, type inference, search trees, and arbitrary-precision integers.

CONTENTS AND SCOPE

Because this book is organized by language, its scope is partly determined by what the bridge languages do and do not offer relative to the originals on which they are based.

- μScheme offers `define`, a `lambda`, and three “let” forms. Values include symbols, machine integers, Booleans, `cons` cells, and functions. There’s no numeric tower and there are no macros.
- μML offers type inference, algebraic data types, and pattern matching. There are no modules, no exceptions, no mutable reference cells, and no value restriction.
- Molecule offers a procedural, monomorphic core language with mutable algebraic types, coupled to a module language that resembles OCaml and Standard ML.
- μSmalltalk offers a pure object-oriented language in which everything is an object; even classes are objects. Control flow is expressed via message passing in a form of continuation-passing style. μSmalltalk provides a modest class hierarchy, which includes blocks, Booleans, collections, magnitudes, and three kinds of numbers. And μSmalltalk includes just enough reflection to enable programmers to add new methods to existing classes.

Each of the languages supports multiple topical themes; the major themes are programming, semantics, and types.

- *Idiomatic programming* demonstrates effective use of proven features that are found in many languages. Such features include functions (μScheme, Chapter 2), algebraic data types (μML, Chapter 8), abstract data types and modules (Molecule, Chapter 9), and objects (μSmalltalk, Chapter 10).
- *Big-step semantics* expresses the meaning of programs in a way that is easily connected to interpreters, and which, with practice, becomes easy to read and write. Big-step semantics are given for Impcore, μScheme, nano-ML, μML, and μSmalltalk (Chapters 1, 2, 7, 8, and 10).
- *Type systems* guide the construction of correct programs, help document functions, and guarantee that language features like polymorphism and data abstraction are used safely. Type systems are given for Typed Impcore, Typed μScheme, nano-ML, μML, and Molecule (Chapters 6 to 9).

In addition the major themes and the concepts listed above, the book addresses, to varying degrees, these other concepts:

- Subtype polymorphism, in μSmalltalk (Chapter 10)
- Light metatheory for both operational semantics and type systems (Chapters 1, 5, and 6)
- Free variables, bound variables, variable capture, and substitution, in both terms and types (Chapters 2, 5, and 6)
- Continuations for backtracking search, for small-step semantics, and for more general control flow (Chapters 2, 3, and 10)
- The propositions-as-types principle, albeit briefly (Chapter 6 Afterword)

A book is characterized not only by what it includes but also by what it omits. To start, this book omits the classic theory results such as type soundness and strong normalization; although learners can prove some simple theorems and look for interesting counterexamples, theory is used primarily to express and communicate ideas, not to establish facts. The book also omits lambda calculus, because lambda calculus is not suitable for programming.

The book omits concurrency and parallelism. These subjects are too difficult and too ramified to be handled well in a broad introductory book.

And for reasons of space and time, the book omits three engaging programming models. One is the pure, lazy language, as exemplified by Haskell. Another is the prototype-based object-oriented language, made popular by JavaScript, but brilliantly illustrated by Self. The third is logic programming, as exemplified by Prolog—although Prolog is explored at length in the Supplement (Appendix D). If you are interested in μHaskell, μSelf, or μProlog, please write to me.

SOFTWARE AND OTHER SUPPLEMENTS

The software described in the book is available from the book's web site, which is `build-prove-compare.net`. The web site also provides a "playground" that allows you to experiment with the interpreters directly in your browser, without having to download anything. And it holds the book's PDF Supplement, which includes additional material on multiprecision arithmetic, extensions to algebraic data types, logic programming, and longer programming examples. The Supplement also describes all the code: both the reusable modules and the interpreter-specific modules.

Acknowledgments

I was inspired by Sam Kamin's 1990 book *Programming Languages: An Interpreter-Based Approach*. When I asked if I could build on his book, Sam gave me his blessing and encouragement. *Programming Languages: Build, Prove, and Compare* is narrower and deeper than Sam's book, but several programming examples and several dozen exercises are derived from Sam's examples and exercises, with permission. I owe him a great debt.

In 1995, the Computer Science faculty at Purdue invited me to visit for a year and teach programming languages. Without that invitation, there might not have been a book.

An enormous book is not among the typical duties of a tenured computer-science professor. Kathleen Fisher made it possible for me to finish this book while teaching at Tufts; I am profoundly grateful.

Andrea Schuler and Jack Davidson helped me get permissions for the epigraphs that appear at the beginning of each chapter.

Russ Cox helped bootstrap the early chapters, especially the C code. His work was supported by an Innovation Grant from the Dean for Undergraduate Education at Harvard.

Many colleagues contributed to the development of Molecule. Matthew Fluet's insights and oversight were invaluable; without him, Chapter 9 would never have been completed. And Andreas Rossberg's chapter review helped get me onto the right track.

Robby Findler suggested that the control operators in μScheme+ be lowered to the `label` and `long-goto` forms. He also suggested the naming convention for μScheme+ exceptions.

David Chase suggested the garbage-collector debugging technique described in Section 4.6.2.

Matthew Fluet found some embarrassing flaws in Typed Impcore and Typed μScheme, which I repaired. Matthew also suggested the example used in Section 6.6.8, which shows that if variable capture is not avoided, Typed μScheme's type system can be subverted.

Benjamin Pierce taught me how to think about the roles of proofs in programming languages; Section 1.7 explains his ideas as I understand them.

Chris Okasaki opened my eyes to a whole new world of data structures.

The work of William Cook (2009) shaped my understanding of the consensus view about what properties characterize an object-oriented language. Any misunderstandings of or departures from the consensus view are my own.

Christian Lindig wrote, in Objective Caml, a prettyprinter from which I derived the prettyprinter in Appendix J.

Sam Guyer helped me articulate my thoughts on why we study programming languages.

Matthew Flatt helped me start learning about macros.

Kathy Gray and Matthias Felleisen developed `check-expect` and `check-error`, which I have embraced and extended.

Cyrus Cousins found a subtle bug in μScheme+.

Mike Hamburg and Inna Zakharevich spurred me to improve the concrete syntax of μSmalltalk and to provide better error messages.

Andrew Black examined an earlier design of μSmalltalk and found it wanting. His insistence on good design and clear presentation spurred innumerable improvements to Chapter 10.

Pharo By Example (Black et al. 2010) explained Smalltalk metaclasses in a way I could understand.

Dan Grossman read an early version of the manuscript, and he not only commented on every detail but also made me think hard about what I was doing. Kathleen Fisher's careful reading spurred me to make many improvements throughout Chapters 1 and 2. Jeremy Condit, Ralph Corderoy, Allyn Dimock, Lee Feigenbaum, Luiz de Figueiredo, Andrew Gallant, Tony Hosking, Scott Johnson, Juergen Kahrs, and Kell Pogue also reviewed parts of the manuscript. Gregory Price suggested ways to improve the wording of several problems. Penny Anderson, Jon Berry, Richard Borie, Allyn Dimock, Sam Guyer, Kathleen Fisher, Matthew Fluet, William Harrison, David Hemmendinger, Tony Hosking, Joel Jones, Giampiero Pecelli, Jan Vitek, and Michelle Strout bravely used preliminary versions in their classes. Penny found far more errors and suggested many more improvements than anyone else; she has my profound thanks.

My students, who are too numerous to mention by name, found many errors in earlier drafts. Students in early classes were paid one dollar per error, from which an elite minority earned enough to recover the cost of their books.

Individual chapters were reviewed by Richard Eisenberg, Mike Sperber, Robby Findler, Ron Garcia, Jan Midtgaard, Richard Jones, Suresh Jagannathan, John Reppy, Dimitrios Vytiniotis, François Pottier, Chris Okasaki, Stephanie Weirich, Roberto Ierusalimschy, Matthew Fluet, Andreas Rossberg, Stephen Chang, Andrew Black, Will Cook, and Markus Triska.

Larry Bacow inspired me to do the right thing and live with the consequences.

Throughout the many years I have worked on this book, Cory Kerens has loved and supported me. And during the final push, she has been the perfect companion. She, too, knows what it is to be obsessed with a creative work—and that shipping is also a feature. Cory, it's time to go adventuring!

Credits

In the epigraph for the Introduction, Russ Cox is quoted by permission.

The epigraph for Chapter 1 is from John Backus 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641. Used by permission.

The epigraphs for Chapters 2 and 4 are from Richard Kelsey, William Clinger, and Jonathan Rees 1998. *Revised*^{5} *Report on the Algorithmic Language Scheme*. Used by permission.

The epigraph for Chapter 3 is from Peter J. Landin 1964. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320. Used by permission.

In the epigraph for Chapter 5, David Hanson and the unnamed student are quoted by permission.

The first epigraph for Chapter 6 is from John C. Reynolds 1974. Towards a theory of type structure. In *Colloque sur la Programmation, Paris, France, LNCS*, volume 19, pages 408–425. Springer-Verlag. Used by permission.

In the second epigraph for Chapter 6, Arvind is quoted by permission.

The epigraph for Chapter 7 is from Robin Milner 1983. How ML evolved. *Polymorphism—The ML/LCF/Hope Newsletter*, 1(1). Used by permission.

The epigraph for Chapter 8 is from Frederick P. Brooks, Jr. 1975. *The Mythical Man-Month*. Addison-Wesley. Used by permission.

The epigraph for Chapter 9 is from Barbara Liskov and Stephen Zilles 1974. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59. Used by permission.

The first epigraph for Chapter 10 is from Alan C. Kay 1993. The early history of Smalltalk. *SIGPLAN Notices*, 28(3):69–95. Used by permission.

The second epigraph for Chapter 10 is from Kristen Nygaard and Ole-Johan Dahl 1978. The development of the SIMULA languages. *SIGPLAN Notices*, 13(8):245–272. Used by permission.

Judgment forms, important functions, & concrete syntax

Evaluation judgments

Language	*Expression or related form*	*Page*	*Definition*	*Page*
Impcore	$\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$	30	$\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \phi' \rangle$	37
μScheme	$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	144	$\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$	151
μScheme+	$\langle e/v, \rho, \sigma, S \rangle \rightarrow \langle e'/v', \rho', \sigma', S' \rangle$	215	$\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$	222
Typed Impcore	(as in Impcore)	30		37
Typed μScheme	$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	380	(as in Impcore)	151
nano-ML	$\langle e, \rho \rangle \Downarrow v$	405	$\langle d, \rho \rangle \rightarrow \rho'$	405
μML	$\langle e, \rho \rangle \Downarrow v$	491	(as in nano-ML)	405
	$\langle p, v \rangle \rightarrowtail r$ (pattern match)	490		
μSmalltalk				
definition	$\langle d, \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle$			684
expression finishes	$\langle e, \rho, c_{\mathsf{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle$			679
expression returns	$\langle e, \rho, c_{\mathsf{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle$			679
expressions return	$\langle [e_1, \ldots, e_n], \rho, c_{\mathsf{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle$			679
expressions finish	$\langle [e_1, \ldots, e_n], \rho, c_{\mathsf{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle [v_1, \ldots, v_n]; \sigma', \mathcal{F}' \rangle$			679
primitive	$\langle p, [v_1, \ldots, v_n], \xi, \sigma, \mathcal{F} \rangle \Downarrow_p \langle v; \sigma', \mathcal{F}' \rangle$			679
method dispatch	$m \rhd c \mathbin{@} \mathit{imp}$			681

Typing judgments

Language	*Expression or related form*	*Page*	*Definition*	*Page*
Typed Impcore	$\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$	335	$\langle d, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma'_\xi, \Gamma'_\phi \rangle$	336
Typed μScheme	$\Delta, \Gamma \vdash e : \tau$	363	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	366
nano-ML	$\Gamma \vdash e : \tau$ (nondeterministic)	413	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	416
	$\theta\Gamma \vdash e : \tau$ (with substitutions)	418	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	416
	$C, \Gamma \vdash e : \tau$ (with constraints)	418	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	416
μML	(as in nano-ML)	418	(as in nano-ML)	416
	$\Gamma, \Gamma' \vdash p : \tau$ (pattern)	496		
Molecule	(14 judgment forms are shown in Chapter 9, Figure 9.14)			565

Well-formedness judgments

Language	*Form*	*Judgment*	*Page*
Typed Impcore	Type	τ is a type	334
Typed μScheme	Kind	κ is a kind	354
	Type	$\Delta \vdash \tau :: \kappa$	355

Evaluation and type-checking functions

	Evaluation				Type checking and elaboration			
Language	*Exp.*	*Page*	*Def.*	*Page*	*Exp.*	*Page*	*Def.*	*Page*
Impcore	`eval`	48	`evaldef`	53				
μScheme	`eval`	155	`evaldef`	159				
μScheme+	`eval`	227	`evaldef`	159				
μScheme (in ML)	`eval`	309	`evaldef`	311				
Typed Impcore	`eval`	S397	`evaldef`	S398	`typeof`	338	`typdef`	341
Typed μScheme	`eval`	S411	`evaldef`	S412	`typeof` ($\mathcal{E}$)	366	`typdef` ($\mathcal{E}$)	366
nano-ML	`eval`	S429	`evaldef`	S430	`typeof`	437	`typdef`	439
μML	`ev`	492	`evalDataDef`	490	`ty`	497	`typeDataDef`	489
μSmalltalk	`eval`	688	`evaldef`	693				

Other judgments

Language	*Concept*	*Judgment*	*Page*
μScheme	Primitive equality	$v_1 \equiv v_2$	150
μScheme+	Tail position	e is in tail position	253
μScheme	Free term variable	$y \in \text{fv}(e)$	316
Typed μScheme	Free type variable	$\alpha \in \text{ftv}(\tau)$	371
Typed μScheme	Type equivalence	$\tau \equiv \tau'$	369
Typed μScheme	Capture-avoiding substitution	$\tau'[\alpha \mapsto \tau] \equiv \tau''$	374
Nano-ML	Constraint satisfied	C is satisfied	428

Tables relating judgments and functions

Language	*Evaluation*	*Type checking*
Impcore	page 40	—
μScheme (C code)	page 153	—
μScheme (ML code)	page 304	—
Typed Impcore	—	page 338
Typed μScheme	—	page 367
nano-ML	—	page 432
μML	page 492	page 491

Concrete syntax

Language	*Page*	*Language*	*Page*
Impcore	18	nano-ML	404
μScheme	93	μML	467
μScheme+	203	Molecule	536
Typed Impcore	330	μSmalltalk	628
Typed μScheme	353		

Symbols and notation, in order of appearance

Impcore

$::=$	defines a syntactic category in a grammar, page 17
$\|$	separates alternatives in a grammar, page 17
$\{\cdots\}$	repeatable syntax in a grammar, page 17
ξ	global-variable environment (“ksee”), page 28
ϕ	function environment (“fee”), page 29
ρ	value environment (“roe”), page 29
x	object-language variable, page 29
v	value, page 29
$\mapsto$	shows binding in function or environment, page 29
y	object-language variable, page 29
$\{\}$	empty environment, page 29
d	definition, page 30
e	expression, page 30
$\langle\cdots\rangle$	brackets wrapping abstract-machine state, page 30
$\oplus$	object-language operator, page 30
$\Downarrow$	relates initial and final states of big-step evaluation (“yields”), page 30
dom	domain of an environment or function, page 32
$\in$	membership in a set, page 32
f	name of object-language function, page 36
$\rightarrow$	relates initial and final states in evalution of definitions, page 37
$\triangleq$	defines syntactic sugar, page 66
$[\![\cdots]\!]$	brackets used to wrap syntax (“Oxford brackets”), page 81
$[\cdots]$	optional syntax in a grammar, page 86

μScheme

$\mathcal{P}$	in a mini-index, marks a primitive function (“primitive”), page 95
$O(\cdots)$	asymptotic complexity, page 100
k	a key in an association list, page 105
a	an attribute in an association list, page 105
$\{\cdots\}$	justification of a step in an equational proof, page 114
$(\!\vert\cdots\vert\!)$	a closure, page 122
$\circ$	function composition (“composed with”), page 125
$::$	infix notation for `cons` (“cons”), page 128
$\vee$	disjunction (“or”), page 139
$\neg$	Boolean complement (“not”), page 139
σ	the store: a mapping of locations to values (“sigma”), page 144
$\subseteq$	the subset relation, reflexively closed (“subset”), page 181

μScheme+

$[\,]$	an empty stack (“empty”), page 210
F	frame on an evaluation stack (“frame”), page 210
S	evaluation stack, page 210
$\bullet$	a hole in an evaluation context (“hole”), page 211
$e \rightsquigarrow e'$	lowering transformation (“lowerexp”), page 214

$\to$	the reduction relation in a small-step semantics (“steps to”), page 215
e/v	abstract-machine component: either an expression or a value, page 215
$\to^*$	the reflexive, transitive closure of the reduction relation (“normalizes to”), page 215
C	an evaluation context in a traditional semantics, page 241
λ	the Greek way of writing `lambda`, page 242

Garbage collection

H	the size of the heap, page 260
L	the amount of live data, page 261
γ	the ratio of heap size to live data (“gamma”), page 262

Type systems

τ	a type (“tau”), page 333
Γ	type environment; maps term variable to its type (“gamma”), page 333
$\to$	in a function type, separates the argument types from the result type (“arrow”), page 334
$\times$	in a function type, separates the types of the arguments (“cross”), page 334
$\vdash$	in a judgment, separates context from conclusion (“turnstile”), page 335
$e : \tau$	ascribes type τ to term e (“e has type τ”), page 335
$\to$	relates type environments before and after typing of definition, page 336
μ	a type constructor (“mew”), page 347
$\times$	forms pair types or product types (multiplication is $\cdot$ on page S15) (“cross”), page 348
$+$	used to form sum types, page 349
$[\![\tau]\!]$	the set of values associated with type τ, page 350
κ	a kind, which classifies types (“kappa”), page 354
$*$	the kind ascribed to types that classify terms (“type”), page 354
$\Rightarrow$	used to form kinds of type *constructors* (“arrow”), page 354
$\tau :: \kappa$	ascribes kind κ to type τ (“τ has kind κ”), page 354
Δ	a kind environment (“delta”), page 354
α, β, γ	type variables (“alpha, beta, gamma”), page 356
$\forall$	used to write quantified, polymorphic types (“for all”), page 356
$(\tau_1, \ldots, \tau_n)\ \tau$	τ applied to type parameters $\tau_1, \ldots, \tau_n$, page 357
$\equiv$	type equivalence, page 369
$\cap$	set intersection, page 374
$\emptyset$	the empty set (“empty”), page 374

Type inference

σ	a type scheme (“sigma”), page 408
θ	a substitution (“THAYT-uh”), page 409
$\leqslant$	the instance relation (“instance of”), page 410
θ_I	the identity substitution, page 411
$\tau \sim \tau'$	simple type-equality constraint (“τ must equal τ'”), page 418
C	type-equality constraint, page 418
$\mathbf{T}$	the trivial type-equality constraint, page 420
$\equiv$	equivalence of constraints, page 432

Abstract data types

$\wr \cdots \wr$	bag brackets, page 550
$<:$	the subtype relation, page 561