

# PROGRAMMING LANGUAGES

## Build, Prove, and Compare

(Abridged Edition)

---

Norman Ramsey  
Tufts University  
Medford, Mass

For COMP 105, Tufts University, Fall 2020  
August, 2020

Copyright © 2020 by Norman Ramsey  
All rights reserved



## PREFACE CONTENTS

---

CONTENTS AND SCOPE	VI	DESIGNING A COURSE TO USE THIS BOOK	VIII
WHAT TO EXPECT FROM THE BRIDGE LANGUAGES	VII	Using the exercises	ix
		ACKNOWLEDGMENTS	X

## Preface

This book is about programming languages: 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 many 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 unlike a real programming language, it is small enough to describe formally and to learn in a week or two—yet still 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 more 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 book includes hundreds of exercises, 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 features themselves, as well as the formalism that describes them, there are programming exercises that extend or modify the interpreters. And for insight into formal mechanisms of description and proof, there are theory exercises. Solutions to many challenging exercises are available to instructors.

The exercises impart skills that use 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 Table 1.

To use the book effectively will require skills in programming and proof:

- To use the bridge languages to solve programming problems, one needs a year's worth of programming instruction at the university level, including data structures. Many people do better with an additional half year of instruction.

<i>Language</i>	<i>Concepts introduced</i>
Impcore	Abstract syntax, big-step operational semantics, definitional interpreter
$\mu$ Scheme	Symbolic computing, functional programming with closures, algebraic laws and equational reasoning, semantics of mutable locations, continuation-passing style
$\mu$ Scheme+	Small-step semantics, control operators, CESK machine, garbage collection
Typed Impcore	Monomorphic type checking
Typed $\mu$ Scheme	Parametric polymorphism, polymorphic type checking
nano-ML	Type inference using equality constraints
$\mu$ ML	Algebraic data types, pattern matching
Molecule	Abstract data types, modules
$\mu$ Smalltalk	Objects, classes, inheritance

Table 1: The bridge languages

- To extend and modify the implementations in Chapters 1 to 4, one needs to be able to read and modify C code. 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, one needs to be able to read and modify Standard ML code. Standard ML is used because it is a simple, powerful language that is ideally suited to writing interpreters. Anyone already familiar with Haskell, OCaml, or a similar language can make a quick transition to Standard ML. Otherwise, Chapters 2, 5, and 8 impart the necessary skills.
- To prove the simpler theorems, one needs to substitute equals for equals or to fill in templates of logical reasoning. To prove the more interesting theorems, one needs proof by induction.

## CONTENTS AND SCOPE

This book is organized by language, and each language supports multiple topical themes. The major themes are programming, semantics, and types.

- *Idiomatic programming* conveys the effective use of proven features that are found in many languages. Such features include functions in  $\mu$ Scheme and  $\mu$ ML (Chapters 2 and 8), algebraic data types in  $\mu$ ML, abstract data types in Molecule (Chapter 9), and objects in  $\mu$ Smalltalk (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,  $\mu$ Scheme, nano-ML,  $\mu$ ML, Molecule, and  $\mu$ Smalltalk (Chapters 1, 2, 7, 8, 9, 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  $\mu$ Scheme, nano-ML,  $\mu$ ML, and Molecule (Chapters 6 to 9).

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

- Subtype polymorphism, in  $\mu$ Smalltalk (Chapter 10)
- Light metatheory for both operational semantics and type systems (Chapters 1, 5, and 6), but without classic results like type soundness or strong normalization, which are beyond the scope of this book
- 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, very briefly (Chapters 6 and 11)

Even more than by what we put in, the scope of a design is characterized by what we leave out. To start, the classic theory results are beyond the scope of the book. In *Programming Languages: Build, Prove, and Compare*, although we prove some simple theorems and we look for interesting counterexamples, our primary goal is to use theory to express and communicate ideas. The lambda calculus is also beyond the scope of the book, because it's a poor fit: While it's easy to encode simple data structures, it's not so easy to write interesting functions or programs, and when we do write functions, the results they return are hard to read.

One important language-design area is also out of scope: concurrency and parallelism. These are too difficult and too ramified to be handled well in a broad introductory book.

I have also, reluctantly, omitted three engaging programming models. One is the pure, lazy language, as exemplified by Haskell, with its pure, monadic I/O. Another is the prototype-based object-oriented language, made popular by JavaScript, but brilliantly illustrated by Self, which, among object-oriented languages, offers the ultimate combination of power and simplicity. The third is logic programming, as exemplified by Prolog. If you are interested in  $\mu$ Haskell,  $\mu$ Self, or  $\mu$ Prolog, please write to me.

## WHAT TO EXPECT FROM THE BRIDGE LANGUAGES

If you already know Scheme, ML, CLU, or Smalltalk, you may wonder if the bridge versions will work for you. If you don't know them, you may wonder if crucial parts are missing. This section explains briefly what's there and what's not.

- $\mu$ Scheme 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.
- $\mu$ ML 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.

- $\mu$ Smalltalk 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. Its modest class hierarchy includes In addition to blocks and Booleans, collections, magnitudes, and three kinds of numbers. There's just enough reflection to add new methods to an existing class.

## DESIGNING A COURSE TO USE THIS BOOK

*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 are relative beginners, 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  $\mu$ Scheme+, and they can implement garbage collectors. Or if they can make a transition from  $\mu$ Scheme to Standard ML, possibly via  $\mu$ ML, they can implement type checkers and possibly type inference.

If your students have some discrete math and maybe two or three semesters of programming experience, you can advance into the Big Three. I have taught such a class, which begins with four homework assignments that span an introduction to the framework, operational semantics, recursive functions, and higher-order functions. After these assignments, my students learn Standard ML, in which they implement first a type checker, then type inference. This schedule leaves a week for the lambda calculus, a couple of weeks for Smalltalk, and a week for programming with modules and data abstraction.

A colleague whose students are similarly experienced begins with Impcore and  $\mu$ Scheme, transitions to Standard ML to work on type systems and type inference, then returns to the bridge languages to explore  $\mu$ Smalltalk,  $\mu$ Prolog, and garbage collection.

If your students have seen interpreters and are comfortable with proof by induction, you can move much more quickly through the foundational material, creating room for other topics. I have taught such a class, which explores everything in my less advanced class, then adds garbage collection, denotational semantics, and logic programming.

- A second strategy tilts your class toward programming practice, either de-emphasizing or eliminating theory. There are many ways to introduce students to programming practice in diverse languages, and *Build, Prove, and Compare* is designed to occupy a sweet spot between two extremes. The first extreme “covers”  $N$  languages in  $N$  weeks. This extreme is great for exposure, but not so good 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 get significant practice with a couple of popular paradigms, perhaps by downloading and experimenting with well-crafted languages like OCaml and Ruby. This extreme offers significant depth, but little breadth. Overheads are high, including the overhead involved in making the software work and the cognitive overhead involved in overcoming gratuitous details and differences that full languages make inevitable.

*Build, Prove, and Compare* offers breadth and depth, without the overhead. If you want to focus on programming practice, a good target is “four languages in ten weeks.” That’s the Full Four:  $\mu$ Scheme,  $\mu$ ML, Molecule, and,  $\mu$ Smalltalk. 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 strategy tilts your class toward 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 *Build, Prove, and Compare* is suitable for case studies in *applied* theory: 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  $\mu$ Scheme, and nano-ML; equational reasoning with  $\mu$ Scheme; and type systems with Typed Impcore, Typed  $\mu$ Scheme, nano-ML,  $\mu$ ML, and Molecule. They can compare how universally quantified types are used in three different designs (Typed  $\mu$ Scheme, nano-ML/ $\mu$ ML, and Molecule).
- Many people have asked me about using the book to study interpreters. If you are interested in *definitional* interpreters, *Build, Prove, and Compare* presents many well-crafted examples. And the Supplement presents a powerful infrastructure that you can use to build more definitional interpreters (chapters F to J). But apart from this infrastructure, I do not discuss what a definitional interpreter is or how to design one.

If you want interpreters that perform well, you need more than this book can provide. Definitional interpreters can be engineered for better performance (Midtgård, Ramsey, and Larsen 2013), but a true high-performance interpreter has a register-based virtual instruction set, and it requires a translator that can do at least a little register allocation and code generation. For help designing and building such interpreters, you would need an additional book.

### Using the exercises

No matter how you balance the elements of your course, you will need to assign exercises. A few exercises are simple enough and easy enough that you can have students work on them for 10 to 20 minutes in class. But most exercises are intended as homework exercises. These divide into several groups.

- To introduce a new language like Impcore,  $\mu$ Scheme, or any of the Big Three, 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.

## 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.

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. I'm also grateful to Andreas Rossberg, whose chapter review helped get me onto the right track.

Robby Findler suggested that the control operators in  $\mu$ Scheme+ be lowered to the `label` and `long-goto` forms. He also suggested the naming convention for  $\mu$ Scheme+ 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  $\mu$ Scheme, which I removed. Matthew also suggested the example used in Section 6.6.8, which shows that if variable capture is not avoided, Typed  $\mu$ Scheme'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 H.

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  $\mu$ Scheme+.

Mike Hamburg and Inna Zakharevich spurred me to improve the concrete syntax of  $\mu$ Smalltalk and to provide better error messages.

Andrew Black examined an earlier design of  $\mu$ Smalltalk 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. (I hope also to receive comments from COMP 105, Tufts University, Fall 2020.) 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.

For chapter reviews, I'm grateful to Richard Eisenberg, Mike Sperber, Robby Findler, Ron Garcia, Jan Midgaard, Richard Jones, Suresh Jagannathan, John Reppy, Dimitrios Vytiniotis, Chris Okasaki, Matthew Fluet, Roberto Ierusalimschy, Stephanie Weirich, 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, I have been loved and supported by Cory Kerens. 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!

Norman Ramsey  
Malden, Massachusetts  
August, 2020

# CONTENTS

---

PREFACE	v
TABLES OF JUDGMENT FORMS AND IMPORTANT FUNCTIONS	xiv
SYMBOLS AND NOTATION	xvi
INTRODUCTION	1
<i>What you will learn and how (1). The book in detail (3). Parting advice (7).</i>	
<b>PART I: FOUNDATIONS</b>	<b>9</b>
<b>1. AN IMPERATIVE CORE</b>	<b>11</b>
§1.1. Looking at languages (13). §1.2. The Impcore language (17). §1.3. Abstract syntax (27). §1.4. Environments and the meanings of names (28). §1.5. Operational semantics (29). §1.6. The interpreter (38). §1.7. Operational semantics revisited: Proofs (56). §1.8. Extending Impcore (67). §1.9. Summary (69). §1.10. Exercises (73).	
<b>2. SCHEME, S-EXPRESSIONS, AND FIRST-CLASS FUNCTIONS</b>	<b>91</b>
§2.1. Overview of $\mu$ Scheme (92). §2.2. Language I: Values, syntax, and initial basis (93). §2.3. Practice I: Recursive functions on lists of values (100). §2.4. Data: Records and trees (109). §2.5. Combining theory and practice: algebraic laws (112). §2.6. Language II: Local variables and let (119). §2.7. Language III: First-class functions (122). §2.8. Practice III: Higher-order functions on lists (129). §2.9. Practice IV: Higher-order functions for polymorphism (133). §2.10. Practice V: Continuation-passing style (138). §2.11. Operational semantics (146). §2.12. The interpreter (154). §2.13. Extending $\mu$ Scheme with syntactic sugar (164). §2.14. Scheme as it really is (170). §2.15. Summary (174). §2.16. Exercises (178).	
<b>3. CONTROL OPERATORS AND A SMALL-STEP SEMANTICS</b>	<b>203</b>
§3.1. The $\mu$ Scheme+ language (204). §3.2. Procedural programming with control operators (207). §3.3. Operational semantics: Evaluation using a stack (212). §3.4. Operational semantics: Lowering to a core language (215). §3.5. A semantics of Core $\mu$ Scheme+ (217). §3.6. The interpreter (225). §3.7. Stacks, control, and semantics as they really are (240). §3.8. Summary (245). §3.9. Exercises (249).	
<b>4. AUTOMATIC MEMORY MANAGEMENT</b>	<b>261</b>
§4.1. What garbage is and where it comes from (262). §4.2. Garbage-collection basics (264). §4.3. The managed heap in $\mu$ Scheme+ (267). §4.4. Mark-and-sweep collection (271). §4.5. Copying collection (276). §4.6. Debugging a garbage collector (284). §4.7. Mark-compact collection (288). §4.8. Reference counting (288). §4.9. Garbage collection as it really is (290). §4.10. Summary (292). §4.11. Exercises (297).	
<b>5. INTERLUDE: <math>\mu</math>SCHEME IN ML</b>	<b>307</b>
§5.1. Names and environments (310). §5.2. Abstract syntax and values (312). §5.3. Evaluation (315). §5.4. Defining and embedding primitives (319). §5.5. Notable differences between ML and C (321). §5.6. Free and bound variables (322). §5.7. Summary (324). §5.8. Exercises (327).	
<b>6. TYPE SYSTEMS FOR IMPCORE AND <math>\mu</math>SCHEME</b>	<b>335</b>
§6.1. Typed Impcore: a statically typed imperative core (337). §6.2. A type-checking interpreter for Typed Impcore (346). §6.3. Extending Typed Impcore with arrays (352). §6.4. Interlude: common type constructors (357). §6.5. Type soundness (359). §6.6. Polymorphic type systems and Typed $\mu$ Scheme (360). §6.7. Type systems as they really are (392). §6.8. Summary (393). §6.9. Exercises (395).	
<b>7. ML AND TYPE INFERENCE</b>	<b>411</b>
§7.1. Nano-ML: a nearly applicative language (412). §7.2. Abstract syntax and values (414). §7.3. Operational semantics (415). §7.4. Type system for nano-ML (417). §7.5. From type rules to type inference (427). §7.6. The interpreter (442). §7.7. Hindley-Milner as it really is (451). §7.8. Summary (452). §7.9. Exercises (454).	

<b>PART II: PROGRAMMING AT SCALE</b>	<b>465</b>
<b>8. USER-DEFINED, ALGEBRAIC TYPES</b>	<b>469</b>
<i>§8.1. Case expressions and pattern matching (470). §8.2. Algebraic data types in μML (478). §8.3. Equational reasoning (488). §8.4. Syntactic sugar: patterns everywhere (491). §8.5. Type generativity and type equivalence (495). §8.6. Abstract syntax and values of μML (497). §8.7. Theory and implementation of user-defined types (498). §8.8. Theory and implementation of case expressions (503). §8.9. Algebraic data types as they really are (510). §8.10. Further reading (512). §8.11. Exercises (513).</i>	
<b>9. MOLECULE, ABSTRACT DATA TYPES, AND MODULES</b>	<b>535</b>
<i>§9.1. The vocabulary of data abstraction (537). §9.2. Introduction: Writing client code (538). §9.3. Implementing an abstraction (541). §9.4. The Molecule language (544). §9.5. The initial basis (554). §9.6. Program Design (555). §9.7. Inspecting multiple representations (565). §9.8. Molecule's type system (568). §9.9. Notes on the interpreter (589). §9.10. Things as they really are (590). §9.11. Summary (595). §9.12. Exercises (598).</i>	
<b>10. SMALLTALK AND OBJECT-ORIENTATION</b>	<b>619</b>
<i>§10.1. Object-oriented programming by example (620). §10.2. Data abstraction all over again (635). §10.3. The language (637). §10.4. The initial basis (645). §10.5. Technique I: Dispatch replaces conditionals (662). §10.6. Technique II: Abstract classes (664). §10.7. Technique III: Multiple representations (670). §10.8. Technique IV: Object-oriented programming with invariants (681). §10.9. Operational semantics (685). §10.10. The interpreter (693). §10.11. Smalltalk as it really is (707). §10.12. Objects and classes as they really are (714). §10.13. Summary (715). §10.14. Exercises (721).</i>	
<b>PART III: AFTERWORD</b>	<b>735</b>
<b>11. AFTERWORD</b>	<b>737</b>
<i>§11.1. Typeful programming (737). §11.2. Propositions as types (737). §11.3. More functions (739). §11.4. More objects (739). §11.5. Functions and objects, together (739). §11.6. Functional animation (739). §11.7. Scripting (740). §11.8. Parallel and distributed computation (740). §11.9. One cool domain-specific language (740). §11.10. Stack-based languages (740). §11.11. Array languages (741). §11.12. Languages based on substitution (741). §11.13. String-processing languages (741). §11.14. Conclusion (742).</i>	
<b>PART IV: BACK MATTER</b>	<b>743</b>
<b>KEY WORDS AND PHRASES</b>	<b>745</b>
<b>BIBLIOGRAPHY</b>	<b>749</b>
<b>CONCEPT INDEX</b>	<b>763</b>

*Evaluation judgments*

<i>Judgment Forms</i>	<i>Language</i>	<i>Expression or related form</i>	<i>Page</i>	<i>Definition</i>	<i>Page</i>
	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
	$\mu\text{Scheme}$	$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	146	$\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$	153
	$\mu\text{Scheme}^+$	$\langle e/v, \rho, \sigma, S \rangle \rightarrow \langle e'/v', \rho', \sigma', S' \rangle$	217	$\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$	224
xiv	Typed Impcore	(as in Impcore)	30		37
	Typed $\mu\text{Scheme}$	$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	389	(as in Impcore)	153
	nano-ML	$\langle e, \rho \rangle \Downarrow v$	415	$\langle d, \rho \rangle \rightarrow \rho'$	415
	$\mu\text{ML}$	$\langle e, \rho \rangle \Downarrow v$ $\langle p, v \rangle \rightarrow r$ (pattern match)	503	(as in nano-ML)	415
	Molecule	—	—	—	—
<hr/>					
$\mu\text{Smalltalk}$					
	definition	$\langle d, \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle$			692
	expression finishes	$\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v, \sigma', \mathcal{F}' \rangle$			686
	expression returns	$\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle$			687
	expressions return	$\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle$			687
	expressions finish	$\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma', \mathcal{F}' \rangle$			687
	primitive	$\langle p, [v_1, \dots, v_n], \xi, \sigma, \mathcal{F} \rangle \Downarrow_p \langle v, \sigma', \mathcal{F}' \rangle$			688
	method dispatch	$m \triangleright c @ \text{imp}$			689

*Typing judgments*

<i>Language</i>	<i>Expression or related form</i>	<i>Page</i>	<i>Definition</i>	<i>Page</i>
Typed Impcore	$\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$	343	$\langle d, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma'_\xi, \Gamma'_\phi \rangle$	345
Typed $\mu\text{Scheme}$	$\Delta, \Gamma \vdash e : \tau$	372	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	375
nano-ML	$\Gamma \vdash e : \tau$ (nondeterministic)	423	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	426
	$\theta\Gamma \vdash e : \tau$ (with substitutions)	428	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	426
	$C, \Gamma \vdash e : \tau$ (with constraints)	428	$\langle d, \Gamma \rangle \rightarrow \Gamma'$	426
$\mu\text{ML}$	(as in nano-ML)	428	(as in nano-ML)	426
	$\Gamma, \Gamma' \vdash p : \tau$ (pattern)	507		
Molecule	—	—	—	—

*Well-formedness judgments*

<i>Language</i>	<i>Form</i>	<i>Judgment</i>	<i>Page</i>
Typed Impcore	Type	$\tau$ is a type	343
Typed $\mu\text{Scheme}$	Kind	$\kappa$ is a kind	363
	Type	$\Delta \vdash \tau :: \kappa$	364
Molecule	—	—	—

Language	Evaluation			Type checking and elaboration				
	Exp.	Page	Def.	Page	Exp.	Page	Def.	Page
Impcore	eval	48	evaldef	54				
$\mu$ Scheme	eval	157	evaldef	161				
$\mu$ Scheme+	eval	229	evaldef	161				
$\mu$ Scheme (in ML)	eval	316	evaldef	318				
Typed Impcore	eval S388		evaldef S389		typeof	347	typdef	350
Typed $\mu$ Scheme	eval S398		evaldef S399		typeof ( $\mathcal{E}$ )	375	typdef ( $\mathcal{E}$ )	375
nano-ML	eval S406		evaldef S407		typeof	448	typdef	449
$\mu$ ML	ev	504	evalDataDef 502		ty	509	typeDataDef	501
Molecule	—	—	—	—	—	—	—	—
$\mu$ Smalltalk	eval	696	evaldef	701				

xv

### Other judgments

Language	Concept	Judgment	Page
$\mu$ Scheme	Primitive equality	$v_1 \equiv v_2$	152
$\mu$ Scheme+	Tail position	$e$ is in tail position	255
$\mu$ Scheme	Free term variable	$y \in \text{fv}(e)$	323
Typed $\mu$ Scheme	Free type variable	$\alpha \in \text{ftv}(\tau)$	380
Typed $\mu$ Scheme	Type equivalence	$\tau \equiv \tau'$	378
Typed $\mu$ Scheme	Capture-avoiding substitution	$\tau'[\alpha \mapsto \tau] \equiv \tau''$	383
Nano-ML	Constraint satisfied	$C$ is satisfied	438

### Tables relating judgments and functions

Language	Evaluation	Type checking
Impcore	page 40	—
$\mu$ Scheme (C code)	page 155	—
$\mu$ Scheme (ML code)	page 311	—
Typed Impcore	—	page 347
Typed $\mu$ Scheme	—	page 376
nano-ML	—	page 443
$\mu$ ML	page 504	page 500
Molecule	—	—

### Concrete syntax

Language	Page	Language	Page
Impcore	18	nano-ML	414
$\mu$ Scheme	95	$\mu$ ML	479
$\mu$ Scheme+	205	Molecule	546
Typed Impcore	339	$\mu$ Smalltalk	637
Typed $\mu$ Scheme	362		

## SYMBOLS AND NOTATION, IN ORDER OF APPEARANCE

*Notation*  


---

xvi

<i>Impcore</i>	
$::=$	defines a syntactic category in a grammar, page 18
	separates alternatives in a grammar, page 18
{ ... }	repeatable syntax in a grammar, page 18
$\xi$	global-variable environment (“ksee”), page 28
$\phi$	function environment (“fee”), page 28
$\rho$	value environment (“roe”), page 28
$x$	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
$\langle \dots \rangle$	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
$\text{dom}$	domain of an environment or function, page 32
$\in$	membership in a set, page 32
$f$	name of object-language function, page 35
$\rightarrow$	relates initial and final states in evalution of definitions, page 37
$\triangleq$	defines syntactic sugar, page 69
$[\![ \dots ]\!]$	brackets used to wrap syntax (“Oxford brackets”), page 83
$[\dots]$	optional syntax in a grammar, page 88

### *$\mu$ Scheme*

$\mathcal{P}$	in a mini-index, marks a primitive function (“primitive”), page 97
$O(\dots)$	asymptotic complexity, page 102
$k$	a key in an association list, page 107
$a$	an attribute in an association list, page 107
{...}	justification of a step in an equational proof, page 116
(...)	a closure, page 124
$\circ$	function composition (“composed with”), page 127
$::$	infix notation for cons (“cons”), page 130
$\vee$	disjunction (“or”), page 141
$\neg$	Boolean complement (“not”), page 141
$\sigma$	the store: a mapping of locations to values (“sigma”), page 146
$\subseteq$	the subset relation, reflexively closed (“subset”), page 183

### *$\mu$ Scheme+*

$[]$	an empty stack (“empty”), page 212
$F$	frame on an evaluation stack (“frame”), page 212
$S$	evaluation stack, page 212
$\bullet$	a hole in an evaluation context (“hole”), page 213
$e \rightsquigarrow e'$	lowering transformation (“lowerexp”), page 216
$\rightarrow$	the reduction relation in a small-step semantics (“steps to”), page 217
$e/v$	abstract-machine component: either an expression or a value, page 217
$\rightarrow^*$	the reflexive, transitive closure of the reduction relation (“normalizes to”), page 217

$C$	an evaluation context in a traditional semantics, page 243
$\lambda$	the Greek way of writing lambda, page 244

### Garbage collection

$H$	the size of the heap, page 264
$L$	the amount of live data, page 265
$\gamma$	the ratio of heap size to live data (“gamma”), page 267

### Type systems

<i>Notation</i>	
	xvii
$\tau$	a type (“tau”), page 342
$\Gamma$	type environment; maps term variable to its type (“gamma”), page 342
$\rightarrow$	in a function type, separates the argument types from the result type (“arrow”), page 342
$\times$	in a function type, separates the types of the arguments (“cross”), page 342
$\vdash$	in a judgment, separates context from conclusion (“turnstile”), page 343
$e : \tau$	ascribes type $\tau$ to term $e$ (“ $e$ has type $\tau$ ”), page 343
$\rightarrow$	relates type environments before and after typing of definition, page 345
$\mu$	a type constructor (“mew”), page 356
$\times$	forms pair types or product types (multiplication is $\cdot$ on page S17) (“cross”), page 357
$+$	used to form sum types, page 358
$[\![\tau]\!]$	the set of values associated with type $\tau$ , page 359
$\kappa$	a kind, which classifies types (“kappa”), page 363
$*$	the kind ascribed to types that classify terms (“type”), page 363
$\Rightarrow$	used to form kinds of type constructors (“arrow”), page 363
$\tau :: \kappa$	ascribes kind $\kappa$ to type $\tau$ (“ $\tau$ has kind $\kappa$ ”), page 363
$\Delta$	a kind environment (“delta”), page 363
$\alpha, \beta, \gamma$	type variables (“alpha, beta, gamma”), page 365
$\forall$	used to write quantified, polymorphic types (“for all”), page 365
$(\tau_1, \dots, \tau_n) \tau$	$\tau$ applied to type parameters $\tau_1, \dots, \tau_n$ , page 366
$\equiv$	type equivalence, page 378
$\cap$	set intersection, page 383
$\emptyset$	the empty set (“empty”), page 383

### Type inference

$\sigma$	a type scheme (“sigma”), page 418
$\theta$	a substitution (“THAYT-uh”), page 419
$\leqslant$	the instance relation (“instance of”), page 420
$\theta_I$	the identity substitution, page 422
$\tau \sim \tau'$	simple type-equality constraint (“ $\tau$ must equal $\tau'$ ”), page 428
$C$	type-equality constraint, page 428
$T$	the trivial type-equality constraint, page 431
$\equiv$	equivalence of constraints, page 443

### Abstract data types

$\{\cdots\}$	bag brackets, page 560
$\mathcal{A}$	for an automatically generated function, used as index in lieu of a page number, page S287



# *Introduction*

*The implementation exercises, for all my frustration while doing them, are tremendously valuable. I find that actually implementing something like type inference or continuations greatly enhances my understanding of it, and testable programs are much easier to play with and build intuition about than are pages of equations.*

From student course evaluations

This book is about programming languages—and also about programming. Each of these things is made better by the other. If you program but you don’t know about programming languages, your code may be longer, uglier, less robust, and harder to debug than it could be. If you know about programming languages but you don’t program, what is your knowledge for? To know a language is good, but to use it well is better.

What should you know about programming languages? Unless you are interested in history, *comprehensive* study of programming languages is not in your best interest. Instead of trying to study as many languages or language families as possible, master a few language-design ideas of lasting value: learn what they are, how to recognize them, and how to use them. Learn the best the field has to offer.

The field of programming languages is about more than just programming. We offer rigorous techniques for describing *all* computational processes, for analyzing language features, and for proving properties of programs. A serious introduction includes formal modeling and analysis of languages and language features. Formal tools are what professionals in the field use to communicate their ideas concisely and effectively. Practice with formal tools will help you to see past superficial differences in programming languages, to recognize old ideas when they appear in new languages, to evaluate new programming languages, and to choose and use programming languages intelligently. But formalism is in second place here; programming comes first.

## WHAT YOU WILL LEARN AND HOW

*Programming Languages: Build, Prove, and Compare* helps you *use* programming languages effectively, *describe* programming languages precisely, and *understand* and *enjoy* the diversity of programming languages. You will learn by experimenting with and comparing code written in different languages. You will use important programming-language features to write interesting code, understand how each feature is implemented, and see how different languages are similar and how each one is distinctive. To have such experiences, you cannot work only with full implementations of real programming languages: it’s too easy to be overwhelmed by superficial differences, and it’s too hard to learn how anything is implemented.

To provide an alternative, I have distilled four major programming languages down to small *bridge languages*, which illuminate essential features you will see repeatedly throughout your career: first-class functions, types, data abstraction, modules, pattern matching, and objects with inheritance.

Each bridge language is small enough to learn, but big enough to act as a bridge to the real thing. And all the bridge languages are written in the same, simple concrete syntax—the parenthesized-prefix syntax developed for Lisp. Uniform syntax helps you ignore superficial differences and focus on essentials. Each bridge language is also implemented by an interpreter, which helps you master the abstract world of formalism—in each chapter, you can compare mathematical descriptions of language ideas with the code that implements those ideas.

The book is accompanied by a Supplement, which presents the interpreters in depth. The interpreters are carefully crafted and documented, and you can use them to create your own language designs. Whether your own design explores a variation on one of mine or goes in a completely different direction, the opportunity to try new design ideas for yourself—and to program with the results—will give you a feel for the problems of language design, which you can't get just by studying existing languages. Don't let other people have all the fun!

The book helps you learn in three ways:

- *Build, and learn by doing.* You will learn by building and modifying programs. You will write code in the bridge languages, and you will modify the interpreters.

Once you build things, you may want to share them with others, such as potential employers. My own students' work is more than worthy of a professional portfolio. But if you share your work using a public site like Github or Bitbucket, please share this part of your portfolio only with individuals that you name—please don't put your work in a public repository.

- *Prove, to keep things simple and precise.* Proofs are not what this book is about, but if you are studying programming languages, you should get some experience with them. Try some exercises in Chapter 1 (language metatheory), Chapter 2 (equational reasoning), Chapters 6 and 7 (type-system metatheory), and Chapter 8 (more equational reasoning).
- *Compare, and find several ways to understand.* We learn more easily when we compare new ideas with what we already know—and with each other. You will learn syntax, for example, by seeing it in two forms: concrete and abstract. (Concrete syntax says how a language is written; it's something every programmer learns. Abstract syntax, which may be new to you, says what the underlying structure of a language is; it's the best way to think about what a language can say.) You can compare not just these two ways of writing one syntax but also compare the syntaxes of *different* languages. Because each new bridge language uses new syntax only when it is needed to express a new feature, features that are found in multiple bridge languages aren't just based on the same ideas; they also look the same when written down.

As another example, you can learn about the meanings of language constructs by comparing an interpreter with an operational semantics (usually the “big-step” or “natural” variety). It is easier to learn about interpretation and operational semantics together than to learn about each separately.

You can compare example programs both large and small; you'll start by comparing example programs written in a single language, but I hope you

will also compare examples written in different languages. From such examples, you will learn about recursion, higher-order functions, polymorphism, and more.

You'll accomplish all this by doing exercises, of which the book includes more than 400. In each chapter, exercises are organized by the skills they require or develop, with cross-reference to the most relevant sections. Each chapter also includes short questions intended for “retrieval practice,” which helps bring knowledge to the front of your mind. Doing exercises will help you learn how a semantics is constructed, how an interpreter works, and most important, how to write great code. Each of these avenues to learning reinforces the others, and you can emphasize what suits you best.

To do exercises, you need languages. The main bridge languages— $\mu$ Scheme,  $\mu$ ML, Molecule, and  $\mu$ Smalltalk—are rich enough to write programs that are interesting. And they are distilled from languages whose greatness is widely acknowledged: designers behind Scheme, ML, CLU, and Smalltalk have all won ACM Turing Awards, which is the highest professional honor a computer scientist can receive. The design work illuminated by the bridge languages has influenced many languages that are fashionable today, including Racket, Clojure, Rust, Haskell, Python, Java, JavaScript, Objective C, Ruby, Swift, and Erlang.

Four other tiny languages are suitable for writing toy programs only; they are intended for conveying ideas, not for programming. Impcore and Typed Impcore introduce operational semantics and type systems, but neither can express really interesting programs. Typed  $\mu$ Scheme introduces polymorphic type systems, and it can express and typecheck many interesting programs—many more than the vast majority of typed programming languages. But Typed  $\mu$ Scheme requires so much bookkeeping that using it is tiresome. Finally, nano-ML introduces type inference. Nano-ML can express interesting programs, but you're better off using  $\mu$ ML, which can express even more.

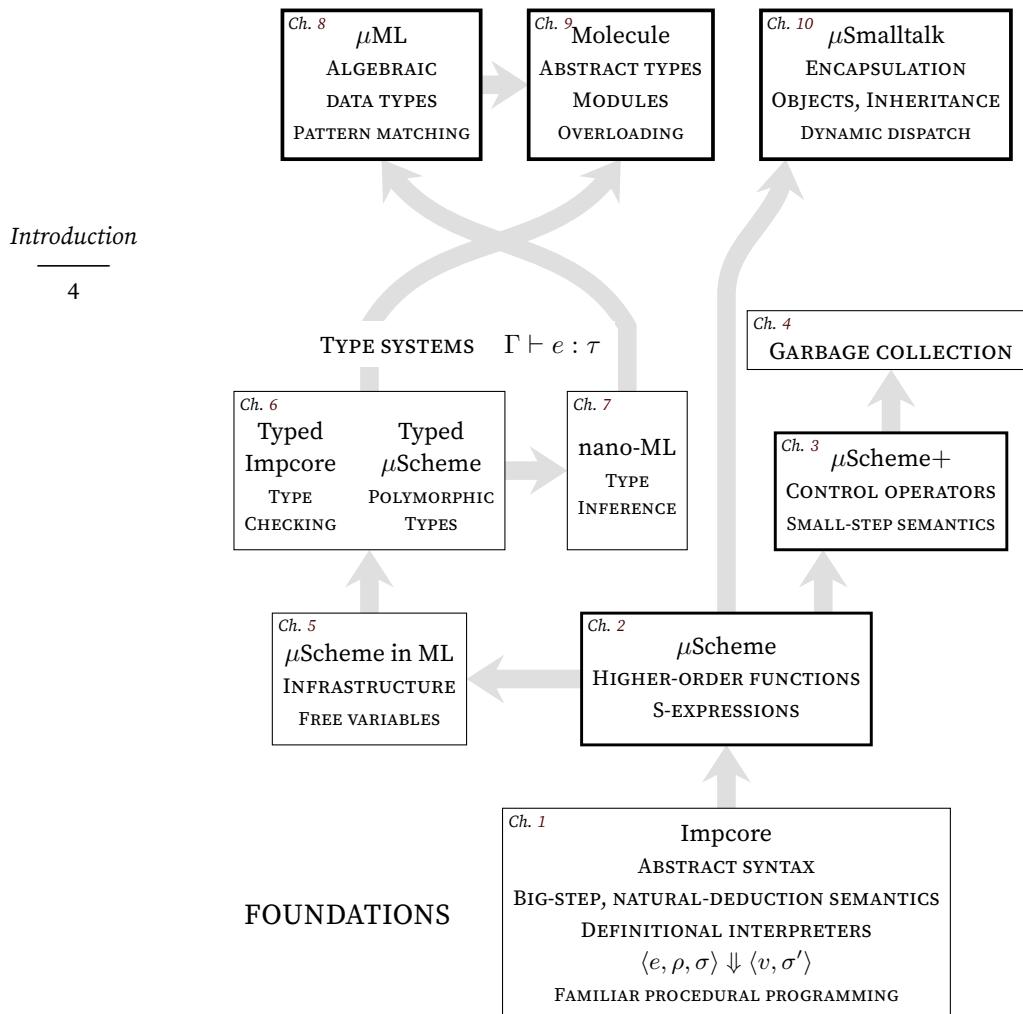
## THE BOOK IN DETAIL

Don't try to read this book cover to cover. Instead, choose languages and chapters that work for you. To help you choose, the languages and chapters are introduced here. The introductions sometimes use jargon like “operational semantics,” “polymorphism,” or “garbage collection,” because such jargon tells an expert exactly what's here. If you're not expert yet, don't worry—there are also some longer explanations. And you can see an overview in Figure 2 on the following page; the thick, gray arrows show which chapters are prerequisite for which others. Figure 2, like the book, is divided into two parts: foundational features and features for programming at scale.

### Foundations

Technical study starts with *abstract syntax* and *operational semantics*, which are formal descriptions of what a language is and what it does. These mathematical ideas are implemented by *definitional interpreters*, whose structure reflects the semantics. Both the mathematical infrastructure and the implementation infrastructure are presented in the context of a tiny imperative language, *Impcore*, which is the subject of Chapter 1. Impcore includes the familiar imperative constructs that are found at the core of mainstream programming languages: loops, conditionals, procedures, and mutable variables. Impcore doesn't introduce any new or unusual language

# PROGRAMMING AT SCALE



(Languages shown in heavy boxes are suitable for coding.)

Figure 2: Topics and languages in this book

features; instead it introduces the professional way of thinking about familiar language features in terms of abstract syntax and operational semantics. Impcore also introduces the interpreters.

Using abstract syntax, operational semantics, and a definitional interpreter,  $\mu$ Scheme (Chapter 2) introduces two new language features. First, it introduces *S-expressions*, a recursive datatype. When processing S-expressions, the natural control structure is recursion, not iteration; this change has far-reaching effects on programming style.  $\mu$ Scheme also introduces *first-class, nested functions*, which are treated as values, can be stored in data structures, can be passed to functions, and can be returned from functions. Functions that accept or return functions are called *higher-order functions*, and their use leads to a concise, powerful, and distinctive programming style: *functional programming*.  $\mu$ Scheme is used to explore simple recursive functions, higher-order functions, standard higher-order functions

on lists, continuation-passing style, and equational reasoning. These new ideas require only a handful of new language features and primitive functions:  $\mu$ Scheme extends Impcore by adding `let`, `lambda`, `cons`, `car`, `cdr`, and `null?`.

Impcore and  $\mu$ Scheme underlie everything else. The remaining foundational chapters are divided into two independent parts: one focused on memory management and one focused on types.  $\mu$ Scheme+ (Chapter 3) extends  $\mu$ Scheme with *control operators*: `break`, `continue`, `return`, `try-catch`, and `throw`. Control operators are supported by a *small-step operational semantics* and a different style of definitional interpreter; both are based on a so-called CESK machine, which uses an explicit stack for evaluation. The new semantics and interpreter are the primary reasons to study Chapter 3; you'll learn how exceptions can be implemented, and you'll see a semantics that can model interaction and nontermination. And in Chapter 4, you can extend the interpreter with *garbage collectors*.

Garbage collection enables programs written in Scheme and other safe languages to allocate new memory as needed, without worrying about where memory comes from or where it goes. Garbage collection simplifies both programming and interface design, and it is a hallmark of civilized programming. It supports all the other languages in the book. In Chapter 4, you build both mark-and-sweep and copying garbage collectors for  $\mu$ Scheme+. You can even build a simple generational collector. If you master chapters 1 to 4, you will have substantial experience connecting programming-language ideas to interpreters.

Independent of  $\mu$ Scheme+ and garbage collection, you can proceed directly from  $\mu$ Scheme to *type systems*. Type systems demand a change in the implementation language: while C is an excellent language for writing garbage collectors, it is not so good for writing type checkers or sophisticated interpreters. For these kinds of tools, you want a language that provides algebraic data types and pattern matching. The simplest, most readily available such language is Standard ML, which is used from Chapter 5 onward. To acclimate you to Standard ML, Chapter 5 reimplements  $\mu$ Scheme using Standard ML. That reimplementation provides infrastructure used in subsequent chapters, including chapters on type systems.

Chapter 6 presents type systems for *Typed Impcore*, a monomorphic, statically typed dialect of Impcore, and for *Typed  $\mu$ Scheme*, a polymorphic, statically typed dialect of  $\mu$ Scheme. Both systems explain formation rules, introduction rules, and elimination rules, with connections to logic. To solidify your understanding of type systems, I guide you in writing *type checkers* for Typed Impcore and Typed  $\mu$ Scheme. A type checker uses type annotations, on formal parameters and elsewhere, to determine the type of every expression in a program.

Typed  $\mu$ Scheme is super expressive, and its type system, when suitably specialized or extended, can describe many real languages, from the simple Hindley-Milner types of Standard ML to the complexities of features like Haskell type classes or Java generics. But considered as a programming language, Typed  $\mu$ Scheme is most unpleasant: it requires a type annotation not just on every function definition, but on every *use* of a polymorphic function. To put in the type annotations automatically, Chapter 7 offers *type inference* for the language *nano-ML*.

Nano-ML, which is derived from  $\mu$ Scheme, is as type-safe as Typed  $\mu$ Scheme, but it does not need to be annotated—the type of every parameter, variable, and function is inferred. Type inference helps make code short, simple, reusable, and reliable, and nano-ML uses almost the same data as  $\mu$ Scheme: numbers, symbols, Booleans, lists, and functions. In Chapter 7, you can implement type inference for yourself. I recommend an algorithm based on a simple solver for conjunctions of equality constraints.

<i>Language</i>	<i>New theory or concepts</i>
Impcore	Big-step operational semantics
$\mu$ Scheme	Mutable locations, capturing environment in a closure
$\mu$ Scheme+	Control operators using small-step semantics
Typed Impcore	Typechecking a monomorphic language
Typed $\mu$ Scheme	Parametric polymorphism
nano-ML	Type inference using equality constraints
$\mu$ ML	Algebraic data types
Molecule	Abstract types, modules, bounded polymorphism, separately compiled interfaces, and operator overloading
$\mu$ Smalltalk	Objects, classes, and inheritance

Table 3: Theory and concepts in each bridge language

<i>Language</i>	<i>New programming technique</i>
Impcore	Basic procedural programming
$\mu$ Scheme	Recursive functions on lists and S-expressions; higher-order functions
$\mu$ Scheme+	Control operators <code>break</code> , <code>continue</code> , <code>return</code> , <code>try-catch</code> , <code>throw</code>
Typed Impcore	Typed procedural programming with numbers, arrays, and Booleans
Typed $\mu$ Scheme	Polymorphic functions with explicit types and explicit generalization and instantiation
nano-ML	Polymorphic functions with inferred types and implicit generalization and instantiation
$\mu$ ML	Algebraic data types and pattern matching
Molecule	Abstract data types and modules; operator overloading
$\mu$ Smalltalk	Objects, classes, and inheritance

Table 4: Programming technique in each bridge language

### *Programming at scale*

Operational semantics, functions, and types are everywhere. Building on these foundations, the second part of the book presents mechanisms that programmers rely on when working at scale. These mechanisms revolve around *data abstraction*, which hides representations that are likely to change. Data abstraction enables components of large systems to be built independently and to evolve independently; it can be implemented using abstract data types, objects, or both.

Representations worth hiding need more ways of structuring data than just the arrays, lists, and atomic types we get from Typed Impcore, Typed  $\mu$ Scheme, and nano-ML. Large programs need to be able to define data using grouping (like `struct`), choices (like `union`), and recursion. Chapter 8 presents  $\mu$ ML, a language that extends nano-ML with inductively defined *algebraic data types*. Algebraic data types provide grouping, choices, and recursion, and they are central to languages like Haskell, Standard ML, OCaml, Scala, Agda, Idris, and Coq. Values of algebraic data type are examined using *case expressions* and *pattern matching*.

Chapter 9 presents *Molecule*, which implements abstract data types on top of  $\mu$ ML’s algebraic data types. Abstract types are defined inside *modules*, and they hide information using types: code can access a representation of abstract type only if it is in the same module as the type’s definition. The restriction is enforced by a polymorphic type checker like the one in Typed  $\mu$ Scheme. Abstract types and modules are found in languages as diverse as CLU, Modula-2, Ada, Oberon, Standard ML, OCaml, and Haskell. (Molecule is a new design inspired by Modula-3, OCaml, and CLU.)

Chapter 10 presents  *$\mu$ Smalltalk*, which implements data abstraction using objects. Objects hide information using names: code associated with an object can name only parts of the object’s own representation, not the representations of other objects. Unlike such hybrid languages as Ada 95, Java, C#, C++, Modula-3, Objective C, and Swift, Smalltalk is *purely* object-oriented: every value is an object, and the basic unit of control flow is message passing. Any message can be sent to any object. Objects are created by sending messages to *classes*, which are also objects, and classes *inherit* state and implementation from parent classes, which enables new forms of code reuse. The mechanisms are simple, but remarkably expressive.

*Parting advice*

7

## PARTING ADVICE

This book is not a meal; it’s a buffet. So don’t try to eat the whole thing. Pick out a few tidbits that look appetizing, try them, and do a few exercises. Digest what you’ve learned, rest, and repeat. If you work hard, then you, like my students, will be impressed at how much skill and knowledge you develop, and how you’ll be able to apply it even to languages you’ve never seen before.



## I. FOUNDATIONS

## CHAPTER CONTENTS

---

1.1	LOOKING AT LANGUAGES	13	1.7.2	Proofs about derivations:	
1.2	THE IMPCORE LANGUAGE	17		Metatheory	60
1.2.1	Lexical structure and concrete syntax	17	1.7.3	How to attempt a metatheoretic proof	62
1.2.2	Talking about syntax: Metavariables	19	1.7.4	Why bother with semantics, proofs, theory, and metatheory?	66
1.2.3	What the syntactic forms do	20	1.8	EXTENDING IMPCORE	67
1.2.4	What the primitive functions do	23	1.9	SUMMARY	69
1.2.5	Extended definitions: Beyond interactive computation	24	1.9.1	Key words and phrases	69
1.2.6	Primitive, predefined, and basis; the initial basis	26	1.9.2	Further reading	73
1.3	ABSTRACT SYNTAX	27	1.10	EXERCISES	73
1.4	ENVIRONMENTS AND THE MEANINGS OF NAMES	28	1.10.1	Retrieval practice	75
1.5	OPERATIONAL SEMANTICS	29	1.10.2	Simple functions using loops or recursion	75
1.5.1	Judgments and rules of inference	30	1.10.3	A simple recursive function	76
1.5.2	Literal values	32	1.10.4	Working with decimal and binary representations	77
1.5.3	Variables	32	1.10.5	Understanding syntactic structure	78
1.5.4	Assignment	32	1.10.6	The language of operational semantics	79
1.5.5	Control Flow	34	1.10.7	Operational semantics: facts about particular expressions	79
1.5.6	Function Application	35	1.10.8	Operational semantics: writing new rules	82
1.5.7	Rules for evaluating definitions	37	1.10.9	Operational semantics: new proof systems	83
1.6	THE INTERPRETER	38	1.10.10	Metatheory: facts about derivations	85
1.6.1	Interfaces	41	1.10.11	Advanced metatheory: facts about implementation	86
1.6.2	Implementation of the evaluator	48	1.10.12	Implementation: New semantics for variables	88
1.6.3	Implementation of environments	55	1.10.13	Extending the interpreter	89
1.7	OPERATIONAL SEMANTICS REVISITED: PROOFS	56	1.10.14	Interpreter performance	89
1.7.1	Proofs about evaluation: Theory	56			

## An imperative core

*Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic... Each assignment statement produces a one-word result. The program must cause these statements to be executed many times in order to make the desired overall change in the store, since it must be done one word at a time.*

John Backus (1978), *Can Programming Be Liberated from the von Neumann Style?*

In your prior programming experience, you may have used a *procedural* language such as Ada 83, Algol 60, C, Cobol, Fortran, Modula-2, or Pascal. Or you may have used a procedural language extended with object-oriented features, such as Ada 95, C#, C++, Eiffel, Java, Modula-3, Objective C, or Python—although these hybrid languages support an object-oriented style, they are often used procedurally. Procedural programming is a well-developed style with identifiable characteristics:

- The fundamental abstraction is the *procedure*: a sequence of *commands*, each of which tells the computer, “do something!” (“Command” is a word from the 1960s; these days, we say “statement.”) A command is executed for its *effects* on the state of the machine; procedures continually change the “mutable state” of the machine—the values contained in the various machine words—by assignment.

Each command in a procedure can itself be implemented by another procedure, and so on; procedures can be designed from the top down using *step-wise refinement* (Wirth 1971).

- Data is processed one word at a time. Source code mentions the computation and movement of each individual word of data; larger computations are made from small ones using sequences and loops. Data is most commonly organized in arrays and records, and both arrays and records are processed one element at a time, not as a whole.
- Most control flow happens in loops, which are typically written using “structured” looping constructs such as `for` and `while`. These constructs match element-by-element processing of arrays. Method calls and recursion are seldom used.
- Both control and data mimic machine architecture. The control constructs `if` and `while` combine conditional and unconditional jump instructions

in simple ways; `goto`, when present, exposes the machine’s unconditional jump. Arrays and records are implemented by contiguous blocks of memory. Pointers are addresses; assignment is often limited to what can be accomplished by a single load or store instruction.

Mimicking a machine has its advantages: the cost of a program is often easy to predict, and a debugger for a procedural language can often be built by adapting a machine-level debugger.

The procedural style can be embodied in a language. In this chapter, that language is *Impcore*. Impcore is named for the standard imperative core—assignments, loops, conditionals, and procedures—that is found in almost all programming languages, including not only “imperative” or “procedural” languages but also many “functional” and “object-oriented” languages. To work up to a nontrivial example, let’s start by defining a global variable `n`, which is initialized to 3:

**12a.** *(transcript 12a)*≡  
-> (val n 3)  
3

12b▷

The arrow “`->` ” is the interpreter’s prompt; text following a prompt is my input; and text on the next line is the interpreter’s response:<sup>1</sup>

To continue, let’s define a function `x-3-plus-1`. It multiplies a number `k` times 3, then adds 1 to the product:

**12b.** *(transcript 12a)*+≡  
-> (define x-3-plus-1 (k)  
(+ (\* 3 k) 1)) ;; returns 3 \* k + 1

△12a 12c▷

The syntax may look different from what you are used to, but before we dive into the differences, let’s do an example that computes with the variable `n` and function `x-3-plus-1` that I just defined. It’s a loop that tries to reduce `n` to 1 by halving `n` when it is even and replacing it with  $3n + 1$  when it is odd. (This loop is believed to terminate for any positive `n`.) To help explain the syntax, each line of Impcore code is commented with analogous C code; if you know C, C++, Java, JavaScript, or something similar, the comments should help you follow along.

**12c.** *(transcript 12a)*+≡  
-> (begin  
;; {  
;; while (> n 1)  
;; {  
;; begin  
;; {  
;; (println n) ;;; printf("%d\n", n);  
;; (if (= (mod n 2) 0) ;;; if (n % 2 == 0)  
;; (set n (/ n 2)) ;;; n = n / 2;  
;; (set n (x-3-plus-1 n)))) ;;; else n = x-3-plus-1(n); }  
;; }  
;; return n; }  
3  
10  
5  
16  
8  
4  
2  
1

△12b 21a▷

<sup>1</sup>Most of the examples in this book are written using the Noweb system for *literate programming*, which enables me to extract the examples from the text and make sure they are consistent with the software. Marginal labels such as 12a identify chunks of examples or code, and pointers such as “12b▷” point to subsequent chunks. A fuller explanation of Noweb appears in Section 1.6 on page 39.

In looking at the examples, you might notice that

- Where C uses curly brackets, Impcore uses begin.
- Impcore doesn't use else or return keywords.
- Impcore uses a lot of ugly parentheses, but no commas.
- Operators like +, \*, >, =, and / aren't where you might expect them to be.

Impcore, like every other language in this book, uses *fully parenthesized, prefix* syntax, in which each operator precedes its arguments. C uses *infix* syntax; each binary operator appears between its arguments. In infix syntax, order of evaluation is determined by "operator precedence," which is fine for stuff you use every day, but is not so good when you have to figure out whether & and && have the same precedence and where they both stand with respect to |. In prefix syntax, order of evaluation is determined by parentheses; it might be ugly, but you don't have to remember any operator precedence. Prefix parenthesized syntax, which is used in Scheme, Common Lisp, Emacs Lisp, Racket, Clojure, and the many other languages of the Lisp family, puts every operator and every programmer on the same footing—the parentheses eliminate any possible ambiguity.

While these examples show Impcore, that's not the real point of the chapter—the point is to start looking at programming languages in a deep, systematic way. To start looking deeper, I continue with examples from a more familiar language: C.

## 1.1 LOOKING AT LANGUAGES

Code is entered into a computer one character at a time. And between the time it is entered and the time it runs, it passes through many different phases, each of which is governed by its own rules. Phases and their rules can be hard to identify, but for any programming language, three sets of rules are essential. They tell us

- How code is formed
- What checking it undergoes before it is deemed OK to run
- What happens when it runs

To start identifying and distinguishing these rules, let's look at some C code that breaks them.

Imagine that I have a two-dimensional point on the plane, and that I want to add 3 to its *x* coordinate. Here is the definition of a point p:

```
struct { int x; int y; } p;
```

To add 3 to the *x* coordinate, I write an assignment statement, but if a letter *x* is deleted by a cat that walks across my keyboard, it might leave this code:

```
p. = p.x + 3;
```

mod

27c

This code is rejected by the C compiler, which reports a *syntax error*. The compiler might flag the = sign after the dot, where it would prefer a name. The code is *ill formed*.

I know I want the = sign, and if I'm programming after midnight, I might just remove the dot:

```
p = p.x + 3;
```

This code is well formed, but a C compiler will report a *type error*: p is not the type of thing you can assign a number to. The code is *ill typed*.

The code I meant to write is

```
p.x = p.x + 3;
```

This code is well formed and well typed, and the compiler is happy. But a happy compiler doesn't guarantee a happy program. Suppose I write

```
int n = 0;  
p.x = p.x / n;
```

This code is also well formed and well typed, and the compiler likes it, but when it runs, nothing good will happen. The best I can hope for is that my operating system will report a *run-time error*, like maybe a “floating-point exception.” (This example exhibits what the C standard calls “undefined behavior,” which the system is not obligated to report.) The code is *ill behaved*.

Useful code is well formed, well typed, and well behaved. Learning what that means, for several different languages, is half of this book. (The other half is learning how to use the languages effectively.) The key concepts are as follows:

- Code is formed according to two sets of rules: Characters are clumped into groups called *tokens* according to *lexical* rules, and tokens are grouped into definitions, statements, and so on according to *syntactic* rules. The syntactic rules are the important ones. Syntax is so important that we talk about two varieties: *concrete syntax*, which is how we write code down, and *abstract syntax*, which is how we think about code's structure. Impcore's concrete and abstract syntax are presented in Sections 1.2 and 1.3, respectively.
- Although there are myriad ways that code can be checked before it is deemed OK to run, the method that is overwhelmingly used in practice is *type checking*, which follows the rules of a *static type system*. Impcore does not have a static type system; type checking and its companion, *type inference*, are not explored in depth until Chapters 6 and 7.
- The behavior of running code is specified by its *semantics* (sometimes called *dynamic semantics*). Semantics can be written in several different styles, but this book uses a style that has dominated the field since the 1990s: *operational semantics*. Operational semantics is a great tool, but there's a learning curve, and to help you climb that curve, each language in this book is implemented by an *interpreter*. The interpreter demonstrates what the language's semantics is trying to tell us, and by enabling us to run code, it also helps us learn to use the language effectively.

The operational semantics of Impcore is presented in Section 1.5. In Section 1.6, the operational semantics is used to guide the construction of Impcore's interpreter, and in Section 1.7, the operational semantics is used to prove properties of Impcore code.

Everything starts with the way programs are formed. To learn the formation rules for *any* programming language, I recommend this process:

1. Understand the lexical structure, especially of things like comments and string literals.
2. Look for familiar syntactic categories, like definitions, declarations, statements, expressions, and types.
3. In each category, look for familiar structures: loops, conditionals, function applications, and so on. To identify what's familiar, mentally translate concrete syntax into abstract syntax.

To carry out this process requires an understanding of lexical structure, grammars, and the two varieties of syntax.

### Lexical structure

Lexical structure rarely requires much thought. If you can spot comments, string literals, and token boundaries, you're good to go. For example, in C, you need to see `3*n+1` as 5 tokens, "`3*n+1`" as a single token (a string literal), and `/*3*n+1*/` as no tokens at all (just a comment). In this book's bridge languages, a comment begins with a semicolon, there are no string literals, and token boundaries are found only at brackets or whitespace. For example, in Impcore, `3*n+1` is a single token (a name).

### Grammars

Concrete syntax is specified using a *grammar*, which tells us what sequences of tokens are well formed. For example, the following toy grammar shows four ways to form an expression *exp*:

$$\text{exp} ::= \text{variable-name} \mid \text{numeral} \mid \text{exp} + \text{exp} \mid \text{exp} * \text{exp}$$

The grammar says that an expression may be a variable, a numeral, the sum of two expressions, or the product of two expressions. Each alternative is a *syntactic form*.

This grammar, like any grammar, can be used to *produce* an expression by replacing *exp* with any of the four alternatives on the right-hand side, and continuing recursively until every *exp* has been replaced with a variable or a numeral. For example, the C expression `3 * n + 1` can be produced in this way.

The toy grammar is *compositional*: you make big expressions by composing smaller ones. Every interesting programming-language grammar supports some kind of composition; even an assembly language allows to you compose long sequences of instructions by concatenating shorter sequences. Compositional syntactic structure is part of what makes something a programming language.

### Two varieties of syntax

When we *write concrete syntax*, we should be *thinking about abstract syntax*. For example, concrete syntax tells us that the "`3n + 1`" loop is written using a `while` keyword, and that in C the loop condition goes between round brackets (parentheses). But we should be thinking "while loop with a condition and a body," and that's what abstract syntax tells us: it names the form (`WHILE`) and tells that it is formed from an expression (the condition) and a statement (the body). Abstract syntax ignores syntactic markers like keywords and brackets.

Abstract syntax helps us recognize familiar forms even when they are clothed in unfamiliar concrete syntax. Here, for example, are loops written in C, Icon, Impcore, Python, Modula-3, Scala, and Standard ML.

```
while (n > 1) n = n / 2;
while n > 1 do n := n / 2
(while (> n 1) (set n (/ n 2))
while n > 1: n = n / 2
WHILE n > 1 DO n := n / 2 END
while (n > 1) { n = n / 2; }
while !n > 1 do n := !n div 2
```

Each language uses a different concrete syntax, but abstract syntax provides a kind of X-ray vision: under their clothes, all these loops are the same.

Abstract syntax is not just a tool for thought. It also gives us *abstract-syntax trees*, the data structure used to represent code in most compilers and interpreters. In this book, abstract-syntax trees appear in C code (Chapters 1 to 3) and in Standard ML code (Chapters 5 to 10). Abstract syntax provides a compact notation for operational semantics and type systems. Using only brackets and commas as delimiters, all the `while` loops above would be written the same: `WHILE(e, s)`, where `e` stands for the condition and `s` stands for the body.

### Familiar syntactic categories and forms

You'll almost always identify abstract syntax by reading a grammar that describes concrete syntax. A real grammar will have many left-hand sides like `exp`; these symbols are called *nonterminal symbols*, or just *nonterminals*. Sometimes the nonterminals tell you exactly what the important phrases in a language are, but if the grammar has been engineered primarily to help a compiler to convert its input into an abstract-syntax tree, many nonterminals will be annoying or distracting. For example, a nonterminal like `expList1` (a list of one or more expressions separated by commas) adds nothing to our understanding.

To understand the structure of a grammar, search the nonterminals for *syntactic categories*. A syntactic category is a group of syntactic forms that all have an important role; for example, the role of an expression is to be evaluated to produce a value (and possibly also have a *side effect*). Typically, two phrases in the same syntactic category can be interchanged without affecting the well-formedness of a program. For example, any of these expressions could be used on the right-hand side of the assignment to `p.x`:

`p.x + 3`      `p.x / n`      `p->next`      `2 * p`

Assigning any of these expressions to `p.x` would be well formed, but as shown above, the division `p.x / n` might not be well behaved, and the assignments of `p->next` and `2 * p` aren't well typed.

In any programming language, there are at least four syntactic categories worth looking for:

- A *definition* introduces a new thing and gives it a name. Forms to look for include forms that define functions, variables, and maybe types; Impcore marks its function-definition and variable-definition forms with keywords `define` and `val`.
- A *declaration* introduces a new thing, with a name, but doesn't yet define the thing—instead, a declaration promises that the thing is defined elsewhere. Anything that can be declared eventually has to be defined, so the forms to look for are forms for declaring things that can be defined. In C and C++, declaration forms are mostly found in `.h` files. Impcore has no declaration forms; this book doesn't use declaration forms until Chapter 9.
- An *expression* is evaluated to produce a value, and possibly also have a side effect. Forms to look for include variables, literal values, function applications, maybe infix operators, and hopefully a conditional form (like C's ternary expression `e1 ? e2 : e3`). Impcore has all these forms except infix operators.
- A *statement* is executed for side effect; it doesn't produce a value. Side effects might include printing, changing the value of a variable, or changing some

value in memory, among others. Forms to look for include loops (`while`, `for`), conditionals (`if`), sequencing (`begin`), and if you’re lucky, a `case` or `switch` statement. Impcore doesn’t actually have statements; its `while` and `if` forms are expressions.

Impcore’s lack of statements might surprise you, but it’s a thing. Providing loops and conditionals as expressions, not statements, makes a language *expression-oriented*. All functional languages are expression-oriented; among procedural languages, Icon is expression-oriented (Griswold and Griswold 1996); and among languages with object-oriented features, Scala is expression-oriented (Odersky, Spoon, and Venners 2019). Making a language expression-oriented simplifies the syntax a bit; for example, an expression-oriented language needs only one conditional form, whereas a language like C has both a conditional statement and a conditional expression.

With your eye out for familiar definition and expression forms, you’re ready to be fully introduced to Impcore.

## 1.2 THE IMPCORE LANGUAGE

An Impcore program is a sequence of *definitions*, each of which contains at least one *expression*. Definitions come in two main forms: a definition of a variable, such as `(val n 5)`, and a definition of a function, such as `(define double (x) (+ x x))`. Impcore is implemented by an interactive interpreter, and because it’s wonderfully convenient to be able to type in an expression, have it evaluated, and see the result, Impcore also counts an expression, such as `(+ 2 2)`, as a “definition”—in this case, a definition of the global variable `it`. The global variable `it` thereby “remembers the last expression typed in.” (The variable `it` has played this special role in interactive interpreters for over thirty years.)

With the big picture of definitions and expressions in mind, we’re ready to look at the complete lexical and syntactic structure of Impcore—how programs are formed—and some more examples.

### 1.2.1 Lexical structure and concrete syntax

The rules for how Impcore programs are formed are divided into two parts: lexical rules and syntactic rules. With only minor variations, one set of lexical rules governs all the languages in the book. The rules are as follows:

- A semicolon starts a comment, which runs to the end of the line on which it appears.
- Each bracket character—`(`, `)`, `[`, `]`, `{`, or `}`—is a token by itself. Impcore uses only the round and square brackets, not the curly ones.
- Other characters are clumped into tokens that are as long as possible; a token ends only at a bracket, a semicolon, or whitespace.
- Aside from its role in delimiting tokens, whitespace is ignored.

If you’re used to C or Java, these rules may surprise you in a one small way: inputs like `x+y` and `3rd` are *single* tokens—in Impcore, each is a valid name!

Syntactic rules—that is, grammars—are written using Extended Backus-Naur Form, usually abbreviated as EBNF. EBNF is based on plain BNF, which is ubiquitous, but different specifications add different extensions, and notations for even the fundamentals of BNF can vary. I explain EBNF in the context of the grammar

```

def      ::= (val variable-name exp)
         | exp
         | (define function-name (formals) exp)
         | (use file-name)
         | unit-test

unit-test ::= (check-expect exp exp)
            | (check-assert exp)
            | (check-error exp)

exp      ::= literal
         | variable-name
         | (set variable-name exp)
         | (if exp exp exp)
         | (while exp exp)
         | (begin {exp})
         | (function-name {exp})

formals ::= {variable-name}

literal  ::= numeral

numeral  ::= token composed only of digits, possibly prefixed with a plus
           or minus sign

*-name   ::= token that is not a bracket, a numeral, or one of the “re-
           served” words shown in typewriter font

```

Figure 1.1: Concrete syntax of Impcore

for Impcore, which appears in Figure 1.1. The notation is explained more fully in Appendix A.

Figure 1.1, like any other grammar, lists nonterminal symbols like *def*, *unit-test*, *exp*, and so on. Each nonterminal is followed by the ::= symbol (pronounced “produces”), followed by the forms of the phrases that the nonterminal can produce. Alternative forms are separated by vertical bars, as (one thing | another). In each syntactic form, a token that is supposed to appear literally (like *val* or *while*) is written in typewriter font; a name that stands for a token or for a sequence of tokens (like *def*, *variable-name*, or *exp*) is written in italic font. Finally, a phrase that can be repeated is written in curly brackets, like the {*exp*} in the *begin* form; a *begin* may contain any number of *exp*s or none at all.

Figure 1.1 confirms that Impcore’s concrete syntax is fully parenthesized; whenever a sequence of tokens appears, that sequence is wrapped in brackets. You may find this syntax unattractive, especially in complex expressions. But when you’re learning multiple languages, it’s great not to have to worry about operator precedence. And if you need to write a deeply nested expression with a ton of brackets, you can show some structure through a mix of round and square brackets—as long as round matches round and square matches square, they are interchangeable.

Let’s look at the syntactic forms, starting with definitions. The definition form (val *x* *e*) defines a new global variable *x* and initializes it to the value of the expression *e*. A global variable must be defined before it is used or assigned to. Any expression *exp* may be used as a definition form; it defines or assigns to the global variable *it*. And the definition form (define *f* (*x*<sub>1</sub> … *x*<sub>*n*</sub>) *e*) defines a function *f* with formal parameters *x*<sub>1</sub> to *x*<sub>*n*</sub> and body *e*.

I call the `val`, `exp`, and `define` forms *true definitions*; these forms should be thought of as part of a program. The remaining forms, which I call *extended definitions*, are more like instructions to the interpreter. The `(use file-name)` form tells the interpreter to read and evaluate the definitions in the named file. A `check-expect`, `check-assert`, or `check-error` form tells the interpreter to remember a test and to run it *after* reading the file in which the test appears.

The expression forms, which all appear in the example while expression at the beginning of the chapter, constitute the bare minimum needed for writing imperative or procedural code. Impcore provides forms for a literal value, a variable, an assignment (`set`), a conditional (`if`), a loop (`while`), a sequence (`begin`), and a function application (any other bracketed form).

Variables and functions are named according to liberal rules: almost any non-bracket token can be a name. Only the words `val`, `define`, `use`, `check-expect`, `check-assert`, `check-error`, `set`, `if`, `while`, and `begin`, are *reserved*—they cannot be used to name functions or variables. And a *numeral* always stands for a number; a numeral cannot be used to name a function or a variable.

When Impcore starts, some names are already defined. These include *primitive* functions `+`, `-`, `*`, `/`, `=`, `<`, `>`, `println`, `print`, and `printu`, and also *predefined* functions `and`, `or`, `not`, `<=`, `>=`, `!=`, and `mod`. A set of defined names form a *basis*; the set of names defined at startup form the *initial basis*. These concepts—primitive, predefined, and basis—are explained in Section 1.2.6 on page 26.

### 1.2.2 Talking about syntax: Metavariables

Once we know how code is formed, we can talk about what happens when we run it. To talk about *any* well-formed code, not just particular codes, we use a notation called *metavariables*. Here are the metavariables used to talk about code in this chapter, and what each one stands for:

---

<i>e</i>	Any expression
<i>d</i>	Any definition
<i>n</i>	Any numeral
<i>x</i>	Any name that is meant to refer to a variable or a parameter
<i>f</i>	Any name that is meant to refer to a function

---

When we want to talk about more than one expression or name, we use subscripts. For example, we write an `if` expression as `(if e1 e2 e3)`. Because each subexpression might be different from the other two, each one is referred to by its own metavariable.

Metavariables are distinct from *program variables*. Program variables *appear* in source code; metavariables *stand for* source code. We write metavariables in *italics* and program variables in *typewriter font*. For example, *x* is a program variable: it is a name that can appear in source code. But *x* is a metavariable: it stands for *any* name that could appear in source code. Metavariable *x* might stand for *x*, but it might also stand for *y*, *z*, *i*, *j*, or any other program variable. To illustrate the difference, when we write `(val x 3)`, we mean the definition of global program variable *x*. But when we write `(val x e)`, we mean a template that can stand for *any* definition of *any* global variable.

Metavariables differ from program variables in one other crucial respect: a metavariable can't be renamed without changing its meaning. So although you can write `(define double (n) (+ n n))` and it means *exactly* the same thing as `(define double (x) (+ x x))`, you *cannot* write `(val m g)` and have it mean the same thing as `(val x e)`: `(val x e)` is a template for an expression, and `(val m g)`

- To evaluate a literal expression, which in Impcore takes the form of a numeral  $n$ , we return the 32-bit integer that  $n$  stands for.
- To evaluate an expression that takes the form of a variable name  $x$ , we find the variable that  $x$  refers to, which must be either a function parameter or a global variable defined with `val`, and we return the value.
- To evaluate an expression of the form `(set x e)`, we evaluate  $e$ , assign its value to the variable  $x$ , and return the value. Variable  $x$  must be a global variable or a formal parameter.
- To evaluate an expression of the form `(if e1 e2 e3)`, we first evaluate  $e_1$ . If the result is nonzero, we evaluate  $e_2$  and return the result; otherwise we evaluate  $e_3$  and return the result.
- To evaluate an expression of the form `(while e1 e2)`, we first evaluate  $e_1$ . If the result is not zero, we evaluate  $e_2$ , then start evaluating the loop again with  $e_1$ . We continue until  $e_1$  evaluates to zero. When  $e_1$  evaluates to zero, looping ends, and the result of the `while` loop is zero. (The result of evaluating a `while` expression is *always* zero, but because `while` is typically evaluated for its side effects, we usually don't care.)
- To evaluate an expression of the form `(begin e1 ... en)`, we evaluate expressions  $e_1$  through  $e_n$  in that order, and we return the value of  $e_n$ .
- To evaluate an expression of the form `(f e1 ... en)`, we evaluate expressions  $e_1$  through  $e_n$  in that order, calling the results  $v_1, \dots, v_n$ . We then apply function  $f$  to  $v_1, \dots, v_n$  and return the result. Function  $f$  may be *primitive* or user-defined; if  $f$  names a user-defined function, we find  $f$ 's definition, let the names in the *formals* stand for  $v_1, \dots, v_n$ , and return the result of evaluating  $f$ 's body. If  $f$  names a primitive function, we apply it as described Section 1.2.4 (page 23).

Figure 1.2(a): Rules for evaluating expressions

is gibberish. If you want a distinct name for a new metavariable, the most you can do is decorate the original name in some way—traditionally with a prime or a subscript. For example, `(val x' e4)` is also a template for an expression, and it's an expression that might be different from `(val x e)`.

### 1.2.3 What the syntactic forms do

Metavariables enable us to talk about *any* definition or expression of a certain form. And what we want to say is *operational*: what happens when a form is evaluated. To explain Impcore's syntactic forms, I want to draw on your experience and intuition, so I use informal English. But informal English lacks precision, so in Section 1.5 (page 29), I also provide a precise, formal semantics.

Expressions are more fundamental than definitions, so we start there. When an expression is evaluated, it returns a value. In Impcore, all values are integers; as in C, `if` and `while` use their conditions by interpreting zero as false and nonzero as true. The evaluation of an expression may also have a *side effect*; in Impcore, possible side effects include printing something or changing the value of a variable.

A literal 3 evaluates to the value 3.

21a. *(transcript 12a)* +≡

-> 3  
3

◀ 12c 21b ▷

The global variable n defined earlier in the chapter evaluates to its value:

21b. *(transcript 12a)* +≡

-> n  
1

◀ 21a 21c ▷

The value of n can be changed by set:

21c. *(transcript 12a)* +≡

-> (set n -13)  
-13  
-> n  
-13

◀ 21b 21d ▷

A conditional produces one value or another depending on the condition. Here's the absolute value of n:

21d. *(transcript 12a)* +≡

-> (if (< n 0) (negated n) n)  
13

◀ 21c 21e ▷

A loop's value is always 0, but it can change the values of variables:

21e. *(transcript 12a)* +≡

-> (while (< n 0) (set n (+ n 10)))  
0  
-> n  
7

◀ 21d 21f ▷

A begin is used to sequence expressions for their side effects, like printing:

21f. *(transcript 12a)* +≡

-> (begin (printu 169) (println 2021) -1)  
@2021  
-1

◀ 21e 21g ▷

Function application may call a primitive, predefined, or user-defined function, like <, negated, +, printu, println, or x-3-plus-1.

Figure 1.2(b): Examples of evaluating expressions

The rules for evaluating expressions are shown in Figure 1.2(a), and corresponding examples are shown in Figure 1.2(b)—except for calls and names, because examples of calls and names won't fit in the figure.

To illustrate calls and names, I start with two function definitions, plus one call to each. After reading the definition of a function, the interpreter echoes its name.

21g. *(transcript 12a)* +≡

-> (define add1 (n) (+ n 1))  
add1  
-> (add1 4)  
5  
-> (define double (n) (+ n n))  
double  
-> (double (+ 3 4))  
14

◀ 21f 22a ▷

negated  
printu  
27c  
B

A call to a user-defined function works much as in C: first the arguments are evaluated; their values are the *actual parameters*. Then the function's body is evaluated with each actual parameter “bound to” (which is to say, named by) the correspond-

ing *formal parameter* from the *formals* in the function’s definition. In the first example, the actual parameter is 4, and  $(+ n 1)$  is evaluated with 4 bound to  $n$ . In the second, the actual parameter is 7, and  $(+ n n)$  is evaluated with 7 bound to  $n$ .

In Impcore, a function call is well behaved only if the number of actual parameters is exactly equal to the number of formal parameters in the function’s definition. Otherwise, things go wrong:

22a. *<transcript 12a>* +≡

-> (add1 17 12)

Run-time error: in (add1 17 12), expected 1 argument but found 2

◀ 21g 22b ▶

Example functions `add1` and `double` use the name `n` as a formal parameter, and inside each function, that parameter is what `n` stands for. But outside any function, `n` continues to stand for the global variable defined on page 12—just as in C, one name can mean different things in different contexts. Also as in C, the meaning of any occurrence is determined by the context in which it occurs: If variable  $x$  occurs in a function definition, and if the function has a formal parameter named with the same  $x$ , then  $x$  refers to that formal parameter. Otherwise  $x$  refers to a global variable. (Impcore has no local variables; to provide them is the object of Exercise 30.) If  $x$  occurs in a top-level expression, outside of any function definition, it necessarily refers to a global variable.

Let’s contrive an example:

22b. *<transcript 12a>* +≡

-> n ; the global n  
7  
-> (define addn (n m) (set n (+ n m))) ; mutates the parameter  
addn  
-> (addn n 1) ; the parameter is set to 8  
8  
-> n ; the global n is unchanged  
7

◀ 22a 23a ▶

Within the body of `addn`, the two occurrences of `n` refer to the formal parameter. But in the top-level expressions `n` and `(addn n 1)`, `n` refers to the global variable. And in the body of `addn`, where `n` is set, changing the formal `n` does not affect anything in the calling context; we say that Impcore passes parameters *by value*. No assignment to a formal parameter ever changes the value of a global variable.

With the details of expressions explored, we turn our attention to definitions. When a definition is evaluated, no value is returned; instead, evaluating a definition updates some part of the interpreter’s state, causing it to “remember” something. Things the interpreter can remember include global variables, function definitions, and pending unit tests.

- To evaluate a definition of the form  $(\text{val } x \ e)$ , we first check to see if a global variable named  $x$  exists, and if not, we create one. We then evaluate  $e$  and assign its value to  $x$ .
- To evaluate a definition that takes the form of an expression  $e$ , we evaluate  $e$  and store the result in the global variable `it`.
- To evaluate a definition of the form  $(\text{define } f \ (x_1 \dots \ x_n) \ e)$ , like the definitions of `add1` or `double`, we remember  $f$  as a function that takes arguments  $x_1, \dots, x_n$  and returns  $e$ .
- To evaluate a definition of the form  $(\text{use } \textit{filename})$ , we look for a file called `filename`, which should contain a sequence of Impcore definitions. We read the definitions and evaluate them in order. And after reading the file, we run any unit tests it contains.

- To evaluate a definition of the form (`check-expect e1 e2`), we remember this test: at the end of the file containing the definition, evaluate both  $e_1$  and  $e_2$ . If their values are equal, the test passes; if not, the test fails.
- To evaluate a definition of the form (`check-assert e`), we remember this test: at the end of the file containing the definition, evaluate  $e$ . If its value is nonzero, the test passes; if not, the test fails.
- To evaluate a definition of the form (`check-error e`), we remember this test: at the end of the file containing the definition, evaluate  $e$ . If evaluating  $e$  triggers a run-time error, like dividing by zero or passing the wrong number of arguments to a function, the test passes; if not, the test fails.

#### 1.2.4 What the primitive functions do

Not every function is defined using `define`; some are built into the interpreter as *primitives*. Each primitive function takes two arguments, except the printing primitives, which take one each. The arithmetic operators `+`, `-`, `*`, and `/` do arithmetic on integers, up to the limits imposed by a 32-bit representation. Each of the comparison operators `<`, `>`, and `=` does a comparison: if the comparison is true, the operator returns 1; otherwise, it returns 0.

The printing primitives demand detailed explanation. Primitive `println` prints a value and then a newline; it's the printing primitive you'll use most often. Primitive `print` prints a value and *no* newline. Primitive `printu` prints a Unicode character and no newline. More precisely, `printu` takes as its argument an integer that stands for a Unicode *code point*—that means it's an integer code that stands for a character in one of a huge variety of alphabets. Primitive `printu` then prints the UTF-8 byte sequence that represents the code point. In most programming environments, this sequence will give you the character you're looking for. For example, `(printu 955)` prints the Greek letter  $\lambda$ . Each printing primitive, in addition to its side effect, also returns its argument.

Using the printing primitives interactively can be confusing, because whenever an expression is evaluated, its value is printed automatically by the interpreter. Don't be baffled by effects like these:

**23a.** *(transcript 12a)*  $\equiv$   
 -> (val x 4)  
 -> (println x)  
 4  
 4

◀ 22b 23b ▶

The 4 is printed twice because `println` is called with actual parameter 4 (the value of  $x$ ), and `println` first *prints* 4, accounting for the first 4, then *returns* 4. The second 4 is printed because the interpreter prints the value of *every* expression, including `(println x)`.

**23b.** *(transcript 12a)*  $\equiv$   
 -> (val y 5)  
 5  
 -> (begin (println x) (println y) (\* x y))  
 4  
 5  
 20

◀ 23a 23c ▶

add1 21g

If you happen to use `print` instead of `println`, you can get some strange output:

**23c.** *(transcript 12a)*  $\equiv$   
 -> (begin (print x) (print y) (\* x y))  
 4520

◀ 23b 24c ▶

Because the interpreter automatically prints the value of each expression you enter, you won't use the printing primitives often—mostly for debugging. I typically use `println`, but when I want fancier output, `printu` and `print` do the job.

### 1.2.5 Extended definitions: Beyond interactive computation

The examples above show transcripts of my interactions with the Impcore interpreter. But interactive code disappears as soon as it is typed; to help you write code that you want to edit or keep, the Impcore interpreter, like all the interpreters, enables you to put it in a file. And when you put code in a file, you can add *unit tests*. Here's an example that computes a greatest common denominator:

**24a.** *(contents of file gcd.imp 24a)* 24b▷

```
(val r 0)
(define gcd (m n)
  (begin
    (while (!= (set r (mod m n)) 0)
      (begin
        (set m n)
        (set n r)))
    n))

(check-expect (gcd 6 15) 3)
```

Since the code is in a file, you don't see arrow prompts. You do see our first unit test; the `check-expect` says that if we call `(gcd 6 15)`, the result should be 3. Here are some more unit tests:

**24b.** *(contents of file gcd.imp 24a)* +≡ ▷24a

```
(check-expect (gcd 15 15) 15)
(check-expect (gcd 14 15) 1)
(check-expect (gcd 14 1) 1)
(check-expect (gcd 72 96) 24)
(check-error (gcd 14 0))
```

The last unit test says that if we evaluate `(gcd 14 0)`, we expect a run-time error.

Unit tests don't get run until after the file is loaded. Then the interpreter says what happened:

**24c.** *(transcript 12a)* +≡ ▷23c 25b▷

```
-> (use gcd.imp)
0
gcd
All 6 tests passed.
```

You can put a unit test *before* the function it tests. This trick can be a great way to plan a function, or to document it. Here's an example using *triangular numbers*. A triangular number is analogous to the square of a number: just as the square of  $n$  is the number of dots needed to form a square array with a side of length  $n$ , the  $n$ th triangular number is the number of dots needed to form an equilateral triangle with  $n$  dots along one side.

```
1 = *
3 = *
      *
6 = * *
      * *
```

If you want to learn to use programming languages well and also to describe them precisely, what languages should you study? Not big industrial languages—you can write interesting programs, but it's hard to say how programs behave, or even what programs are well behaved. And not a tiny artificial language (or a “core calculus” like the famous *lambda calculus*)—its behavior can be described very precisely, but it's hard to write any interesting programs. That's why I've designed the bridge languages: to bridge the gap between industrial languages and core calculi.

The bridge languages are small enough to be described precisely, but big enough for interesting programs. (I won't pretend you can write interesting programs in Impcore, but you *can* write interesting programs in  $\mu$ Scheme,  $\mu$ ML, Molecule, and  $\mu$ Smalltalk.) But a few features are too complicated to define precisely, yet too useful to leave out. They are the “extended definitions”: the `use`, `check-expect`, `check-assert`, and `check-error` forms. The `val`, `define`, and top-level expression forms, which *are* defined precisely are the “true definitions.” The true definitions are part of the language, and the extensions are there to make you more productive as a programmer.

You can compute a triangular number using the `sigma` function in Exercise 2 on page 76, but there's a shortcut:

**25a.** *(triangle.imp 25a)*  $\equiv$

```
(check-expect (triangle 1) 1)
(check-expect (triangle 2) 3)
(check-expect (triangle 3) 6)
(check-expect (triangle 4) 10)
```

```
(define triangle (n)
  (/ (* n (+ n 1)) 2))
```

**25b.** *(transcript 12a)*  $+ \equiv$

```
-> (use triangle.imp)
triangle
All 4 tests passed.
```

$\triangleleft 24c\ 25d \triangleright$

When writing `triangle`, I botched my first attempt. My unit tests caught the botch. Here's what it looks like:

**25c.** *(botched-triangle.imp 25c)*  $\equiv$

```
(check-expect (triangle 1) 1)
(check-expect (triangle 2) 3)
(check-expect (triangle 3) 6)
(check-expect (triangle 4) 10)
```

```
(define triangle (n)      ; botched version
  (/ (* n (- n 1)) 2))
```

**25d.** *(transcript 12a)*  $+ \equiv$

```
-> (use botched-triangle.imp)
triangle
Check-expect failed: expected (triangle 1) to evaluate to 1, but it's 0.
Check-expect failed: expected (triangle 2) to evaluate to 3, but it's 1.
Check-expect failed: expected (triangle 3) to evaluate to 6, but it's 3.
Check-expect failed: expected (triangle 4) to evaluate to 10, but it's 6.
All 4 tests failed.
```

$\triangleleft 25b\ 26b \triangleright$

The check-expect form provides what you'll need to test most Impcore code: because every Impcore value is an integer, your unit tests can usually just say what value you expect. But in Chapter 2 and beyond, you'll have access to more structured values, like records and lists, and you want to talk about *properties*, like “the list is sorted.” For that purpose, each of the bridge languages provides a check-assert form. The form is intended for use with Booleans. Here are a few assertions about properties of products.

**26a.** *(arith-assertions.imp 26a)*≡

```
(check-assert (< (* 1 2) (+ 1 2))) ; product is smaller than sum
(check-assert (> (* 2 3) (+ 2 3))) ; product is bigger than sum
(check-assert (not (< (* -1 -1) 0))) ; product of negatives is not negative
```

Function `not` is predefined; its definition appears in Figure 1.3 on page 27.

**26b.** *(transcript 12a)*+≡

```
-> (use arith-assertions.imp)
All 3 tests passed.
```

◀ 25d 29 ▷

### 1.2.6 Primitive, predefined, and basis; the initial basis

Programmers like big languages with lots of data types and syntactic forms, but implementors want to keep primitive functionality small and simple. (So do semanticists!) To reconcile these competing desires, language designers have found two strategies: translation into a *core language* and definition of an *initial basis*.

Using a core language, you stratify your language into two layers. The inner layer defines or implements its constructs directly; it constitutes the core language. The outer layer defines additional constructs by translating them into the core language; these constructs constitute *syntactic sugar*. This chapter treats Impcore as a core layer to which you can add syntactic sugar (Section 1.8). For example, you can extend Impcore with a `for` expression, which is defined by translation into `begin` and `while`.

A language alone doesn't get much done; programmers also need a standard library. The theory word is *basis*; basis is the collective term for all the things that can be named in definitions. In Impcore, these would be functions and global variables. A language's *initial basis* is the set of named things that are available in a fresh interpreter or installation—the things you have access to even before evaluating your own code.

Just like the language, Impcore's initial basis is stratified into two layers. The inner layer includes all the functions that are defined directly by C code in the interpreter; these are called *primitive*. The outer layer includes functions that are built into the interpreter, but are defined in terms of the primitives, using ordinary Impcore source code; these are called *predefined*.

Stratifying the initial basis makes life easy for everyone. Implementors make life easy for themselves by defining just a few primitives, and they can make life easy for programmers by defining lots of predefined functions. Predefined functions are just ordinary code, and writing them is lots easier than defining new primitives. Impcore's predefined functions are defined by the code in Figure 1.3.

Just like any other function, a primitive or predefined function can be redefined using `define`. This trick can be useful—for example, to count the number of times a function is called. But if you redefine an initial-basis function, take care not to change the results it returns! Such bugs are too hard to find.

Before going on to the next sections, work some of the exercises in Sections 1.10.2 to 1.10.4, starting on page 75.

We write Boolean connectives using `if` expressions.

27a. *(predefined Impcore functions 27a)*  $\equiv$

```
(define and (b c) (if b c b))  
(define or (b c) (if b b c))  
(define not (b) (if b 0 1))
```

27b ▷

Unlike the similar constructs built into the syntax of many languages, these versions of `and` and `or` always evaluate both of their arguments. Section G.7 shows how you can use syntactic sugar to define *short-circuit* variations that evaluate a second expression only when necessary.

§1.3

Abstract syntax

27

We add new arithmetic comparisons.

27b. *(predefined Impcore functions 27a)*  $\equiv$

◁ 27a 27c ▷

```
(define <= (x y) (not (> x y)))  
(define >= (x y) (not (< x y)))  
(define != (x y) (not (= x y)))
```

Finally, we use primitive arithmetic to define modulus and negation.

27c. *(predefined Impcore functions 27a)*  $\equiv$

▷ 27b

```
(define mod (m n) (- m (* n (/ m n))))  
(define negated (n) (- 0 n))
```

The C code to install the initial basis is shown in chunk *(install the initial basis in functions S293a)*, which is continued in chunk S293c.

Figure 1.3: Predefined functions in Impcore’s initial basis

### 1.3 ABSTRACT SYNTAX

In every interpreter in this book, programs are represented as *abstract-syntax trees* (ASTs). And as noted in Section 1.1, abstract syntax isn’t just a great representation; it is the best way to *think* about syntax. There’s just one issue: in C, *coding* tree representations is a pain. For that reason, the C representations in this book are generated from a little language used only to describe abstract-syntax trees—which can also help with our thinking. As an example, the abstract syntax of an Impcore definition is

27d. *(simplified example of abstract syntax for Impcore 27d)*  $\equiv$

27e ▷

```
Def = VAL      (Name, Exp)  
| EXP       (Exp)  
| DEFINE    (Name, Namelist, Exp)
```

This code defines `Def` as a category of abstract-syntax tree, which includes only the true definitions; extended definitions are `XDefs`. Each form of `Def` is named (`VAL`, `EXP`, and `DEFINE`), and following the name of the form is a comma-separated list showing what values or subtrees each form is made from.

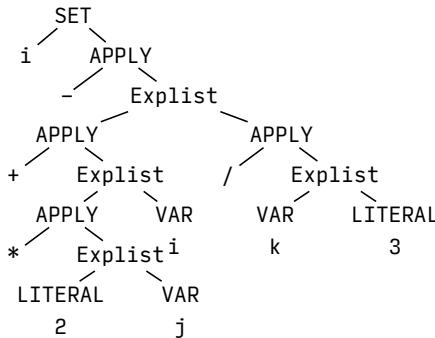
The abstract representation of an expression is specified using the same language, but with more forms:

27e. *(simplified example of abstract syntax for Impcore 27d)*  $\equiv$

▷ 27d

```
Exp = LITERAL (Value)  
| VAR      (Name)  
| SET      (Name, Exp)  
| IF       (Exp, Exp, Exp)  
| WHILE    (Exp, Exp)  
| BEGIN    (Explist)  
| APPLY    (Name, Explist)
```

This description is worth comparing this with the description of `exp` in the grammar for Impcore (Figure 1.1 on page 18).



An abstract-syntax tree is much easier to analyze, manipulate, or interpret than source code. And ASTs focus our attention on structure and semantics. Concrete syntax becomes a separate concern; one could easily define a version of Impcore with C-like concrete syntax, but with identical abstract syntax.

Abstract syntax is created from concrete input by a *parser*, which also identifies and rejects ill-formed phrases such as `(if x 0)` or `(val y)`. Parsing is covered in a large body of literature; for textbook treatments, try Appel (1998) or Aho et al. (2007). A parser for Impcore, which you can easily extend, can be found in Appendix G.

## 1.4 ENVIRONMENTS AND THE MEANINGS OF NAMES

An expression like  $(* x 3)$  cannot be evaluated by itself; we need to know what  $x$  is. Similarly, the effect of an expression such as  $(\text{set } x 1)$  depends on whether  $x$  is a formal parameter, a global variable, or something else entirely. In programming-language theory, the meanings of names are defined by an *environment*, which is usually a mapping from names to meanings. Environments are often implemented as hash tables or search trees, and in an implementation, an environment is sometimes called a *symbol table*.

Knowing what can be in an environment tells you what kinds of things a name can stand for, which is a good first step in understanding a new programming language. Impcore uses three environments, each of which is written using its own metavariable:

- Environment  $\xi$  (xi, pronounced “ksee”) holds values of global variables.
  - Environment  $\phi$  (phi, pronounced “fee”) holds definitions of functions.
  - Environment  $\rho$  (rho, pronounced “roe”) holds values of a function’s parameters.

Environments  $\xi$  and  $\phi$  are global and shared, but every function call has its own  $\rho$ . Together, the contents of the three environments comprise Impcore's *basis*.

A name can be defined in all three environments at once. Here's a maddening example:

29. *<transcript 12a>* +≡  
-> (val x 2)  
2  
-> (define x (y) (+ x y)) ; pushing the boundaries of knowledge...  
-> (define z (x) (x x)) ; and sanity  
-> (z 4)  
6

◀ 26b 65 ▶

The first definition introduces a global variable  $x$ , which is bound to value 2 in environment  $\xi$ . The second defines a function  $x$  that adds its argument to the global variable  $x$ ; that function is bound to name  $x$  in environment  $\phi$ . The third definition defines  $z$ , which passes its formal parameter  $x$  to the function  $x$ ; when  $z$  is called,  $x$  is bound to 4 in environment  $\rho$ . This example is designed to push you to understand the rules of Impcore; to follow it, you have to know not only that Impcore has three environments but also how the environments are used. Of course, no sane person programs this way; production code is written to be *easy* to understand, even by readers who may have forgotten some details of the language.

Mathematically, an environment is a function with a finite domain, from a name to whatever. In Impcore, environments  $\xi$  and  $\rho$  map each defined name to a value;  $\phi$  maps each defined name to a function. Whatever a name is mapped to, all environments are manipulated using the same notation, which is mostly function notation. For example, whatever is associated with name  $x$  in the environment  $\rho$  is written  $\rho(x)$ . The set of names bound in environment  $\rho$  is written  $\text{dom } \rho$ . An extended environment,  $\rho$  plus a binding of the name  $x$  to  $v$ , is written  $\rho\{x \mapsto v\}$ . In an extended environment, the new binding hides previous bindings of  $x$ , so lookup is governed by this equation:

$$\rho\{x \mapsto v\}(y) = \begin{cases} v, & \text{when } y = x \\ \rho(y), & \text{when } y \neq x \end{cases}$$

Finally, an empty environment, which does not bind any names, is written  $\{\}$ .

One environment can be combined with another, but because combining environments is not useful in Impcore, the notation is deferred to Chapter 2.

## 1.5 OPERATIONAL SEMANTICS

Section 1.2 describes Impcore informally and concisely, but imprecisely. Section 1.6 implements Impcore in C. An implementation is precise, but it is much longer and harder to understand than an informal description. It also lacks focus; code embodies many irrelevant decisions, including decisions about the *representations* of names, environments and abstract syntax. Those decisions are part of the implementation, not part of the language.

This section defines Impcore using a technique that is concise, focused, and precise: formal *operational semantics*. Operational semantics specifies behavior precisely while hiding implementation details. Although the notation can be intimidating at first, with practice you can read an operational semantics as easily as informal English.

Impcore's operational semantics defines an *abstract machine* and rules for its execution. The semantics defines the machine's states, including its start state and its acceptable final states, and presents rules for making transitions from one state to another. By applying these rules repeatedly, the machine can go from a start state like "I just turned on and have this program to evaluate" to an accepting state like "the answer is 42."

A machine may reach a state from which it cannot make progress; for example, a machine evaluating  $(/ 1 0)$  probably cannot make a transition. When a machine reaches such a state, we say it “gets stuck” or “goes wrong.” An implementation might indicate a run-time error.

To describe the Impcore machine’s state and transitions, the semantics uses the same metavariables as in Section 1.2.2, plus metavariable  $v$  for values and a metavariable for each of the three environments (Table 1.4 on the facing page).

The state of an Impcore machine has four parts: a *definition*  $d$  or *expression*  $e$  being evaluated; a value environment  $\xi$ , which holds the values of global variables; a function-definition environment  $\phi$ ; and a value environment  $\rho$ , which holds the values of formal parameters. Definitions do not appear inside functions, so when the machine is evaluating a definition  $d$ , there are no formal parameters, and its state is written as  $\langle d, \xi, \phi \rangle$ . When the machine is evaluating an expression  $e$ , its state is written as  $\langle e, \xi, \phi, \rho \rangle$ . When the machine is resting between evaluations, it remembers only the values of global variables and the definitions of functions, and its state is written as  $\langle \xi, \phi \rangle$ .

In Impcore, the values stored in  $\xi$  and  $\rho$  are integers. What’s stored in  $\phi$  is either a primitive function or a user-defined function. A primitive function is written as  $\text{PRIMITIVE}(\oplus)$ , where  $\oplus$  is the name of an operator like  $+$ ,  $=$ , or  $*$ . A user-defined function is written as  $\text{USER}(\langle x_1, \dots, x_n \rangle, e)$ , where the  $x_i$ ’s are the formal parameters and  $e$  is the body.

In the initial state of the Impcore machine,  $\xi = \{\}$ , because there are no global variables defined, but  $\phi = \phi_0$ , where  $\phi_0$  is preloaded with the definitions of primitive functions as well as the user-defined functions in the initial basis (Figure 1.3 on page 27).

### 1.5.1 Judgments and rules of inference

Transitions of the abstract machine are described using *judgments*, which take one form for definitions and another form for expressions. The judgment for the evaluation of an expression,  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , means “evaluating expression  $e$  produces value  $v$ .” More precisely, it means “in the environments  $\xi$ ,  $\phi$ , and  $\rho$ , evaluating  $e$  produces a value  $v$ , and it also produces new environments  $\xi'$  and  $\rho'$ , while leaving  $\phi$  unchanged.”<sup>2</sup> This judgment uses eight metavariables;  $e$  stands for an expression,  $\xi$ ,  $\phi$ , and  $\rho$  stand for environments, and  $v$  stands for a value. Different metavariables are distinguished by giving them subscripts, or as with the environments, by using primes.

Just by looking at the *form* of the judgment  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , we can learn a few things:

- Evaluating an expression always produces a value, unless of course the machine gets stuck. Even expressions like `SET` and `WHILE`, which are typically evaluated only for side effects, produce values.<sup>3</sup>
- Evaluating an expression might change the value of a global variable (from  $\xi$ ) or a formal parameter (from  $\rho$ ).

<sup>2</sup>A judgment describes a relation, not a function. In principle, it is possible to have  $v_1 \neq v_2$  such that  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$  and also  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi', \phi, \rho' \rangle$ . A language that permits such ambiguity is *nondeterministic*. All the languages in this book are deterministic, but multithreaded languages like Java or C# can be nondeterministic. Programs written in such languages can produce different answers on different runs. Languages that do not specify the order in which expressions are evaluated, like C, can also be nondeterministic. Programs written in such languages can produce different answers when translated with different compilers. (See also Exercise 27 on page 86.)

<sup>3</sup>This property distinguishes expression-oriented languages, like Impcore, ML, and Scheme, from statement-oriented languages like C. All these languages have imperative constructs that are evaluated only for side effects, but only in C do these constructs return no values.

---

$e, e_i$	An expression
$d$	A definition
$x, x_i$	A name that refers to a variable or a parameter
$f$	A name that refers to a function
$v, v_i$	A value
$\xi, \xi', \dots$	A global-variable environment
$\phi, \phi', \dots$	A function-definition environment
$\rho, \rho', \dots$	A formal-parameter environment

---

Table 1.4: Metavariables of Impcore’s operational semantics

- Evaluating an expression never adds or changes a function definition (because  $\phi$  is unchanged).

One thing we *can’t* learn from the form of the judgment is whether evaluating an expression can introduce a new variable. In fact it can’t, but to learn this requires that we study the full semantics and write an inductive proof (Exercise 24 on page 85).

The form of the evaluation judgment also gives this semantics part of its name: no matter how much computation is required to get from  $e$  to  $v$ , the judgment  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$  encompasses all that computation in one big step. It is therefore called a *big-step* judgment and is part of a *big-step* semantics.

The judgment for a definition is simpler;  $\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \phi' \rangle$  means “evaluating definition  $d$  in the environments  $\xi$  and  $\phi$  yields new environments  $\xi'$  and  $\phi'$ . ” The different arrow helps distinguish this judgment from an expression judgment.

Not all judgments describe real program behaviors. For example, it seems reasonable to claim that  $\langle (+ 1 1), \xi, \phi, \rho \rangle \Downarrow \langle 2, \xi, \phi, \rho \rangle$ , but unless some joker changes the binding of the name `+` in  $\phi$ ,  $\langle (+ 1 1), \xi, \phi, \rho \rangle \Downarrow \langle 4, \xi, \phi, \rho \rangle$  doesn’t describe what Impcore code does.

To say which judgments describe real program behaviors, an operational semantics uses *rules of inference*. Each rule has the form

$$\frac{\text{premises}}{\text{conclusion}} \quad (\text{NAME OF RULE})$$

If we can prove each of the premises, we can use the rule to prove the conclusion.

For example, the rule

$$\frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle}, \quad (\text{IFTRUE})$$

which is part of the semantics of Impcore, says that whenever  $\langle e_1, \xi, \phi, \rho \rangle$  evaluates to some nonzero value  $v_1$ , the expression  $\text{IF}(e_1, e_2, e_3)$  evaluates to the result of evaluating  $e_2$ . Because  $e_2$  is evaluated in the environment produced by evaluation of  $e_1$ , if  $e_1$  contains side effects, such as assigning to a variable, the results of those side effects are visible to  $e_2$ . The premises of this rule don’t even mention  $e_3$ , because if  $v_1 \neq 0$ ,  $e_3$  is never evaluated.

How do we know that  $e_1$  is evaluated before  $e_2$ ? We look at which environments go where. The side effects of  $e_1$  are captured in environments  $\xi'$  and  $\rho'$ , and these are the environments used to evaluate  $e_2$ . Order of evaluation is determined not by the order in which the premises are written, but by the flow of data (in this case, the environments) through the computation. That said, a rule’s premises are conventionally written in the same order as the evaluations they describe. Any other order would be, shall we say, discourteous.

The rules of Impcore’s semantics belong to a larger family of reasoning techniques called *natural deduction*. This family gives Impcore’s semantics the rest of its name: it is a *big-step, natural-deduction* semantics.

Once you have learned to read rules of inference, you’ll be able to translate them into recursive code. A complete example is done for you in Section 1.6: Impcore’s expression-evaluation judgment is implemented by function eval. Calling  $\text{eval}(e, \xi, \phi, \rho)$  returns  $v$  and has side effects on  $\xi$  and  $\rho$  with the result that  $\langle e, \xi_{\text{before}}, \phi, \rho_{\text{before}} \rangle \Downarrow \langle v, \xi_{\text{after}}, \phi, \rho_{\text{after}} \rangle$ . It works by looking at the form of  $e$  and examining rules that have  $e$ ’s of that form in their conclusions. Evaluation judgments in premises are implemented by recursive calls. For example, to evaluate an IF expression, eval first makes a recursive call to itself to find  $v_1, \xi'$ , and  $\rho'$  such that  $\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$ . Then, if  $v_1 \neq 0$ , it makes another recursive call to find  $v_2, \xi''$ , and  $\rho''$  such that  $\langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle$ . Having satisfied all the premises of rule IFTRUE, it then returns  $v_2$  and the modified environments.

The rules for every possible form of expression (and definition) are presented in the rest of this section.

### 1.5.2 Literal values

Literal values evaluate to themselves without changing any environments.

$$\frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \quad (\text{LITERAL})$$

This rule has no premises, so at a LITERAL node, the recursive implementation of eval terminates.

### 1.5.3 Variables

If a name is bound in the parameter or global environment, then the variable with that name evaluates to the value associated with it by the environment. Otherwise, no rules apply, the machine gets stuck, and the computation does not continue.

Parameters hide global variables. To see if  $x$  is a parameter, we check if  $x \in \text{dom } \rho$ , where  $\text{dom } \rho$  is the domain of  $\rho$ , i.e., the set of names bound by  $\rho$ .

$$\frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle} \quad (\text{FORMALVAR})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle} \quad (\text{GLOBALVAR})$$

The premises of these rules involve only membership tests on environments, not other evaluation judgments, so at a VAR node, the recursive implementation of eval also terminates.

### 1.5.4 Assignment

Assignment follows the same rules as variable lookup: if the name is known in the parameter environment, we change the binding there; otherwise we look in the global environment.

$$\frac{x \in \text{dom } \rho \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \{x \mapsto v\} \rangle} \quad (\text{FORMALASSIGN})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi' \{x \mapsto v\}, \phi, \rho' \rangle} \quad (\text{GLOBALASSIGN})$$

*“May differ,” “must equal,” but not “must differ”*

When we specify evaluation using a judgment form, we are using a form of mathematical logic. In this logic, it can be hard to learn how to use primes and subscripts on the metavariables. To understand what primes and subscripts can say, let’s look at the global-variable environment  $\xi$  in three examples:

- *The general form of judgment*  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$

The general form shows  $\xi$  on the left and  $\xi'$  on the right. Environments  $\xi$  and  $\xi'$  look different, but the notation means only that they *may differ*. The form says, “evaluating an unknown expression  $e$  *may* change a global variable.” What actually happens depends on what rules you use and what judgment you prove.

- *Conclusion of LITERAL*  $\langle \text{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle$

The LITERAL rule proves a judgment that shows  $\xi$  on both the left and the right. Because  $\xi$  *must equal*  $\xi$ , the judgment says, “evaluating a literal expression  $\text{LITERAL}(v)$  *does not* change or add a global variable.”

- *Conclusion GLOBALASSIGN*  $\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi'\{x \mapsto v\}, \phi, \rho' \rangle$

It looks like  $\xi$  and  $\xi'\{x \mapsto v\}$  must differ. But it’s not so:  $\xi$  may equal  $\xi'$ , and  $\xi'$  may map  $x$  to  $v$ . In that case,  $\xi$  and  $\xi'\{x \mapsto v\}$  are equal. Just imagine evaluating  $(\text{set } x \ 0)$  when  $x$  is *already* equal to zero.

Metavariables can say only that two environments “may differ” or “must equal” each other. In the rule for IFTRUE on page 34, for example,  $\xi$  may equal  $\xi'$ , which may equal  $\xi''$ , as in  $(\text{if } (> n 0) n (-\theta n))$ . Or  $\xi$  may equal  $\xi'$ , but they may both differ from  $\xi''$ , as in  $(\text{if } (> n 0) (\text{set sign } 1) (\text{set sign } -1))$ .

Suppose  $\xi$  and  $\xi'$  *must differ*? That can’t be said with primes or subscripts; primes and subscripts can say only “may differ” or “must equal.” To say that two things must differ, use an explicit  $\neq$  sign, as in “ $v_1 \neq 0$ ” or “ $\xi \neq \xi'$ .”

The difference between “may differ” and “must equal” is most important when you write your own derivations. A common beginner’s mistake is to add primes or subscripts as they appear in the rules. But a prime or subscript says “I don’t know; it may differ,” and in a derivation, this is almost always wrong—we do know. Here’s an invalid derivation, intended to describe the evaluation of  $(\text{if } (> n 0) n (-\theta n))$  when  $n$  is 7 (I’ve taken a minor liberty with the notation):

$$\frac{\dots}{\langle (\text{if } (> n 0), \xi, \phi, \rho) \Downarrow \langle 1, \xi, \phi, \rho \rangle \quad 1 \neq 0 \quad \langle n, \xi, \phi, \rho \rangle \Downarrow \langle 7, \xi, \phi, \rho \rangle \rangle} \langle \text{IF}((> n 0), n, (-\theta n)), \xi, \phi, \rho \rangle \Downarrow \langle 7, \xi'', \phi, \rho'' \rangle$$

The conclusion is bogus. Because  $\xi''$  doesn’t appear anywhere else in the derivation, there is nothing that  $\xi''$  must equal, and that means the judgment says, “evaluating the if expression produces 7, and afterward, global variables (and formal parameters) have *arbitrary* values.” To avoid this kind of problem, remember that in any judgment that you prove, the elements of the final state *must equal* something that you specify. Do that and you’ll write good derivations.

The premises of both SET rules involve evaluation judgments on the right-hand side  $e$ , so at a SET node, the recursive implementation of eval always makes a recursive call.

In Impcore, it is possible to assign only to previously defined variables; given a SET node where  $x \notin \text{dom } \rho$  and  $x \notin \text{dom } \xi$ , the machine gets stuck. In many languages, like Awk for example, assignment to an undefined and undeclared variable creates a new global variable (Aho, Kernighan, and Weinberger 1988). The following rule might be used in an operational semantics for Awk:

**1** *An imperative core*  
34

$$\frac{x \notin \text{dom } \rho \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi' \{x \mapsto v\}, \phi, \rho' \rangle} \quad (\text{GLOBALASSIGN for Awk})$$

To spot such subtle differences, you have to read inference rules carefully. In Impcore, it is possible to create new global variables only by means of a VAL definition, as shown below in rule DEFINEGLOBAL (page 37).

### 1.5.5 Control Flow

#### Conditional evaluation

The expression  $\text{IF}(e_1, e_2, e_3)$  first evaluates the expression  $e_1$  to produce value  $v_1$ . If  $v_1$  is nonzero, the result of the IF expression is the result of evaluating  $e_2$ , otherwise, it's the result of evaluating  $e_3$ . It is simplest to have different rules for the two cases.

$$\frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle} \quad (\text{IFTRUE})$$

$$\frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0 \quad \langle e_3, \xi', \phi, \rho' \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle} \quad (\text{IFFALSE})$$

Even though there are two rules with IF in the conclusion, only one can apply at one time, because the premises  $v_1 \neq 0$  and  $v_1 = 0$  are mutually exclusive. This property keeps the evaluation of Impcore programs deterministic.

#### Loops

To evaluate a WHILE loop, we first evaluate the condition  $e_1$  to produce value  $v_1$ . If  $v_1$  is nonzero, evaluation continues with the body  $e_2$ , and then we evaluate the WHILE loop again.

$$\frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle \quad \langle \text{WHILE}(e_1, e_2), \xi'', \phi, \rho'' \rangle \Downarrow \langle v_3, \xi''', \phi, \rho''' \rangle}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi''', \phi, \rho''' \rangle} \quad (\text{WHILEITERATE})$$

The outcome of any evaluation judgment is captured by the four elements to the right of the  $\Downarrow$  arrow. In the conclusion of this rule, those four elements are  $v_3$ ,  $\xi'''$ ,  $\phi$ , and  $\rho'''$ . If you study the rule carefully, you'll see that all the intermediate outcomes contribute to this final outcome, except one: value  $v_2$  is not used. In informal English, that tells us that the body  $e_2$  is evaluated only for its side effects, i.e., for the new environments  $\xi''$  and  $\rho''$ .

If the condition in a WHILE loop evaluates to zero, the WHILE loop terminates, and the value of the loop is also zero.

$$\frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi', \phi, \rho' \rangle} \quad (\text{WHILEEND})$$

In Exercise 23 on page 85, you can prove that even when rule WHILEITERATE is used, the value of a WHILE expression is always zero. A WHILE expression is therefore executed for its side effects.

### *Sequential execution*

BEGIN requires two rules: one for the normal case and one for the empty BEGIN. (I allow empty BEGIN expressions because this possibility simplifies the implementation.)

The empty BEGIN evaluates to zero.

$$\frac{}{\langle \text{BEGIN}(), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle} \quad (\text{EMPTYBEGIN})$$

A nonempty BEGIN evaluates its expressions left to right.

$$\frac{\begin{array}{c} \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \end{array}}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle} \quad (\text{BEGIN})$$

As expected, the values  $v_1, \dots, v_{n-1}$  are ignored, but the environments from evaluating  $e_1$  are used when evaluating  $e_2$ , and so on. By seeing how the environment from one expression is used to evaluate the next, we can understand the order of evaluation of expressions. The order of the premises themselves is irrelevant. For example, the BEGIN rule might equally well have been written this way:

$$\frac{\begin{array}{c} \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\ \langle e_{n-1}, \xi_{n-2}, \phi, \rho_{n-2} \rangle \Downarrow \langle v_{n-1}, \xi_{n-1}, \phi, \rho_{n-1} \rangle \\ \vdots \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \end{array}}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle} \quad (\text{equivalent BEGIN})$$

This equivalent rule still specifies that  $e_1$  is evaluated before  $e_2$ , etc., but when the rule is written this way, it is not as easy to understand.

### 1.5.6 Function Application

#### *User-defined functions*

The description of user-defined functions begins to show one of the advantages of an operational semantics: it is much more concise than an implementation.

$$\frac{\begin{array}{c} \phi(f) = \text{USER}(\langle x_1, \dots, x_n \rangle, e) \\ x_1, \dots, x_n \text{ all distinct} \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\ \langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle \end{array}}{\langle \text{APPLY}(f, e_1, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi', \phi, \rho_n \rangle} \quad (\text{APPLYUSER})$$

As in the BEGIN rule, expressions  $e_1$  through  $e_n$  are evaluated in order. We then create a new, unnamed formal-parameter environment that maps the formal parameter names  $x_1, \dots, x_n$  to the results of evaluating the expressions, and we evaluate the body of the function in this new environment. By reading this rule carefully, we can draw several conclusions:

- The behavior of a function doesn't depend on the function's name, but only on the definition to which the name is bound.
- The body of a function can't get at the formal parameters of its caller, since the body  $e$  is evaluated in a state that does not contain  $\rho_0, \dots, \rho_n$ .
- If a function assigns to its own formal parameters, its caller can't see the new values because the caller has no access to the environment  $\rho'$ .
- After the body of a function is evaluated, the environment  $\rho'$  containing the values of its formal parameters is thrown away. This fact matters to implementors of programming languages, who can use temporary space (in registers and on the stack) to implement formal-parameter environments.

Taken together, these facts mean that the formal parameters of a function are private to that function—neither its caller nor its callees can see them or modify them. The privacy of formal parameters is an essential part of what language designers call “functional abstraction,” which both programmers and implementors rely on.

### *Primitive functions*

Evaluation of primitives is very similar to evaluation of user-defined functions. We evaluate the arguments and then perform the operation.

As a representative of the arithmetic primitives, here is addition:

$$\begin{array}{c} \phi(f) = \text{PRIMITIVE}(+) \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\ -2^{31} \leq v_1 + v_2 < 2^{31} \\ \hline \langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1 + v_2, \xi_2, \phi, \rho_2 \rangle \end{array} \quad (\text{APPLYADD})$$

The final condition on the sum  $v_1 + v_2$  ensures that the result can be represented in a 32-bit signed integer.

As a representative of the comparison primitives, here is the test for equality. As with the `if` expression, we use two rules with mutually exclusive premises ( $v_1 = v_2$  and  $v_1 \neq v_2$ ):

$$\begin{array}{c} \phi(f) = \text{PRIMITIVE}(=) \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\ \hline \frac{v_1 = v_2}{\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 1, \xi_2, \phi, \rho_2 \rangle} \quad (\text{APPLYEQTRUE}) \\ \\ \phi(f) = \text{PRIMITIVE}(=) \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\ \hline \frac{v_1 \neq v_2}{\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 0, \xi_2, \phi, \rho_2 \rangle} \quad (\text{APPLYEQFALSE}) \end{array}$$

The other arithmetic and comparison primitives are so similar to addition and equality that they are not worth including here.

Because our formal model does not include output, the final primitive, `println`, appears to be the identity function.

$$\frac{\phi(f) = \text{PRIMITIVE}(\text{println})}{\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle} \quad (\text{APPLYPRINTLN})$$

$$\langle \text{APPLY}(f, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle \quad \text{while printing } v$$

The semantics of `print` and `printu` are the same.

Modeling the printing primitives formally would not be difficult; we could extend the program state to include a sequence of all characters ever printed. I prefer, however, not to clutter our state and rules with such a list. It is OK to leave the specification of printing informal because our operational semantics is not intended to nail down every last detail; it's intended to convey understanding.

### 1.5.7 Rules for evaluating definitions

The rules above apply to the evaluation of expressions. The remaining rules apply to evaluation of true definitions. These rules use the judgment  $\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \phi' \rangle$ , and they are implemented by the `evaldef` function in Section 1.6.2. Extended definitions don't have semantics, but in Exercises 17 and 21 on pages 82 and 85, you can try designing some.

#### *Variable definition*

Each global variable must be defined before use.  $\text{VAL}(x, e)$  adds the binding  $x \mapsto v$  to the global environment  $\xi'$ , where  $v$  is the result of evaluating  $e$ . When an expression is evaluated at top level, there are no formal parameters, so  $e$  is evaluated with an empty  $\rho$ . When  $x$  is already a global variable,  $\text{VAL}(x, e)$  behaves just like  $\text{SET}(x, e)$ . The `DEFINEGLOBAL` rule is almost identical to the `GLOBALASSIGN` rule, but the `DEFINEGLOBAL` rule does not require  $x \in \text{dom } \xi$ .

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{VAL}(x, e), \xi, \phi \rangle \rightarrow \langle \xi' \{x \mapsto v\}, \phi \rangle} \quad (\text{DEFINEGLOBAL})$$

#### *Function definition*

To process a function definition, the interpreter packages the formal parameters and body as  $\text{USER}(\langle x_1, \dots, x_n \rangle, e)$ , then binds the package into the function-definition environment  $\phi$ . The `DEFINEFUNCTION` rule enforces the invariant that the names of formal parameters are not duplicated.

$$\frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi \{ f \mapsto \text{USER}(\langle x_1, \dots, x_n \rangle, e) \} \rangle} \quad (\text{DEFINEFUNCTION})$$

#### *Top-level expression*

Evaluating an expression can modify the global-variable environment  $\xi$  but not the function environment  $\phi$ . In fact, evaluating a top-level expression  $e$  *always* modifies the global environment  $\xi$ : in addition to whatever is modified during the evaluation of  $e$ , in the final environment, the special variable `it` is bound to  $v$ , the result of evaluating  $e$ .

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{EXP}(e), \xi, \phi \rangle \rightarrow \langle \xi' \{ \text{it} \mapsto v \}, \phi \rangle} \quad (\text{EVALEXP})$$

If you read the rules carefully, you should see that evaluating the “definition” `EXP(e)` has exactly the same effects as evaluating the more conventional definition `VAL(it, e)`.

### *Extended definitions*

As mentioned in the sidebar on page 25, I don’t use formal semantics to say what `use`, `check-expect`, `check-assert`, and `check-error` do. Evaluating a `use` is about the same as evaluating all the definitions contained in the file named by the `use`, then running unit tests. And evaluating a `check-expect`, a `check-assert`, or a `check-error` remembers a unit test. Modeling `use` formally would require a model of files with names and contents. Modeling unit tests would require adding a set of pending unit tests to our abstract machine, plus judgments that describe what it means for a test to succeed or fail. Such things would complicate our formal semantics while distracting us from what the semantics is meant to do: help us understand and compare programming languages.

To understand these issues for yourself, try designing some operational semantics for extended definitions. I recommend that you define the success or failure of `check-expect` (Exercise 17) or the success of `check-error` (Exercise 21). To specify `check-error`, you will need to design a proof system that says what it means for evaluation to halt with an error.

## 1.6 THE INTERPRETER

A bridge language is meant to be small enough to learn, small enough to specify, small enough to implement, but big enough to write interesting programs in. (Or in the case of Impcore, not quite big enough to write interesting programs in.) To write programs, you need an implementation, but why talk about it? Why not bury it in a repository somewhere? Because the implementation of each interpreter has something to tell you about a language and its semantics, and code speaks to you in a way that nothing else can. And when you want to try your own experiments in language design, you can build on or change my code. To make such exploration possible, I can’t just hand you the code; I have to explain it. But I can’t explain all of it—the explanations would add over 300 pages to this book. In this chapter, I explain just the most important parts.

- What you need most is the code that is most relevant to the study of programming languages: the data structures for the crucial abstractions (environments and abstract-syntax trees), and the functions that evaluate expressions and definitions. All that appears in this chapter.
- The crucial functions and abstractions are supported by general-purpose infrastructure: error handling, parsing, printing, test reporting, and so on. That infrastructure is lovingly described in a Supplement to this book.<sup>4</sup> The Supplement also includes some special-purpose modules that are used only to implement Impcore, but are not sufficiently relevant to programming languages to warrant inclusion in the chapter. These pieces include the implementation of function environments, which is nearly identical to the implementation of value environments; functions for printing abstract-syntax trees and values; and functions for running unit tests.

There’s no magic; everything is built on top of the standard C library, which as libraries go, is simple and low-level. With the Supplement in hand, you can under-

---

<sup>4</sup>As these words are written, I don’t know whether the Supplement will be published as a separate volume or will simply be available from the book’s web site.

stand as much or as little as you need, as suits your purposes. Now, how is the code explained?

The code is presented using the Noweb system for *literate programming*, which enables me to split the code into named “code chunks” and to surround each code chunk with textual explanations. The code chunks are written in the order best suited to explaining the interpreter, not the order dictated by a C compiler. Chunks contain source code and references to other chunks. The names of chunks appear italicized and in angle brackets, as in *<evaluate e->ifx and return the result 50b>*. The label “50b” shows where to find the definition: the number identifies the page, and when the page contains more than one chunk, each chunk gets its own lowercase letter, which is appended to the page number. The label also appears on the first line of the definition, in bold. Each definition is shown using an  $\equiv$  sign. A definition can be continued in a later chunk; Noweb concatenates the contents of all definitions of the same chunk. A definition that continues a previous definition is identified by a  $+ \equiv$  sign in place of the  $\equiv$  sign. When a chunk’s definition is continued, Noweb includes pointers to the previous and next definitions, written “ $\triangleleft 48c$ ” and “ $S291a \triangleright$ ”; these pointers appear at the right margin. The notation “(48d)” shows where a chunk is used.

To help you make connections between chunks that appear on different pages, the Noweb system does some *identifier cross-reference*. For example, if you look on page 47 in the right margin you’ll see an alphabetical list of identifiers: a *mini-index*. The mini-index tells you, for example, that function `fetchval` is declared in chunk 42e on page 42. It also tells you that types `Def` and `Exp` are not defined by hand-written code; they are generated automatically. In any mini-index,  $A$  stands for automatically generated code, and  $B$  stands for a basis function from C’s standard library. Finally,  $P$  stands for a *primitive* function that is defined in an interpreter, in which case a chunk number is also given. Examples of primitives appear starting in Chapter 2.

In addition to the mini-indices, there are two large indices at the back of the book. The “concept index” on page 763 provides the sort of index you might find in any textbook. And the “code index” on page S587 allows you to look up the definition—or for C code, the declaration—of any function in any interpreter.

In addition to the general-purpose cross-reference and indexing tools provided by Noweb, I’ve put a short guide to this chapter’s code in Table 1.5. Think of this table as a kind of Rosetta stone that will help you to learn the important parts of the code, and to connect them to the math. (The important parts of the code—the ones I’m calling relevant to the study of programming languages—are all there in the metavariables and math symbols from Section 1.5.)

Once you find the code you’re looking for, you may have an easier time reading it if you know my programming conventions. For example, when I introduce a new type, I use `typedef` to give it a name that begins with a capital letter, like `Name`, or `Exp`, or `Def`. The representation of such a type might be *exposed*, in which case you get to see the entire definition of the type, and you can get at fields of structures and so on. A type whose representation is exposed is called *manifest*. Types `Exp` and `Def` are manifest. The representation of such a type might also be *abstract*, in which case you *can’t* get at the representation. In C, the only way to make a type abstract is to make it a pointer to a named struct, but not to give the fields of the struct. Type `Name` is abstract; you can store a `Name` in a field or a variable, and you can pass a `Name` to a function, but you can’t look inside a `Name` to see how it is represented. (Strictly speaking, *you* can see everything; as explained in Chapter 9, it is your client code, and mine, that is not allowed access to the representations of abstract types.)

I write the names of functions using lowercase letters only, except for some automatically generated functions used to build lists.

Semantics	Concept	Interpreter
$d$	True definition	Def (page 42)
$e$	Expression	Exp (page 43)
$x, f$	Name	Name (page 43)
$v$	Value	Value (page 44)
USER( $\dots$ )	Function	Userfun (page 42), Func (page 44)
PRIMITIVE( $\oplus$ )	Function	Name (page 43), Func (page 44)
$\xi, \rho$	Value environment	Valenv (page 44)
$\phi$	Function environment	Funenv (page 44)
$\langle e, \xi, \phi, \rho \rangle \Downarrow$ $\langle v, \xi', \phi, \rho' \rangle$	Expression evaluation	$\text{eval}(e, \xi, \phi, \rho) = v$ , with $\xi$ and $\rho$ updated to $\xi'$ and $\rho'$ (page 48)
$\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \rho' \rangle$	Definition evaluation	$\text{evaldef}(e, \xi, \phi, \text{echo})$ updates $\xi$ to $\xi'$ and $\phi$ to $\phi'$ (page 54)
$x \in \text{dom } \rho$	Definedness	isvalbound (page 45)
$f \in \text{dom } \phi$	Definedness	isfunbound (page 45)
$\rho(x), \xi(x)$	Lookup	fetchval (page 45)
$\phi(f)$	Lookup	fetchfun (page 45)
$\rho\{x \mapsto v\}$	Binding	bindval (page 45)
$\phi\{f \mapsto \dots\}$	Binding	bindfun (page 45)

Table 1.5: Correspondence between Impcore semantics and code

To the degree that C permits, I distinguish interfaces from implementations. An interface typically includes some or all of these elements:

- Types, which may be manifest or abstract
- Invariants of manifest types, if any
- Prototypes of functions
- Documentation explaining how the types and functions should be used

An *atypical* interface might also include declarations of global variables, macros, or other arcana. But no interface, typical or atypical, ever includes the implementations of its functions.

Interface documentation explains not only how to use functions but also what happens when a function is used incorrectly; in particular, it explains who is responsible for detecting or avoiding an error. A *checked run-time error* is a mistake that the implementation guarantees to detect; a typical example would be passing a NULL pointer to a function. The implementation need not *recover* from a checked error, and indeed, many of my implementations simply halt with assertion failures.

An *unchecked* run-time error is more insidious; this is a mistake that it is up to the C programmer to avoid. If client code causes an unchecked run-time error, the implementation provides no guarantees; anything can happen. Unchecked run-time errors are part of the price we pay for programming in C.

Once you've read and understood an interface, you should be able to use its functions without needing to look at their implementations. But of course the crucial implementations, of functions like eval and evaldef, are intended for you to

## Safety

A language in which all errors are checked is called *safe*. Safety is usually implemented by a combination of compile-time and run-time checking. Popular safe languages include Awk, C#, Haskell, Go, Java, JavaScript, Lua, ML, Perl, Python, Ruby, Rust, Scheme, and Smalltalk. Another way to characterize a safe language is that “there are no unexplained core dumps”; a program that halts always issues an informative error message.

A language that permits unchecked errors is called *unsafe*. Unsafe languages put an extra burden on the programmer, but they provide extra expressive power. This extra power is needed to write things like garbage collectors and device drivers; systems programming languages, like Bliss and C, have historically been unsafe. C++ is an anomaly: it is ostensibly intended for high-level problem-solving, but it is nevertheless unsafe.

A few well-designed systems-programming languages are safe by default, but have unsafe features that can be turned on explicitly at need, usually by a keyword `UNSAFE`. The best known of these may be Cedar and Modula-3.

look at. When you start looking at my implementations, you’ll see I follow a couple of conventions there as well:

- Within reason, I narrow the scope of each local variable to the region in which it is used, rather than allowing local variables to scope over an entire function definition.
- When possible, I initialize each variable where it is declared, just like a `val` definition in Impcore.

On to the code! The presentation begins with the interfaces in Section 1.6.1. These interfaces include not only the interfaces associated with the evaluator, but also some interfaces associated with general-purpose utility code from Appendix F. You have to understand these interfaces in order to understand the code, but the most interesting stuff—the implementation of Impcore’s operational semantics—is the implementation in Section 1.6.2, which starts on page 48.

### 1.6.1 Interfaces

The part of the Impcore interpreter that interests us the most is the evaluator, which implements the operational semantics. That evaluator is supported by interfaces for the central structures of a programming language: syntax, names, values, environments, and lists thereof. I also show a little of the underlying infrastructure: interfaces used for printing and for reporting errors.

#### *Interface to abstract syntax: manifest types and creator functions*

The abstract-syntax interface exposes all the representations of definitions, expressions, and so on. In addition to the exposed representations, the interface also provides convenient *creator functions* which are used to build abstract-syntax trees.

The type of an abstract-syntax tree is a *sum type*, also known as a *discriminated-union type*. Any value of such a type is one of a list of alternatives, which are given explicitly in the definition of the sum type. Sum types play an essential role in symbolic computing, but they are not directly supported in C, which provides only an

*undiscriminated* union, sometimes also called an “unsafe” union. A C union provides a list of alternatives, but there is no way to tell which alternative was last assigned to the union. Accordingly, to represent a sum type in C, we require both a union, to hold the alternatives, and a *discriminant* (or “tag”), to show which alternative is actually in use. For convenience, I put the union and the tag together in a structure. I call the tag `alt` (for “alternative”), but I don’t name the union. By using an *anonymous* union, I make it possible for code to refer directly not only to the tag, as in `e->alt`, but also to any alternative by name, as in `e->var`.

I use the name of each alternative in two places: in an enumeration type, which lists the possible values of `alt`, and in the union type, which lists the possible members of the union. And in keeping with common C practice, when a name is used as an enumeration literal, I write it using all capital letters, but it is used to name a member of a union, I write the lowercase equivalent.

My representation of sum types is safe and systematic, but it requires unwieldy notation. For example, to refer to the name of a function in a definition `d`, I must write `d->define.name`. Although something like `d->name` might be more convenient, referring to `define` enables the compiler to keep me from making mistakes; in particular, I can’t accidentally grab a name field from the wrong part of the sum. Other languages provide better support for sum types, with better notation. For example, C++ and some nonstandard dialects of C provide anonymous unions and structures, which can provide less unwieldy notation for the same representation of sum types. And Standard ML provides direct support for sum types, including a convenient pattern-matching notation (Chapter 5).

Using my conventions, here is the exposed representation of a true definition `Def`, a simplified version of which appears in chunk *(simplified example of abstract syntax for Impcore 27a)*. A `Def` is a sum type with three alternatives, called `VAL`, `EXP`, and `DEFINE`.

**42a.** *(type and structure definitions for Impcore 42a)*≡  
44d▷

```
typedef struct Userfun Userfun;
struct Userfun { Namelist formals; Exp body; };

typedef struct Def *Def;
typedef enum { VAL, EXP, DEFINE } Defalt;
struct Def {
    Defalt alt;
    union {
        struct { Name name; Exp exp; } val;
        Exp exp;
        struct { Name name; Userfun userfun; } define;
    };
};
```

Writing such type definitions and creator functions by hand is tedious and prone to error, especially when type definitions change. I therefore generate them automatically. Appendix H shows an ML program that generates such code from the following descriptions.

**42b.** *(definition.t 42b)*≡

```
Userfun = (Namelist formals, Exp body)
Def*   = VAL          (Name name, Exp exp)
      | EXP          (Exp)
      | DEFINE       (Name name, Userfun userfun)
```

A valid `Userfun` satisfies the invariant that the names in `formals` are all distinct.

Here is a description of the abstract syntax for `Exp`,<sup>5</sup> which you might wish to compare with the concrete syntax given for `exp` on page 18:

43a.  $\langle \text{exp.t } 43a \rangle \equiv$

```
Exp* = LITERAL (Value)
| VAR      (Name)
| SET      (Name name, Exp exp)
| IFX      (Exp cond, Exp truex, Exp falsex)
| WHILEX   (Exp cond, Exp exp)
| BEGIN    (Explist)
| APPLY    (Name name, Explist actuals)
```

§1.6  
The interpreter

43

The descriptions above are slightly elaborated versions of *(simplified example of abstract syntax for Impcore 27d)*. I use similar descriptions for much of the C code in this book.

The true definitions and the expressions are the essential elements of abstract syntax, and if you understand how they work, it will help you connect the operational semantics and the code. Impcore's *extended* definitions, including unit tests, are described in the Supplement.

#### Interface to names: an abstract type

Programs are full of names. To make it easy to compare names and look them up in tables, I define an abstract type to represent them. The essential feature of this abstract type is that a name has no internal structure; names are atomic objects which can efficiently be compared for equality.

For real implementations, it is convenient to build names from strings. Unlike C strings, names are immutable, and they can be compared using pointer equality.

43b.  $\langle \text{shared type definitions } 43b \rangle \equiv$

(S290)

```
typedef struct Name *Name;
typedef struct Namelist *Namelist; // list of Name
```

Pointer comparison is built into C, but I provide two other operations on names.

43c.  $\langle \text{shared function prototypes } 43c \rangle \equiv$

(S290) 46c ▷

```
Name strtoname(const char *s);
const char *nametostr(Name x);
```

These functions satisfy the following algebraic laws:

```
strcmp(s, nametostr(strtoname(s))) == 0
strcmp(s, t) == 0 if and only if strtoname(s) == strtoname(t)
```

Informally, the first law says if you build a name from a string, `nametostr` returns a copy of your original string. The second law says you can compare names using pointer equality.

Because `nametostr` returns a string of type `const char*`, a client of `nametostr` cannot modify that string without subverting the type system. Modification of the string is an unchecked run-time error.

The `Name` type is abstract; the interface hides the members of `struct Name`. A client should create new values of type `Name*` only by calling `strtoname`; to do so by casting other pointers is a violation of the type system and an unchecked run-time error.

nametostr S293e  
strtoname S294a

<sup>5</sup>Why are the alternatives for `if` and `while` named `IFX` and `WHILEX`, not `IF` and `WHILE`? Because corresponding to each alternative, there is a field of a union that uses the same name in *lower case*. For example, if `e` is a `LITERAL` expression, the literal `Value` is found in field `e->literal`. And I can't name a structure field `if` or `while`, because the names `if` and `while` are *reserved words*—they may be used only to mark C syntax. So I call these alternatives `IFX` and `WHILEX`, which I encourage you to think of as “`if-expression`” and “`while-expression`.” For similar reasons, the two branches of the `IFX` are called `truex` and `falsex`, not `true` and `false`. And in Chapter 2, you'll see `LETX` and `LAMBDAx` instead of `LET` and `LAMBDA`, so that I can write an interpreter for  $\mu$ Scheme in  $\mu$ Scheme.

The value interface defines the type of value that our expressions evaluate to. Impcore supports only integers. A `Valuelist` is a list of `Value`s.

**44a.** *(type definitions for Impcore 44a)*  $\equiv$  (S290) 44b ▷

```
typedef int32_t Value;
typedef struct Valuelist *Valuelist;      // list of Value
```

### Interface to functions, both user-defined and primitive

In the Impcore interpreter, the type “function” is another discriminated-union type. There are two alternatives: user-defined functions and primitive functions. Just like the operational semantics, which represents a user-defined function as  $\text{USER}(\langle x_1, \dots, x_n \rangle, e)$ , the interpreter represents a user-defined function as a pair containing formals and body. The interpreter represents each primitive by its name.

**44b.** *(type definitions for Impcore 44a)*  $+ \equiv$  (S290) ▲ 44a 44f ▷

```
typedef struct Funclist *Funclist; // list of Func
```

**44c.** *(fun.t 44c)*  $\equiv$

```
Func = USERDEF  (Userfun)
      | PRIMITIVE (Name)
```

From this description, these type and structure definitions are generated automatically:

**44d.** *(type and structure definitions for Impcore 42a)*  $+ \equiv$  △ 42a

```
typedef struct Func Func;
typedef enum { USERDEF, PRIMITIVE } Funcalt;
struct Func { Funcalt alt; union { Userfun userdef; Name primitive; } ; };
```

Also generated automatically are these prototypes for creator functions.

**44e.** *(function prototypes for Impcore 44e)*  $\equiv$  (S290) 45a ▷

```
Func mkUserdef(Userfun userdef);
Func mkPrimitive(Name primitive);
```

### Interface to environments: more abstract types

In the operational semantics, the environments  $\rho$  and  $\xi$  hold values, and the environment  $\phi$  holds functions. To represent these two kinds of environments, C offers these choices:

- We can define one C type for environments that hold a `Value` and another for environments that hold a `Func`, and we can define two versions of each function. This choice guarantees type safety, but requires duplication of code.
- We can define a single C type for environments that hold a `void*` pointer, define a single version of each function, and use type casting to convert a `void*` to a `Value*` or `Func*` as needed. This choice duplicates no code, but it is unsafe; if we accidentally put a `Value*` in an environment intended to hold a `Func*`, it is an error that neither the C compiler nor the run-time system can detect.

In the interests of safety, I duplicate code. Chapter 5 shows how in another implementation language, ML, we can use *polymorphism* to achieve type safety without duplicating code.

**44f.** *(type definitions for Impcore 44a)*  $+ \equiv$  (S290) ▲ 44b

```
typedef struct Valenv *Valenv;
typedef struct Funenv *Funenv;
```

A new environment may be created by passing a list of names and a list of associated values or function definitions to `mkValenv` or `mkFunenv`. For example, calling `mkValenv(⟨x1, …, xn1, …, vn⟩)` returns  $\{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ . If the two lists are not the same length, it is a checked run-time error.

**45a.** *(function prototypes for Impcore 44e) +≡*

(S290) ▷ 44e 45b ▷

```
Valenv mkValenv(Namelist vars, Valuelist vals);
Funenv mkFunenv(Namelist vars, Funclist defs);
```

To retrieve a value or function definition, we use `fetchval` or `fetchfun`. In the operational semantics, I write the lookup  $\text{fetchval}(x, \rho)$  simply as  $\rho(x)$ .

**45b.** *(function prototypes for Impcore 44e) +≡*

(S290) ▷ 45a 45c ▷

```
Value fetchval(Name name, Valenv env);
Func fetchfun(Name name, Funenv env);
```

§1.6  
The interpreter

45

If the given name does not appear in the environment, it is a checked run-time error. To avoid such errors, we can call `isvalbound` or `isfunbound`; they return 1 if the given name is in the environment, and 0 otherwise. Formally,  $\text{isvalbound}(x, \rho)$  is written  $x \in \text{dom } \rho$ .

**45c.** *(function prototypes for Impcore 44e) +≡*

(S290) ▷ 45b 45d ▷

```
bool isvalbound(Name name, Valenv env);
bool isfunbound(Name name, Funenv env);
```

To add new bindings to an environment, use `bindval` and `bindfun`. Unlike previous operations on environments, `bindval` and `bindfun` cannot be specified as pure functions. Instead, `bindval` and `bindfun` *mutate* their environments, replacing the old bindings with new ones. Calling `bindval(x, v, ρ)` is equivalent to performing the assignment  $\rho := \rho \{x \mapsto v\}$ . Because  $\rho$  is a *mutable* abstraction, the caller can see the modifications to the environment.

**45d.** *(function prototypes for Impcore 44e) +≡*

(S290) ▷ 45c 45e ▷

```
void bindval(Name name, Value val, Valenv env);
void bindfun(Name name, Func fun, Funenv env);
```

These functions can be used to replace existing bindings or to add new ones.

### Interface to the evaluator

The evaluator's interface is not very interesting: it is the *implementation*, which starts on page 48, that is interesting. The evaluator works with abstract syntax and values, whose representations are exposed, and with names and environments, whose representations are not exposed. The interface exports functions `eval` and `evaldef`, which evaluate expressions and true definitions, respectively. (Function `readevalprint`, which evaluates extended definitions, is described in the Supplement.)

Function `eval` corresponds to the  $\Downarrow$  relation in our operational semantics. For example,  $\text{eval}(e, \xi, \phi, \rho)$  finds a  $v, \xi'$ , and  $\rho'$  such that  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , assigns  $\rho := \rho'$  and  $\xi := \xi'$ , and returns  $v$ . The function `evaldef` similarly corresponds to the  $\rightarrow$  relation.

**45e.** *(function prototypes for Impcore 44e) +≡*

(S290) ▷ 45d

```
Value eval (Exp e, Valenv globals, Funenv functions, Valenv formals);
void evaldef(Def d, Valenv globals, Funenv functions, Echo echo_level);
```

bindfun	S301b
bindval	S5c
type Def	A
type Echo	S289b
eval	48d
evaldef	54a
type Exp	A
fetchfun	S301a
fetchval	56b
type Func	A
isfunbound	S300f
isvalbound	56a
mkFunenv	S300d
mkValenv	S5c
type Name	43b
type Namelist	43b
type Userfun	A

The `echo_level` parameter to `evaldef` controls printing; if it is ECHOES, `evaldef` prints the values and names of top-level expressions and functions. If `echo_level` is NO\_ECHOES, nothing is printed.

The types of the functions can tell us something, just as in the operational semantics, the form of a judgment can tell us something. Here, the result types tell us that evaluating an `Exp` produces a value, but evaluating a `Def` does not. Both kinds of evaluations can have side effects on environments.

The interpreter uses lists of names, values, functions, expressions, unit tests, and parenthesized inputs. As with environments, C offers two choices. We can define a distinct list type for each item type, getting safety at the cost of some code bloat, or we can use lists of `void*`, getting compact code at the cost of losing safety. Again, I choose safety. The interpreter defines types `Parlist`, `UnitTestlist`, `Explist`, `Namelist`, `Valuelist`, and `Funclist`.

Each list type is a recursive data type. A list is either empty or is a pointer to a pair (`hd`, `tl`), where `hd` is the first element of the list and `tl` is the rest of the list. An empty list is represented by a null pointer. Here's an example definition:

**46a.** *(example structure definitions for Impcore 46a)≡*

```
struct Explist {
    Exp hd;
    struct Explist *tl;
};
```

The interpreter's list definitions are generated by a Lua script, which searches header files for lines of the form

```
typedef struct Foo *Foo; // list of Foo
```

For each type of list, the script also generates a length function, an extractor, a creator function, and a print function. These functions are named `lengthNL`, `nthNL`, `mkNL`, and `printNL`, where  $N$  is the first letter of the list type. The length of the NULL list is zero; the length of other lists is the number of elements. Elements are numbered from zero, and asking for `nthNL(xs, n)` when  $n \geq \text{lengthNL}(xs)$  is a checked run-time error. Calling `mkNL` creates a fresh list with the new element at the head; it does not mutate the old list.

Here are the prototypes for the expression-list functions:

**46b.** *(example function prototypes for Impcore 46b)≡*

```
int     lengthEL(Explist es);
Exp    nthEL    (Explist es, unsigned n);
Explist mkEL    (Exp e, Explist es);
Explist popEL   (Explist es);
```

Definitions and function prototypes for all the list types can be found in the interpreter's `all.h` file. Because of the replication, this code is tedious to read, but generated the code automatically makes the tedium bearable. As shown in Chapter 5, ML's polymorphism enables a simpler solution.

### Interface to infrastructure: Printing

After every definition, the interpreter prints a name or a value. And if an error occurs or a unit test fails, the interpreter may also print an expression or a definition. For printing, the C standard library provides `printf`, but `printf` and its siblings are not well suited to print messages that include renderings of expressions or definitions. To address this problem, this interface defines functions `print` and `fprint`, which support direct printing of `Names`, `Exps`, and so on.

**46c.** *(shared function prototypes 43c) +≡* (S290) ▷ 43c 47 ▷

```
void print (const char *fmt, ...); // print to standard output
void fprintf(FILE *output, const char *fmt, ...); // print to given file
```

Functions `print` and `fprint` resemble `printf` and `fprintf`: the `fmt` parameter is a “format string” that contains “conversion specifications.” Our conversion specifications are like those used by `printf`, but much simpler. A conversion specification is two characters: a percent sign followed by a character like `d` or `s`, which is called a *conversion specifier*. Our conversion specifications don’t contain minus signs, numbers, or dots. The Impcore interpreter uses the conversion specifications shown here in Table 1.6. By convention, lowercase specifiers print individual values and uppercase specifiers print lists. To print a `Def`, I cannot use `%d`, because `%d` is firmly established as a specification for printing print decimal integers. Instead I use `%t`, for “top-level,” which is where a `Def` appears.

<code>%%</code>	Print a percent sign
<code>%d</code>	Print an integer in decimal format
<code>%e</code>	Print an <code>Exp</code>
<code>%f</code>	Print a <code>Func</code>
<code>%E</code>	Print an <code>Explist</code> (list of <code>Exp</code> )
<code>%n</code>	Print a <code>Name</code>
<code>%N</code>	Print a <code>Namelist</code> (list of <code>Name</code> )
<code>%p</code>	Print a <code>Par</code> (see Appendix K)
<code>%P</code>	Print a <code>Parlist</code> (list of <code>Par</code> )
<code>%s</code>	Print a <code>char*</code> (string)
<code>%t</code>	Print a <code>Def</code> ( <code>t</code> stands for “top-level”, which is where definitions appear)
<code>%v</code>	Print a <code>Value</code>
<code>%V</code>	Print a <code>Valuelist</code> (list of <code>Value</code> )

Table 1.6: Conversion specifiers for impcore

Functions `print` and `fprint` are *unsafe*; if you pass an argument that is not consistent with the corresponding conversion specifier, it is an *unchecked* run-time error.

#### *Interface to infrastructure: Error handling*

When given a faulty Impcore program, the interpreter complains and recovers by calling a function in an error-handling interface. In general, a fault occurs whenever a program is ill formed, ill typed, or ill behaved, but in Impcore, there is no static type system, so the interpreter reports only ill-formed and ill-behaved programs:

- When it detects an ill-formed program during parsing, the interpreter signals a *syntax error* by calling `synerror`.
- When it detects an ill-behaved program at run time, the interpreter signals a *run-time error* by calling `runerror`.

Before initiating error recovery, each error-signaling function prints a message, and for that reason, an error-signaling function’s interface resembles `print`. But because different information is available at parse time and at run time, the two functions have different interfaces.

<code>fprint</code>	S190c
<code>print</code>	S190b
<code>runerror</code>	S194a

The simpler of the two error-signaling functions is `runerror`. During normal operation, `runerror` prints to standard error and then `longjmps` to `errorjmp`.

47. *<shared function prototypes 43c>* +≡ (S290) ▷46c 48a▷

```
void runerror (const char *fmt, ...);
extern jmp_buf errorjmp;           // longjmp here on error
```

During unit testing, `runerror` operates in *testing* mode, and it behaves a little differently. The details are in Section F.4.1 on page S194.

Function `synerror` is like `runerror`, except that before its format string, it takes an argument of type `Sourceloc`, which tracks the source-code location being read at the time of the error. The location can be printed as part of the error message.

**48a.** `<shared function prototypes 43c> +≡` (S290) ▲47 48b▷  
`void synerror (Sourceloc src, const char *fmt, ...);`

Error *handling*, as opposed to error signaling, is implemented by calling `setjmp` on `errorjmp`. Client code must ensure that `setjmp` is called before any error-signaling function. It is an unchecked run-time error to call `runerror` or `synerror` except while a `setjmp` involving `errorjmp` is active on the C call stack.

One common run-time error is that an Impcore function is called with the wrong number of arguments. That error is detected by function `checkargc`. The Exp holds the call in which the error might occur.

**48b.** `<shared function prototypes 43c> +≡` (S290) ▲48a  
`void checkargc(Exp e, int expected, int actual);`

### 1.6.2 Implementation of the evaluator

The part of the interpreter that illustrates Impcore's semantics is the evaluator: functions `eval` and `evaldef` evaluate expressions and true definitions, respectively.

#### Evaluating expressions

The function `eval` implements the  $\Downarrow$  relation from the operational semantics. Calling `eval( $e, \xi, \phi, \rho$ )` finds a  $v, \xi', \phi, \rho'$  such that  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , assigns  $\rho := \rho'$  and  $\xi := \xi'$ , and returns  $v$ . Because it is unusual to use Greek letters in C code, I have chosen these names instead:

$\xi$     globals  
 $\phi$     functions  
 $\rho$     formals

Function `eval` is mutually recursive with a private helper function, `evallist`:

**48c.** `<eval.c 48c> ≡` 48d▷  
`static Valuelist evallist(Explist es, Valenv globals, Funenv functions,  
                              Valenv formals);`

Like any other recursive interpreter, `eval` implements the operational semantics by examining the inference rules from the bottom up. The first step in evaluating  $e$  is to discover what the syntactic form of  $e$  is, by looking at the tag (the `alt` field) in the discriminated union.

**48d.** `<eval.c 48c> +≡` △48c 52a▷  
`Value eval(Exp e, Valenv globals, Funenv functions, Valenv formals) {  
    switch (e->alt) {  
        case LITERAL: (evaluate e->literal and return the result 49a)  
        case VAR:     (evaluate e->var and return the result 49b)  
        case SET:     (evaluate e->set and return the result 50a)  
        case IFX:     (evaluate e->ifx and return the result 50b)  
        case WHILEX: (evaluate e->whilex and return the result 50c)  
        case BEGIN: (evaluate e->begin and return the result 51a)  
        case APPLY: (evaluate e->apply and return the result 51b)  
    }  
    assert(0);  
}`

The assertion at the end of `eval` might seem superfluous, but it isn't; it helps protect me, and you, from mistakes. If I forget a case in the `switch`, the evaluator will halt with an error message, rather than silently do something unpredictable. Another way to protect yourself from mistakes is to turn on all compiler warnings, which I do; calling `assert(0)`, which the compiler knows can never return, keeps the C compiler from issuing a warning that `eval` might not return.

As I hope is evident from the code, function `eval` works by case analysis over the syntactic forms of `Exp`. To write the implementation, we consider one syntactic form at a time, and we consult the operational semantics to find the rules that have the form on the left-hand sides of their *conclusions*.

Only one rule has `LITERAL` in its conclusion.

$$\frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \quad (\text{LITERAL})$$

The implementation simply returns the literal value.

**49a.** *(evaluate e->literal and return the result 49a) ≡* (48d)  
`return e->literal;`

Two rules have variables in their conclusions.

$$\frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle} \quad (\text{FORMALVAR})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle} \quad (\text{GLOBALVAR})$$

We know which rule to use by checking  $x \in \text{dom } \rho$ , which is implemented in C by calling `isvalbound(e->var, formals)`. If  $x \notin \text{dom } \rho$  and  $x \notin \text{dom } \xi$ , the operational semantics gets stuck—so the interpreter issues an error message. Less formally, we look up the variable by checking first the local environment and then the global environment.

**49b.** *(evaluate e->var and return the result 49b) ≡* (48d)  
`if (isvalbound(e->var, formals))  
 return fetchval(e->var, formals);  
else if (isvalbound(e->var, globals))  
 return fetchval(e->var, globals);  
else  
 runerror("unbound variable %n", e->var);`

The call to `runerror` illustrates the convenience of the extensible printer; I use `%n` to print a `Name` directly, without needing to convert it to a string.

Setting a variable is very similar. Again there are two rules, and again we distinguish by looking at the domain of  $\rho$  (`formals`).

$$\frac{x \in \text{dom } \rho \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \{x \mapsto v\} \rangle} \quad (\text{FORMALASSIGN})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi' \{x \mapsto v\}, \phi, \rho' \rangle} \quad (\text{GLOBALASSIGN})$$

checkargc	S195d
checkoverflow	S197a
evalalist	52a
type Exp	A
type Explist	S288c
fetchval	45b
type Funenv	44f
runerror	47
type Sourceloc	
	S289d
synerror	S194b
type Valenv	44f
type Value	44a
type Valuelist	44a

Because both rules require the premise  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , we evaluate the right-hand side first and put the result in  $v$ .

**50a.** *(evaluate e->set and return the result 50a)≡* (48d)

```

{
    Value v = eval(e->set.exp, globals, functions, formals);

    if (isvalbound(e->set.name, formals))
        bindval(e->set.name, v, formals);
    else if (isvalbound(e->set.name, globals))
        bindval(e->set.name, v, globals);
    else
        runerror("tried to set unbound variable %n in %e", e->set.name, e);
    return v;
}

```

To evaluate `ifx`, we again have two rules.

$$\frac{\begin{array}{c} \langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle \\ \langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle \end{array}}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle} \text{ (IFTRUE)}$$

$$\frac{\begin{array}{c} \langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0 \quad \langle e_3, \xi', \phi, \rho' \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle \\ \langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle \end{array}}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle} \text{ (IFFALSE)}$$

Both rules have the same first premise:  $\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$ . To get  $v_1$ ,  $\xi'$ , and  $\rho'$ , we call `eval(e->ifx.cond, globals, functions, formals)` recursively. This call is safe only because the new environments  $\xi'$  and  $\rho'$  are used in the third premises of *both* rules. If  $\xi'$  and  $\rho'$  were not always used, we would have had to make copies and pass the copies to the recursive call.

Once we have  $v_1$ , testing it against zero tells us which rule to use, and the third premises of both rules require recursive calls to `eval`. The expression  $e_2$  is `e->ifx.true`;  $e_3$  is `e->ifx.false`.

**50b.** *(evaluate e->ifx and return the result 50b)≡* (48d)

```

if (eval(e->ifx.cond, globals, functions, formals) != 0)
    return eval(e->ifx.true, globals, functions, formals);
else
    return eval(e->ifx.false, globals, functions, formals);

```

There are also two rules for while loops.

$$\frac{\begin{array}{c} \langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \\ \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle \quad \langle \text{WHILE}(e_1, e_2), \xi'', \phi, \rho'' \rangle \Downarrow \langle v_3, \xi''', \phi, \rho''' \rangle \end{array}}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi''', \phi, \rho''' \rangle} \text{ (WHILEITERATE)}$$

$$\frac{\begin{array}{c} \langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0 \\ \langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi', \phi, \rho' \rangle \end{array}}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi', \phi, \rho' \rangle} \text{ (WHILEEND)}$$

A systematic translation of  $\langle \text{WHILE}(e_1, e_2), \xi'', \phi, \rho'' \rangle \Downarrow \langle v_3, \xi''', \phi, \rho''' \rangle$ , where it appears as a premise of the first rule, would produce a recursive call to `eval(e, ...)`. Because we know  $e$  is a while loop, however, we can turn the recursion into iteration, writing the interpretation of Impcore's while loop as a while loop in C. Such an optimization is valuable because during the execution of a long while loop in Impcore, it can keep the C stack from overflowing.

**50c.** *(evaluate e->whilex and return the result 50c)≡* (48d)

```

while (eval(e->whilex.cond, globals, functions, formals) != 0)
    eval(e->whilex.exp, globals, functions, formals);
return 0;

```

For the `begin` expression, I use a `for` loop to evaluate all the premises in turn. Local variable `lastval` remembers the value of the last expression in the `begin`. In the pointless case where the `begin` doesn't contain any expressions, I have crafted the operational semantics to match the implementation.

$$\langle \text{BEGIN}(), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle \quad (\text{EMPTYBEGIN})$$

$$\begin{aligned} \langle e_1, \xi_0, \phi, \rho_0 \rangle &\Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle &\Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \end{aligned}$$

§1.6  
*The interpreter*

51

$$\vdots$$

$$\langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle$$

$$\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \quad (\text{BEGIN})$$

**51a.** *(evaluate `e->begin` and return the result 51a) ≡* (48d)

```
{  
    Value lastval = 0;  
    for (Explist es = e->begin; es; es = es->t1)  
        lastval = eval(es->hd, globals, functions, formals);  
    return lastval;  
}
```

There are many rules for applying functions, but I divide them into two classes. One class contains only the rule for user-defined functions; the other class contains the rules for primitives. To apply the function named  $f$ , the interpreter looks at the form of  $\phi(f)$ . In the code,  $f$  is `e->apply.name`.

**51b.** *(evaluate `e->apply` and return the result 51b) ≡* (48d)

```
{  
    Func f;  
    <make f the function denoted by e->apply.name, or call runerror 51c>  
    switch (f.alt) {  
        case USERDEF: <apply f.userdef and return the result 52b>  
        case PRIMITIVE: <apply f.primitive and return the result 52c>  
        default: assert(0);  
    }  
}
```

If the lookup  $\phi(f)$  fails, we call `runerror`.

**51c.** *(make f the function denoted by e->apply.name, or call runerror 51c) ≡* (51b)

```
if (!isfunbound(e->apply.name, functions))  
    runerror("call to undefined function %n in %e", e->apply.name, e);  
f = fetchfun(e->apply.name, functions);
```

Applying a user-defined function has something in common with `begin`, because arguments  $e_1, \dots, e_n$  have to be evaluated. The difference is that `begin` keeps only result  $v_n$  (in variable  $v$  in chunk 51a), where function evaluation keeps all the result values, to bind into a new environment.

$$\begin{aligned} \phi(f) &= \text{USER}(\langle x_1, \dots, x_n \rangle, e) \\ &\quad x_1, \dots, x_n \text{ all distinct} \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle &\Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ &\vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle &\Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\ \langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\} \rangle &\Downarrow \langle v, \xi', \phi, \rho' \rangle \\ \langle \text{APPLY}(f, e_1, \dots, e_n), \xi_0, \phi, \rho_0 \rangle &\Downarrow \langle v, \xi', \phi, \rho_n \rangle \quad (\text{APPLYUSER}) \end{aligned}$$

bindval	45d
eval	45e
type Explist	S288c
fetchfun	45b
formals	48d
type Func	A
functions	48d
globals	48d
runerror	47
type Value	44a

To produce values  $v_1, \dots, v_n$ , I define the auxiliary function `evalist`, which is given  $e_1, \dots, e_n$  along with  $\xi_0, \phi$ , and  $\rho_0$ . It evaluates  $e_1, \dots, e_n$  in order, and it

mutates the environments so that when it is finished,  $\xi = \xi_n$  and  $\rho = \rho_n$ . Finally, `evalist` returns the list  $v_1, \dots, v_n$ .

**52a.**  $\langle eval.c\ 48c \rangle + \equiv$   $\triangleleft 48d\ 54a \triangleright$

```
static Valuelist evalist(Explist es, Valenv globals, Funenv functions,
                         Valenv formals)
{
    if (es == NULL) {
        return NULL;
    } else {
        Value v = eval(es->hd, globals, functions, formals);
        return mkVL(v, evalist(es->tl, globals, functions, formals));
    }
}
```

The rules of Impcore require that `es->hd` be evaluated before `es->tl`. To ensure the correct order of evaluation, we call `eval(es->hd, ...)` and `evalist(es->tl, ...)` in separate C statements. Writing both calls as parameters to `mkVL` would be a mistake because C makes no guarantees about the order in which the actual parameters of a function are evaluated.

The premises of the `APPLYUSER` rule require that the list of  $f$ 's formal parameters be the same length as the list of actual parameters in the call. I let `xs` represent the formals  $x_1, \dots, x_n$  and `vs` represent the actuals  $v_1, \dots, v_m$ . If the formals and actuals are the same length, so  $m = n$ , I use `mkValenv(xs, vs)` to create a fresh environment  $\{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$  in which to evaluate the body.

**52b.**  $\langle apply\ f.\ userdef\ and\ return\ the\ result\ 52b \rangle \equiv$  (51b)

```
{
    Namelist xs = f.userdef.formals;
    Valuelist vs = evalist(e->apply.actuals, globals, functions, formals);
    checkargc(e, lengthNL(xs), lengthVL(vs));
    return eval(f.userdef.body, globals, functions, mkValenv(xs, vs));
}
```

Impcore has few primitive operators, and they are simple. The printing primitives are handled separately from the arithmetic primitives. More general techniques for implementing primitives, which are appropriate for larger languages, are shown in the implementation of  $\mu$ Scheme in Section 2.12.2 on page 156.

**52c.**  $\langle apply\ f.\ primitive\ and\ return\ the\ result\ 52c \rangle \equiv$  (51b)

```
{
    Valuelist vs = evalist(e->apply.actuals, globals, functions, formals);
    if (f.primitive == strtoname("print"))
        (apply Impcore primitive print to vs and return 52d)
    else if (f.primitive == strtoname("println"))
        (apply Impcore primitive println to vs and return S300a)
    else if (f.primitive == strtoname("printu"))
        (apply Impcore primitive printu to vs and return S300b)
    else
        (apply arithmetic primitive to vs and return 53a)
}
```

Primitive `print` is shown here; `println` and `printu` appear in Appendix K.

**52d.**  $\langle apply\ Impcore\ primitive\ print\ to\ vs\ and\ return\ 52d \rangle \equiv$  (52c)

```
{
    checkargc(e, 1, lengthVL(vs));
    Value v = nthVL(vs, 0);
    print("%v", v);
    return v;
}
```

Each arithmetic primitive expects on exactly two arguments, which I put in C variables `v` and `w`. The name of the primitive goes in `s`.

**53a.** *⟨apply arithmetic primitive to vs and return 53a⟩* ≡ (52c)

```

{
    checkargc(e, 2, lengthVL(vs));
    Value v = nthVL(vs, 0);
    Value w = nthVL(vs, 1);
    const char *s = nametToStr(f.primitive);
    ⟨if operation s would overflow on v and w, call runerror 53c⟩
    ⟨return a function of v and w determined by s 53b⟩
}

```

§1.6  
*The interpreter*

53

Because the name of each arithmetic primitive has just one character, I switch on that character. This technique works only because the set of primitives is small and fixed.

**53b.** *⟨return a function of v and w determined by s 53b⟩* ≡ (53a)

```

assert(strlen(s) == 1);
switch (s[0]) {
    case '<':
        return v < w;
    case '>':
        return v > w;
    case '=':
        return v == w;
    case '+':
        return v + w;
    case '-':
        return v - w;
    case '*':
        return v * w;
    case '/':
        if (w == 0)
            runerror("division by zero in %e", e);
        return v / w;
    default:
        assert(0);
}

```

checkargc	48b
checkarith	S198b
eval	45e
evalList	48c
type ExprList	S288c
formals	48d
functions	48d
type FunEnv	44f
globals	48d
lengthNL	A
lengthVL	A
mkValEnv	45a
mkVL	A
type Namelist	43b
nametToStr	43c
nthVL	A
print	46c
runerror	47
strToName	43c
type ValEnv	44f
type Value	44a
type Valuelist	44a

The code depends on the fact that Impcore shares C's rules for the values of comparison expressions. Impcore does *not* share C's other rules of arithmetic: in Impcore, if the result of an arithmetic operation does not fit in the range  $-2^{31}$  to  $2^{31}$ , the operation causes a checked run-time error. Function `checkarith` is defined in Appendix F.

**53c.** *⟨if operation s would overflow on v and w, call runerror 53c⟩* ≡ (53a)

```
checkarith(s[0], v, w, 32);
```

### Evaluating true definitions

As noted on page 24, there are two forms of definition: True definitions are specific to Impcore, and they have an operational semantics; they include `val` and `define`. Extended definitions are shared across languages, and they don't have an operational semantics; they include `use` and `check-expect`. Evaluation of true definitions is described here; evaluation of the extended definitions is described in the Supplement.

Function `evalDef` implements the  $\rightarrow$  relation on the true definitions, from the operational semantics. That is, calling `evalDef(d, xi, phi, echo)` finds a  $\xi'$  and  $\phi'$  such

that  $\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \phi' \rangle$ , and evaldef mutates the C representation of the environments so the global-variable environment becomes  $\xi'$  and the function environment becomes  $\phi'$ . If echo is ECHOES, evaldef also prints the interpreter's response to the user's input. Printing the response is evaldef's job because only evaldef can tell whether to print a value (for EXP and VAL) or a name (for DEFINE).

Just like eval, evaldef looks at the conclusions of rules, and it discriminates on the syntactic form of d.

**54a.**  $\langle \text{eval.c 48c} \rangle + \equiv$

△ 52a

```
void evaldef(Def d, Valenv globals, Funenv functions, Echo echo) {
    switch (d->alt) {
        case VAL:
            ⟨evaluate d->val, mutating globals 54b⟩
            return;
        case EXP:
            ⟨evaluate d->exp and possibly print the result 54c⟩
            return;
        case DEFINE:
            ⟨evaluate d->define, mutating functions 55a⟩
            return;
    }
    assert(0);
}
```

The operational semantics dictates the cases for VAL, EXP, and DEFINE.

A variable definition updates  $\xi$ .

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{VAL}(x, e), \xi, \phi \rangle \rightarrow \langle \xi' \{x \mapsto v\}, \phi \rangle} \quad (\text{DEFINEGLOBAL})$$

The premise shows we must call eval to get value  $v$  and environment  $\xi'$ . When we call eval, we must use an empty environment as  $\rho$ . The rule says the new environment  $\xi'$  is retained, and the value of the expression,  $v$ , is bound to  $x$  in it. The implementation may also print  $v$ .

**54b.**  $\langle \text{evaluate d->val, mutating globals 54b} \rangle \equiv$

△ 54a

```
{ Value v = eval(d->val.exp, globals, functions, mkValenv(NULL, NULL));
  bindval(d->val.name, v, globals);
  if (echo == ECHOES)
    print("%v\n", v);
}
```

Evaluating a top-level expression is just like evaluating a definition of it.

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{EXP}(e), \xi, \phi \rangle \rightarrow \langle \xi' \{ \text{it} \mapsto v \}, \phi \rangle} \quad (\text{EVALEXP})$$

**54c.**  $\langle \text{evaluate d->exp and possibly print the result 54c} \rangle \equiv$

△ 54a

```
{ Value v = eval(d->exp, globals, functions, mkValenv(NULL, NULL));
  bindval(strtoname("it"), v, globals);
  if (echo == ECHOES)
    print("%v\n", v);
}
```

A function definition updates  $\phi$ . Our implementation also prints the name of the function.

$$\frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi \{ f \mapsto \text{USER}(\langle x_1, \dots, x_n \rangle, e) \} \rangle} \quad (\text{DEFINEFUNCTION})$$

55a. *(evaluate d->define, mutating functions 55a)*≡ (54a)

```
bindfun(d->define.name, mkUserdef(d->define.userfun), functions);
if (echo == ECHOES)
    print("%n\n", d->define.name);
```

The evaluator does not check to see that the  $x_1, \dots, x_n$  are all distinct—the  $x_i$ 's are checked when the definition is parsed, by function `check_def_duplicates` in chunk S217d.

### 1.6.3 Implementation of environments

§1.6  
The interpreter

55

In the interest of type safety, all the environment code is implemented twice: once for value environments and once for function environments. Only the value-environment code appears here; the function-environment code appears in Appendix K.

I represent an environment as two parallel lists: one holding names and one holding values. The representation's key invariant is that the lists have the same length.

55b. *(env.c 55b)*≡

55c▷

```
struct Valenv {
    Namelist xs;
    Valuelist vs;
    // invariant: lists have the same length
};
```

Given the representation, creating an environment is simple. To prevent the invariant from being violated, I assert that `xs` and `vs` have equal length.

55c. *(env.c 55b) +≡*

△ 55b 55d▷

```
Valenv mkValenv(Namelist xs, Valuelist vs) {
    Valenv e = malloc(sizeof(*e));
    assert(e != NULL);
    assert(lengthNL(xs) == lengthVL(vs));
    e->xs = xs;
    e->vs = vs;
    return e;
}
```

Three environment functions (`fetchval`, `isvalbound`, and `bindval`) have to search the list of names. I want to implement that search just once, so there is a single point of truth about how to do it. I therefore introduce a private function `findval`. Given a name  $x$ , it searches the environment. If it finds  $x$ , it returns a pointer to the value associated with  $x$ . If it doesn't find  $x$ , it returns `NULL`. The pointer can be used to test for binding (`isvalbound`), to fetch a bound value (`fetchval`), or to change an existing binding (`bindval`). Code for the corresponding functions `findfun`, `isfunbound`, `fetchfun`, and `bindfun` appears in Appendix K.

Like `strtoname`, `findval` uses linear search. Hash tables or search trees would be faster but more complicated.

bindfun	45d
bindval	45d
type Def	A
type Echo	S289b
type Funenv	44f
lengthNL	A
lengthVL	A
mkUserdef	44e
type Name	43b
type Namelist	43b
print	46c
strtoname	43c
type Valenv	44f
type Value	44a
type Valuelist	44a

55d. *(env.c 55b) +≡*

△ 55c 56a▷

```
static Value* findval(Name x, Valenv env) {
    Namelist xs;
    Valuelist vs;

    for (xs=env->xs, vs=env->vs; xs && vs; xs=xs->t1, vs=vs->t1)
        if (x == xs->hd)
            return &vs->hd;
    return NULL;
}
```

A name is bound if there is a value associated with it.

```
56a. ⟨env.c 55b⟩+≡
    bool isvalbound(Name name, Valenv env) {
        return findval(name, env) != NULL;
    }

    We fetch a value through the pointer returned by findval, if any.

56b. ⟨env.c 55b⟩+≡
    Value fetchval(Name name, Valenv env) {
        Value *vp = findval(name, env);
        assert(vp != NULL);
        return *vp;
    }
```

You might think that to add a new binding to an environment, we would have to insert a new name and value at the beginning of the lists. But I can get away with an optimization. If  $x \in \text{dom } \rho$ , instead of extending  $\rho$  by making  $\rho\{x \mapsto v\}$ , I overwrite the old binding of  $x$ . This optimization is safe only because no program written in Impcore can tell that it is there. Proving that the optimization is safe requires reasoning about the rules of the operational semantics, which show that in any context where  $\rho\{x \mapsto v\}$  appears, there is no way to get to the old  $\rho(x)$ . (See Exercise 29 on page 87.)

```
56c. ⟨env.c 55b⟩+≡
    void bindval(Name name, Value val, Valenv env) {
        Value *vp = findval(name, env);
        if (vp != NULL)
            *vp = val; // safe optimization
        else {
            env->xss = mkNL(name, env->xss);
            env->vs = mkVL(val, env->vs);
        }
    }
```

## 1.7 OPERATIONAL SEMANTICS REVISITED: PROOFS

If the interpreter in Section 1.6 is correct, then calling  $\text{eval}(e, \xi, \phi, \rho)$  returns a value  $v$  if and only if there is a *proof* of the judgment  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle$  using the rules from Section 1.5. Proofs in programming languages are not very interesting; or rather, they are interesting only when they are wrong. A wrong proof, or the failure to find a proof, tells you that you got something wrong in your language design—just like a bug in a program.

But there's more to proofs than being interesting. What a proof in programming languages *can* do for you is convey *certainty*. A proof provides a way of being sure that your program does what you think it does, or that your language does what you think it does.

### 1.7.1 Proofs about evaluation: Theory

A proof about the evaluation of code—that is, a call to `eval` or `evaldef`—takes the form of a *derivation*. This form of proof, which is borrowed from formal logic, is a tree in which each node is an *instance* of an inference rule; an instance of a rule is obtained by substituting for the rule's metavariables. If every substitution is done consistently, and if, in the resulting derivation, every proposition holds, then the derivation is *valid*. A valid derivation justifies the judgment claimed in the conclusion of its root. These concepts are illustrated in the next few pages.

A derivation is also called a *proof tree*; the root contains the conclusion, and each subtree is also a derivation. We write a derivation tree with its root at the bottom; as in a single rule, the conclusion of a derivation appears on the bottom, below a horizontal line. The leaf nodes appear at the top; each leaf node is an instance of an inference rule that has no evaluation judgments among its premises, like the LITERAL rule or the FORMALVAR rule (page 32). A leaf node corresponds to a computation in which eval returns a result without making a recursive call.

Each node in a derivation tree is obtained by *instantiating* a rule and then deriving that rule's premises. Instantiation may substitute for none, some, or all of a rule's metavariables. Substitution that replaces all metavariables, leaving only complete environments, syntax, and data, describes a single run of eval. For example, suppose we evaluate a literal 83 in a context where there are no global variables, no functions, and no formal parameters:

```
eval(mkLiteral(83),
      mkValenv(NULL, NULL), mkFunenv(NULL, NULL), mkValenv(NULL, NULL));
```

The resulting run can described by an instance of the LITERAL rule. Here is the rule as it appears in the semantics:

$$\overline{\langle \text{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \quad (\text{LITERAL})$$

To get the instance needed to describe the run, I substitute 83 for  $v$ ,  $\{\}$  for  $\xi$ ,  $\{\}$  for  $\phi$ , and  $\{\}$  for  $\rho$ :

$$\overline{\langle \text{LITERAL}(83), \{\}, \{\}, \{\} \rangle \Downarrow \langle 83, \{\}, \{\}, \{\} \rangle}$$

Because the LITERAL rule has no premises above the line, this instance is actually a complete, valid derivation all by itself.

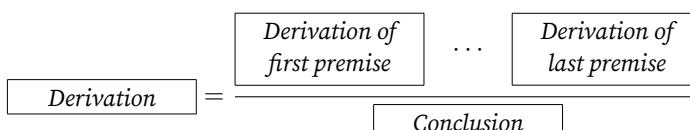
The preceding example is awfully specific. I'd like to prove that a literal 83 evaluates to 83 *regardless* of the presence of functions or variables—and that the evaluation does not change the values of any variables. I can do that by creating a different instance of LITERAL, in which I substitute only for  $v$ , leaving  $\xi$ ,  $\phi$ , and  $\rho$  as they are written in the rule:

$$\overline{\langle \text{LITERAL}(83), \xi, \phi, \rho \rangle \Downarrow \langle 83, \xi, \phi, \rho \rangle}$$

This instance is also a complete, valid derivation, and it describes the evaluation of a literal 83 in any possible environment.

A complete derivation tree ends in a single rule, but if that rule has evaluation judgments above the line, like the APPLY rules, then each of those judgments—the premises—must be derived as well. Here's a kind of a schematic view:

findval	55d
mkNL	A
mkVL	A
type Name	43b
type Valenv	44f
type Value	44a



This schematic view simplifies things a little bit: it assumes that every premise must be justified by a derivation. In practice, only an evaluation judgment can be justified by a derivation. Other premises, like  $x \in \text{dom } \rho$  or  $\rho(x) = 3$ , have to be justified by appealing to what we know about  $x$  and  $\rho$ .

Here's an example of a derivation tree with more than one node. The expression being evaluated is  $(+ (* x x) (* y y))$ , which computes the sum of two squares. It's evaluated in an environment where we know that  $\rho$  binds  $x$  to 3 and  $y$  to 4.

$$\frac{\text{APPLYADD} \quad \begin{array}{c} \text{FORMALVAR } \frac{x \in \text{dom } \rho \quad \rho(x) = 3}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle} \quad \text{FORMALVAR } \frac{x \in \text{dom } \rho \quad \rho(x) = 3}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle} \\ \text{APPLYMUL} \end{array}}{\langle \text{APPLY}(*, \text{VAR}(x), \text{VAR}(x)), \xi, \phi, \rho \rangle \Downarrow \langle 9, \xi, \phi, \rho \rangle} \dots$$

$$\langle \text{APPLY}(+, \text{APPLY}(*, \text{VAR}(x), \text{VAR}(x)), \text{APPLY}(*, \text{VAR}(y), \text{VAR}(y))), \xi, \phi, \rho \rangle \Downarrow \langle 25, \xi, \phi, \rho \rangle$$

In this example, each node is labeled with the name of the rule to which it corresponds. Because derivation trees take so much space, I've elided the subtree that proves

$$\langle \text{APPLY}(*, \text{VAR}(y), \text{VAR}(y)), \xi, \phi, \rho \rangle \Downarrow \langle 16, \xi, \phi, \rho \rangle.$$

Derivation trees can get big, and to fit them into small spaces, we often take liberties with notation. For example, instead of writing abstract syntax like  $\text{APPLY}(*, \text{VAR}(y), \text{VAR}(y))$ , we can write concrete syntax like  $(* y y)$ . The resulting notation is easier to digest, but it is less obvious that each node is an instance of a semantic rule:

$$\frac{\text{APPLYADD} \quad \begin{array}{c} \text{FORMALVAR } \frac{x \in \text{dom } \rho \quad \rho(x) = 3}{\langle x, \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle} \quad \text{FORMALVAR } \frac{x \in \text{dom } \rho \quad \rho(x) = 3}{\langle x, \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle} \\ \text{APPLYMUL} \end{array}}{\langle (* x x), \xi, \phi, \rho \rangle \Downarrow \langle 9, \xi, \phi, \rho \rangle} \dots$$

$$\langle (+ (* x x) (* y y)), \xi, \phi, \rho \rangle \Downarrow \langle 25, \xi, \phi, \rho \rangle$$

Even if I use a smaller font and don't label the nodes, the full derivation tree sticks into the margin:

$$\frac{\begin{array}{cccc} x \in \text{dom } \rho & \rho(x) = 3 & y \in \text{dom } \rho & \rho(y) = 4 \\ \langle x, \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle & \langle x, \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle & \langle y, \xi, \phi, \rho \rangle \Downarrow \langle 4, \xi, \phi, \rho \rangle & \langle y, \xi, \phi, \rho \rangle \Downarrow \langle 4, \xi, \phi, \rho \rangle \\ \langle (* x x), \xi, \phi, \rho \rangle \Downarrow \langle 9, \xi, \phi, \rho \rangle & & \langle (* y y), \xi, \phi, \rho \rangle \Downarrow \langle 16, \xi, \phi, \rho \rangle & \end{array}}{\langle (+ (* x x) (* y y)), \xi, \phi, \rho \rangle \Downarrow \langle 25, \xi, \phi, \rho \rangle}$$

*What is a valid derivation?*

A derivation is *valid* if every node is obtained by instantiating a rule and every premise is provable. To explain this idea, it helps to name the rule that is instantiated at the root. Here is what it means for a derivation  $\mathcal{D}$  to be a valid derivation ending in an application of rule  $\mathcal{R}$ :

- The derivation is obtained by substituting for the metavariables in  $\mathcal{R}$ . Substitution must respect the intent of the metavariables: we must substitute a well-formed expression for  $e$ , a value for  $v$ , a well-formed environment for  $\rho$ , and so on. While whatever we substitute must be well formed, it need not be completely specified: whatever we substitute for a metavariable may itself contain metavariables. For example, it's OK to substitute  $\text{IF}(e_1, e_2, e_3)$  for  $e$ .
- Rule  $\mathcal{R}$  may include evaluation judgments above the line, as premises. After substitution, each of these premises must be justified by a derivation of its own.
- After substitution, every other premise in rule  $\mathcal{R}$  must also be provable. Premises that aren't evaluation judgments are usually proved by set theory, arithmetic, or appeal to assumptions.

Let's see an example: if  $n$  is a formal parameter, evaluating  $(\text{set } n \ 0)$  sets  $n$  to zero and returns zero. The judgment is

$$\langle \text{SET}(n, \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \{ n \mapsto 0 \} \rangle$$

I construct a derivation  $\mathcal{D}$  ending in rule FORMALASSIGN, which I reproduce here:

$$\frac{x \in \text{dom } \rho \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \{ x \mapsto v \} \rangle} \quad (\text{FORMALASSIGN})$$

To get the judgment I want, I substitute  $n$  for  $x$ , LITERAL(0) for  $e$ ,  $\xi$  for  $\xi'$ , 0 for  $v$ , and  $\rho$  for  $\rho'$ . With that substitution, my derivation in progress looks like this:

$$\frac{n \in \text{dom } \rho \quad \langle \text{LITERAL}(0), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle}{\langle \text{SET}(n, \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \{ n \mapsto 0 \} \rangle}$$

To complete the derivation, I derive judgment  $\langle \text{LITERAL}(0), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle$ , with a new derivation  $\mathcal{D}_1$ . Derivation  $\mathcal{D}_1$  ends in rule LITERAL, and to construct  $\mathcal{D}_1$ , I substitute 3 for  $v$  in rule LITERAL. Rule LITERAL has no premises, so that substitution completes  $\mathcal{D}_1$ :

$$\mathcal{D}_1 = \frac{}{\langle \text{LITERAL}(0), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle}$$

Plugging  $\mathcal{D}_1$  into my previous work completes  $\mathcal{D}$ :

$$\mathcal{D} = \frac{n \in \text{dom } \rho \quad \overline{\langle \text{LITERAL}(0), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle}}{\langle \text{SET}(n, \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \{ n \mapsto 0 \} \rangle}$$

For derivation  $\mathcal{D}$  to be valid, premise  $n \in \text{dom } \rho$  must also be proved. It's true by assumption:  $n$  is a formal parameter, which is what  $n \in \text{dom } \rho$  means.

*How do we construct valid derivations?*

A derivation is valid if we substitute consistently for each metavariable, replace every evaluation premise with a valid derivation, and can prove the other premises. But how do we know what to substitute? There's an algorithm, and it's the same algorithm that's used in eval to interpret code. That's no surprise—every derivation tree is intimately related to an application of eval. The derivation expresses, in a data structure, the steps that eval goes through to interpret the code.

To construct a derivation, we start with an initial state—the left-hand side of an evaluation judgment. We usually know some or all of the syntax, and we may know something about environments as well. The algorithm works like this:

1. Find a rule  $\mathcal{R}$  whose *conclusion* matches the left-hand side we are interested in. By “matches,” I mean, that the left-hand side we are interested can be obtained by substituting for metavariables in the conclusion of  $\mathcal{R}$ . For example, in the derivation for the SET expression above, there are only two rules that can conclude in a SET expression: FORMALASSIGN and GLOBALASSIGN.

In the Impcore interpreter, the “find a rule” step corresponds to the main switch statement in function eval (chunk 48d), which identifies possible rules by looking at  $e \rightarrow \text{alt}$ .

2. Substitute in  $\mathcal{R}$  so that the left-hand side of the conclusion is equal to the initial state of interest. Now look at each of  $\mathcal{R}$ 's premises.

- If a premise does not involve an evaluation judgment, is it provable? If so, continue building the derivation. Otherwise, try some other rule.

In an interpreter, provability corresponds to a check on run-time data. Premises like  $x \in \text{dom } \xi$  or  $x \notin \text{dom } \rho$  are checked by calling `isvalbound`, and premises like  $v_1 \neq 0$  or  $v_1 = 0$  are checked by comparing a run-time value with 0.

- If a premise *does* involve an evaluation judgment, recursively build a derivation of that judgment, starting with the right-hand side. In the Impcore interpreter, recursively building a derivation corresponds to a recursive call to `eval`.

This algorithm eventually dictates what needs to be substituted on the right-hand side of the evaluation judgment. As an example, I sketch the construction of a derivation for  $(\text{set } n (+ n 1))$ , in an environment in which  $n$  is 7. The sketch appears in Figure 1.7, which shows a sequence of snapshots. In each snapshot, the partially constructed derivation is shown in black, and the parts that are not yet constructed are shown in gray. The derivation begins with the expression  $(\text{set } n (+ n 1))$  and an environment  $\rho$  in which  $\rho(n) = 8$ . (Because  $\xi$  and  $\phi$  do not take part and do not change, they are omitted to save space.) Each step fills something in; first, check  $n \in \text{dom } \rho$ ; next, evaluate  $(+ n 1)$ , and so on. The sketch should suggest that the derivation, which is a data structure, is a spacelike representation of a timelike computation. This is true of every derivation.

#### *What can we do with derivations?*

If we know something about the environments, we can use derivation trees to answer questions about the evaluation of expressions and definitions. One example is the evaluation of the expression in Exercise 12 on page 79. This kind of application of the language semantics is called the *theory* of the language. But we can answer much more interesting questions if we *prove facts about derivations*. For example, Exercise 13 on page 79 asks you to show that in Impcore, the expression  $(\text{if } x x 0)$  is always equivalent to  $x$ . Because a computation is a sequence of events in time, proving facts about computations can be difficult. But if every terminating computation is described by a derivation, a derivation is just a data structure, and proving facts about data structures is much easier. Reasoning about derivations is called *metatheory*.

#### 1.7.2 *Proofs about derivations: Metatheory*

If you already know that you want to study metatheory, you're better off with another book. In this book, I hope only to suggest what kinds of things metatheory is good for, so that when you consider whether you want to study metatheory, you'll know something about it.

Here's a claim about Impcore programs: in any program, we can replace the expression  $(+ x 0)$  by the expression  $x$ , and this replacement doesn't change the meaning of the program. This claim is not very interesting to a programmer, but it's important to a compiler writer, who might use it to create an "optimization" that improves the performance of programs. If you're going to create an optimization, you must be certain that it doesn't change the meaning of the program. Providing certainty is where proofs—and proofs about proofs—are useful.

What do we know about derivations that refer to the expression  $(+ x 0)$ ? We know that if there is a derivation, it is going to contain a judgment of the form

$$\langle (+ x 0), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle.$$

$$\begin{array}{c}
 n \in \text{dom } \rho \\
 \frac{n \in \text{dom } \rho}{\langle n, \rho \rangle \Downarrow \langle 7, \rho \rangle} \quad \frac{\langle 1, \rho \rangle \Downarrow \langle 1, \rho \rangle}{\langle (+ n 1), \rho \rangle \Downarrow \langle 8, \rho \rangle} \\
 \langle (\text{set } n (+ n 1)), \rho \rangle \Downarrow \langle 8, \rho \{n \mapsto 8\} \rangle
 \end{array}$$
  

$$\begin{array}{c}
 n \in \text{dom } \rho \\
 \frac{n \in \text{dom } \rho}{\langle n, \rho \rangle \Downarrow \langle 7, \rho \rangle} \quad \frac{\langle 1, \rho \rangle \Downarrow \langle 1, \rho \rangle}{\langle (+ n 1), \rho \rangle \Downarrow \langle 8, \rho \rangle} \\
 \langle (\text{set } n (+ n 1)), \rho \rangle \Downarrow \langle 8, \rho \{n \mapsto 8\} \rangle
 \end{array}$$
  

$$\begin{array}{c}
 n \in \text{dom } \rho \\
 \frac{n \in \text{dom } \rho}{\langle n, \rho \rangle \Downarrow \langle 7, \rho \rangle} \quad \frac{\langle 1, \rho \rangle \Downarrow \langle 1, \rho \rangle}{\langle (+ n 1), \rho \rangle \Downarrow \langle 8, \rho \rangle} \\
 \langle (\text{set } n (+ n 1)), \rho \rangle \Downarrow \langle 8, \rho \{n \mapsto 8\} \rangle
 \end{array}$$
  

$$\begin{array}{c}
 n \in \text{dom } \rho \\
 \frac{n \in \text{dom } \rho}{\langle n, \rho \rangle \Downarrow \langle 7, \rho \rangle} \quad \frac{\langle 1, \rho \rangle \Downarrow \langle 1, \rho \rangle}{\langle (+ n 1), \rho \rangle \Downarrow \langle 8, \rho \rangle} \\
 \langle (\text{set } n (+ n 1)), \rho \rangle \Downarrow \langle 8, \rho \{n \mapsto 8\} \rangle
 \end{array}$$
  

$$\begin{array}{c}
 n \in \text{dom } \rho \\
 \frac{n \in \text{dom } \rho}{\langle n, \rho \rangle \Downarrow \langle 7, \rho \rangle} \quad \frac{\langle 1, \rho \rangle \Downarrow \langle 1, \rho \rangle}{\langle (+ n 1), \rho \rangle \Downarrow \langle 8, \rho \rangle} \\
 \langle (\text{set } n (+ n 1)), \rho \rangle \Downarrow \langle 8, \rho \{n \mapsto 8\} \rangle
 \end{array}$$

Figure 1.7: Construction of a derivation (omitting unchanging  $\xi$  and  $\phi$ )

Let's consider the sub-derivation rooted in that judgment. What do we know about it? It must apply a rule that permits the application of the `+` operator on the left-hand side of its conclusion. *This kind of reasoning is exactly the kind of reasoning used to write the interpreter code in chunk 48d.* In both cases, we look for inference rules with `APPLY(+, ...)` in their conclusions.

If we consult Section 1.5.6, we see there are two such rules: `APPLYUSER` and `APPLYADD`. And now we see that the claim is false! If  $\phi(+)$  refers to a user-defined function, the `APPLYUSER` rule kicks in, and it is *not* safe to replace  $(+ x 0)$  with  $x$ . Here's a demonstration:

```

61. <terrifying transcript 61>≡
-> (define + (x y) y) ; no sane person would do this
-> (define addzero (x) (+ x 0))
-> (addzero 99)
0
-> (define addzero2 (x) x)
-> (addzero2 99)
99

```

If a compiler writer wants to be able to replace an occurrence of  $(+ x 0)$  with  $x$ , he or she will first have to prove that the environment  $\phi$  in which  $(+ x 0)$  is evaluated *never* binds `+` to a user-defined function. (Compilers typically include lots of

infrastructure for proving facts about environments, but such infrastructure is well beyond the scope of this book.)

Metatheory enables you to prove properties like these (Section 1.10 on page 73):

- Expression `(if x x 0)` is equivalent to `x` (Exercise 13).
- If evaluation of a `while` loop terminates, its value is zero (Exercise 23).
- Evaluating an expression can't create a new variable (Exercise 24).
- In Impcore, evaluating an expression in the same context always produces the same result—which isn't true of languages that support parallel execution (Exercise 27).
- Impcore programs can be evaluated using a stack of mutable environments, as the implementation in Section 1.6 does (Exercise 29).

A little metatheory goes a long way.

### 1.7.3 How to attempt a metatheoretic proof

A metatheoretic proof works by induction on the structure of valid derivations. A derivation is a tree, and we can assume an induction hypothesis holds for any proper subtree—or, if you prefer, we can prove metatheorems by induction on the height of a derivation tree.

A valid derivation can end in any rule, so a proof by induction on a derivation's structure has a case for every rule. In a language as big as Impcore, that's a lot of cases. To make my proofs readable and convincing, I organize each case in exactly the same way. Here's an example case from the proof that evaluating an expression doesn't create any new global variables, which is to say that if derivation  $\mathcal{D}$  proves  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , then  $\text{dom } \xi = \text{dom } \xi'$ :

- When the last rule used in  $\mathcal{D}$  is FORMALASSIGN, the derivation must have the following form:

$$\mathcal{D} = \frac{\mathcal{D}_1}{\begin{array}{c} x \in \text{dom } \rho \\ \langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle \\ \langle \text{SET}(x, e_1), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \{x \mapsto v\} \rangle \end{array}} \text{ FORMALASSIGN}$$

The form of  $e$  is `SET(x, e1)`. Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove that  $\text{dom } \xi = \text{dom } \xi'$ . But because derivation  $\mathcal{D}_1$  is smaller than derivation  $\mathcal{D}$ , we are permitted to assume the induction hypothesis, which tells us that  $\text{dom } \xi = \text{dom } \xi'$ . Our obligation is met.

This example, like all the cases in my metatheoretic proofs, uses this template:

1. When the last rule used in  $\mathcal{D}$  is RULENAME, and RULENAME has conclusion  $C$  and premises  $P_1$  to  $P_n$ , the derivation must have the following form:

$$\mathcal{D} = \frac{P_1 \quad \cdots \quad P_n}{C} \text{ RULENAME}$$

*Commentary: The conclusion C is the evaluation judgment of which D is a proof. If any particular P<sub>i</sub> is also an evaluation judgment, I write its derivation above it,*

as in

$$\mathcal{D} = \frac{\begin{array}{cccc} \mathcal{D}_1 & & \mathcal{D}_i & & \mathcal{D}_n \\ \hline P_1 & \dots & P_i & \dots & P_n \end{array}}{C} \text{ RULENAME}$$

A premise like “ $x \notin \text{dom } \rho$ ” is not an evaluation judgment and is not supported by a subderivation.

2. The form of  $e$  is syntactic form, and whatever additional analysis goes with that syntactic form and with rule RULENAME.
3. Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove whatever it is.
4. Identifying each premise  $P_i$  that is an evaluation judgment, because derivation  $\mathcal{D}_i$  is strictly smaller than derivation  $\mathcal{D}$ , we are permitted to assume that the induction hypothesis applies to derivation  $\mathcal{D}_i$ . This assumption gives us whatever it gives us.
5. From the truth of premises  $P_1$  to  $P_n$ , plus the information from the induction hypothesis, we show that the induction hypothesis holds for the judgment below the line.
6. Our obligation is met.

This template has served me well, but part of it may surprise you: it doesn’t distinguish between “base cases” and “inductive cases.” The distinction is there—a base case has no evaluation judgments above the line—but in a programming-language proof, the distinction is not terribly useful. For example, the base cases might or might not be easy, and they might or might not be the ones that fail.

The template is instantiated for every case in a proof. As a demonstration, I instantiate the template to try to prove the metatheoretic conjecture,

If an expression  $e$  is evaluated successfully, then every variable in  $e$  is defined.

(This conjecture isn’t true, but that’s a good thing—it’s the things we try to prove that aren’t so that teach us the most.)

To begin, I must state my conjecture formally. And that means I must formalize the idea of “every variable in  $e$ .” I use the function  $\text{fv}(e)$ , short for “free variables of  $e$ ,” which is defined in Figure 1.8 on page 65. Figure 1.8 uses a simplified dialect of Impcore, which makes the metatheoretic proof a little easier:

- In Simplified Impcore, there is no `begin` expression.
- In Simplified Impcore, every function application has exactly two arguments.

(You can work out how to eliminate `begin` in Exercise 14 on page 81.) Informally speaking,  $\text{fv}(e)$  is the set of variables mentioned in  $e$ . A variable can be mentioned only in a VAR expression or in a SET expression. In other forms of expression, the free variables are the free variables of the subexpressions.

Now I can state my conjecture: if  $\mathcal{D}$  is a valid derivation of  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , then  $\text{fv}(e) \subseteq \text{dom } \xi \cup \text{dom } \rho$ . My conjecture is also my induction hypothesis.

I now instantiate the proof template for each rule of Simplified Impcore. (Rules are listed in Figure 1.11 on page 80.)

- When the last rule used in  $\mathcal{D}$  is LITERAL, the derivation must have the following form:

$$\mathcal{D} = \frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \text{LITERAL}$$

The form of  $e$  is  $\text{LITERAL}(v)$ . Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove  $\text{fv}(\text{LITERAL}(v)) \subseteq \text{dom } \xi \cup \text{dom } \rho$ . According to the definition of  $\text{fv}$  in Figure 1.8,  $\text{fv}(\text{LITERAL}(v)) = \emptyset$ , and the empty set is a subset of any set. Our obligation is met.

- When the last rule used in  $\mathcal{D}$  is FORMALVAR, the derivation must have the following form:

$$\mathcal{D} = \frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle} \text{FORMALVAR}$$

The form of  $e$  is  $\text{VAR}(x)$ . Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove  $\text{fv}(\text{VAR}(x)) \subseteq \text{dom } \xi \cup \text{dom } \rho$ . According to the definition of  $\text{fv}$  in Figure 1.8,  $\text{fv}(\text{VAR}(x)) = \{x\}$ . And from the first and only premise of the derivation, we know that  $x \in \text{dom } \rho$ . Therefore

$$\text{fv}(\text{VAR}(x)) = \{x\} \subseteq \text{dom } \rho \subseteq (\text{dom } \rho \cup \text{dom } \xi).$$

Our obligation is met.

- When the last rule used in  $\mathcal{D}$  is GLOBALVAR, the derivation must have the following form:

$$\mathcal{D} = \frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle} \text{GLOBALVAR}$$

The form of  $e$  is  $\text{VAR}(x)$ . Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove  $\text{fv}(\text{VAR}(x)) \subseteq \text{dom } \xi \cup \text{dom } \rho$ . According to the definition of  $\text{fv}$  in Figure 1.8,  $\text{fv}(\text{VAR}(x)) = \{x\}$ . And from the second premise of the derivation, we know that  $x \in \text{dom } \xi$ . Therefore

$$\text{fv}(\text{VAR}(x)) = \{x\} \subseteq \text{dom } \xi \subseteq (\text{dom } \rho \cup \text{dom } \xi).$$

Our obligation is met.

- When the last rule used in  $\mathcal{D}$  is FORMALASSIGN, the derivation must have the following form:

$$\mathcal{D} = \frac{x \in \text{dom } \rho \quad \frac{\mathcal{D}_1}{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}}{\langle \text{SET}(x, e_1), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \{x \mapsto v\} \rangle} \text{FORMALASSIGN}$$

The form of  $e$  is  $\text{SET}(x, e_1)$ . Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove that  $\text{fv}(\text{SET}(x, e_1)) \subseteq \text{dom } \xi \cup \text{dom } \rho$ . According to the definition of  $\text{fv}$  in Figure 1.8,  $\text{fv}(\text{SET}(x, e_1)) = \{x\} \cup \text{fv}(e_1)$ . It therefore suffices to prove these two inclusions:

$$\begin{aligned}
\text{fv}(\text{LITERAL}(v)) &= \emptyset \\
\text{fv}(\text{VAR}(x)) &= \{x\} \\
\text{fv}(\text{SET}(x, e)) &= \{x\} \cup \text{fv}(e) \\
\text{fv}(\text{IF}(e_1, e_2, e_3)) &= \text{fv}(e_1) \cup \text{fv}(e_2) \cup \text{fv}(e_3) \\
\text{fv}(\text{WHILE}(e_1, e_2)) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv}(\text{APPLY}(f, e_1, e_2)) &= \text{fv}(e_1) \cup \text{fv}(e_2)
\end{aligned}$$

Figure 1.8: Free variables of an expression in Simplified Impcore

65

- (a)  $\{x\} \subseteq \text{dom } \xi \cup \text{dom } \rho$
- (b)  $\text{fv}(e_1) \subseteq \text{dom } \xi \cup \text{dom } \rho$

From the first premise of the derivation, we know that  $x \in \text{dom } \rho$ , and as before, that implies (a). From the second premise of the derivation, we can apply the induction hypothesis to  $\mathcal{D}_1$ , which gives us (b). Our obligation is met.

- When the last rule used in  $\mathcal{D}$  is GLOBALASSIGN, the reasoning is so similar to what we have already done that it's not worth repeating.
- When the last rule used in  $\mathcal{D}$  is IFTRUE, the derivation must have the following form:

$$\mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle}} \text{IFTRUE}$$

The form of  $e$  is  $\text{IF}(e_1, e_2, e_3)$ . Our obligation is to prove that the induction hypothesis holds for the judgment below the line. We must therefore prove  $\text{fv}(\text{IF}(e_1, e_2, e_3)) \subseteq \text{dom } \xi \cup \text{dom } \rho$ . According to the definition of fv in Figure 1.8,  $\text{fv}(\text{IF}(e_1, e_2, e_3)) = \text{fv}(e_1) \cup \text{fv}(e_2) \cup \text{fv}(e_3)$ . It therefore suffices to prove these three inclusions:

- (a)  $\text{fv}(e_1) \subseteq \text{dom } \xi \cup \text{dom } \rho$
- (b)  $\text{fv}(e_2) \subseteq \text{dom } \xi \cup \text{dom } \rho$
- (c)  $\text{fv}(e_3) \subseteq \text{dom } \xi \cup \text{dom } \rho$

By applying the induction hypothesis to  $\mathcal{D}_1$ , we get (a). By applying the induction hypothesis to  $\mathcal{D}_2$ , we get (b). But there is no subderivation corresponding to the evaluation of  $e_3$ . And in fact we cannot prove that  $\text{fv}(e_3) \subseteq \text{dom } \xi \cup \text{dom } \rho$ , because it isn't true! Expression  $e_3$  is not evaluated, and its free variables might not be defined. Here's an example:

**65.** *(transcript 12a) +≡*  
*-> (if 1 7 undefined)*  
*7*

◀ 29

Our obligation cannot be met.

- We could continue with all the other cases. The proof fails for IFFALSE as well as IFTRUE, but it succeeds for EMPTYBEGIN, BEGIN, and APPLYUSER. Analysis of rules WHILEITERATE and WHILEEND is left for Exercise 26.

The conjecture isn't actually a theorem, and in the process of working out a proof, we found a counterexample. To guarantee that every variable in an expression is defined, we need something stronger than a successful evaluation. (Chapter 6 shows how to provide just that guarantee by writing a type checker.)

#### 1.7.4 Why bother with semantics, proofs, theory, and metatheory?

What's up with all the Greek letters and horizontal lines? What's the point? Isn't it easier just to look at the code? No. The great thing about an operational semantics is that it glosses over all sorts of "implementation details" that would otherwise impede our understanding of how a language works. For example, to a compiler writer, the representation of an environment is super important—where values are stored has a huge impact on the performance of programs. But if we just want to understand how programs behave, we don't care. And once you get used to the Greek letters and horizontal lines, you'll find them easier to read than code—*much* easier. That's why when you find a new idea in a professional paper, the idea is usually nailed down using operational semantics. When you can read operational semantics, you'll be able to learn about new ideas for yourself, direct from the sources, instead of having to find somebody to explain them to you.

What about proof theory and metatheory? Theory involves making derivations—typically one derivation at a time. It's a great guide to an implementor, because it tells you just what each construct is supposed to do. In principle, theory could also be a guide to a programmer, who also needs to know what programs are supposed to do. But in practice, derivations of operational semantics work at too low a level. A programmer is much more likely to use something like the *algebraic laws* in the next chapter. The programmer—or perhaps a specialist—uses operational semantics to show that the laws are sound, and after that, programming proceeds by appealing to the laws, not to the operational semantics directly.

So theory is good for building implementations and for establishing algebraic laws, both of which are useful for programmers. What is metatheory good for? Metatheory involves *reasoning about derivations*. In particular, can metatheory reveal universal truths about derivations, which correspond to facts about all programs in a given language. Depending on the truths in question, they might interest implementors, programmers, or even policy makers. Here are some examples:

- If you're implementing C or Impcore, you can keep the local variables and formal parameters of *all* functions on a stack (Exercise 29). This stack is called the *call stack*.
- In Impcore, no function can change the value of a formal parameter (or local variable) belonging to any other function.
- In C, a function *can* change the value of a formal parameter (or local variable) belonging to another function, but only if at some point the & operator was applied to the parameter or variable in question—or if somebody has exploited "undefined behavior" with pointers.
- If an ordinary device driver fails, it can take down a whole operating-system kernel, resulting in a "blue screen of death." But if a device driver is written in the special-purpose language Sing#, the worst thing that can happen is that you lose the device—metatheory guarantees that the operating-system kernel and the other drivers are unaffected.

Serious metatheory is well worth learning, but this book isn't the right book. What you *can* do with this book, using a few of the exercises in this chapter, is learn

67. ⟨answer transcript 67⟩≡

```

-> (define add-odds-to (n)
  [locals i sum]
  (begin
    (set i 1)
    (set sum 0)
    (while (<= i n)
      (begin
        (set sum (+ sum i))
        (set i (+ i 2))))
    sum))
-> (add-odds-to 3)
4
-> (add-odds-to 5)
9
-> (add-odds-to 7)
16

```

Figure 1.9: Programming with local variables

the difference between theory and metatheory, get an idea of how a metatheoretic proof can go, and get an idea of what metatheory can do for you.

## 1.8 EXTENDING IMPCORE

Impcore is a great “starter kit” for learning about abstract syntax, operational semantics, and interpreters. But it’s not a useful programming language—to do much of anything interesting, you probably want values that go beyond integers. New, more interesting values are coming in Chapter 2. And Chapter 6 adds arrays and Booleans to Impcore, making it a little more useful. But even with only integer values, Impcore can still be extended in two useful ways: with local variables and with looping constructs.

Any language that even pretends to be useful benefits from some kind of local variables. Exercise 30 on page 88 asks you to add local variables to Impcore. With this extension you’ll be able to define functions like the one shown in Figure 1.9, which adds up the odd numbers from 1 to n.

Adding local variables requires you to *change* the abstract syntax of Userfun so that it includes not only a body and a list of formal parameters, but also a list of local variables. And to account for the semantics of local variables, you’ll need to change the evaluator. But other kinds of extensions, including new looping constructs, can be implemented *without* touching the abstract syntax or the evaluator. You add only concrete syntax, which is implemented in terms of the abstract syntax you have already. This kind of new concrete syntax is called *syntactic sugar*.

As examples of syntactic sugar, I suggest several new ways to write loops. Let’s begin with an ordinary while loop, like this one:

```
(while (<= i n)
  (begin
    (set sum (+ sum i))
    (set i (+ i 2))))
```

The `begin`, at least to some programmers, seems like a lot of syntactic overhead. Suppose we define a new form, which I’ll call `while*`, in which the condition is still

*What is syntactic sugar and who benefits?*

As noted in the text, syntactic sugar is the name we give to a technique of defining syntax in terms of other syntax, without any operational semantics. Some examples of syntax that can be defined in this way appear in the text: C's `do...while` loop executes the body first, then the condition. And C's `for` loop has four parts: initialization, test, post-loop update, and loop body. As shown on page 69, these constructs can be defined as syntactic sugar for various combinations of `while` and `begin`. Here I discuss who benefits from this kind of definition: programmers, implementors, designers, theorists, or other tool builders.

Programmers are best served by syntactic sugar when they don't know it's there. Syntactic sugar is definition by translation, and for most people, translation is not easy to think about. If every time you want to use a `do-while` you first have to mentally translate it into something else, that's not an aid; it's a stumbling block.

Implementors sometimes benefit a little bit from syntactic sugar. You implement the translation, which is called *desugaring*, and then without any other change to your compiler or interpreter, you have a new language feature. You'll realize this benefit if you tackle Exercise 34. But as soon as your implementation gets serious—say you want to check types, as in Chapter 6, or you want to provide source-level-debugging—the syntactic sugar is not so useful, because you need to report errors or status in terms of the syntax the user wrote originally, not the desugared form.

Language designers and theorists benefit the most from syntactic sugar. For example, let's say you've completed Exercise 29: you've proven that Impcore can be evaluated on a stack. Now you want to add `do-while`, or `for`, or `while*`, or some other shiny new syntax. If all the new syntax is syntactic sugar—that is, if it is all defined by translation into the original syntax, which you used in your proof—then you know the new, extended Impcore can *still* be evaluated on a stack. You don't have to consider any new cases in your proof, and you don't have to revisit any cases that you've already proven. In my opinion, this scenario describes the most common and the most valuable use of syntactic sugar: a careful language designer benefits from small language, which is easy to prove things about, but the users benefit from a larger language, which is more attractive and makes it easier to say things idiomatically. Using syntactic sugar, a designer can have both.

In judging the utility of syntactic sugar, I assume that only language designers and implementors can create new syntactic sugar. This assumption holds not only for C and Impcore, but also for the vast majority of other languages. But for languages in the Lisp/Scheme family, ordinary programmers can create new syntactic sugar. This capability changes the game completely—it gives programmers many of the same powers as language designers. I visit the topic in more detail in Section 2.14.4 on page 173.

a single expression, but the body is a *sequence* of expressions. Now we don't need that begin:

```
(while* (≤ i n)
        (set sum (+ sum i))
        (set i (+ i 2)))
```

The `while*` loop can be defined as syntactic sugar:

$$(\text{while* } \textit{condition} \textit{ e}_1 \cdots \textit{ e}_n) \triangleq (\text{while } \textit{condition} \text{ (begin } \textit{ e}_1 \cdots \textit{ e}_n\text{)))$$

As another example, C's `do...while` loop executes the body first, then the condition. In Impcore, we might define a `do-while` as syntactic sugar:

$$(\text{do-while } \textit{body} \textit{ condition}) \triangleq (\text{begin } \textit{body} \text{ (while } \textit{condition} \textit{ body}))$$

Finally, C's complicated four-part `for` loop can also be defined as syntactic sugar:

$$(\text{for } \textit{pre} \textit{ test} \textit{ post} \textit{ body}) \triangleq (\text{begin } \textit{pre} \text{ (while } \textit{test} \text{ (begin } \textit{body} \textit{ post))))$$

You can implement these alternatives in Exercise 34 on page 89. For some sample code that adds syntactic sugar to Impcore, consult the Supplement (Section G.7 on page S217).

## 1.9 SUMMARY

Impcore is a toy, but its simple, imperative control constructs—procedure definitions, conditionals, and loops—model the structure of all procedural languages and parts of many other languages, including languages that call themselves “object-oriented,” “scripting,” and even some “functional” languages. More important, Impcore serves as a tiny, familiar medium with which to introduce two foundational ideas: abstract syntax and operational semantics. Finally, Impcore embodies the most distinctive feature of this book: to help you learn about both programming and programming languages, it provides a definitional interpreter.

### 1.9.1 Key words and phrases

#### Programming-language theory

**JUDGMENT** A judgment is a claim in a formal system of proof. In programming languages, a judgment is often about the evaluation of some syntactic form; for example,  $\langle (+ 2 2), \xi, \phi, \rho \rangle \Downarrow \langle 4, \xi, \phi, \rho \rangle$  is a judgment. (Later chapters present judgments about forms' types.)

**JUDGMENT FORM** A template for creating JUDGMENTS. For example, the form of the evaluation judgment for Impcore is  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle$ . You get from a JUDGMENT FORM to a JUDGMENT by substituting real things (e.g., ABSTRACT SYNTAX, values, ENVIRONMENTS) for METAVARIABLES.

**METAVARIABLE** A variable used in a JUDGMENT FORM or elsewhere in a THEORY, which stands for something described by the judgment form or the theory. Example metavariables include

- $e$  for an expression
- $d$  for a definition
- $x$  for a *program* variable
- $\rho$  for an ENVIRONMENT
- $\sigma$  for a store (Chapter 2)

Metavariables convey a lot of information to a skilled reader, but unfortunately, no two authors agree on what metavariables to use for what purpose. For example, while many authors, like Harper (2012) use a mix of Greek and Roman letters for metavariables, both Pierce (2002) and Cardelli (1989) use only Roman letters.

**SYNTACTIC FORM** A template for creating a phrase in a programming language, like  $(\text{set } x \ e)$ .

**SYNTACTIC CATEGORY** A group of SYNTACTIC FORMS that are *grammatically* interchangeable. Examples of common syntactic categories include expressions, definitions, statements, and types.

**GRAMMAR** A set of formal rules that enumerates all the SYNTACTIC FORMS in each SYNTACTIC CATEGORY. A grammar produces the set of all programs that are grammatically well formed. Depending on the form of the grammar, there is usually a simple decision procedure for deciding if a particular utterance was produced by the grammar, and if so, how. This decision procedure is embodied in a *parser*.

**CONCRETE SYNTAX** The means by which a program's source code is written, as a *sequence* of characters or tokens. Specifies such details as what shape of brackets to use and whether to use commas or semicolons. When we ask a computer to run a program, we express the program using concrete syntax. Likewise, when talking with people about programs, we write concrete syntax.

**ABSTRACT SYNTAX** The underlying *tree* structure of a program's source code. Called "abstract" in part because it abstracts away from such details as whether source code is written using round brackets and square brackets or whether it is written using semicolons and curly braces. To a programming-languages person, abstract syntax expresses the important truth about a program.

**SYNTACTIC SUGAR** A means of adding to concrete syntax without changing abstract syntax. Syntactic sugar is concrete syntax that is translated into existing abstract syntax. The programmer sees it, but the theory and the EVALUATOR do not. Section 1.8 on page 67 suggests several ways of using syntactic sugar to add new loop forms to Impcore.

**ENVIRONMENT** An environment stores information about names. For example, in Impcore, it stores the value of each name. Old-school compiler writers may refer to an environment as a *symbol table*.

**SYMBOL TABLE** A compiler writer's word for ENVIRONMENT.

**OPERATIONAL SEMANTICS** A precise way of describing the evaluation of a program. Usually comprises PROOF RULES for JUDGMENTS about program evaluation. The operational semantics gives enough information to write an EVALUATOR for a language.

**BIG-STEP SEMANTICS** A species of OPERATIONAL SEMANTICS in which each JUDGMENT FORM expresses, in one step, the evaluation of syntax to produce a result. For example, a single judgment  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle$  shows how an expression is evaluated to produce a value, or a judgment  $\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \phi' \rangle$  shows how a definition is evaluated to produce a new ENVIRONMENT. Big-step semantics is well aligned with the way we think about programs.

**SMALL-STEP SEMANTICS** A species of OPERATIONAL SEMANTICS in which each JUDGMENT FORM expresses the smallest possible increment of computation. Such a semantics exposes the intermediate steps of the computation. An example appears in Chapter 3. Small-step semantics can express more kinds of program behaviors than big-step semantics, and the METATHEORETIC proof techniques used with small-step semantics tend to be simpler.

**NATURAL DEDUCTION** A style of proof which is expressed using INFERENCE RULES with JUDGMENTS in the premises and conclusion. (A logician would call these judgments “propositions.”) This style, which describes the big-step semantics we are using, supposedly accords with a “natural” way of reasoning. It is associated with the German mathematician Gerhard Gentzen. The natural-deduction style of OPERATIONAL SEMANTICS was introduced by Kahn (1987).

§1.9. *Summary*

71

**INFERENCE RULE or PROOF RULE** The basic unit of a SYNTACTIC PROOF SYSTEM, an inference rule provides a means of proving one JUDGMENT. That judgment appears below a horizontal line and is called the *conclusion*. Any number of other judgments, including zero, may appear above the line; they are the *premises*. If you have a SYNTACTIC PROOF of each premise, you may combine them by *applying* or *appealing to* the inference rule (just like applying a function to arguments<sup>6</sup>), and the result is a proof of the conclusion. Because JUDGMENT FORMS often use Greek letters as METAVARIABLES and inference rules are written with horizontal lines, we sometimes joke that programming-language theory is “the theory of Greek letters and horizontal lines.”

**SYNTACTIC PROOF or DERIVATION** A formal proof formed by instantiating INFERENCE RULES to form a tree. A proof that obeys the rules is called *valid*. Given a proof and a set of rules, it is easy to decide if the proof is valid.

**ABSTRACT MACHINE** An abstraction used to evaluate programs in OPERATIONAL SEMANTICS. The state of an abstract machine may include any well-specified mathematical object; you may find an ENVIRONMENT, ABSTRACT SYNTAX, a *store* (Chapter 2), or other kind of state. An operational semantics typically specifies state transitions of an abstract machine, uses the state as context to evaluate an expression, or both.

**PARSING** The process of transforming CONCRETE SYNTAX into ABSTRACT SYNTAX. More generally, the process of recognizing concrete syntax. (Some compilers and interpreters skip the abstract-syntax step; instead they generate code directly in the parser.)

**EVALUATION** When an ordinary programmer talks about “running code,” a pointy-headed theorist talks about “evaluating ABSTRACT SYNTAX.”

**THEORY** Theorems about programs. More broadly, mathematical tools that specify meaning and behavior of programs. Useful for proving theorems about individual programs and as a specification for an evaluator or type checker. In this book, the theory of a language is its operational semantics plus, in Chapters 6 to 9, its type system. A language’s theory may be used to create a DEFINITIONAL INTERPRETER.

---

<sup>6</sup>More accurately, if you have read Chapter 8, it is like applying a value constructor to arguments.

**METATHEORY** Theorems about proofs. More broadly, mathematical tools for showing properties that are true for the execution of any program. Example properties might be that evaluating an expression never introduces a new global variable, or that Impcore can be evaluated on a stack, or that a secure language does not leak information.

**METATHEORETIC PROOF** A proof of a fact that is true of all valid derivations. Normally proceeds by STRUCTURAL INDUCTION on derivations.

**EXPRESSION-ORIENTED LANGUAGE** A programming language in which conditional constructs, control-flow constructs, and assignments, like `if`, `while`, `begin`, and `set`, are expressions, not statements—and evaluating each produces a value.

**PRIMITIVE FUNCTION** A function that is built into a language or its implementation, like the `+` function in Impcore. A primitive function typically cannot be defined using the other parts of its language, so it is considered a sort of a part of its language.

**PREDEFINED FUNCTION** A function that is available to every program written in a language, because by the time a user's code is examined, the function's definition has already been evaluated. Function `mod` is an example of a predefined function in Impcore. A predefined function is not part of its language; it can be defined using primitive functions.

**BASIS** A *basis* comprises all the information available about a *particular* set of names. It is typically a set of environments. In Impcore, a set of environments  $\langle \phi, \xi \rangle$  constitute a basis. Each definition is evaluated in the context of a basis, and evaluating the definition typically extends or alters the current basis.

**INITIAL BASIS** The BASIS used when first evaluating a user's code. The initial basis contains all the PRIMITIVE FUNCTIONS and all the PREDEFINED FUNCTIONS

**PROCEDURAL PROGRAMMING** A style of programming in which the primary mechanism of composition is sequential composition of side-effecting computations, which may be called *commands*, *statements*, or *procedures*. Procedural programs tend to manipulate one machine word at a time, and they operate primarily on mutable data. Because it offers only one data type, the number, which is immutable, Impcore is not a very good example of a procedural programming language.

### Language implementation

**READ-EVAL-PRINT LOOP** A model of interaction between a programmer and an implementation of a language. In this model, a definition or an extended definition is first *read* from standard input, then evaluated, and its result printed. And then the implementation loops, waiting for the next definition. A read-eval-print loop can also be part of a graphical user interface or other software-development environment. Alternatives to a read-eval-print loop include command-line batch development, where a programmer uses operating-system commands to compile and run code, and app development, where code is developed and packaged on one platform and then

shipped to run on another platform. The primary advantage of a read-eval-print loop is that it enables a programmer to use individual functions and to manipulate individual variables, and this facility is provided for all functions and all variables with zero effort on the programmer's part.

**PARSER** That part of a language's implementation which translates CONCRETE SYNTAX to ABSTRACT SYNTAX. (A parser may also translate concrete syntax directly to intermediate code.)

**EVALUATOR** A part of a language implementation that evaluates code directly. In this book, the evaluators evaluate ABSTRACT SYNTAX, but an evaluator may also evaluate intermediate code, virtual-machine code, or even machine code.

**DEFINITIONAL INTERPRETER** A complete implementation of a language that is intended to be a “direct” implementation of the language’s theory, or that is intended to illustrate a language’s theory. Almost sure to include a PARSE and EVALUATOR. All the interpreters in this book are definitional.

**COMPILER** A language implementation that works by translating syntax into machine code. The machine code may be for a real hardware machine made by a manufacturer like Intel or ARM, in which case the compiler is called a *native-code* compiler. Or the machine code may be for a *virtual machine* like the Java Virtual Machine or the Squeak virtual machine.

### 1.9.2 Further reading

Literate programming was invented by Knuth (1984); Noweb is described by Ramsey (1994).

The term “basis” is taken from the Definition of Standard ML (Milner et al. 1997), where it refers to a collection of environments binding not only values but types, “signatures,” and other entities.

The modern movement toward operational semantics was launched by Plotkin (1981), who describes a style of operational semantics that is better suited to proving properties of programs than the natural-deduction style that we use. The natural-deduction style was introduced by Kahn (1987); it is better suited to specifying evaluators.

The first extensible printer for C programs that I know of was created by Ken Thompson; his implementation appeared in Ninth Edition Research Unix. Another implementation can be found in Chapter 14 of Hanson (1996).

The Singularity project, which relies on the metatheory of Sing# to guarantee the behavior of its device drivers, is described by Hunt and Larus (2007).

## 1.10 EXERCISES

If you read this book without doing any of the exercises, you’ll miss most of what the book has to offer. But if you try to do *all* the exercises, you’ll die of overwork. To help you find exercises that are good for you, I’ve organized them by the skill they demand. In each chapter, you’ll find a table of exercises that lists each group of exercises along with the skills they develop and the reading that is most necessary. Skills often include programming in a bridge language, working with a semantics, and modifying an interpreter—but there are others as well. In this chapter, the skills are listed in Table 1.10 on the following page. If you choose your exercises well, doing them will be the best part of your experience.

Exercises	Section	Notes
1 to 5	1.2	Writing Impcore syntax; simple functions that could use loops or recursion.
6 to 8	1.2	Simple functions that demand recursion; functions that manipulate decimal or binary representations.
9	1.3	Syntactic structure and syntactic categories.
10 and 11	1.4, 1.5	Reading and writing judgments of operational semantics.
12 to 14	1.7	Using operational semantics to prove facts about the evaluation of particular expressions.
15 to 17	1.5	Writing new rules of operational semantics, for new features or for features not specified in the chapter.
18 to 21	1.5	Writing new proof systems for new forms of judgment.
22 to 27	1.7	Using metatheory to prove facts about all expressions.
28 and 29	1.6, 1.7	Using metatheory to prove facts about the implementation.
30 to 32	1.6	Implementing new semantics for variables.
33 and 34	1.6, 1.8	Extend the interpreter with a new primitive or new syntax.
35 and 36	1.6	Improving the performance or error behavior of the interpreter.

Table 1.10: Synopsis of all the exercises, with most relevant sections

Each chapter’s exercises are preceded by a list of *highlights*. The highlights list a few of the exercises that are my personal favorites, or that I think are the best, or that I often assign to my students. And each chapter’s exercises begin with questions that support *retrieval practice*. These very short questions will help you keep the essential ideas at the surface of your mind, so you can do the exercises fluently. And if you are a student in a university course, they may help you study for exams.

Here are the highlights of this chapter’s exercises:

- Of all the exercises on programming with numbers, Exercise 8 on page 78 is my favorite—there’s a clean, minimal solution, but to find it, you have to develop some insight into the inductive structure of numerals. If you have trouble, start with Exercise 7 on page 77.
- Exercise 16 on page 82 invites you to do a little language design: what if variables didn’t have to be declared before use?
- Exercise 12 on page 79 asks you to write a derivation; to start reasoning about derivations, follow up with Exercise 13 or 23 on pages 79 or 85.
- This is the only chapter in which you get to do much metatheory. The very best of the metatheoretic exercises are Exercises 25 and 29 on pages 85 and 87. You may have to work up to them, but if you tackle either, or better yet both, you will understand which rules of the operational semantics are boring and straightforward and which rules have interesting and important consequences. Exercises 24 and 27 on pages 85 and 86 are significantly easier but also worthwhile.
- Exercise 30 asks you to add local variables to Impcore. It’s good practice for modifying the interpreter, and it will help you think about the connection between semantics and implementation.

### 1.10.1 Retrieval practice and other short questions

- A. What's the difference between function – and function negated?
- B. What does `check-expect` do?
- C. What is the value of the global variable `it`?
- D. What is the difference between concrete syntax and abstract syntax?
- E. An example on page 28 shows an example expression along with its abstract-syntactic tree. The expression shown in the example uses the syntactic forms `SET`, `APPLY`, `VAR`, and `LITERAL`. What syntactic forms are used in the expression `(if (< x 0) (negated x) x)`?
- F. What syntactic forms are used in the expression `(* (+ x y) (- x y))`?
- G. What do the metavariables `e` and `d` stand for?
- H. What does the metavariable `x` stand for?
- I. When you evaluate `(print 4)`, why does the interpreter print 44? What should you evaluate instead?
- J. Which Greek letter holds the values of all the global variables?
- K. Which Greek letter holds the values of all of a function's formal parameters?
- L. What does  $\rho\{x \mapsto v\}$  stand for?
- M. How should you pronounce  $\rho\{x \mapsto v\}$ ?
- N. What aspect of the implementation does  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$  stand for?
- O. What is being claimed by a rule of the form 
$$\frac{J_1 \cdots J_n}{J}, \text{ where each } J \text{ is some kind of judgment?}$$

### 1.10.2 Simple functions using loops or recursion

Some of the exercises in these first three sections are adapted from Kamin (1990, Chapter 1), with permission.

1. *Understanding scope, from C to Impcore.* The scope rules of Impcore are identical to that of C. Consider this C program:

75. `⟨mystery.c 75⟩`≡

```
int x;

void R(int y) {
    x = y;
}

void Q(int x) {
    R(x + 1);
    printf("%d\n", x);
}

void main(void) {
    x = 2;
    Q(4);
    printf("%d\n", x);
}
```

- (a) For each occurrence of  $x$  in the C program, identify whether the occurrence refers to a global variable or a formal parameter.
  - (b) Say what the C program prints.
  - (c) Write, in Impcore, a sequence of four definitions that correspond to the C program. Instead of `printf`, call `println`.
2. *Sums of consecutive integers.* Define a function `sigma` satisfying the equation  $(\text{sigma } m \ n) = m + (m + 1) + \dots + n$ . The right-hand side is the sum of the elements of the set  $\{i \mid m \leq i \leq n\}$ .
  3. *Exponential and logarithm.* Define functions `exp` and `log`. When base  $b$  and exponent  $n$  are nonnegative,  $(\text{exp } b \ n) = b^n$ , and when  $b > 1$  and  $m > 0$ ,  $(\text{log } b \ m)$  is the smallest integer  $n$  such that  $b^{n+1} > m$ . On inputs that don't satisfy the preconditions, your implementation may do anything you like—even fail to terminate.
  4. *The  $n$ th Fibonacci number.* Define a function `fib` such that  $(\text{fib } n)$  is the  $n$ th Fibonacci number. The Fibonacci numbers are a sequence of numbers defined by these laws:

$$\begin{aligned}(\text{fib } 0) &= 0 \\ (\text{fib } 1) &= 1 \\ (\text{fib } n) &= (\text{fib } (-n \ 1)) + (\text{fib } (-n \ 2)) \quad \text{when } n > 1\end{aligned}$$

5. *Prime-number functions.* Define functions `prime?`, `nthprime`, `sumprimes`, and `relprime?` meeting these specifications:

$(\text{prime? } n)$  is nonzero (“true”) if  $n$  is prime and 0 (“false”) otherwise.  
 $(\text{nthprime } n)$  is the  $n$ th prime number. Consider 2 to be the first prime number, so  $(\text{nthprime } 1) = 2$ , so  $(\text{nthprime } 2) = 3$ , and so on.  
 $(\text{sumprimes } n)$  is the sum of the first  $n$  primes.  
 $(\text{relprime? } m \ n)$  is nonzero (“true”) if  $m$  and  $n$  are relatively prime—that is, their only common divisor is 1—and zero (“false”) otherwise.

Functions `prime?` and `sumprimes` expect nonnegative integers. Functions `nthprime` and `relprime?` expect *positive* integers.

### 1.10.3 A simple recursive function

6. *Binomial coefficients without arithmetic overflow.* Define a function `choose` such that  $(\text{choose } n \ k)$  is the number of ways that  $k$  distinct items can be chosen from a collection of  $n$  items. Assume that  $n$  and  $k$  are nonnegative integers. The value  $(\text{choose } n \ k)$  is called a *binomial coefficient*, and it is usually written  $\binom{n}{k}$ . It can be defined as  $\frac{n!}{k!(n-k)!}$ , but this definition presents computational problems: even for modest values of  $n$ , computing  $n!$  can overflow machine arithmetic. If you need to compute a binomial coefficient, you're better off using these identities:

$$\begin{aligned}\binom{n}{0} &= 1 && \text{when } n \geq 0 \\ \binom{n}{n} &= 1 && \text{when } n \geq 0 \\ \binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} && \text{when } n > 0 \text{ and } k > 0\end{aligned}$$

The advantage of these identities is that if the answer is small enough to fit in a machine word, then the results of all of the intermediate computations are also small enough to fit in a machine word.

#### 1.10.4 Working with decimal and binary representations

7. *Properties of decimal representations.* In this exercise, you write functions that take numbers apart and look at properties of their decimal digits. You'll need to understand how to define decimal representations inductively, as described in Exercise 18 below.

- (a) Write a function `given-positive-all-fours?`, which when given a positive number, returns 1 if its decimal representation is all fours and 0 otherwise.

**77a.** *(exercise transcripts 77a)*≡

```
-> (given-positive-all-fours? 4)  
1  
-> (given-positive-all-fours? 44444)  
1  
-> (given-positive-all-fours? 44443)  
0
```

77c▷

- (b) Write a function `all-fours?`, which when given *any* number, returns 1 if its decimal representation is all fours and 0 otherwise. You could define a function like this:

**77b.** *(unsatisfying answer 77b)*≡

```
(define all-fours? (n)  
(if (> n 0) (given-positive-all-fours? n) 0))
```

But that's unsatisfying; instead, define just one function `all-fours?`, which you could then use *in place of* `given-positive-all-fours?`.

**77c.** *(exercise transcripts 77a)*+≡

△77a 77d▷

```
-> (all-fours? 0)  
0  
-> (all-fours? -4)  
0  
-> (all-fours? 4)  
1
```

- (c) Define a function `all-one-digit?` which when given a number, returns 1 if the decimal representation of that number uses just one of the ten digits, and zero otherwise.

**77d.** *(exercise transcripts 77a)*+≡

△77c 78▷

```
-> (all-one-digit? 0)  
1  
-> (all-one-digit? -4)  
1  
-> (all-one-digit? 44443)  
0  
-> (all-one-digit? 33)  
1
```

- (d) Define a function `increasing-digits?` which when given a number, returns 1 if in the decimal representation of that number, the digits are strictly increasing, and returns zero otherwise.

78. *(exercise transcripts 77a)*  $\equiv$   
     $\rightarrow (\text{increasing-digits? } 1)$   
    1  
     $\rightarrow (\text{increasing-digits? } 1123)$   
    0  
     $\rightarrow (\text{increasing-digits? } 12435)$   
    0  
     $\rightarrow (\text{increasing-digits? } 489)$   
    1

◀ 77d

If you define these functions and are feeling bold, try Exercise 6 on page 181.

8. *Decimal-to-binary conversion.* Define a function `binary` such that  $(\text{binary } m)$  is the number whose decimal representation looks like the binary representation of  $m$ . For example  $(\text{binary } 12) = 1100$ , since  $1100_2 = 12_{10}$ . An ideal implementation of `binary` will work on any integer input, including negative ones. For example,  $(\text{binary } -5) = -101$ .

This exercise, like the previous one, will be easier if you know how to define decimal and binary representations inductively, as described in Exercise 18 below.

#### 1.10.5 Understanding syntactic structure

9. *Syntactic categories in another language.* Here's a discussion problem: Pick your favorite programming language and identify the syntactic categories.

- What syntactic categories are there?
- How are the different syntactic categories related? In particular, when can a phrase in one syntactic category be a direct part of a phrase in another (or the same) syntactic category?
- Do all the phrases in each individual syntactic category share a similar job? What is that job?
- Which syntactic categories do you have to know about to understand how the language is structured? Which categories are details that apply only to one corner of the language, and aren't important for overall understanding?

In Impcore, we might answer these questions as follows:

- The syntactic categories of Impcore are definitions and expressions.
- A definition may contain an expression, and an expression may contain another expression, but an expression may not contain a definition.
- In Impcore, the job of an expression is to be evaluated to produce a value. The job of a definition is to introduce a new name into an environment.

### 1.10.6 The language of operational semantics

10. *Understanding what judgments mean.* Take each of the following formal statements and restate it using informal English.
- Either  $x \in \text{dom } \rho$  or  $x \in \text{dom } \xi$ .
  - If  $x \in \text{dom } \xi$ , then  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle \emptyset, \xi', \phi, \rho' \rangle$ .
  - $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ .
  - $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle$ .
  - If  $\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle$  and  $\langle e_2, \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi_1, \phi, \rho_1 \rangle$ , then  $v_1 = v_2$ .
11. *Translating English into formal judgments.* Take each of the following informal statements and restate it using the formalism of operational semantics.
- Expression  $e$  can be evaluated successfully even if global variable  $x$  is not defined.
  - The result of evaluating  $e$  doesn't depend on the value of global variable  $x$ .
  - Evaluating  $e$  doesn't change the values of any global variables.
  - Evaluating  $e$  does not define any new global variables.
  - Unless function `prime?` is defined, expression  $e$  cannot evaluate successfully.
  - The evaluation of  $e$  terminates and the result is zero.
  - If the evaluation of  $e$  terminates, the result is zero.

§1.10. Exercises

79

### 1.10.7 Operational semantics: facts about particular expressions

12. *Proof of the result of evaluation.* Use the operational semantics to prove that if you evaluate `(begin (set x 3) x)` in an environment where  $\rho(x) = 99$ , then the result of the evaluation is 3. In your proof, use a formal derivation tree like the example on page 58.
13. *Proof of equivalence of two expressions.* In this exercise, you show that expression `(if x x 0)` is *observationally equivalent* to just  $x$ . That is, the two expressions can be interchanged in any program, and if we run both variants, we won't be able to observe any difference in behavior.
- Use the operational semantics to show that if there exist environments  $\xi, \phi$ , and  $\rho$  (and  $\xi', \rho', \xi'',$  and  $\rho''$ ) such that
$$\langle \text{IF}(\text{VAR}(x), \text{VAR}(x), \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$$
and
$$\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle$$
then  $v_1 = v_2$ .

**1** An imperative core  
—  
80

$$\begin{array}{c}
 \frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \quad (\text{LITERAL}) \\[10pt]
 \frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle} \quad (\text{FORMALVAR}) \\[10pt]
 \frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle} \quad (\text{GLOBALVAR}) \\[10pt]
 \frac{x \in \text{dom } \rho \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \{x \mapsto v\} \rangle} \quad (\text{FORMALASSIGN}) \\[10pt]
 \frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi' \{x \mapsto v\}, \phi, \rho' \rangle} \quad (\text{GLOBALASSIGN}) \\[10pt]
 \frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle} \quad (\text{IFTTRUE}) \\[10pt]
 \frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0 \quad \langle e_3, \xi', \phi, \rho' \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle} \quad (\text{IFFALSE}) \\[10pt]
 \frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0}{\langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle} \quad (\text{WHILEITERATE}) \\[10pt]
 \frac{\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi', \phi, \rho' \rangle} \quad (\text{WHILEEND}) \\[10pt]
 \frac{}{\langle \text{BEGIN}(), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle} \quad (\text{EMPTYBEGIN}) \\[10pt]
 \frac{\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \quad \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \quad \vdots \quad \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle} \quad (\text{BEGIN}) \\[10pt]
 \frac{\phi(f) = \text{USER}(\langle x_1, \dots, x_n \rangle, e) \quad x_1, \dots, x_n \text{ all distinct} \quad \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \quad \vdots \quad \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle}{\langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle} \quad (\text{APPLYUSER})}
 \end{array}$$

Figure 1.11: Summary of operational semantics (expressions)

$$\begin{array}{c}
 \frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{VAL}(x, e), \xi, \phi \rangle \rightarrow \langle \xi' \{x \mapsto v\}, \phi \rangle} \quad (\text{DEFINEGLOBAL}) \\[10pt]
 \frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi \{f \mapsto \text{USER}(\langle x_1, \dots, x_n \rangle, e)\} \rangle} \quad (\text{DEFINEFUNCTION}) \\[10pt]
 \frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{EXP}(e), \xi, \phi \rangle \rightarrow \langle \xi' \{\text{it} \mapsto v\}, \phi \rangle} \quad (\text{EVALEXP})
 \end{array}$$

### §1.10. Exercises

81

Figure 1.12: Summary of operational semantics (definitions)

- (b) Now use the operational semantics to show that there exist environments  $\xi, \phi, \rho, \xi'$ , and  $\rho'$  and a value  $v_1$  such that

$$\langle \text{IF}(\text{VAR}(x), \text{VAR}(x), \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$$

if and only if there exist environments  $\xi, \phi, \rho, \xi''$ , and  $\rho''$  and a value  $v_2$  such that

$$\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle.$$

Give necessary and sufficient conditions on the environments  $\xi, \phi$ , and  $\rho$  such that both expressions evaluate successfully.

- (c) Finally, extend your proof in the first part to show that not only are the results equal, but the final environments are also equal.

14. *Proof that begin can be eliminated.* Impcore can be simplified by eliminating the begin expression—every begin can be replaced with a combination of function calls. For this problem, assume that  $\phi$  binds the function second according to the following definition:

`(define second (x y) y)`

I claim that if  $e_1$  and  $e_2$  are arbitrary expressions, you can always write `(second e1 e2)` instead of `(begin e1 e2)`.

- (a) Using evaluation judgments, take the claim “you can always write `(second e1 e2)` instead of `(begin e1 e2)`” and restate the claim in precise, formal language.

*Hint:* The claim is related to the claims in Exercise 13.

- (b) Using operational semantics, prove the claim.

- (c) Define a translation for `(begin e1 ... en)` such that the translated code behaves exactly the same as the original code, but in the result of the translation, every remaining begin has exactly two subexpressions. For example, you might translate

`(begin e1 e2 e3)`

into

`(begin e1 (begin e2 e3))`

If you apply the translation recursively, then replace every begin with a call to `second`, you can eliminate begin entirely.

### 1.10.8 Operational semantics: writing new rules

- 1 An imperative core
- 
- 82
15. *Operational semantics of a for loop.* Give operational semantics for a C-like  $\text{FOR}(e_1, e_2, e_3, e_4)$ . Like a while expression, a for expression is evaluated for its side effects, so the value it returns is unimportant. Choose whatever result value you like.
  16. *Extending Impcore to allow references to unbound variables.* In Impcore, it is an error to refer to a variable that is not bound in any environment. In Awk, the first use of such a variable (either for its value or as the target of an assignment) implicitly creates a new global variable with value 0. In Icon, the rule is similar, except the implicitly created variable is a local variable, whose scope is the entire procedure in which the assignment appears.
    - (a) Change the rules of Impcore as needed, and add as many new rules as needed, to give Impcore Awk-like semantics for unbound variables.
    - (b) Change the rules of Impcore as needed, and add as many new rules as needed, to give Impcore Icon-like semantics for unbound variables.<sup>7</sup>
    - (c) Which of the two changes do you prefer, and why?
    - (d) Create a program that can distinguish standard Impcore semantics from the Awk-like and Icon-like extensions described above. In particular, create a source file `awk-icon.imp` containing a sequence of definitions with the following properties:
      - Every definition in the sequence is *syntactically* valid Impcore.
      - If you present the sequence of definitions to a standard Impcore interpreter, the result is a checked run-time error.
      - If you present the sequence of definitions to an Impcore interpreter that has been extended with the Awk-like semantics, the last thing the interpreter does is print 1.
      - If you present the sequence of definitions to an Impcore interpreter that has been extended with the Icon-like semantics, the last thing the interpreter does is print 0.
  17. *Formal semantics of unit tests.* We don't try to give a formal semantics to extended definitions. But we can write some of the pieces easily enough. These three exercises have simple solutions:
    - (a) Design a judgment form to express the idea that "a check-expect test succeeds." Your judgment form should include environments  $\phi$  and  $\xi$ . Write a proof rule for the new judgment form.
    - (b) Design a judgment form to express the idea that "a check-expect test fails." Write a proof rule for it.
    - (c) Design a judgment form to express the idea that "a check-error test fails." Write a proof rule for it.

The success of a check-error test is another matter entirely: it requires more than just a single rule. For that, look at the next exercise.

---

<sup>7</sup>Impcore has top-level expressions whereas Icon does not. For purposes of this problem, assume that every top-level expression is evaluated in its own, anonymous procedure.

### 1.10.9 Operational semantics: new proof systems

18. *Meanings of numerals.* A *numeral* is what we use to write numbers. Like an Impcore expression or definition, a numeral is syntax. We could define decimal numerals using a grammar: a decimal numeral  $N_{10}$  is composed of decimal digits  $d$ :

$$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$N_{10} ::= d \mid N_{10}d$$

§1.10. Exercises

83

In informal English, we might say that a decimal numeral is either a single decimal digit, or it is a (smaller) decimal numeral followed by a decimal digit.

The *meaning* of a numeral is a *number*—a numeral is syntax, and a number is a value. The numerals above are written in typewriter font; I write numbers in ordinary math font. The meaning of a *decimal* numeral  $N_{10}$  can be specified by a function  $\mathcal{D}$ . Function  $\mathcal{D}$  is specified by a set of recursion equations, each of the form  $\mathcal{D}[\![N_{10}]\!] = e$ , where  $e$  is a mathematical formula. (The  $[\![\dots]\!]$  symbols are called “Oxford brackets,” and they are used to wrap syntax.) There is one equation for each decimal digit, and one rule for the case where a numeral is a numeral followed by a digit:

$$\mathcal{D}[\![0]\!] = 0 \quad \mathcal{D}[\![2]\!] = 2 \quad \mathcal{D}[\![4]\!] = 4 \quad \mathcal{D}[\![6]\!] = 6 \quad \mathcal{D}[\![8]\!] = 8$$

$$\mathcal{D}[\![1]\!] = 1 \quad \mathcal{D}[\![3]\!] = 3 \quad \mathcal{D}[\![5]\!] = 5 \quad \mathcal{D}[\![7]\!] = 7 \quad \mathcal{D}[\![9]\!] = 9$$

$$\mathcal{D}[\![N_{10}d]\!] = 10 \cdot \mathcal{D}[\![N_{10}]\!] + \mathcal{D}[\![d]\!]$$

(The expression  $10 \cdot \mathcal{D}[\![N_{10}]\!]$  means “10 times  $\mathcal{D}[\![N_{10}]\!]$ ; as described in Appendix B, this book uses the  $\times$  symbol only for type theory.) In this exercise, you write a similar specification for binary numerals.

- (a) Define precisely what is a binary digit.
  - (b) Give an inductive definition of binary numerals.
  - (c) Using a meaning function  $\mathcal{B}$ , give recursion equations that define the meaning of a binary numeral.
19. *Proof systems for program analysis: having set.* Impcore is an *imperative* core because it uses side effects. Of these side effects, the most important is *mutation*, also known as assignment.<sup>8</sup> In Impcore, assignment is implemented by `set`. In this exercise, you can use proof theory to reason about a very simple property: whether an expression has `set` in it. Looking forward, in Exercise 25, you can see what you can prove if you know an expression *doesn't* have a `set`.

The easy way to see if an expression has `set` is to just look at it and see if `set` is used. But that's an instruction for a person, not an algorithm for a computer or a set of rules for a proof. Here is a proof system for a new judgment form  $\boxed{e \text{ has } \text{SET}}$ , which is intended to mean “expression  $e$  has a `set` in it.”

---

<sup>8</sup>The other side effects are printing and use.

$$\begin{array}{c}
 \text{SET} \frac{}{\text{SET}(x, e) \text{ has SET}} \\[10pt]
 \text{IF1} \frac{e_1 \text{ has SET}}{\text{IF}(e_1, e_2, e_3) \text{ has SET}} \quad \text{IF2} \frac{e_2 \text{ has SET}}{\text{IF}(e_1, e_2, e_3) \text{ has SET}} \quad \text{IF3} \frac{e_3 \text{ has SET}}{\text{IF}(e_1, e_2, e_3) \text{ has SET}} \\[10pt]
 \text{WHILE1} \frac{e_1 \text{ has SET}}{\text{WHILE}(e_1, e_2) \text{ has SET}} \quad \text{WHILE2} \frac{e_2 \text{ has SET}}{\text{WHILE}(e_1, e_2) \text{ has SET}} \\[10pt]
 \text{BEGIN} \frac{e_i \text{ has SET}}{\text{BEGIN}(e_1, \dots, e_n) \text{ has SET}}, i \in \{1, \dots, n\} \\[10pt]
 \text{APPLY} \frac{e_i \text{ has SET}}{\text{APPLY}(f, e_1, \dots, e_n) \text{ has SET}}, i \in \{1, \dots, n\}
 \end{array}$$

Here are some things to notice about the proof system:

- There are no rules for variables or for literal values. And that's as it should be: variables and literal values are expressions that don't have set.
- There's no premise on the rule for SET. A set expression definitely has set, no matter what's true about its subexpressions.
- Any of the other expressions has set if and only if one of its proper subexpressions has set. For IF and WHILE, I can write out all the cases explicitly, but for BEGIN and APPLY, I have to write *rule schemas*. The notation  $i \in \{1, \dots, n\}$  on the right-hand side of a rule means that the rule is repeated  $n$  times: once for each value of  $i$ .

Solve the following two problems:

- (a) Prove that the expression

```
(while (> x 1)
  (if (= 0 (mod x 2))
    (set x (/ x 2))
    (set x (+ (* 3 x) 1))))
```

has set.

- (b) Show that *having set* isn't the same as *evaluating set*. Give an example of an expression  $e$  such that  $e$  has SET, but you can guarantee that evaluating  $e$  never evaluates a set.

20. *Proof systems for program analysis: lacking set.* The proof system in the previous problem does a perfect job telling us that an expression has set. But if an expression *doesn't* have set, that proof system can't tell us anything! Knowing that expression doesn't have set requires its own proof system. Develop a proof system for yet another judgment form:  $\boxed{e \text{ hasn't SET}}$ . There should be a derivation of  $e \text{ hasn't SET}$  exactly when expression  $e$  doesn't have a set in it.

Your proof system should have a structure that is closely related to the structure of the proof system for  $e$  has SET. The relationship is what a mathematician would call a *dual* relationship:

- Where  $e$  has SET lacks proof rules, such as for literals and variables,  $e$  hasn't SET will have trivial proof rules with no premises.
- Where  $e$  has SET has a trivial proof rule with no premises, such as for set,  $e$  hasn't SET will lack proof rules. (There's no way you can prove that a set expression doesn't have set.)
- Where an expression has subexpressions, for  $e$  has SET it is sufficient to prove that *any* of the subexpressions has a set. But for  $e$  hasn't SET it is necessary to prove that *all* of the subexpressions haven't got set. (This duality is an instance of DeMorgan's Law.)

### §1.10. Exercises

85

The payoff for this exercise and the previous exercise lies in proving consistency: that every expression either has SET or it doesn't (Exercise 22).

21. *Proof system for checked run-time errors.* To reason about the success of a check-error test, we need to be able to prove that evaluation of an expression terminates with an error. This kind of proof requires a pretty big proof system: not quite as big as the complete operational semantics of Impcore, but bigger than the proof systems for  $e$  has SET and  $e$  hasn't SET in Exercise 19.
  - Design a judgment form to express the idea that evaluation of an expression terminates with an error. Your form will need all the same environments as the form for evaluating an expression that produces a value.
  - Write a proof system for this judgment form.
  - Design a judgment form to express the idea that “a check-error test succeeds.” Using your proof system from part (b), write a proof rule for your new judgment form.
  - If a run-time error occurs during the evaluation of a check-expect test, that test is deemed to fail. To cover this possibility, write additional proof rules for the judgment that “a check-expect test fails.”

#### 1.10.10 Metatheory: facts about derivations

22. *An expression either has SET or it doesn't.* Show that the two judgments in Exercises 19 and 20 are mutually exclusive and cover all cases. That is, for any expression  $e$ , there is a valid derivation of exactly one of the two judgments  $e$  has SET and  $e$  hasn't SET. I recommend using proof by induction on the syntactic structure of  $e$ .
23. *A WHILE expression evaluates to zero.* Prove that the value of a WHILE expression is always zero. That is, given any  $\xi, \phi, \rho, e_1$ , and  $e_2$ , if there exist a  $\xi', \rho'$ , and  $v$  such that there is a derivation of  $\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , then  $v = 0$ . Use structural induction on the derivation.
24. *Expression evaluation doesn't add or remove variables.* Prove that the execution of an Impcore expression does not change the set of variables bound in the environment. That is, prove that if  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ , then  $\text{dom } \xi = \text{dom } \xi'$ .
25. *Program analysis and expression evaluation: does lacking SET guarantee unchanged variables?* Is it true or false that evaluating an expression without a SET node does not change any environment? Use metatheory to justify your

answer. To be sure you understand what it means to have a SET node, see Exercise 19.

26. Does evaluation guarantee defined variables? Section 1.7.3 on page 62 shows how to attempt a metatheoretic proof, and it examines the conjecture “if expression  $e$  evaluates, all its variables are defined.” Section 1.7.3 addresses every rule except for WHILEITERATE and WHILEEND.
  - (a) For derivations ending in WHILEITERATE, either prove the conjecture or show an expression whose evaluation is a counterexample.
  - (b) For derivations ending in WHILEEND, either prove the conjecture or show an expression whose evaluation is a counterexample.
  - (c) Explain in informal English what is going on with while loops—when, whether, and how do we know if a while loop’s variables are defined?

27. Impcore is deterministic. Prove that Impcore is deterministic. That is, for any  $e$  and any environments, there is at most one  $v$  such that  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ .

The challenge here is that you have to reason about two potentially different derivations, each describing its own evaluation of  $e$ . Think carefully about your induction hypothesis. For example, if you want to prove that expression (begin  $e_0$   $x$ ) evaluates to at most one  $v$ , what do you need to know about the evaluation of  $e_0$ ?

#### 1.10.11 Advanced metatheory: facts about implementation

28. Updating environments in place. If we write the evaluation judgment in the form  $\langle e, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi_n, \phi, \rho_n \rangle$ , the notation suggests that between the initial state  $\langle e, \xi_0, \phi, \rho_0 \rangle$  and the final state  $\langle v, \xi_n, \phi, \rho_n \rangle$  there are  $n$  steps of computation. For example, we evaluate a variable in one step, an expression like (set  $x$  3) in two steps, and so on. I propose, as a metatheoretic conjecture, that once a step is complete, the environments in its initial state can be discarded. For example, here is a derivation  $\mathcal{D}$  for (set  $x$  3):

$$\frac{x \in \text{dom } \xi_0 \quad \frac{\xi_1 = \xi_0 \quad \rho_1 = \rho_0}{\langle 3, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 3, \xi_1, \phi, \rho_1 \rangle} \quad \xi_2 = \xi_1 \{x \mapsto 3\} \quad \rho_2 = \rho_1}{\langle (\text{set } x \text{ } 3), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 3, \xi_2, \phi, \rho_2 \rangle}$$

I now use  $\mathcal{D}$  as a subderivation in a larger derivation for (set  $y$  (set  $x$  3)):

$$\frac{\frac{y \in \text{dom } \rho_0 \quad \mathcal{D}}{\langle (\text{set } x \text{ } 3), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 3, \xi_2, \phi, \rho_2 \rangle} \quad \xi_3 = \xi_2 \quad \rho_3 = \rho_2 \{y \mapsto 3\}}{\langle (\text{set } y \text{ } (\text{set } x \text{ } 3)), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 3, \xi_3, \phi, \rho_3 \rangle}$$

These derivations are written in such a way that each part of each state is named with a metavariable. For example, in the first derivation, instead of simply writing  $\xi_0 \{x \mapsto 3\}$ , I write  $\xi_2$ , together with equations  $\xi_1 = \xi_0$  and  $\xi_2 = \xi_1 \{x \mapsto 3\}$ . The metavariables may help you see that, for example,  $\xi_0$  and  $\rho_0$  are used multiple times, but that after we prove the judgment  $\langle 3, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle 3, \xi_1, \phi, \rho_1 \rangle$ , we use  $\xi_1$  and  $\rho_1$  henceforth—environments  $\xi_0$  and  $\rho_0$  are never used again. The transition from  $\xi_1$  and  $\rho_1$  to  $\xi_2$  and  $\rho_2$  is similar, and so on. This observation suggests an optimization that is used throughout the code in this chapter: where the derivation says something

like  $\xi_2 = \xi_1\{x \mapsto 3\}$ , we don't actually have to build a fresh environment  $\xi_2$ . We can instead *implement* the semantics with a data structure that is simply updated: first the data structure holds  $\xi_1$ , then after the update, it holds  $\xi_2$ .

This exercise has two parts:

- (a) Prove that in any valid derivation, evaluation judgments can be totally ordered by their use of the global-variable environment. That is, the premises required to prove any judgment can be ordered in such a way that after the proof of every evaluation judgment, each of which takes the form  $\langle e_i, \xi_i, \phi, \rho_i \rangle \Downarrow \langle v_i, \xi_{i+1}, \phi, \rho_{i+1} \rangle$ , global-variable environment  $\xi_i$  is never used again and can be discarded.  
This metatheorem justifies using `bindval(e->set.name, v, globals)` to *overwrite* the `globals` environment when `set` is evaluated.
  - (b) The same conjecture does not hold of the formal-parameter environment. Demonstrate, via a counterexample, a valid derivation containing a judgment of the form  $\langle e_i, \xi_i, \phi, \rho_i \rangle \Downarrow \langle v_i, \xi_{i+1}, \phi, \rho_{i+1} \rangle$ , such that the next evaluation judgment has an initial state of the form  $\langle e, \xi_{i+1}, \phi, \rho \rangle$ , where  $\rho$  is independent of  $\rho_{i+1}$ , and furthermore, some *other* evaluation judgment has an initial state that depends on  $\rho_{i+1}$ . Formal-parameter environments cannot simply be mutated in place.
29. *Keeping environments on a stack.* Although proving properties of an implementation is a best practice in programming languages, it is also a bit tricky, because you save the most work by proving properties of an implementation that you have not yet built. Here I'm asking you to *imagine* an implementation that uses an *explicit* stack of environments. And to imagine the implementation, you should write a new operational semantics. (The implementation I'm asking you to imagine is halfway between the operational semantics of this chapter and a stack machine that is described in Chapter 3.)

Your new semantics should use an abstract machine that is almost identical to the Impcore machine, except the last element of the state should be a *non-empty stack* of formal-parameter environments. You can write a nonempty stack as  $\rho :: S$ , where  $S$  is a (possibly empty) stack of formal-parameter environments. A rule should affect only the top of the stack, so the judgment form should be  $\langle e, \xi, \phi, \rho :: S \rangle \Downarrow \langle v, \xi', \phi, \rho' :: S' \rangle$ .

Under this new semantics, we can think of a procedure `eval` which takes  $\rho$  off of the stack, does a bunch of computation, and just before it terminates, pushes  $\rho'$  onto the stack. The “bunch of computation” in the middle may include pushes, pops, and recursive calls to `eval`. The key property is that once  $\rho$  is popped off the stack, it is used only to make  $\rho'$  and not in any other way—so it is safe to implement  $\rho'$  by mutating  $\rho$  in place.

Prove that the following properties hold.

- (a) In `eval`, the implementation of every proof rule that ends in the judgment form  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$  can be changed to pop  $\rho$  off the stack and push  $\rho'$  onto the stack. (It is possible that  $\rho' = \rho$ .)
- (b) If  $\rho' = \rho$ , then the only copy of  $\rho$  is the one on top of the stack. If  $\rho' \neq \rho$ , then once  $\rho$  is popped off the stack, it is thrown away and never used again. In particular, no environment ever needs to be copied anywhere except on the stack; that is, the stack holds all the environments that will ever be used in any future evaluation.

From this lemma, it is an easy step to say that the operation “pop  $\rho$ ; push  $\rho'$ ” can be replaced by the operation “mutate  $\rho$  in place to become  $\rho'$ . In particular, it is safe to implement  $\rho\{x \mapsto v\}$  by mutating an existing binding, rather than by constructing a new binding. Part (b) of the lemma is crucial; it is safe to mutate  $\rho$  only because the sole copy is on top of the stack.

Your proof of the lemma should be by induction on the height of a derivation of  $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ . The base cases are the rules that have no evaluation judgments in the premises, such as the LITERAL or FORMALVAR rules. The induction steps are the rules that do have evaluation judgments as premises, such as FORMALASSIGN.

This theorem justifies our implementation of bindval, as referred to in Section 1.6.3.

#### 1.10.12 Implementation: New semantics for variables

30. *Adding local variables to Impcore.* Extend function definitions so that an Impcore function may have local variables. That is, change the concrete and abstract syntax of definitions to:

```
(define function-name (formals) [(locals locals)] expression)
Userfun = (Namelist formals, Namelist locals, Exp body)
```

where *locals*, having the same syntax as *formals*, names the function’s local variables. The square brackets in “[*(locals locals)*]” means that the *locals* declaration can be omitted; if the concrete syntax has no *locals* declaration, give the abstract syntax an empty list of *locals*.

If a local variable has the same name as a formal parameter, then in the body of the function, that name refers to the local variable. And before the body of the function is evaluated, each local variable should be initialized to zero.

The definitions of struct Userfun and function mkUserfun have to change, as does the relevant case in reduce\_to\_xdef in chunk S205d in Section G.2. But the interesting changes are in the evaluator, in eval.c. Make sure that after your changes, the interpreter still checks that the number of actual parameters is correct.

31. *Extending Impcore to work with unbound variables.* Implement your solutions to Exercise 16. Use your implementation to test the code you write to distinguish the new semantics.
32. *Passing parameters by reference.* Change the Impcore interpreter to pass parameters by *reference* instead of by value. For example, if a variable *x* is passed to a function *f*, function *f* can modify *x* by assigning to a formal parameter. If non-variable expression is passed as an argument to a function, assignments to formal parameters should have no effect outside the function. (In particular, it should not be possible to change the value of an integer literal by assignment to a formal parameter.)

To implement this change, change the return type of eval and fetchval to be Value\*, and make Valuelists hold Value\*s rather than Values. Type checking in your C compiler should help you find the other parts that need to change. No change in syntax is needed.

- (a) Is the bindval function in the environment interface still necessary?

- (b) Write a function that uses call by reference. A good candidate is a function that wants to return multiple values, like a division function that wants to return both quotient and remainder.
- (c) How does call by reference affect the truth of the assertion (page 22): “no assignment to a formal parameter can ever change the value of a global variable.”?
- (d) What are the advantages and disadvantages of reference parameters? Do you prefer Impcore with call by reference or call by value? If arrays were added to Impcore, as in Chapter 6, how would your answers change?

Justify your answers, preferably using examples and scenarios.

### §1.10. Exercises

89

#### 1.10.13 Extending the interpreter

If you intend to do some of the interpreter exercises in Chapters 2 to 4, the exercises below will help you get started (as will Exercise 30 above).

33. *Adding a new primitive.* Add the primitive `read` to the Impcore interpreter and the initial basis. Function `read` is executed for its side effect; it takes no arguments, reads a number from standard input, and returns the number.
34. *Adding new concrete syntax.* Using syntactic sugar, extend Impcore with the looping constructs discussed in Section 1.8:
  - (a) Implement the C-style `do-while`.
  - (b) Implement `while*`, which allows you to code directly for a loop that does multiple operations, without needing `begin`.
  - (c) Implement the C-style `for` loop, described as `FOR` in Exercise 15.

For some sample code that adds syntactic sugar to Impcore, see Section G.7 on page S217.

35. *Recovering lost file descriptors.* The implementation of `use` in chunk `<evaluate d->use, possibly mutating globals and functions S291c>` leaks open file descriptors when files have bugs. Explain how you would fix the problem.

#### 1.10.14 Interpreter performance

36. *Profiling.* Write an Impcore program that takes a long time to execute. Profile the interpreter.
  - (a) Approximately what fraction of time is spent in linear search? Approximately how much faster might the interpreter run if you used search trees? What about hash tables?
  - (b) Download code from Hanson (1996)<sup>9</sup> and use it to implement names and environments. How much speedup do you actually get?
  - (c) What other “hot spots” can you find? What is the best way to make the interpreter run faster?

An Impcore “program” is simply a sequence of definitions.

---

<sup>9</sup>See URL <http://www.cs.princeton.edu/software/cii>.

# CHAPTER CONTENTS

---

2.1	OVERVIEW OF $\mu$ SCHEME	92	2.10.2	Continuations for backtracking	140
2.2	LANGUAGE I: VALUES, SYNTAX, AND INITIAL BASIS	93	2.11	OPERATIONAL SEMANTICS	146
2.3	PRACTICE I: RECURSIVE FUNCTIONS ON LISTS OF VALUES	100	2.11.1	Abstract syntax and values	146
2.3.1	List basics	100	2.11.2	Variables and functions	147
2.3.2	List reversal and the method of accumulating parameters	101	2.11.3	Rules for other expressions	151
2.3.3	Lists of numbers: insertion sort	103	2.11.4	Rules for definitions	153
2.3.4	Lists of prime numbers	103	2.12	THE INTERPRETER	154
2.3.5	Coding with S-expressions	105	2.12.1	Representation of values	154
2.3.6	Inspecting multiple inputs: Equality on S-expressions	106	2.12.2	Interfaces	155
2.3.7	Lists that represent sets	106	2.12.3	Implementation of the evaluator	157
2.3.8	Association lists	107	2.12.4	Evaluating true definitions	161
2.3.9			2.12.5	Implementations of primitives	162
2.3.10			2.12.6	Memory allocation	164
2.4	DATA: RECORDS AND TREES	109	2.13	EXTENDING $\mu$ SCHEME WITH SYNTACTIC SUGAR	164
2.4.1	Records	109	2.13.1	Sugar for LET forms	165
2.4.2	Binary trees	111	2.13.2	Sugar for cond	165
2.5	COMBINING THEORY AND PRACTICE: ALGEBRAIC LAWS	112	2.13.3	Sugar for conditionals: avoiding variable capture	166
2.5.1	Laws of list primitives	113	2.13.4	Hygienic substitution	167
2.5.2	Laws and classification	113	2.13.5	Sugar for BEGIN	169
2.5.3	Boolean laws	114	2.13.6	Sugar for record definitions	169
2.5.4	Abstract finite-map laws	114	2.14	SCHEME AS IT REALLY IS	170
2.5.5	Laws that state properties of functions	115	2.14.1	Language differences	170
2.5.6	Laws and proof	115	2.14.2	Proper tail calls	172
2.5.7	Equational reasoning	116	2.14.3	Data types	172
2.5.8	Source of the induction principle: Inductively defined data	118	2.14.4	From syntactic sugar to syntactic abstraction: Macros	173
2.5.9			2.14.5	call/cc	174
2.6	LANGUAGE II: LOCAL VARIABLES AND let	119	2.15	SUMMARY	174
2.7	LANGUAGE III: FIRST-CLASS FUNCTIONS	122	2.15.1	Key words and phrases	174
2.7.1	How lambda works: Closures with mutable locations	124	2.15.2	Further reading	176
2.7.2	Useful higher-order functions	127	2.16	EXERCISES	178
2.8	PRACTICE III: HIGHER-ORDER FUNCTIONS ON LISTS	129	2.16.1	Retrieval practice	178
2.8.1	Using standard functions	129	2.16.2	Functions on lists	180
2.8.2	Visualizing the functions	131	2.16.3	Accumulating parameters	181
2.8.3	Implementing the functions	131	2.16.4	Generalizing to S-expressions	182
2.9	PRACTICE IV: HIGHER-ORDER FUNCTIONS FOR POLYMORPHISM	133	2.16.5	Records and trees	183
2.9.1	Approaches	135	2.16.6	Functional data structures	183
2.9.2	Polymorphic sort	137	2.16.7	First-order reasoning	185
2.10	PRACTICE V: CONTINUATION-PASSING STYLE	138	2.16.8	Higher-order reasoning	185
2.10.1	Continuation-passing style	140	2.16.9	Standard higher-order functions	185
2.10.2			2.16.10	Functions as arguments	187
2.10.3			2.16.11	Functions as results	189
2.10.4			2.16.12	Continuations	191
2.10.5			2.16.13	Semantics and proof	195
2.10.6			2.16.14	Semantics and design	196
2.10.7			2.16.15	Metatheory	197
2.10.8			2.16.16	Syntax and sugar	197
2.10.9			2.16.17	Implementing primitives	200
2.10.10			2.16.18	Interpreter improvement	200

## Scheme, S-expressions, and first-class functions

*Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.*

Jonathan Rees and William Clinger, editors,  
*The Revised<sup>3</sup> Report on the Algorithmic Language Scheme*

Scheme combines power and simplicity. Scheme is derived from Lisp, which John McCarthy developed—inspired in part by Alonzo Church’s work on the  $\lambda$ -calculus—while exploring ideas about computability, recursive functions, and models of computation. McCarthy intended Lisp for computing with symbolic data he called *S-expressions*. S-expressions are based on lists, and the name “Lisp” was formed from “list processing.” Lisp programs can be concise and natural programs, and they often resemble mathematical definitions of the functions they compute. Lisp has been used heavily in artificial intelligence for over fifty years, and in 1971, McCarthy received ACM’s Turing Award for contributions to artificial intelligence.

Lisp spawned many successor dialects, of which the most influential have been Common Lisp and Scheme. Common Lisp was designed to unify many of the dialects in use in the 1980s; its rich programming environment has attracted many large software projects. Scheme was designed to be small, clean, and powerful; its power and simplicity have attracted many teachers and authors like me.

Scheme was created by Guy Steele and Gerry Sussman, who introduced the main ideas in a classic series of MIT technical reports in the late 1970s, all bearing titles of the form “LAMBDA: The Ultimate (blank).” And Scheme may have been made famous by Abelson and Sussman (1985), who show off its ability to express many different programming-language ideas and to build programs in many different styles.

So what is Scheme? To answer such a question, set aside the syntax; the essence of a language lies in its values. If the essence of C is pointer arithmetic, the essence of Perl is regular expressions, and the essence of Fortran is arrays, the essence of Scheme is *lists and functions*.<sup>1</sup>

---

<sup>1</sup>Today, any list of Scheme’s essential aspects would also include hygienic macros. But hygienic macros were developed relatively late, in the 1980s and 1990s, well after the other foundations of Scheme were laid down. And unlike those foundations, macros have not colonized other languages. In this book, macros, substitution, and hygiene are just barely touched on.

Lists come from original Lisp. Lists can contain other lists, so they can be used to build records and trees. Lists of key-value pairs can act as tables. Add numbers and symbols, and lists provide everything you need for symbolic computation.

In addition to lists, Scheme provides first-class, higher-order, *nested* functions. Nested functions are created at run time by evaluating lambda expressions, which should dramatically change our thinking about programming and computation.

Scheme was meant to be small, but full Scheme is still too big for this book. Instead, we use  $\mu$ Scheme (pronounced “micro-Scheme”), a distillation of Scheme’s essential features. In this book, “Scheme” refers to ideas that Scheme and  $\mu$ Scheme share. “Full Scheme” and “ $\mu$ Scheme” refer to the large and small languages, respectively.

## 2.1 OVERVIEW OF $\mu$ SCHEME AND THIS CHAPTER

Scheme shares some central ideas with Impcore, which you know:

- Scheme encourages interactive programming with functions. Scheme programmers don’t “write a program”; they “define a function”. They don’t “run a program”; they “evaluate an expression.”
- Scheme has simple, regular syntax. It has no infix operators and therefore no operator precedence.

$\mu$ Scheme goes beyond Impcore in five significant ways:

- $\mu$ Scheme makes it easy to define local variables, which are introduced and initialized by `let` expressions.
- In addition to machine integers,  $\mu$ Scheme provides Booleans values, symbols, functions, lists of values, and a species of record structure.
- Instead of loops, Scheme programmers write recursive functions.<sup>2</sup>
- $\mu$ Scheme’s functions, unlike Impcore’s functions, are values, which can be used in the same ways as any other value. In particular, functions can be passed as arguments to other functions and returned as results from other functions; they can also be assigned to variables and even stored in lists. Such functions are said to be *first class*.

Functions that accept functions as arguments or return functions as results are called *higher-order functions*; functions that accept and return only non-functional values are called *first-order functions*. Only first-order functions are found in Impcore.

- Anonymous, nested functions are defined by a new form of expression, the `lambda` expression.

These language changes wind up changing our programming style:

- Much as arrays, loops, and assignment statements lead to a procedural style of programming, S-expressions, recursive functions, and `let` binding lead to a new, *applicative* style of programming.

<sup>2</sup>Although mathematicians have used recursion equations for centuries, computer scientists took a long time to recognize recursion as *practical*. Alan Perlis said that when some of the world’s leading computer scientists met in 1960 to design the language Algol-60, “We really did not understand the implications of recursion, or its value, ... McCarthy did, but the rest of us didn’t.” (Wexelblat 1981, p. 160).

Values	Rules
Ordinary S-expressions	1 and 2
Fully general S-expressions	1, 2, and 4
Values	1 to 4

1. A symbol, number, or Boolean is a value.
2. A list of values is a value.
3. A function is a value.
4. If  $v_1$  and  $v_2$  are values,  $(\text{cons } v_1 \ v_2)$  produces a value.

§2.2  
Language I:  
Values, syntax,  
and initial basis

93

Figure 2.1: Summary of rules for  $\mu$ Scheme values

- The ability to define first-class, *nested* functions leads to a powerful new programming technique: *higher-order programming*.

The addition of nested functions, with the ability of one function to change an enclosing function’s variables, requires a change in the semantics:

- In Impcore, environments map names to values, and assignment (`set!`) is implemented by binding new values. In  $\mu$ Scheme, environments map names to mutable *locations*, which contain values. Assignment is implemented by changing the contents of locations (Section 2.7.1).

Details are found throughout the chapter, organized as follows:

- S-expressions and other values are formed according to four simple rules (Section 2.2).
- Recursion exemplifies applicative programming (Section 2.3).
- List elements are addressed by position; to enable us to refer to an element by its name,  $\mu$ Scheme provides a record construct. Records in turn are used to represent trees (Section 2.4).
- Applicative code is clearly and succinctly specified by *algebraic laws*. Algebraic laws are much easier to work with than operational semantics; we can prove facts about applicative code using just simple algebra (Section 2.5).
- Local names are introduced by `let` bindings (Section 2.6).
- First-class, nested functions are created by lambda expressions (Section 2.7). They support list processing (Section 2.8), code reuse via *polymorphism* (Section 2.9), and backtracking search via *continuation passing* (Section 2.10).
- $\mu$ Scheme’s run-time behavior is specified by an operational semantics (Section 2.11) and implemented by an interpreter (Section 2.12).
- Conveniences like `record`, short-circuit conditionals, and other conditional forms can be implemented as syntactic sugar. Making the sugar work correctly requires attention to *substitution* and *hygiene* (Section 2.13).

## 2.2 LANGUAGE I: VALUES, SYNTAX, AND INITIAL BASIS

Scheme’s values include not only integers but also Booleans, functions, *symbols*, and *lists* of values, all of which are described below. The ones that can easily be written down—the ones that don’t involve functions—are called *S-expressions*, which is short for “symbolic expressions.”

Scheme's least familiar sort of value is the *symbol*; a symbol is a value that is a name. To paraphrase Kelsey, Clinger, and Rees (1998), what matters about symbols is that two symbols are identical if and only if their names are spelled in the same way. That is, symbols behave like Name values from the Impcore interpreter. And like Names, symbols are often used to represent identifiers in programs. They are also used in the same way that enumeration literals are used in C, C++, or Java.

The other forms of  $\mu$ Scheme value are more familiar: numbers, Booleans, lists, and functions. They are described by three inductive rules, which are summarized in Figure 2.1 on the previous page:

1. A symbol is a value, and so is a number. The Boolean values #t and #f are values; #t and #f, not 1 and 0, canonically represent truth and falsehood. Values defined by this rule are *atomic*: like atoms in the ancient Greek theory of matter, they have no observable internal structure and cannot be “taken apart.” Atomic values are also called *atoms*.
2. If  $v_1, \dots, v_n$  are values, then that list of  $n$  values is also a value, and it is written  $(v_1 \dots v_n)$ . Even when  $n = 0$ , the empty list, which is written  $()$ , is a value. And the empty list is considered to be an atom as well as a list.
3. Every function is a value.

Although rules 1 to 3 cover the common cases,  $\mu$ Scheme provides one more form of value, which can be explained only by referring to  $\mu$ Scheme's *cons* primitive, which is described below.

Values formed by rules 1 and 2 are *ordinary S-expressions*, usually called just “S-expressions.” An ordinary S-expression can be written directly in source code, preceded by a quote mark. Examples might include the following:

```
3 10 -39 44 ; numbers
#t #f ; Booleans
hello frog ; symbols
(80 87 11) ; list of numbers
(frog newt salamander) ; list of symbols
(10 lords a-leaping) ; list of mixed values
((9 ladies dancing) (8 maids a-milking)) ; list of S-expressions
```

From values we move to syntax, which is shown in Figure 2.2 on the facing page. Much of the core syntax should be familiar from Impcore; new forms include the *let* family (Section 2.6), *lambda* (Section 2.7), and lots of new *literals*. Some syntactic sugar is also new. Record definitions, short-circuit conditionals, and the *cond* form are expanded as described in Section 2.13; *when* and *unless* are expanded as described in Chapter 3 (Table 3.3 on page 206).

After values and syntax, the last element of the language is the initial basis, starting with the primitives.

- Primitives shared with Impcore include +, -, \*, /, <, and >, which implement arithmetic and comparisons. The comparisons return Booleans, not numbers: #t if the condition holds, and otherwise, #f. Applying any of these functions to a non-number is a checked run-time error, as is division by zero.

Primitives that are new with  $\mu$ Scheme include *type predicates*, which are used to identify a value's form.

- Each type predicate, *symbol?*, *number?*, *boolean?*, *null?*, or *function?*, returns #t if its argument is of the named form, #f otherwise. Predicates

```

def      ::= (val variable-name exp)
         | exp
         | (define function-name (formals) exp)
★ | (record record-name [field-name])
         | (use file-name)
         | unit-test

unit-test ::= (check-expect exp exp)
         | (check-assert exp)
         | (check-error exp)

exp    ::= literal
         | variable-name
         | (set variable-name exp)
         | (if exp exp exp)
         | (while exp exp)
         | (begin {exp})
         | (exp {exp})
         | (let-keyword ({[variable-name exp]}) exp)
         | (lambda (formals) exp)
★ | (&& {exp}) | (|| {exp})
★ | (cond {[question-exp answer-exp]})
★ | (when exp {exp}) | (unless exp {exp})

let-keyword ::= let | let* | letrec

formals  ::= {variable-name}

literal   ::= numeral | #t | #f | 'S-exp | (quote S-exp)

S-exp     ::= symbol-name | numeral | #t | #f | ({S-exp})

numeral   ::= token composed only of digits, possibly prefixed with a plus
             or minus sign

*-name   ::= token that is not a bracket, a numeral, or one of the “re-
             served” words shown in typewriter font

```

Tokens are as in Impcore, except that if a quote mark ' occurs at the beginning of a token, it is a token all by itself; e.g., 'yellow is two tokens.

Each quoted S-expression is converted to a literal value by the parser. And each record definition is expanded to a sequence of true definitions, also by the parser; in other words, a record definition is syntactic sugar (Section 1.8 on page 68), as marked by the \*. Five forms of conditional expression are also syntactic sugar. All the other forms are handled by the eval function.

Figure 2.2: Concrete syntax of  $\mu$ Scheme

`symbol?`, `number?`, `boolean?`, and `null?` identify atoms—`null?` asks if the value is an empty list. Predicate `function?` identifies functions.<sup>3</sup>

Other new primitives support lists:

- Function `cons` adds one element to the *front* of a list. If  $vs$  (pronounced “veez”) is the empty list, then  $(\text{cons } v \text{ } vs)$  is the singleton list  $(v)$ . If  $vs$  is the non-empty list  $(v_1 \dots v_n)$ , then  $(\text{cons } v \text{ } vs)$  is the longer list  $(v \text{ } v_1 \dots v_n)$ .

Scheme,

S-expressions, and  
first-class functions

2

96

In Scheme, `cons` has a quirk not common in other functional languages: it can be applied to any two values, not just to an element and a list. This quirk demands a fourth rule for the formation of  $\mu$ Scheme values:

4. If  $v_1$  and  $v_2$  are values, then  $(\text{cons } v_1 \text{ } v_2)$  produces a value.

Scheme values formed by rules 1, 2, and 4 are called *fully general S-expressions*.

Calling  $(\text{cons } v_1 \text{ } v_2)$  always produces a value, but the result a *list* of values if and only if  $v_2$  is a list of values. Values made with `cons` are identified by type predicate `pair?`.

- Calling  $(\text{pair? } v)$  returns `#t` if and only if  $v$  is a value produced by `cons`.

Lists are interrogated by other primitives:

- Function `null?` is used to distinguish an empty list from a nonempty list.
- Function `car` returns the first element of a *nonempty* list; if  $vs$  is the non-empty list  $(v_1 \dots v_n)$ , then  $(\text{car } vs)$  is  $v_1$ . If  $vs$  is any other value made with `cons`, like  $(\text{cons } v_1 \text{ } v_2)$ , then  $(\text{car } vs)$  is also  $v_1$ . If  $vs$  is  $()$  or is not made with `cons`, applying `car` to it is a checked run-time error.
- Function `cdr` returns the remaining elements of a *nonempty* list. If  $vs$  is the nonempty list  $(v_1 \dots v_n)$ , then  $(\text{cdr } vs)$  is the list  $(v_2 \dots v_n)$ , which contains all the elements of  $vs$  except the first. If  $vs$  is the singleton list  $(v_1)$ ,  $(\text{cdr } vs)$  is  $()$ . If  $vs$  is any other value made with `cons`, like  $(\text{cons } v_1 \text{ } v_2)$ , then  $(\text{cdr } vs)$  is  $v_2$ . If  $vs$  is  $()$  or is not made with `cons`, applying `cdr` to it is a checked run-time error.

The word “`cdr`” is pronounced as the word “could” followed by “er.”

Primitives `car` and `cdr` can safely be applied to any value made with `cons`.

The name `cons` stands for “construct,” which makes some kind of sense. The names `car` and `cdr`, by contrast, stand for “contents of the address part of register” and “contents of the decrement part of register,” which make sense only if we are thinking about the machine-language implementation of Lisp on the IBM 704. In the Racket dialect of Scheme, `car` and `cdr` are called by the more sensible names `first` and `rest` (Felleisen et al. 2018).

$\mu$ Scheme includes an equality primitive that works on more than just numbers:

- Function `=` tests atoms for equality. Calling  $(= v_1 \text{ } v_2)$  returns `#t` if  $v_1$  and  $v_2$  are the same *atom*. That is, they may be the same symbol, the same number, or the same Boolean, or they may both be the empty list. Given any two values that are not the same atom, including any functions or nonempty lists,  $(= v_1 \text{ } v_2)$  returns `#f`. (To compare nonempty lists, we use the non-primitive function `equal?`, which is shown in chunk 106a.)

<sup>3</sup>If you already know Scheme, or if you learn Scheme, you’ll notice some differences. For example, in full Scheme, functions are called “procedures.” S-expressions are called “datums.” The syntax of the `define` form is different, and the `val` form uses the `define` keyword. The `=` primitive works only on numbers, and it is complemented by other equality functions like `eq?`, `eqv?`, and `equal?`. Full Scheme’s `and` and `or` forms are syntax, not functions, so they short-circuit, like  $\mu$ Scheme’s `&&` and `||`.

Last, the printing primitives are like Impcore’s printing primitives. Primitives `print` and `println` work with any value.

- Calling `(print v)` prints a representation of `v`, but what you normally want is `(println v)`, which prints the value and a newline. With either one, if `v` is a fully general S-expression, then the primitive prints everything known about it. But when `v` is a function, the primitives print only “`<function>`.”
- Function `printu` prints UTF-8: If `n` is a number that corresponds to a Unicode code point, `(printu n)` prints the UTF-8 encoding of the code point. Code points newline, space, semicolon, quotemark, left-round, right-round, left-curly, right-curly, left-square, and right-square are predefined.

As in Impcore, only `println`, `print`, and `printu` have side effects; other primitives compute new values without changing anything. For example, applying `cons`, `car`, or `cdr` to a list does not change the list.

The primitives that  $\mu$ Scheme shares with Impcore are demonstrated in Chapter 1. The list primitives are demonstrated below, by applying them to values that are written as *literals*. Literals include numerals, Boolean literals, and *quoted* S-expressions, as shown in Figure 2.2 on page 95. Using quoted S-expressions, we can demonstrate `cons`, `car`, and `cdr`:

**97a.** *transcript 97a*  $\equiv$

```

-> (cons 'a '())
(a)
-> (cons 'a '(b))
(a b)
-> (cons '(a) '(b))
((a) b)
-> (cdr '(a (b (c d))))
((b (c d)))
-> (car '(a (b (c d))))
a

```

97b ▷

Primitive `null?` finds that the empty list is empty, but it finds that a singleton list containing the empty list is not empty:

**97b.** *transcript 97a*  $+ \equiv$

```

-> (null? '())
#t
-> (null? '(()))
#f

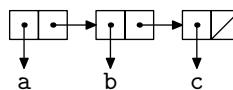
```

◁ 97a 100 ▷

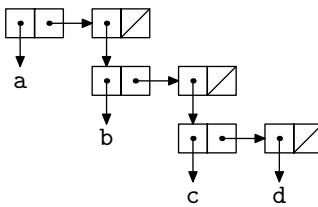
In more complex examples, the primitives’ definitions have to be used carefully; a literal S-expression might look like a long list even when it’s not. For example, `list (a (b (c d)))` looks long, but it has only two elements: the symbol `a` and the list `(b (c d))`. Its `cdr` is therefore the single-element list `((b (c d)))`.

Primitives `cons`, `car`, and `cdr` are often explained with diagrams. Any non-empty list can be drawn as a box that contains two pointers, one of which points to the `car`, and the other to the `cdr`. This box helps explain not only the behavior but also the cost of running Scheme programs, so it has a name—it is a *cons cell*. If the `cdr` of a cons cell is the empty list, there’s nothing to point to; instead, it is drawn as a slash. Using these conventions, the list `(a b c)` is drawn like this:

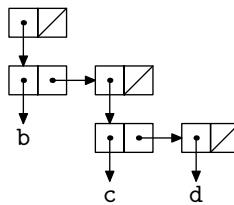
<code>car</code>	$\mathcal{P}$ 164a
<code>cdr</code>	$\mathcal{P}$ 164a
<code>cons</code>	$\mathcal{P}$ S307e
<code>null?</code>	$\mathcal{P}$ 164a



The car or cdr of a nonempty list is found by following the arrow that leaves the left or right box of the first cons cell. As another example, the S-expression (`a (b (c d))`) is drawn like this:



Its cdr is simply what the first cell's right arrow points at, which is, as above, (`((b (c d)))`):



When complex structures are built from cons cells, car and cdr are often applied several times in succession. Such applications have traditional abbreviations:

**98a.** *(predefined μScheme functions 98a)*≡

```
(define caar (xs) (car (car xs)))
(define cadr (xs) (car (cdr xs)))
(define cdar (xs) (cdr (car xs)))
```

98b▷

These definitions appear in chunk *(predefined μScheme functions 98a)*, from which they are built into the μScheme interpreter itself and are evaluated when the interpreter starts. Definitions are built in for all combinations of car and cdr up to depth five, ending with cddddd, but the others are relegated to the Supplement.

If applying car or cdr several times in succession is tiresome, so is applying cons several times in succession. Common cases are supported by more predefined functions:

**98b.** *(predefined μScheme functions 98a)*+≡

```
(define list1 (x) (cons x '()))
(define list2 (x y) (cons x (list1 y)))
(define list3 (x y z) (cons x (list2 y z)))
```

◁ 98a 101b▷

More cases, for list4 to list8, are defined in the Supplement. In full Scheme, all possible cases are handled by a single, variadic function, list, which takes any number of arguments and returns a list containing those arguments (Exercise 56).

Three predefined functions are similar but not identical to functions found in Impcore: the Boolean functions and, or, and not. Instead of Impcore's 1 and 0, they return Boolean values.

**98c.** *(definitions of predefined μScheme functions and, or, and not 98c)*≡

```
(define and (b c) (if b c b))
(define or (b c) (if b b c))
(define not (b) (if b #f #t))
```

Functions and and or inconveniently evaluate both arguments, even when the first argument determines the result. To evaluate only as much as is needed to make a decision, use the syntactic forms && and || (Exercise 53).

=, !=	Equality and inequality on atoms
equal?	Recursive equality on fully general S-expressions (isomorphism, not object identity)
/, *, -, +, mod	Integer arithmetic
>, <, >=, <=	Integer comparison
lcm, gcd, min, max	Binary operations on integers
lcm*, gcd*, max*, min*	The same operations, but taking one non-empty list of integers as argument
not, and, or	Basic operations on Booleans, which, unlike their counterparts in full Scheme, evaluate all their arguments
symbol?, number?, boolean?, null?, pair?, function?	Type predicates
atom?	Type predicate saying whether a value is an atom (not a function and not a pair)
cons, car, cdr	The basic list operations
caar, cdar, cadr, cddr, ...	Abbreviations for combinations of list operations, including also caaar, cdaar, caadr, cdadr, and so on, all the way to cddddr.
list1, list2, list3, list4, list5, list6, list7, list8	Convenience functions for creating lists, including also list5 to list8
append	The elements of one list followed by the elements of another
revapp	The elements of one list, reversed, followed by another
reverse	A list reversed
bind, find	Insertion and lookup for association lists
filter	Those elements of a list satisfying a predicate
exists?	Does any element of a list satisfy a predicate?
all?	Do all elements of a list satisfy a predicate?
map	List of results of applying a function to each element of a list
takewhile	The longest prefix of a list satisfying a predicate
dropwhile	What's not taken by takewhile
foldl, foldr	Elements of a list combined by an operator, which associates to left or right, respectively
o	Function composition
curry	The curried function equivalent to some binary function
uncurry	The binary function equivalent to some curried function
println, print	Primitives that print one value
printu	Primitive that prints a Unicode character
error	Primitive that aborts the computation with an error message

Table 2.3: The initial basis of  $\mu$ Scheme

The functions above are all part of  $\mu$ Scheme's initial basis. The entire basis, including both primitive and predefined functions, is summarized in Table 2.3 on the preceding page. The primitives are discussed above, and predefined functions that are unique to  $\mu$ Scheme are shown in this chapter in code chunks named *⟨predefined  $\mu$ Scheme functions 98a⟩*. Other predefined functions are defined exactly as in Impercore; their definitions are relegated to the Supplement.

## 2.3 PRACTICE I: RECURSIVE FUNCTIONS ON LISTS OF VALUES

Primitives `null?`, `car`, and `cdr` are used to write functions that inspect lists and their elements. In Scheme, such functions are typically recursive. Even functions that iterate are usually written recursively; Scheme programmers rarely use `set!` or `while`. To become fluent in writing your own recursive functions, you can imitate the many examples in this section:

- Functions that are agnostic about the types of list elements
- Functions that operate on lists of numbers
- Functions that operate on lists of lists—that is, S-expressions
- Functions that use lists to represent sets or finite maps

### 2.3.1 Basic principles of programming with lists

In Scheme, a list is created in one of two ways: by using `'()` or `cons`. In the style of the preceding section,

- L1. The empty list `'()` is a list of values.
- L2. If  $v$  is a value and  $vs$  is a list of values, `(cons v vs)` is a list of values.

Like any other data type that can be created in multiple ways, a list is normally consumed by a function that begins with case analysis. Such a function distinguishes cases using `null?`, and when the list is not null—it was created by using `cons`—a typical function calls itself recursively on the rest of the list (the `cdr`).

One of the simplest such functions finds the length of a list of values:

100. *⟨transcript 97a⟩* +≡  
    → (define length (xs)  
        (if (null? xs)  
            0  
            (+ 1 (length (cdr xs)))))

△97b 101a△

The `length` function typifies recursive functions that *consume* lists: when it sees an empty list, its recursion stops (the *base case*); when it sees a nonempty list, it calls itself recursively on `(cdr xs)` (an *induction step*). The name `xs` (pronounced “exes”) suggests a list of elements of unknown type; a single `x` suggests one such element.

The behavior of `length` can be summarized in two equations: one for each form that its argument can take (forms L1 and L2 above).

$$\begin{aligned} (\text{length } '()) &= 0 \\ (\text{length } (\text{cons } v \text{ } vs)) &= (+ 1 \text{ (length } vs)) \end{aligned}$$

These equations are *algebraic laws* (Section 2.5). Algebraic laws often tell us exactly what to do with each possible form of input, making them a great way to plan an implementation. In practice, algebraic laws are also used for specification, proof, optimization, and testing.

The algebraic laws for `length` specify exactly what `length` does; if you want to *use* `length` and you understand the laws, you never need to look at the code. But if

it's not obvious how the code works, it might help to look more closely at an example call. When `(length '(a b))` is evaluated, here's what happens:

- In the initial call,  $\text{xs} = (\text{a } \text{b})$ .
- The list  $(\text{a } \text{b})$  is not the empty list, so `(null? xs)` returns `#f`.
- The expression `(+ 1 (length (cdr xs)))` is evaluated, where  $\text{xs} = (\text{a } \text{b})$ . Calling `(cdr xs)` returns list  $(\text{b})$ . When `length` is applied to  $(\text{b})$ ,
  - List  $\text{xs} = (\text{b})$ .
  - List  $(\text{b})$  is not the empty list, so `(null? xs)` returns `#f`.
  - Expression `(+ 1 (length (cdr xs)))` is evaluated. Calling `(cdr xs)` returns the empty list. When `length` is applied to the empty list,
    - \* List  $\text{xs} = ()$ .
    - \* `(null? xs)` returns `#t`.
    - \* `length` returns `0`.
  - The call `(length (cdr xs))` returns `0`, so `length` returns `1`.
- The call `(length (cdr xs))` returns `1`, so `length` returns `2`.

As another example of recursive code specified using algebraic laws, let's look at the predefined function `append`. Function `append` takes two lists,  $\text{xs}$  and  $\text{ys}$ , and it returns a list that contains the elements of  $\text{xs}$  followed by the elements of  $\text{ys}$ :

**101a.** *(transcript 97a)* +≡  
 $\rightarrow (\text{append } '(\text{moon over}) '(\text{miami vice}))$   
 $(\text{moon over miami vice})$

△100 102a ▷

Interestingly, `append` never looks at  $\text{ys}$ ; it inspects only  $\text{xs}$ . And like any list,  $\text{xs}$  is formed using either `'()` or `cons`. If  $\text{xs}$  is empty, `append` returns  $\text{ys}$ . If  $\text{xs}$  is `(cons z zs)`, `append` returns  $z$  followed by  $zs$  followed by  $\text{ys}$ . The behavior of `append` can be specified precisely using two algebraic laws:

$$\begin{aligned} (\text{append } '() \text{ ys}) &= \text{ys} \\ (\text{append } (\text{cons } z \text{ zs}) \text{ ys}) &= (\text{cons } z \text{ (append } zs \text{ ys)}) \end{aligned}$$

In the code, argument  $\text{xs}$  holds  $\text{xs}$ , argument  $\text{ys}$  holds  $\text{ys}$ ,  $z$  is `(car xs)`, and  $zs$  is `(cdr xs):`

**101b.** *(predefined μScheme functions 98a)* +≡  
 $\begin{aligned} (\text{define } \text{append } (\text{xs } \text{ys})) \\ (\text{if } (\text{null? } \text{xs}) \\ \quad \text{ys} \\ \quad (\text{cons } (\text{car } \text{xs}) \text{ (append } (\text{cdr } \text{xs}) \text{ ys}))) \end{aligned}$

△98b 102b ▷

### 2.3.2 List reversal and the method of accumulating parameters

cdr P 164a  
null? P 164a

Algebraic laws can also help us design a list-reversal function—and make it efficient. To clarify the design, I condense the notation used to write laws, eliminating keywords and parentheses:

Concept	μScheme	Condensed form
Empty list	<code>'()</code>	$\epsilon$
Element followed by list	<code>(cons z zs)</code>	$z \cdot zs$
List followed by list	<code>(append xs ys)</code>	$xs \cdot ys$
List reversed	<code>(reverse xs)</code>	$R(xs)$

In condensed form, the append laws look like this:

$$\begin{aligned}\epsilon \cdot ys &= ys \\ (z \cdot zs) \cdot ys &= z \cdot (zs \cdot ys)\end{aligned}$$

And the reversal laws look like this:

Scheme,

S-expressions, and  
first-class functions

2

102

$$\begin{aligned}R(\epsilon) &= \epsilon \\ R(z \cdot zs) &= R(zs) \cdot z\end{aligned}$$

Translated back to Scheme, the reversal laws are

$$\begin{aligned}(\text{simple-reverse } '()) &= '() \\ (\text{simple-reverse } (\text{cons } z \text{ zs})) &= (\text{append } (\text{simple-reverse } zs) (\text{list1 } z))\end{aligned}$$

The code looks like this:

**102a.** *(transcript 97a)* +≡  
    → (define simple-reverse (xs)  
        (if (null? xs)  
            xs  
            (append (simple-reverse (cdr xs)) (list1 (car xs)))))  
    → (simple-reverse '(my bonny lies over))  
    (over lies bonny my)  
    → (simple-reverse '(a b (c d) e))  
    (e (c d) b a)

▷ 101a 102d ▷

This `simple-reverse` function is expensive: `append` takes  $O(n)$  time and space, and so `simple-reverse` takes  $O(n^2)$  time and space, where  $n$  is the length of the list. But list reversal can be implemented in linear time. In Scheme, reversal is made efficient by using a trick: take two lists,  $xs$  and  $ys$ , and return the reverse of  $xs$ , followed by (unreversed)  $ys$ . List  $xs$  is either empty or is  $z$  followed by  $zs$ , and the computation obeys these laws:

$$\begin{aligned}R(\epsilon) \cdot ys &= ys \\ R(z \cdot zs) \cdot ys &= (R(zs) \cdot z) \cdot ys = R(zs) \cdot (z \cdot ys)\end{aligned}$$

Translated back to Scheme, the laws for “reverse-append” are

$$\begin{aligned}(\text{revapp } '() \text{ ys}) &= ys \\ (\text{revapp } (\text{cons } z \text{ zs}) \text{ ys}) &= (\text{revapp } zs (\text{cons } z \text{ ys}))\end{aligned}$$

The code looks like this:

**102b.** *(predefined μScheme functions 98a)* +≡  
    (define revapp (xs ys) ; (reverse xs) followed by ys  
        (if (null? xs)  
            ys  
            (revapp (cdr xs) (cons (car xs) ys))))

▷ 101b 102c ▷

Function `revapp` takes time and space linear in the size of  $xs$ . Using it with an empty list makes predefined function `reverse` equally efficient.

**102c.** *(predefined μScheme functions 98a)* +≡  
    (define reverse (xs) (revapp xs '()))

▷ 102b 105a ▷

**102d.** *(transcript 97a)* +≡  
    → (reverse '(the atlantic ocean))  
    (ocean atlantic the)

▷ 102a 103a ▷

The trick used to make reversal efficient also applies to other problems. Because the parameter *ys* is used to accumulate the eventual result, the trick is called the *method of accumulating parameters*. In full Scheme, a recursive function with accumulating parameters is typically compiled into a very tight loop.

### 2.3.3 Lists of numbers: insertion sort

Functions like `length`, `append`, and `reverse` don't inspect the elements of any list—so they work on all lists of values. But other recursive functions may inspect elements and use the values to make decisions. As an example, I present a function that works only on lists of numbers: insertion sort.

Insertion sort considers one case for each form of input: an empty list of numbers is sorted, and a nonempty list (`cons m ms`), where *m* is a number and *ms* is a list of numbers, is sorted by recursively sorting *ms*, then inserting *m* into its proper position in the sorted list.

Inserting *m* into its proper position is not trivial. Function `insert` must inspect the list of numbers into which *m* is inserted, so it too is recursive. Its behavior, which can depend on the relative order of two numbers *m* and *k*, is described by these laws:

$$\begin{aligned} (\text{insert } m \ '()) &= (\text{list1 } m) \\ (\text{insert } m \ (\text{cons } k \ ks)) &= (\text{cons } m \ (\text{cons } k \ ks)), \text{ when } m < k \\ (\text{insert } m \ (\text{cons } k \ ks)) &= (\text{cons } k \ (\text{insert } m \ ks)), \text{ when } m \geq k \end{aligned}$$

In the code, if `sorted` is (`cons k ks`), then *k* is (`car sorted`) and *ks* is (`cdr sorted`).

**103a.** *(transcript 97a)* +≡  
`-> (define insert (m sorted)
 (if (null? sorted)
 (list1 m)
 (if (< m (car sorted))
 (cons m sorted)
 (cons (car sorted) (insert m (cdr sorted))))))`

◀102d 103b▶

Function `insert` is now used, in `insertion-sort`, to insert *m* into a recursively sorted list:

```
(insertion-sort '()) = '()
(insert (cons m ms)) = (insert m (insertion-sort ms))
```

**103b.** *(transcript 97a)* +≡  
`-> (define insertion-sort (ns)
 (if (null? ns)
 '()
 (insert (car ns) (insertion-sort (cdr ns)))))`  
`-> (insertion-sort '(4 3 2 6 8 5))
(2 3 4 5 6 8)`

◀103a 104a▶

<code>car</code>	$\mathcal{P}$ 164a
<code>cdr</code>	$\mathcal{P}$ 164a
<code>cons</code>	$\mathcal{P}$ S307e
<code>null?</code>	$\mathcal{P}$ 164a

### 2.3.4 Lists of prime numbers

As another example of a recursive function that uses the values of list elements, I implement a well-known algorithm for finding prime numbers. The algorithm starts with a sequence of numbers from 2 to *n*, and it produces a prime *p* by taking the first number in the sequence, then continuing recursively after removing all

multiples of  $p$ . Because it identifies a multiple of  $p$  by trying to divide by  $p$ , the algorithm is called *trial division*.<sup>4</sup>

The sequence of numbers from 2 to  $n$  is created by `(seq 2 n)`; in general, `(seq m n)` returns a list containing numbers  $m, m + 1, m + 2, \dots, n$ . A multiple of  $p$  is identified by calling `(divides? p n)`.

**104a.** *(transcript 97a)*  $\equiv$

$\triangleleft 103b\ 104b\triangleright$

```
-> (define seq (m n)
  (if (> m n) '() (cons m (seq (+ 1 m) n))))
-> (seq 3 7)
(3 4 5 6 7)
-> (define divides? (p n) (= (mod n p) 0))
```

Multiples of  $p$  are removed by calling `(remove-multiples p ns)`, which returns those elements of  $ns$  that are not multiples of  $p$ . It considers both possible forms of  $ns$ : `'()` and `(cons m ms)`. And when the input is `(cons m ms)`, the behavior depends on a relation between  $p$  and  $m$ :

```
(remove-multiples p '()) = '()
(remove-multiples p (cons m ms)) = (remove-multiples p ms), when  $p$  divides  $m$ 
(remove-multiples p (cons m ms)) = (cons m (remove-multiples p ms)), otherwise
```

The code looks like this:

**104b.** *(transcript 97a)*  $\equiv$

$\triangleleft 104a\ 104c\triangleright$

```
-> (define remove-multiples (p ns)
  (if (null? ns)
    '()
    (if (divides? p (car ns))
      (remove-multiples p (cdr ns))
      (cons (car ns) (remove-multiples p (cdr ns))))))
-> (remove-multiples 2 '(2 3 4 5 6 7))
(3 5 7)
```

Removal of multiples is the key step in trial division. The algorithm iterates over a sequence of numbers that contains a *smallest* prime  $p$  and no multiples of any prime smaller than  $p$ . Removing  $p$  and its multiples produces a prime, plus a new, smaller sequence with the same properties as the original sequence. Iteration proceeds until there are no more primes, as described by these algebraic laws:

```
(primes-in (cons p ms)) = (cons p (primes-in (remove-multiples p ms)))
(primes-in '()) = '()
```

The code looks like this:

**104c.** *(transcript 97a)*  $\equiv$

$\triangleleft 104b\ 104d\triangleright$

```
-> (define primes-in (ns)
  (if (null? ns)
    '()
    (cons (car ns) (primes-in (remove-multiples (car ns) (cdr ns))))))
```

Applying `primes-in` to `(seq 2 n)` produces all the primes up to  $n$ .

**104d.** *(transcript 97a)*  $\equiv$

$\triangleleft 104c\ 105b\triangleright$

```
-> (define primes<= (n) (primes-in (seq 2 n)))
-> (primes<= 10)
(2 3 5 7)
-> (primes<= 50)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
```

---

<sup>4</sup>This algorithm is sometimes called the Sieve of Eratosthenes, but don't be fooled: O'Neill (2009) will convince you that this algorithm is not what Eratosthenes had in mind.

### 2.3.5 Lists of lists: programming with S-expressions

Even more recursion happens when a list element is itself a list, which can contain other lists, and so on. Such lists, together with the atoms (rule 1, page 94), constitute the *ordinary S-expressions*.

An ordinary S-expression is either an atom or a list of ordinary S-expressions.<sup>5</sup> An atom is identified by predefined function `atom?`:

**105a.** *(predefined μScheme functions 98a)* +≡

```
(define atom? (x)
  (or (symbol? x) (or (number? x) (or (boolean? x) (null? x)))))
```

◀102c 106a▶

When an ordinary S-expression `sx` is passed to a function, that function must consider all possible forms of `sx`. As an example, function `has?` tells if an ordinary S-expression `sx` “has” an atom `a`.

- If `sx` is an atom, then `sx` “has” `a` if and only if `sx` is `a`. Equality of atoms is tested using the primitive function `=`.
- If `sx` is the empty list of S-expressions, it doesn’t have `a`.
- If `sx` is the nonempty list (`cons y ys`), then `sx` has `a` if either `y` has `a` or `ys` has `a`.

This specification is best written using algebraic laws:

$$\begin{aligned} (\text{has? } sx \text{ } a) &= (= sx a), \text{ when } x \text{ is an atom} \\ (\text{has? } (\text{cons } y \text{ } z) \text{ } a) &= (\text{has? } y \text{ } a) \text{ or } (\text{has? } z \text{ } a) \end{aligned}$$

These laws work with *fully general* S-expressions, which can be formed using value rule 4, not just rules 1 and 2 (Figure 2.1 on page 93). That’s why, in the law, the `cons` cell is written `(cons y z)` and not `(cons y ys)`.

In the code, if `sx` is `(cons y z)`, then `y` is `(car sx)` and `z` is `(cdr sx)`.

**105b.** *(transcript 97a)* +≡

```
> (define has? (sx a)
  (if (atom? sx)
    (= sx a)
    (or (has? (car sx) a) (has? (cdr sx) a))))
```

◀104d 105c▶

This code calls `has?` twice. It could be made faster by using an `if` expression instead of `or`, or by using short-circuit operator `||` (Section 2.13.3 on page 166).

Function `has?` can search a list of lists of symbols:

**105c.** *(transcript 97a)* +≡

```
> (val pangrams    ;; www.rinkworks.com/words/pangrams.shtml, June 2018
  '((We promptly judged antique ivory buckles for the next prize.)
    (The quick red fox jumps over a lazy brown dog.)
    (Amazingly few discotheques provide jukeboxes.)
    (Heavy boxes perform quick waltzes and jigs.)
    (Pack my box with five dozen liquor jugs.)))
> (has? pangrams 'fox)
#t
-> (has? pangrams 'box)
#t
-> (has? pangrams 'cox)
#f
```

◀105b 106b▶

car	$\mathcal{P}$ 164a
cdr	$\mathcal{P}$ 164a
cons	$\mathcal{P}$ S307e
null?	$\mathcal{P}$ 164a

<sup>5</sup>The empty list `'()` is both an atom and a list of ordinary S-expressions.

$(\text{equal? } sx_1 \ sx_2)$ $(\text{equal? } sx_1 \ (\text{cons } w \ z))$ $(\text{equal? } (\text{cons } x \ y) \ sx_2)$ $(\text{equal? } (\text{cons } x \ y) \ (\text{cons } w \ z))$	$= (= sx_1 \ sx_2),$ $\quad \quad \quad \text{if } sx_1 \text{ is an atom and } sx_2 \text{ is an atom}$ $= \#f, \text{ if } sx_1 \text{ is an atom}$ $= \#f, \text{ if } sx_2 \text{ is an atom}$ $= (\text{and} (\text{equal? } x \ w) (\text{equal? } y \ z))$
--	---

These laws call for four cases, but in an implementation, the first two laws can be combined: the second law calls for `equal?` to return `#f`, but when  $sx_1$  is an atom and  $sx_2$  is  $(\text{cons } w \ z)$ ,  $(= sx_1 \ sx_2)$  always returns false, so both cases where  $sx_1$  is an atom may use `=`:

**106a.** *⟨predefined μScheme functions 98a⟩* +≡

▷ 105a 108a ▷

```
(define equal? (sx1 sx2)
  (if (atom? sx1)
      (= sx1 sx2)
      (if (atom? sx2)
          #f
          (and (equal? (car sx1) (car sx2))
                (equal? (cdr sx1) (cdr sx2)))))))
```

The expected behavior is easily confirmed:

**106b.** *⟨transcript 97a⟩* +≡

▷ 105c 107a ▷

```
-> (equal? 'a 'b)
#f
-> (equal? '(a (1 3) c) '(a (1 3) c))
#t
-> (equal? '(a (1 3) d) '(a (1 3) c))
#f
-> (equal? '(a b c) '(a b))
#f
-> (equal? #f #f)
#t
```

More rigorous testing confirms both `#f` and (when possible) `#t` results for all four cases.

### 2.3.7 Lists that represent sets

In Scheme, as in any other language, a list with no repeated elements can represent a set. As long as the set is small, this representation is efficient, and the set operations are easy to write and to understand. Operations `emptyset`, `member?`, `add-element`, `size`, and `union` are shown below, along with their algebraic laws. Other operations appear at the end of the chapter (Exercise 3).

Function `member?` requires explicit recursion.

**107a.** *(transcript 97a)* +≡ ▷106b 107b▷

```
-> (val emptyset '())
-> (define member? (x s)      ; (member? x '())           = #f
      (if (null? s)          ; (member? x (cons x ys)) = #t
          #f                  ; (member? x (cons y ys)) = (member? x ys),
          (if (equal? x (car s)) ; when x differs from y
              #t                  ; (member? x (cdr s))))))
```

Function `add-element` might be implemented recursively, but instead it calls `member?`.

**107b.** *(transcript 97a)* +≡ ▷107a 107c▷

```
-> (define add-element (x s) ; (add-element x s) = xs, when x is in s
      (if (member? x s)        ; (add-element x s) = (cons x xs),
          s                    ; when x is not in s
          (cons x s)))
-> (val s (add-element 3 (add-element 'a emptyset)))
(3 a)
-> (member? 'a s)
#t
```

Functions `size` and `union`, like `member?`, are recursive.

**107c.** *(transcript 97a)* +≡ ▷107b 107d▷

```
-> (define size (s)          ; (size '())           = 0
      (if (null? s)          ; (size (cons x xs)) = (+ 1 (size xs))
          0
          (+ 1 (size (cdr s)))))
-> (define union (s1 s2)     ; (union '() s2)       = s2
      (if (null? s1)         ; (union (cons x xs) s2) =
          s2                  ; (add-element x (union xs s2))
          (add-element (car s1) (union (cdr s1) s2))))
-> (union s (add-element 2 (add-element 3 emptyset)))
(a 2 3)
```

Because `member?` tests for identity using `equal?`, it can recognize a list as an element of a set:

**107d.** *(transcript 97a)* +≡ ▷107c 108d▷

```
-> (val t (add-element '(a b) (add-element 1 emptyset)))
((a b) 1)
-> (member? '(a b) t)
#t
```

If `member?` used = instead of `equal?`, this last example wouldn't work; I encourage you to explain why (Exercise 4).

### 2.3.8 Association lists

A list of ordered pairs can represent a classic data structure of symbolic computing: the *finite map* (also called *associative array*, *dictionary*, and *table*). Finite maps are ubiquitous; for example, in this book they are used to represent the *environments* found in operational semantics and in interpreters. (In an interpreter or compiler, an environment is often called a *symbol table*.)

A small map is often represented as an *association list*. An association list has the form  $((k_1 a_1) \dots (k_m a_m))$ , where each  $k_i$  is a symbol, called a *key*, and each  $a_i$  is an arbitrary value, called an *attribute*. A pair  $(k_i a_i)$  is made

§2.3  
Practice I:  
Recursive  
functions on lists of  
values

107

functions on lists of  
values

107

&lt;p

with function `make-alist-pair` and inspected with functions `alist-pair-key` and `alist-pair-attribute`:

```
(alist-pair-key      (make-alist-pair k a)) = k
(alist-pair-attribute (make-alist-pair k a)) = a
```

The pair is represented by a two-element list, so the three `alist-pair` functions are implemented as follows:

**108a.** *(predefined μScheme functions 98a)* +≡

△106a 108b△

```
(define make-alist-pair      (k a)  (list2 k a))
(define alist-pair-key       (pair) (car  pair))
(define alist-pair-attribute (pair) (cadr pair))
```

A list of these pairs forms an association list, and when an association list is nonempty, the key and attribute of the first pair are retrieved by these auxiliary functions:

**108b.** *(predefined μScheme functions 98a)* +≡

△108a 108c△

```
(define alist-first-key      (alist) (alist-pair-key      (car alist)))
(define alist-first-attribute (alist) (alist-pair-attribute (car alist)))
```

An association list is operated on primarily by functions `bind` and `find`, which add bindings and retrieve attributes. Their behavior is described by these laws:

```
(bind k a '())           = (cons (make-alist-pair k a) '())
(bind k a (cons (make-alist-pair k' a') ps)) = (cons (make-alist-pair k a) ps)
(bind k a (cons (make-alist-pair k' a') ps)) =
                                              (cons (make-alist-pair k' a') (bind k a ps)),
                                              when k and k' are different
(find k '())             = '()
(find k (cons (make-alist-pair k a) ps)) = a
(find k (cons (make-alist-pair k' a) ps)) = (find k ps)
                                              when k and k' are different
```

A missing attribute is retrieved as `'()`.

**108c.** *(predefined μScheme functions 98a)* +≡

△108b 127a△

```
(define bind (k a alist)
  (if (null? alist)
    (list1 (make-alist-pair k a))
    (if (equal? k (alist-first-key alist))
      (cons (make-alist-pair k a) (cdr alist))
      (cons (car alist) (bind k a (cdr alist))))))
(define find (k alist)
  (if (null? alist)
    '()
    (if (equal? k (alist-first-key alist))
      (alist-first-attribute alist)
      (find k (cdr alist)))))
```

Function `bind` is illustrated by this contrived example, which shows how the attribute for `I` is replaced by `bind-ing I` a second time:

**108d.** *(transcript 97a)* +≡

△107d 109a△

```
-> (val demo-alist (bind 'I 'Ching '()))
((I Ching))
-> (val demo-alist (bind 'E 'coli demo-alist))
((I Ching) (E coli))
-> (val demo-alist (bind 'I 'Magnin demo-alist))
((I Magnin) (E coli))
-> (find 'I demo-alist)
Magnin
```

As a more realistic example, an association list can store the locations of groceries. Association list *aisles* shows where to find yams and butter, but not milk:

109a. *(transcript 97a)* +≡ ◀108d 109b▶

```
-> (val aisles (bind 'nog 'dairy '()))
  ((nog dairy))
-> (val aisles (bind 'apple 'produce aisles))
  ((nog dairy) (apple produce))
-> (val aisles (bind 'yam 'produce aisles))
  ((nog dairy) (apple produce) (yam produce))
-> (val aisles (bind 'butter 'dairy aisles))
  ((nog dairy) (apple produce) (yam produce) (butter dairy))
-> (val aisles (bind 'chex 'cereal aisles))
  ((nog dairy) (apple produce) (yam produce) (butter dairy) (chex cereal))
-> (find 'yam aisles)
  produce
-> (find 'butter aisles)
  dairy
-> (find 'milk aisles)
()
```

§2.4

Data: Records and  
trees

---

109

## 2.4 MORE DATA: RECORDS AND TREES

S-expressions can code all forms of lists and trees: a lot of structured data. But when all data are S-expressions, every interesting structure is made with cons cells. Such structures are traversed with combinations of car and cdr, and using car and cdr for everything is too much like programming in assembly language.

Functions like car and cdr are perfect for data whose size or structure may vary. But many data structures store a fixed number of elements in known locations, like a C struct. In Scheme, such structures can be represented by *records*. In this section, records are introduced by example, then used to represent binary trees.

### 2.4.1 Records

As an example of a record, a “frozen dinner” is a container that holds these parts:

- A protein
- A starch
- A vegetable
- A dessert

In  $\mu$ Scheme, such a record is defined like this:

109b. *(transcript 97a)* +≡ ◀109a 110a▶

```
-> (record frozen-dinner [protein starch vegetable dessert])
```

The record is a lot like a C struct or a Java object:

- Like a C struct, a frozen-dinner record always holds exactly four values: the protein, the starch, the vegetable, and the dessert. As in C, these values are called the *members* or *fields* of the record.
- As in C, each field is accessed by using its name. But where C uses *dot notation*, writing .starch after a struct (or quite commonly, ->starch after a pointer to a struct),  $\mu$ Scheme uses an *accessor function*, applying frozen-dinner-starch to a record.

$\mu$ Scheme records are allocated and initialized differently from C structs. A struct is allocated by `malloc` (or in C++, `new`), and each individual field is initialized by an individual assignment. A  $\mu$ Scheme record is allocated *and* initialized by a *constructor function*, which receives the initial values of *all* the fields as its arguments. The constructor function is a bit like a constructor in C++ or Java, except its behavior is determined by the language and cannot be changed by the programmer. The constructor function expects one argument per field of the record, in the order in which they appear, as follows:

**110a.** *(transcript 97a)*  $\equiv$  △109b 110b▷  
-> (make-frozen-dinner 'steak 'potato 'green-beans 'pie)  
(make-frozen-dinner steak potato green-beans pie)  
-> (make-frozen-dinner 'beans 'rice 'tomatillo 'flan)  
(make-frozen-dinner beans rice tomatillo flan)  
-> (frozen-dinner-starch it)  
rice

The accessor functions and constructor function are defined automatically by the record definition form, which also defines a type predicate. The type predicate `frozen-dinner?` may receive *any* value, and it returns `#t` if and only if the value was built using constructor function `make-frozen-dinner`:<sup>6</sup>

**110b.** *(transcript 97a)*  $\equiv$  △110a 110c▷  
-> (frozen-dinner? (make-frozen-dinner 'beans 'rice 'tomatillo 'flan))  
#t  
-> (frozen-dinner? '(beans rice tomatillo flan))  
#f

When a record form is evaluated, the interpreter prints the names of the functions that it defines:

**110c.** *(transcript 97a)*  $\equiv$  △110b 110d▷  
-> (record frozen-dinner [protein starch vegetable dessert])  
make-frozen-dinner  
frozen-dinner?  
frozen-dinner-protein  
frozen-dinner-starch  
frozen-dinner-vegetable  
frozen-dinner-dessert

A long name like `frozen-dinner-protein` is necessary because some other record might also have a `protein` field:

**110d.** *(transcript 97a)*  $\equiv$  △110c 111a▷  
-> (record nutrition [protein fat carbs])  
make-nutrition  
nutrition?  
nutrition-protein  
nutrition-fat  
nutrition-carbs

A short name like `.protein` won't work in Scheme, because there is no type system to tell us *which* record is meant. Long names, like `nutrition-protein` and `frozen-dinner-protein`, say explicitly which record is meant (nutritional-information records and frozen-dinner records, respectively).

---

<sup>6</sup>Not quite: In  $\mu$ Scheme, but not in full Scheme, the type predicate can be fooled by a record that is forged using `cons` (Section 2.13.6).

Functions defined by record satisfy algebraic laws, which say that we get out what we put in:

```
(frozen-dinner? (make-frozen-dinner p s v d)) = #t
(frozen-dinner? v) = #f,
where v is not made by make-frozen-dinner
(frozen-dinner-protein (make-frozen-dinner p s v d)) = p
(frozen-dinner-starch (make-frozen-dinner p s v d)) = s
(frozen-dinner-vegetable (make-frozen-dinner p s v d)) = v
(frozen-dinner-dessert (make-frozen-dinner p s v d)) = d
```

§2.4  
Data: Records and  
trees

---

111

#### 2.4.2 Binary trees

Records, like C structs, make great tree nodes. For example, in a binary tree, a node might contain a tag and two subtrees. A binary tree is either such a node or is empty. A node can be represented as a node record, and the empty tree can be represented as #f.

**111a.** *(transcript 97a)* +≡

```
-> (record node [tag left right])
make-node
node?
node-tag
node-left
node-right
```

△110d 111b▷

The node functions obey these laws:

```
((node?) (make-node t left right)) = #t
((node?) atom) = #f
(node-tag (make-node t left right)) = t
(node-left (make-node t left right)) = left
(node-right (make-node t left right)) = right
```

The set of tagged binary trees can be defined precisely, by induction. The set  $BINTREE(T)$  contains tagged binary trees with tags drawn from set  $T$ :

$$\frac{\#f \in BINTREE(T)}{l \in T \quad t_1 \in BINTREE(T) \quad t_2 \in BINTREE(T) \quad (make-node l t1 t2) \in BINTREE(T)}$$

Such a tree is either built with `make-node` or is #f, as in this example:

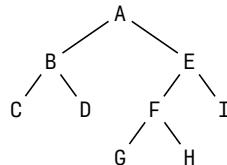
**111b.** *(transcript 97a)* +≡

```
-> (val example-sym-tree
  (make-node 'A
    (make-node 'B
      (make-node 'C #f #f)
      (make-node 'D #f #f)))
  (make-node 'E
    (make-node 'F
      (make-node 'G #f #f)
      (make-node 'H #f #f)))
  (make-node 'I #f #f))))
```

△111a 112a▷

frozen-dinner-	
starch	109b
frozen-dinner?	109b
make-frozen-dinner	109b

The example tree might be drawn as follows:



Scheme,

S-expressions, and  
first-class functions

2

112

Tagged binary trees are consumed by functions whose internal structure follows the structure of the input. Just as a list-consuming function must handle two forms, `cons` and `'()`, a tree-consuming function must handle two forms: `make-node` and `empty-tree`. The forms can be distinguished by predicate `empty-tree?`:

**112a.** *(transcript 97a)*  $\equiv$   $\triangleleft$  111b 112b  $\triangleright$   
-> (define empty-tree? (tree) (= tree #f))

One example tree-consuming function is a classic tree traversal: the preorder traversal:

**112b.** *(transcript 97a)*  $\equiv$   $\triangleleft$  112a 120a  $\triangleright$   
-> (define preorder (tree)  
 (if (empty-tree? tree)  
 '()  
 (cons (node-tag tree)  
 (append  
 (preorder (node-left tree))  
 (preorder (node-right tree)))))))  
-> (preorder example-sym-tree)  
(A B C D E F G H I)

Inorder and postorder traversals are left for you to implement (Exercise 11).

## 2.5 COMBINING THEORY AND PRACTICE: ALGEBRAIC LAWS

Throughout this chapter, functions are described by algebraic laws. The laws help us understand what the code does (or what it is supposed to do). Algebraic laws also help with design: they can show what cases need to be handled and what needs to be done in each case. Translating such laws into code is much easier than writing correct code from scratch.

Algebraic laws have many other uses. Any algebraic law can be turned into a test case by substituting example data for metavariables; for example, QuickCheck (Claessen and Hughes 2000) automatically substitutes a random input for each metavariable. Algebraic laws are also used to specify the behavior of abstract types, to simplify code, to improve performance, and even to prove properties of code.

Algebraic laws work by specifying equalities: in a valid law, whenever values are substituted for metavariables, the two sides are equal. (The values substituted must respect the conditions surrounding the law. For example, if `ns` stands for a list of numbers, we may not substitute a Boolean for it.) The substitution principle extends beyond values; a valid law also holds when *program variables* are substituted for metavariables, and even when *pure expressions* are substituted for metavariables. A pure expression is one whose evaluation has no side effects: it does not change the values of any variables and does not do any input or output. And for our purposes, a pure expression runs to successful completion; if an expression's evaluation doesn't terminate or triggers a run-time error, the expression is considered impure.

The equality specified by an algebraic law is a form of *observational equivalence*: if  $e_1 = e_2$ , and a program contains  $e_1$ , we can replace  $e_1$  with  $e_2$ , and the program

### 2.5.1 Laws of list primitives

Algebraic laws can be used to specify the behaviors of primitive functions. After all, programmers never need to see implementations of `cons`, `car`, `cdr`, and `null?`; we just need to know how they behave. Their behavior can be specified by operational semantics, but operational semantics often gives more detail than we care to know. For example, if we just want to be able to use `car` and `cdr` effectively, everything we need to know is captured by these two laws:

$$\begin{aligned} (\text{car } (\text{cons } x \ y)) &= x \\ (\text{cdr } (\text{cons } x \ y)) &= y \end{aligned}$$

These laws also tell us something about `cons`: implicitly, the laws confirm that `cons` may be applied to *any* two arguments  $x$  and  $y$ , even if  $y$  is not a list (rule 4, page 93). Use of `cons` cells and '`()`' to represent lists is merely a programming convention.

To capture `cons` completely also requires laws that tell us how a `cons` cell is viewed by a type predicate, as in these examples:

$$\begin{aligned} (\text{pair? } (\text{cons } x \ y)) &= \#t \\ (\text{null? } (\text{cons } x \ y)) &= \#f \end{aligned}$$

The laws above suffice to enable us to use `cons`, `car`, and `cdr` effectively. Nothing more is required, and any implementation that satisfies the laws is as correct as any other. To develop an unusual implementation, try Exercise 39.

### 2.5.2 Developing laws by classifying operations

How many laws are enough? To know if we have enough laws to describe a data type  $T$ , we analyze the functions that involve values of type  $T$ .

- A function that makes a new value of type  $T$  is a *creator* or a *producer*. A creator is either a value of type  $T$  all by itself, or it is a function that returns a value of type  $T$  without needing any arguments of type  $T$ . As an example, '`()`' is a creator for lists. A producer is a function that takes at least one argument of type  $T$ , and possibly additional arguments, and returns a value of type  $T$ . As an example, `cons` is a producer for lists.

<code>cons</code>	$\mathcal{P}$	S307e
<code>example-sym-tree</code>		111b
<code>node-left</code>	111a	
<code>node-right</code>	111a	
<code>node-tag</code>	111a	

Creators and producers are sometimes grouped into a single category called *constructors*, but “constructor” is a slippery word. The grouping usage comes from algebraic specification, but “constructor” is also used in functional programming and in object-oriented programming—and in each community, it means something different.

- A function that takes an argument of type  $T$  and gets information out of it is an *observer*. An observer “looks inside” its argument and produces some fact about the argument, or perhaps some constituent value, which may or

may not also be of type  $T$ . Observers are sometimes also called *selectors* or *accessors*. As examples, primitives `car`, `cdr`, `pair?`, and `null?` are observers for lists.

- Creators, producers, and observers have no side effects. A function that has side effects on an existing value of type  $T$  is a *mutator*. Mutators, too, can fit into the discipline of algebraic specification. An explanation in depth is beyond the scope of this book, but a couple of simple examples appear in Section 9.6.2 on page 557. For more, see the excellent book by Liskov and Guttag (1986).

This classification of functions tells us how many laws are enough: there are enough laws for type  $T$  if the laws specify the result of every permissible combination of observers applied to creators and producers. By this criterion, our list laws aren't yet complete; they don't specify what happens when observers are applied to the empty list. Such observations are specified by these laws:

$$\begin{aligned} (\text{pair? } '()) &= \#f \\ (\text{null? } '()) &= \#t \end{aligned}$$

Not all observations of the empty list are permissible. An observation would cause an error, like `(car '())`, isn't specified by any law, and so it is understood that the observation is impermissible. This convention resembles the convention of the operational semantics, where if an evaluation causes an error, no rule applies.

Laws for rich data structures can be extensive. Because S-expressions include both lists and atoms, they have lots of creators, producers, and especially observers. Laws for all combinations would be overwhelming; only a few Boolean laws are sketched below.

### 2.5.3 Boolean laws

For the Booleans, values `#t` and `#f` act as creators, the syntactic form `if` acts like an observer, and the predefined function `not` is a producer. Their interactions are described by these laws:

$$\begin{aligned} (\text{if } \#t \ x \ y) &= x \\ (\text{if } \#f \ x \ y) &= y \\ (\text{if } (\text{not } p) \ x \ y) &= (\text{if } p \ y \ x) \\ (\text{if } p \ \#f \ \#t) &= (\text{not } p) \end{aligned}$$

If  $p$  is guaranteed to be Boolean, one more law is valid:

$$(\text{if } p \ \#t \ \#f) = p$$

This law, which is frequently overlooked, is also valid if the result of the `if` expression is used only as a condition in other `if` expressions (or `while` expressions). Whenever possible, it should be used to simplify code.

### 2.5.4 Abstract finite-map laws

Laws for association lists (Section 2.3.8) tell us that `find` returns the most recent property added with `bind`:

$$\begin{aligned} (\text{find } k \ (\text{bind } k \ a \ m)) &= a \\ (\text{find } k \ (\text{bind } k' \ a \ m)) &= (\text{find } k \ m), \text{ when } k \text{ is different from } k' \end{aligned}$$

These laws are justified by appealing to the implementations in Section 2.3.8. But when a finite map gets large, those implementations aren't good enough; we need more a efficient data structure, like a binary search tree or even a red-black tree. These data structures, or any data structure that obeys the same laws, can serve as a drop-in replacement for the less efficient association list. (In another data structure, application of `find` to the empty map might cause an error, so no law for that case is included.)

### 2.5.5 Laws that state properties of functions

Not every set of laws fully specifies the behavior of the functions it describes. Sometimes a law states a property that a function ought to have, without nailing down its behavior in every case—as in the following examples:

$$\begin{aligned}
 (\text{and } p \ q) &= (\text{and } q \ p) \\
 (\text{and } p \ \#t) &= p \\
 (\text{and } p \ \#f) &= \#f \\
 (\text{or } p \ q) &= (\text{or } q \ p) \\
 (\text{or } p \ \#t) &= \#t \\
 (\text{or } p \ \#f) &= p \\
 (\text{append } (\text{cons } x \ '()) \ xs) &= (\text{cons } x \ xs) \\
 (\text{append } (\text{append } xs \ ys) \ zs) &= (\text{append } xs \ (\text{append } ys \ zs)) \\
 (\text{reverse } (\text{reverse } xs)) &= xs
 \end{aligned}$$

These laws, which I often use in my own programming, are great for simplifying code. They also make excellent laws for “property-based” testing (Claessen and Hughes 2000).

### 2.5.6 Laws and proof

Algebraic laws enable a new, elegant form of proof. What good is a new form of proof? It proves more interesting facts with less work than operational semantics.

In principle, anything we might prove about a  $\mu$ Scheme program can be proved using the operational semantics. But using operational semantics to prove properties of programs is like using assembly language to write them: it operates at so low a level that it's practical only for small problems. Operational semantics is best used only to prove laws about primitive functions and syntactic forms. Those laws can then be used to prove laws about functions, which can be used to prove laws about other functions, and so on. In proof, laws play a similar role to the role that functions play in coding: they enable us to break problems down hierarchically.

These hierarchical proofs are founded on proofs about primitives and syntax. Unfortunately, the foundational proofs can be quite challenging; for example, although no program can tell the difference between the two expressions `(let ([x 1983]) x)` and just `1983`, they do not have exactly the same semantics: one allocates a fresh location and the other doesn't. The extra location can't be observed, but to prove it requires techniques that are far beyond the scope of this book.

What we do in this book is freely substitute one pure expression for another, provided they always evaluate to equal values—even if they don't have identical effects on the store. This *substitution of equals for equals* is a simple, powerful proof technique, and it works not just on primitives and simple syntactic forms, but also on function applications. To substitute for a function application, we expand the body of the function by replacing each formal parameter with the corresponding

actual parameter. As long as there are no side effects, this too is substitution of equals for equals.

Substitution is justified by algebraic laws: an algebraic law says, “these two sides are equal, and so one may be freely substituted for the other.” This technique is called *equational reasoning*, and the resulting proofs are sometimes called *calculational proofs*. It is demonstrated in the next section.

Scheme,

S-expressions, and

first-class functions

2

116

### 2.5.7 Using equational reasoning to write proofs

Equational reasoning can be used to prove the laws for append and length given in Section 2.3.1. The first append law on page 101 says that  $(\text{append } '() \ ys) = ys$ . To prove it, I first substitute for append’s formal parameters, then apply the “null-empty” law from Section 2.5.2 and the “if-true” law from Section 2.5.3:

```
(append '() ys)
  = {substitute actual parameters '() and ys in definition of append}
(if (null? '()))
  ys
  (cons (car '()) (append (cdr '()) ys)))
  = {null-empty law}
(if #t
  ys
  (cons (car '()) (append (cdr '()) ys)))
  = {if-#t law}
ys
```

Each step in the proof is a single equality, presented in this form:

$$\frac{\begin{matrix} term_1 \\ = \{ \text{justification that } term_1 = term_2 \} \\ term_2 \end{matrix}}{}$$

These steps are chained together to show that every term is equal to every other term, and in particular that the first term is equal to the last term. In the example above, the chain of equalities establishes that  $(\text{append } '() \ ys) = ys$ . The append-cons law,  $(\text{append } (\text{cons } z \ zs) \ ys) = (\text{cons } z \ (\text{append } zs \ ys))$ , is proved in similar fashion.

The key step in the proof is the expansion of the definition of append. As another example of such an expansion, I prove that the length of a cons cell is one more than the length of the second argument:

$$(\text{length } (\text{cons } y \ ys)) = (+ 1 \ (\text{length } ys))$$

Again the expansion is the first step: in the definition of length, the actual parameter  $(\text{cons } y \ ys)$  is substituted for the formal parameter xs:

```
(length (cons y ys))
  = {substitute actual parameter in definition of length}
(if (null? (cons y ys))
  0
  (+ 1 (length (cdr (cons y ys)))))
  = {null?-cons law}
```

```

(if #f
  0
  (+ 1 (length (cdr (cons y ys)))))
  = {if-#f law}
(+ 1 (length (cdr (cons y ys))))
  = {cdr-cons law}
(+ 1 (length ys))

```

Expanding a function's definition works, but it can make a proof long, verbose, and hard to follow. A function's definition should be expanded as little as possible—just enough to prove the laws that describe its implementation. Then, in future proofs, only those laws are needed. As an example of using such a law, I prove that appending a singleton list is equivalent to `cons`. I still substitute actual parameters, but now instead of substituting them for the formal parameters of `append`, I substitute them for the metavariables of the `append-cons` law on the preceding page. The key step is again the first step, where I substitute `x` for `z` and `'()` for `zs`, leaving `ys` unchanged:

```

(append (cons x '()) ys)
  = {substitute actual parameters in the append-cons law}
(cons x (append '() ys))
  = {apply the append-empty law, substituting the right-hand side for the left}
(cons x ys)

```

The examples above prove properties about the application of `append` or `length` to a list that is known to be short (empty or singleton). But useful properties aren't limited to short lists; for example, appending two lists adds their lengths, even when the first argument to `append` is arbitrarily long. The computation involves a recursive call to `append`, and proving general facts about recursive functions usually demands proof by induction. In particular, laws about lists and S-expressions are proved by *structural induction*:

- Prove the law holds for every base case. In the case of a list, prove that the law holds when the list is empty.
- Prove every induction step by assuming the law holds for the constituents. Again in the case of a list, prove the case for a nonempty list `(cons z zs)` by assuming the induction hypothesis for the smaller list `zs`.

As an example, I prove the law

$$(\text{length} (\text{append} xs ys)) = (+ (\text{length} xs) (\text{length} ys))$$

The proof is by structural induction on `xs`. In the base case, `xs` is empty:

```

(length (append '() ys))
  = {append-empty law}
(length ys)
  = {zero is the additive identity}
(+ 0 (length ys))
  = {length-empty law, from right to left}
(+ (length '()) (length ys))

```

In the induction step,  $xs$  is not empty, and therefore there exist a  $z$  and  $zs$  such that  $xs = (\text{cons } z \ zs)$ .

$$\begin{aligned}
& (\text{length} (\text{append } xs \ ys)) \\
&= \{\text{by assumption that } xs \text{ is not empty, } xs = (\text{cons } z \ zs)\} \\
& (\text{length} (\text{append} (\text{cons } z \ zs) \ ys)) \\
&= \{\text{appeal to the append-cons law}\} \\
& (\text{length} (\text{cons } z (\text{append } zs \ ys))) \\
&= \{\text{appeal to the length-cons law}\} \\
& (+ 1 (\text{length} (\text{append } zs \ ys))) \\
&= \{\text{apply the induction hypothesis}\} \\
& (+ 1 (+ (\text{length } zs) (\text{length } ys))) \\
&= \{\text{associativity of } +\} \\
& (+ (+ 1 (\text{length } zs)) (\text{length } ys)) \\
&= \{\text{length-cons law, from right to left}\} \\
& (+ (\text{length} (\text{cons } z \ zs)) (\text{length } ys)) \\
&= \{\text{by the initial assumption that } xs = (\text{cons } z \ zs)\} \\
& (+ (\text{length } xs) (\text{length } ys))
\end{aligned}$$

These examples should teach you enough so that you can do Exercises 16 to 26 starting on page 185.

### 2.5.8 Source of the induction principle: Inductively defined data

For a law like  $(\text{length} (\text{append } xs \ ys)) = (+ (\text{length } xs) (\text{length } ys))$ , what cases need to be proved? One for each possible way that  $xs$  can be constructed. Since  $xs$  is a list of values, I claim there are only two ways:  $'()$  and  $\text{cons}$ . This claim can be made precise with the help of a proof system that says what a list is.

“List” is a *parametric abstraction*; for example, there are “lists of numbers,” “lists of Booleans,” “lists of values,” and so on. To notate the set of all lists whose elements are in set  $A$ , I write  $\text{LIST}(A)$ . This set can be defined by a proof system for set membership. The judgment form is  $v \in \text{LIST}(A)$ , and it is proved by showing that  $v$  is either  $'()$  or a suitable  $\text{cons}$  cell:

$$\frac{\text{EMPTYLIST}}{'() \in \text{LIST}(A)} \qquad \frac{\text{CONSLIST} \quad a \in A \quad as \in \text{LIST}(A)}{(\text{cons } a \ as) \in \text{LIST}(A)}$$

A value  $v$  is in  $\text{LIST}(A)$  if and only if there is a proof of  $v \in \text{LIST}(A)$ .

Now let’s prove that some property  $P$  holds for every  $v$  in  $\text{LIST}(A)$ . Formally that means, “if there is a derivation of  $v \in \text{LIST}(A)$ , then  $P(v)$ .” The proof is a metatheoretic proof! It’s just like the metatheoretic proofs in Chapter 1 (page 60), except the underlying proof system isn’t the operational semantics of Impcore—it’s the proof system for  $v \in \text{LIST}(A)$ . But the structure is the same; in particular, a proof about  $\text{LIST}(A)$  needs a case for each rule in the proof system for  $v \in \text{LIST}(A)$ , and that means one case for  $'()$  and one for  $\text{cons}$ .

Metatheoretic proof techniques apply to other sets of values, provided those sets are defined inductively, by a proof system. For example, S-expressions can be defined by a proof system. The forms of S-expression that are of most interest to Scheme programmers are the *ordinary S-expressions*  $\text{SEXP}_O$  (the ones that can be written with quote) and the *fully general S-expressions*  $\text{SEXP}_{FG}$  (the ones that are made with atoms and cons). To write their proof systems, I write  $\text{SYM}$ ,  $\text{NUM}$ ,

and *BOOL* for primitive sets of atoms. An ordinary S-expression is either an atom or a list of S-expressions, and its judgment form is  $v \in \text{SEXP}_O$ :

$$\frac{v \in \text{SYM}}{v \in \text{SEXP}_O} \quad \frac{v \in \text{NUM}}{v \in \text{SEXP}_O} \quad \frac{v \in \text{BOOL}}{v \in \text{SEXP}_O} \quad \frac{v \in \text{LIST}(\text{SEXP}_O)}{v \in \text{SEXP}_O}$$

A fully general S-expression is either an atom or a pair of S-expressions:

$$\frac{v \in \text{SYM}}{v \in \text{SEXP}_{FG}} \quad \frac{v \in \text{NUM}}{v \in \text{SEXP}_{FG}} \quad \frac{v \in \text{BOOL}}{v \in \text{SEXP}_{FG}} \quad \frac{}{'() \in \text{SEXP}_{FG}}$$

$$\frac{v_1 \in \text{SEXP}_{FG} \quad v_2 \in \text{SEXP}_{FG}}{(\text{cons } v_1 v_2) \in \text{SEXP}_{FG}}$$

§2.6

Language II: Local variables and let

119

Not all fully general S-expressions can be written with  $\mu$ Scheme's quote form; for example, the result of evaluating  $(\text{cons } 2 2)$  cannot be written with a quote.

Inductively defined data is so common that it's useful to have a shorthand notation for it. The most common notations, including the datatype definition forms of Standard ML (Chapter 5) and  $\mu$ ML (Chapter 8), are related to *recursion equations*. For example, the simplest recursion equation for  $\text{LIST}(A)$  looks like this:

$$\text{LIST}(A) = \{ '() \} \cup \{ (\text{cons } v vs) \mid v \in A \wedge vs \in \text{LIST}(A) \}$$

This equation can't quite be taken as a definition, unless we say that  $\text{LIST}(A)$  is the *smallest* set that satisfies the equation.

S-expressions can be described by similar equations:

$$\begin{aligned} \text{SEXP}_O &= \text{SYM} \cup \text{NUM} \cup \text{BOOL} \cup \text{LIST}(\text{SEXP}_O) \\ \text{SEXP}_{FG} &= \text{SYM} \cup \text{NUM} \cup \text{BOOL} \cup \{ '() \} \cup \\ &\quad \{ (\text{cons } v_1 v_2) \mid v_1 \in \text{SEXP}_{FG} \wedge v_2 \in \text{SEXP}_{FG} \} \end{aligned}$$

## 2.6 LANGUAGE II: LOCAL VARIABLES AND *let*

While a lot can be done with just formal parameters, big functions need local variables. In a procedural language like C or Impcore, local variables can be introduced without being initialized, and they are often assigned to more than once. But in a functional language like Scheme, local variables are *always* initialized when introduced, and afterward, they are rarely assigned to again.

In Scheme, local variables are introduced by *let binding*. A let binding is often hand-written as “**let**  $x = e'$  **in**  $e$ ,” which means “evaluate  $e'$ , let  $x$  stand for the resulting value, and evaluate  $e$ .” In Scheme, the same binding is written  $(\text{let } ([x e']) e)$ . In general, Scheme's *let* form binds a *collection* of values:

$$(\text{let } ([x_1 e_1] \cdots [x_n e_n]) e).$$

This *let* expression is evaluated as follows: First evaluate the *right-hand sides*  $e_1$  through  $e_n$ ; call the results  $v_1, \dots, v_n$ . Next, *extend* the local environment so that  $x_1$  stands for  $v_1$ , and so on. Finally, in the extended environment, evaluate the *body*  $e$ ; the value of the body becomes the value of the entire *let* expression.

The *let* form helps us avoid repeating computation, and it helps make code readable. To enhance its readability, I write its bindings in square brackets. (In  $\mu$ Scheme, in full Scheme, and in all the bridge languages, square brackets mean the same as round brackets (“parentheses”). I typically use round brackets to wrap expressions, definitions, and lists of formal parameters; I use square brackets to wrap other kinds of syntax, like binding pairs or local-variable declarations.)

A let form can be thought of as *naming the result of a computation*. As an example, let's compute the roots of the quadratic equation:  $ax^2 + bx + c = 0$ . From the quadratic formula, the roots are  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ . Although this formula is not much use without real numbers, it can still be implemented in  $\mu$ Scheme, and it can compute roots of such equations as  $x^2 + 3x - 70 = 0$ . Decent candidates to be named with let are the values of subformulas  $-b$ ,  $\sqrt{b^2 - 4ac}$ , and  $2a$ :

**120a.** *(transcript 97a)*  $\equiv$  △ 112b 120b ▷  
*(definition of sqrt S310c)*  
 $\rightarrow (\text{define roots} (\text{a b c})$   
 $\quad (\text{let} ([\text{minus-b}] (\text{negated b}))$   
 $\quad [\text{discriminant} (\text{sqrt} (- (* \text{b} \text{b}) (* 4 (* \text{a} \text{c}))))])$   
 $\quad [\text{two-a}] (* 2 \text{a}))]$   
 $\quad (\text{list2} (/ (+ \text{minus-b} \text{discriminant}) \text{two-a})$   
 $\quad (/ (- \text{minus-b} \text{discriminant}) \text{two-a})))$   
 $\rightarrow (\text{roots} 1 3 -70)$   
 $(7 -10)$

In a let expression, all of the right-hand sides are evaluated before any of the  $x_i$ 's are bound. It is often more useful to evaluate and bind one expression at a time, in sequence, so that right-hand side  $e_i$  can refer to the values named  $x_1, \dots, x_{i-1}$ . Sequential binding is implemented by the let\* form. This form has the same structure as let, but after evaluating the first right-hand side  $e_1$ , it extends the environment by binding the result to  $x_1$ . Then it evaluates  $e_2$  in the extended environment, binds the result to  $x_2$ , and so on.

The difference between let and let\* can be illustrated by a contrived example:

**120b.** *(transcript 97a)*  $\equiv$  △ 120a 120c ▷  
 $\rightarrow (\text{val x 'global-x})$   
 $\rightarrow (\text{val y 'global-y})$   
 $\rightarrow (\text{let}$   
 $\quad ([\text{x 'local-x}]$   
 $\quad [\text{y x}])$   
 $\quad (\text{list2 x y}))$   
 $(\text{local-x global-x})$

In this example, because the right-hand sides in a let are evaluated in the original environment, the x in [y x] refers to the *global* definition of x. Using let\*, the same structure works differently:

**120c.** *(transcript 97a)*  $\equiv$  △ 120b 121a ▷  
 $\rightarrow (\text{val x 'global-x})$   
 $\rightarrow (\text{val y 'global-y})$   
 $\rightarrow (\text{let*}$   
 $\quad ([\text{x 'local-x}]$   
 $\quad [\text{y x}])$   
 $\quad (\text{list2 x y}))$   
 $(\text{local-x local-x})$

In this example, because the right-hand sides in a let\* are evaluated and bound in sequence, the x in [y x] refers to the *local* definition of x.

Any let\* expression can be simulated with a nested sequence of let expressions, but let\* is more readable and more convenient. As evidence of let\*'s utility, I show a *level-order traversal* (also called breadth-first traversal) of a binary tree. This traversal visits every node on one level before visiting any node on the next level. In effect, it visits nodes in order of distance from the root. For example, level-order traversal of the tree on page 111 visits the nodes in the order (A B E C D F I G H).

Level-order traversal uses an auxiliary data structure: a queue of nodes not yet visited. Traversal starts with a queue containing only the root, and it continues until the queue is empty. When the queue is not empty, the traversal visits the node at the

front of the queue, then enqueues the node's children at the end. The implementation needs queue operations `emptyqueue`, `front`, `without-front`, and `enqueue`:

```
121a. (transcript 97a) +≡
-> (val emptyqueue '())
-> (define front          (q) (car q))
-> (define without-front (q) (cdr q))
-> (define enqueue (t q)
  (if (null? q)
    (list t)
    (cons (car q) (enqueue t (cdr q)))))
-> (define empty? (q) (null? q))
```

△120c 121b ▷

§2.6

Language II: Local variables and let

121

This implementation of queues is woefully inefficient—it has the same cost as `append`—but it's simple. I encourage you to do better (Exercise 13) and also to describe the queue's behavior using algebraic laws (Exercise 14).

The traversal is performed recursively by auxiliary function `level-order-of-q`, which receives an initial queue that contains only the tree to be traversed. It uses `let*` to bind the first element to the name `hd`, which is then used both to make a new queue `newq` and to make the result in the body.

```
121b. (transcript 97a) +≡
-> (define level-order-of-q (queue)
  (if (empty? queue)
    '()
    (let* ([hd   (front      queue)]
          [t1   (without-front queue)])
      [newq (if (empty-tree? hd)
                t1
                (enqueue (node-right hd)
                         (enqueue (node-left hd) t1)))]
      (if (node? hd)
        (cons (node-tag hd) (level-order-of-q newq))
        (level-order-of-q newq))))
  -> (define level-order (t)
    (level-order-of-q (enqueue t emptyqueue)))
  -> (level-order example-sym-tree)
  (A B E C D F I G H))
```

△121a 121c ▷

The example code names three queues: `queue`, `t1`, and `newq`. Picking the wrong one can cause an error; for example, if `queue` is used place of `t1`, the function will loop. In an idiomatic imperative function, such an error wouldn't occur; the function would use just one variable, `queue`, whose value would change over time. In an idiomatic applicative function, the same error can *also* be avoided; by clever use of `let*`, the name `queue` can be repeatedly *rebound* so that it always refers to the queue of interest:

```
121c. (transcript 97a) +≡
-> (define level-order-of-q (queue)
  (if (empty? queue)
    '()
    (let* ([hd   (front      queue)]
          [queue (without-front queue)])
      [queue (if (empty-tree? hd)
                  queue
                  (enqueue (node-right hd)
                           (enqueue (node-left hd) queue)))]
      (if (node? hd)
        (cons (node-tag hd) (level-order-of-q queue))
        (level-order-of-q queue))))
```

car	§P 164a
cdr	§P 164a
cons	§P S307e
empty-tree?	112a
example-sym-tree	111b
node-left	111a
node-right	111a
node-tag	111a
node?	111a
null?	§P 164a
sqrt	S310c

122a. *(transcript 97a)* +≡ ▷ 121c 122b ▷

```
-> (define level-order (t)
  (level-order-of-q (enqueue t emptyqueue)))
-> (level-order example-sym-tree)
(A B E C D F I G H)
```

Both the imperative and the applicative idioms accomplish the same goal: they use just one name, queue, and at each point in the program, it means the right thing.

The let and let\* forms have a recursive sibling, letrec, which has yet a third set of rules for the visibility of the bound names  $x_i$ . In a let expression, none of the  $x_i$ 's can be used in any of the  $e_i$ 's. In a let\* expression, each  $x_i$  can be used in any  $e_j$  with  $j > i$ , that is, an  $x$  can be used in all the  $e$ 's that follow it. In a letrec expression, all of the  $x_i$ 's can be used in all of the  $e_i$ 's, regardless of order. The letrec form is used to define recursive functions. A simple example appears on page 137, but a detailed explanation is best deferred to the formal treatment of lambda and closures (Section 2.11.2).

2  
122

Scheme,  
S-expressions, and  
first-class functions

## 2.7 LANGUAGE III: FIRST-CLASS FUNCTIONS, lambda, AND LOCATIONS

In Impcore, as in C, functions can be defined only at top level. And in Impcore, functions are not values. In C, functions are not quite values either, but a *pointer* to a function is an ordinary value, and it can be used like other values. In Scheme, functions are values, and a function can be defined anywhere—even inside another function. A function is defined by evaluating a new form of expression: the *lambda expression*.

The expression

```
(lambda (x1 x2 ... xn) e)
```

denotes the function that takes values  $v_1, v_2, \dots, v_n$  and returns the result of evaluating  $e$  in an environment where  $x_1$  is bound to  $v_1$ ,  $x_2$  is bound to  $v_2$ , and so on. Together with the LET expressions, the lambda expression is what distinguishes μScheme syntax from Impcore syntax; roughly speaking, μScheme is Impcore plus LET expressions, LAMBDA expressions, and S-expression data. The lambda expression is the key to many useful programming techniques, the most important and widely used of which are presented in this chapter.

By itself, lambda is not obviously powerful; one lambda is simple and innocuous. For example,  $(\lambda(x\ y)\ (+\ (*\ x\ x)\ (*\ y\ y)))$  denotes the function that, given values  $v$  and  $w$ , returns  $v^2 + w^2$ . Its only novelty is that it is *anonymous*; unlike Impcore functions and C functions, Scheme functions need not be named.

122b. *(transcript 97a)* +≡

▷ 122a 123a ▷

```
-> ((lambda (x y) (+ (* x x) (* y y))) 3 4)
25
-> ((lambda (x y) (+ (* x x) (* y y))) 707 707)
999698
-> ((lambda (x y z) (+ x (+ y z))) 1 2 3)
6
-> ((lambda (y) (* y y)) 7)
49
```

The lambda looks like a new way of defining functions, so you might think that μScheme has two kinds of functions, one created with lambda and one created with define. But define is an abbreviation—another form of syntactic sugar. A define form can be *desugared* into a combination of val and lambda:

$$(\text{define } f (x_1 \dots x_n) e) \stackrel{\Delta}{=} (\text{val } f (\lambda (x_1 \dots x_n) e)).$$

As described in the sidebar on page 68, this kind of syntactic sugar is popular; it gives programmers a nice big language to use, while keeping the core language and its operational semantics (and the proofs!) small.

The `define` form can be desugared into a `val` only because in  $\mu$ Scheme, a function is just another kind of value—and all values, including functions, are bound in the same environment (Section 2.11). In Impcore, where functions and values are bound in different environments, `define` could not be desugared into a `val`.

Desugaring `define` into `val` is one way to exploit the idea that a function defined with `lambda` is just another value. Besides putting it on the right-hand side of a `val`, what else can you do with such a function? What you can do with any other value:

- Pass it as an argument to a function
- Return it from a function
- Store it in a global variable, using `set!`
- Save it in a data structure, using `cons` or a record constructor

These capabilities can help you assess the role of any species of value in any programming language; if you can do all these things with a value, that value is sometimes said to be *first class*. “First-class functions” is a phrase sometimes used to characterize Scheme and other functional languages, but it’s not quite right: what’s essential about Scheme is that it provides first-class, *nested* functions.

To see what difference nesting makes, let’s start with a non-nested example. Because functions are first class, a  $\mu$ Scheme function (defined with `define` or with `lambda`) can be passed as an argument to another function. What can the function that receives the argument do? All the standard things listed above, which it can do with any first-class value. And one more—the only *interesting* thing to do with a function: *apply* it. As a simple example, function `apply-n-times` receives a function `f`, an integer `n`, and an argument `x`, and it returns the result of applying `f` to `x` `n` times:

**123a.** *(transcript 97a)* +≡

```
-> (define apply-n-times (n f x)
  (if (= 0 n)
      x
      (apply-n-times (- n 1) f (f x))))
-> (apply-n-times 77 not #t)
#f
-> (apply-n-times 78 not #t)
#t
```

◁ 122b 123b ▷

Applying `n` times is more interesting with numbers:

**123b.** *(transcript 97a)* +≡

```
-> (define twice (n) (* 2 n))
-> (define square (n) (* n n))
-> (apply-n-times 2 twice 10)
40
-> (apply-n-times 2 square 10)
10000
-> (apply-n-times 10 twice 1)
1024
-> (apply-n-times 10 square 1)
1
```

◁ 123a 124 ▷

emptyqueue	121a
enqueue	121a
example-sym-tree	111b
level-order-of-q	121c

A function like `apply-n-times` is described by a handy piece of jargon: a function that takes another function as an argument is called a *higher-order function*. The term “higher-order function” is inclusive; it also describes a function that re-

*turns another function. Functions like `twice` and `square`, which neither take functions as arguments nor return functions as results, are called *first-order functions*.*

Not all higher-order functions are created equal. Some, like `apply-n-times`, could be implemented in C just as easily as in Scheme. What *can't* be implemented in C is a function that evaluates a *nested lambda* expression inside its own body, then returns the result of that evaluation, which is called a *closure*. Such a function creates a new function anonymously, “on the fly”; the closure is the *representation* of the new function. As a toy example, function `add` returns a closure:

2 *Scheme, S-expressions, and first-class functions* 124

124. *(transcript 97a)* +≡

```
-> (val add (lambda (x) (lambda (y) (+ x y))))
-> (val add1 (add 1))
-> (add1 4)
5
```

◀123b 125a▶

Function `add` is defined by the outer `lambda` expression, which takes `x` as an argument. It's not so interesting. What's interesting is inner `lambda`, which takes `y` as an argument: every time it is evaluated, it creates a new function. The two `lambda`s work together: when `add` receives argument `m`, it returns a function which, when it receives argument `n`, returns `m + n`. (Therefore, `add1` is a one-argument function which adds 1 to its argument.) Applying `add` to different integers can create *arbitrarily many* functions. Not only `(add 1)` but `(add 2)` and `(add 100)` can be functions. Even an expression like `(add (length xs))` creates a function.

New functions can be created because `(lambda (y) (+ x y))` is nested inside the outer `lambda` that defines `add`. Every time `add` is called, the inner `lambda` is evaluated, and in Scheme, every time a `lambda` expression is evaluated, a new function is created. In languages like Impcore and C, which don't have `lambda`, a new function can be created only by writing it explicitly in the source code.

### 2.7.1 How `lambda` works: Closures with mutable locations

The expression `(lambda (y) (+ x y))` can be evaluated to produce more than one function. Such a function is represented by pairing the `lambda` expression with an environment that says what `x` stands for. This pair forms the closure, which we write in “banana brackets,” as in `((lambda (y) (+ x y)), {x ↦ 1})`. The banana brackets emphasize that a closure cannot be written directly using Scheme syntax; the closure is a value that Scheme builds from a `lambda` expression. A closure is the simplest way to implement first-class, nested functions.

A closure is defined as a pair: code and an environment. In  $\mu$ Scheme, the code is a `lambda` expression, but what is the environment? In Impcore, an environment maps a name to a value (or a function). But in Scheme, an environment maps a name to a *mutable location*. You can think of a mutable location as a box containing a value, which can change, or you can wear your C programmer's hat and think of it as a location in memory, which the environment can point to.

A Scheme environment maps each name to a mutable location because in a world where there are closures, this is the easiest way to express the way assignment (`set!`) works. In Impcore, `set!` simply replaces an old environment with a new one (and can even update the environment in place). But `set!` can be implemented in this way only because no environment is ever copied. In Scheme, by contrast, an environment can be copied into any number of closures, and there is no easy way for `set!` to update them all. Instead, when a name is assigned to, `set!` updates the mutable location associated with that name.

In the following example, `set!` updates a mutable location that is accessible only from within a `lambda`. The inner `lambda` evaluates to a closure in which `x` points to

a mutable location. That closure is stored in global variable ten, and every time ten is called, it updates x and returns the new value:

125a. *(transcript 97a)* +≡  
-> (val counter-from  
<function>  
> (lambda (x)  
   (lambda () (set x (+ x 1)))))  
> (val ten (counter-from 10))  
<function>  
> (ten)  
11  
> (ten)  
12  
> (ten)  
13

△124 125b ▷

§2.7  
*Language III:*  
*First-class  
functions*

125

When ten is defined, counter-from is applied, and that application allocates a location  $\ell$  to hold the value of x, which is 10. The environment in which the inner lambda is evaluated therefore binds x to  $\ell$ , and the inner lambda evaluates to the closure  $((\lambda() (\text{set } x (+ x 1))), \{x \mapsto \ell, \dots\})$ , which is the value of ten.<sup>7</sup>

A mutable location can be shared among multiple closures, which can communicate with each other by mutating it. For example, a mutable number n can be shared by counter and reset functions, which are stored in a counter record:

125b. *(transcript 97a)* +≡  
-> (record counter [step reset])  
> (val resettable-counter-from  
<function>  
> (lambda (n) ; create a counter  
   (make-counter (lambda () (set n (+ n 1)))  
   (lambda () (set n 0)))))

△125a 125c ▷

The two innermost lambda expressions refer to the *same* n, so that when a counter's reset function is called, its step function starts over at 0. And two *different* applications of resettable-counter-from refer to *different* n's, so that two independent counters never interfere.

A counter is stepped or reset in two steps: extract a function from the counter record, then apply it.

125c. *(transcript 97a)* +≡  
-> (val step (lambda (counter) ((counter-step counter))))  
> (val reset (lambda (counter) ((counter-reset counter))))

△125b 125d ▷

The double applications are easy to overlook. A function is taken from the counter record in an inner application, such as (counter-step counter). That function, which takes no parameters, is called in an outer application, such as ((counter-step counter)).

In the following transcript, two counters, hundred and twenty, count independently and are reset independently.

125d. *(transcript 97a)* +≡  
-> (val hundred (resettable-counter-from 100))  
> (val twenty (resettable-counter-from 20))  
> (step hundred)  
101  
> (step hundred)  
102  
> (step twenty)  
21

△125c 126a ▷

<sup>7</sup>As suggested by the ellipsis, the environment contains other bindings, but the only bindings that affect computation are the binding of x to  $\ell$  and of + to a location containing the primitive addition function.

Resetting hundred doesn't affect twenty.

**126a.** *(transcript 97a)* +≡  
-> (reset hundred)  
0  
-> (step hundred)  
1  
-> (step twenty)  
22

△125d 126b▷

The mutable `n` that the `step` and `reset` function share is an example of *shared mutable state*. Shared mutable state isn't new; it's available in any language that has mutable global variables. But the shared mutable state provided by `lambda` is more flexible and better controlled than a global variable.

- It's more flexible because the amount of shared mutable state isn't limited by the number of global variables.
- It's better controlled because not every piece of code has access to it—the only code that has access is code that has the relevant variables in a closure.

The mutable state that is stored in a closure is *private* to a function but also *persistent* across calls to that function. Private, persistent mutable state is sometimes provided as a special language feature, usually called "own variables." An own variable is local to a function, but its value is preserved across calls of the function. The name comes from Algol-60, but own variables are also found in C, where they are defined using the keyword `static`.

As another example of private, persistent mutable state, I define a simple random-number generator.<sup>8</sup> It's a higher-order function that takes a `next` function as parameter, and it keeps a private, mutable variable `seed`, which is initially 1:

**126b.** *(transcript 97a)* +≡  
-> (define mk-rand (next)  
 (let ([seed 1])  
 (lambda () (set seed (next seed)))))

△126a 126c▷

Parameter `next` can be any function that takes a number and returns another number. To make a *good* random-number generator requires a function that satisfies some sophisticated statistical properties. A simple approximation uses the *linear congruential* method (Knuth 1981, pp. 9–25) on numbers in the range 0 to 1023:

**126c.** *(transcript 97a)* +≡  
-> (define simple-next (seed) (mod (+ (\* seed 9) 5) 1024))

△126b 126d▷

This generator is not, in any statistical sense, good. For one thing, after generating only 1024 different numbers, it starts repeating. But it's usable:

**126d.** *(transcript 97a)* +≡  
-> (val irand (mk-rand simple-next))  
-> (irand)  
14  
-> (irand)  
131  
-> (irand)  
160  
-> (val repeatable-irand (mk-rand simple-next))  
-> (repeatable-irand)  
14  
-> (irand)  
421

△126c 127b▷

<sup>8</sup>Technically a generator of *pseudorandom* numbers.

Function `irand` has its own private copy of `seed`, which only it can access, and which it updates at each call. And function `repeatable-irand`, which might be used to replay an execution for debugging, has *its* own private seed. So it repeats the *same* sequence [1, 14, 131, 160, 421, ...] no matter what happens with `irand`.

### 2.7.2 Simple, useful higher-order functions

The `lambda` expression does more than just encapsulate mutable state; `lambda` helps express and support not just algorithms but also *patterns of computation*. What a “pattern of computation” might be is best shown by example.

One minor example is the function `mk-rand`: it can be viewed as a pattern that says “if you tell me how to get from one number to the next, I can deliver an entire sequence of numbers starting with 1.” This pattern of computation, while handy, is not used often. More useful patterns can make new functions from old functions or can express common ways of programming with lists, like “do something with every element.” Such patterns are presented in the next few sections.

#### Composition

One of the simplest ways to make a new function is by composing two old ones. Function `o` (pronounced “circle” or “compose”) returns the composition of two one-argument functions, often written  $f \circ g$ . Composition is described by the algebraic law  $(f \circ g)(x) = f(g(x))$ , and like any function that makes new functions, it returns a `lambda`:

**127a.** *(predefined μScheme functions 98a)* +≡ ▷108c 128c▷  
    `(define o (f g) (lambda (x) (f (g x))))` ;  $((o\ f\ g)\ x) = (f\ (g\ x))$

Function composition can negate a predicate by composing `not` with it:

**127b.** *(transcript 97a)* +≡ ▷126d 127c▷  
    `-> (define even? (n) (= 0 (mod n 2)))`  
    `-> (val odd? (o not even?))`  
    `-> (odd? 3)`  
    `#t`  
    `-> (odd? 4)`  
    `#f`

In large programs, function composition can be used to improve modularity: an algorithm can be broken down into functions that are connected using composition.

#### Currying and partial application

A `lambda` can be used to change the interface to a function, as in the example of `add` (page 124):

**127c.** *(transcript 97a)* +≡ ▷127b 128a▷  
    `-> (val add (lambda (x) (lambda (y) (+ x y))))`

Function `add` does what `+` does, but `add` takes one argument at a time, whereas `+` takes both its arguments at once. For example, `(add 1)` is a function that adds 1 to its argument, while `(+ 1)` is an error. Similarly, `(+ 1 2)` is 3, while `(add 1 2)` is an error. But `add` and `+` are really two forms of the same function; they differ only in the way they take their arguments. The forms are named: `add` is the *curried* form, and `+` is the *uncurried* form. The names honor the logician Haskell B. Curry.

Any function can be put into curried form; the curried form simply takes its arguments one at a time. If it needs more than one argument, it takes the first argument, then returns a `lambda` that expects the remaining arguments, also one

hundred	125d
reset	125c
step	125c
twenty	125d

at a time. As another example, the curried form of the `list3` function uses three lambdas:

**128a.** *(transcript 97a)* +≡ ◁127c 128b ▷  
 -> (val curried-list3 (lambda (a) (lambda (b) (lambda (c) (list3 a b c)))))  
 -> (curried-list3 'x)  
 <function>  
 -> ((curried-list3 'x) 'y)  
 <function>  
 -> (((curried-list3 'x) 'y) 'z)  
 (x y z)

Curried functions don't mesh well with Scheme's concrete syntax. Defining one requires lots of lambdas, and applying one requires lots of parentheses.<sup>9</sup> If currying is so awkward, why bother with it? To get *partial applications*. A curried function is *partially applied* when it is applied to only *some* of its arguments, and the resulting function is saved to be applied later. If the function's arguments are expected in the right order, partial applications can be quite useful. For example, the curried form of `<` can be partially applied to 0, and the resulting partial application takes any  $m$  and says whether  $0 < m$ :

**128b.** *(transcript 97a)* +≡ ◁128a 128d ▷  
 -> (val <-curried (lambda (n) (lambda (m) (< n m))))  
 -> (val positive? (<-curried 0))  
 -> (positive? 0)  
 #f  
 -> (positive? 8)  
 #t  
 -> (positive? -3)  
 #f

Functions needn't always be curried by hand. Any binary function can be converted between its uncurried and curried forms using the predefined functions `curry` and `uncurry`:

**128c.** *(predefined μScheme functions 98a)* +≡ ◁127a 131 ▷  
 (define curry (f) (lambda (x) (lambda (y) (f x y))))  
 (define uncurry (f) (lambda (x y) ((f x) y)))

**128d.** *(transcript 97a)* +≡ ◁128b 128e ▷  
 -> (val zero? ((curry =) 0))  
 -> (zero? 0)  
 #t  
 -> (val add1 ((curry +) 1))  
 -> (add1 4)  
 5

Functions `curry` and `uncurry` obey these algebraic laws:

$$\begin{aligned} ((\text{curry } f) x) y &= (f x y) \\ (\text{uncurry } f) x y &= ((f x) y) \end{aligned}$$

As can be proved using the laws, the two functions are inverses; for example, if I `curry` `+`, then `uncurry` the result, I get `+` back again:

**128e.** *(transcript 97a)* +≡ ◁128d 129a ▷  
 -> (val also+ (uncurry (curry +)))  
 -> (also+ 1 4)  
 5

---

<sup>9</sup>More recently developed functional languages, like Standard ML, OCaml, and Haskell, use notations and implementations that encourage currying and partial application, so much so that in OCaml and Haskell, function definitions produce curried forms by default. For partial application in Scheme, see the proposal by Eigner (2002).

If currying makes your head hurt, don't panic—I expect it. Both currying and composition are easier to understand when we they are used to make functions that operate on values in lists.

## 2.8 PRACTICE III: HIGHER-ORDER FUNCTIONS ON LISTS

If higher-order functions embody “patterns of computation,” what are those patterns? The patterns you’re best equipped to appreciate are patterns of computation on lists. Some of the most popular and widely reused patterns can be described informally as follows:

- Pick just some elements from a list.
- Make a new list from an existing list by transforming each element.
- Search a list for an element.
- Check every element of a list to make sure it is OK.
- Visit every list element, perform a computation there, and accumulate a result. Or more loosely, “do something” to every element of a list.

These patterns, and more besides, are embodied in the higher-order functions `filter`, `map`, `exists?`, `all?`, and `foldr`, all of which are in  $\mu$ Scheme’s initial basis. (If you’ve heard of Google MapReduce, the Map is the same, and Reduce is a parallel variant of `foldr`.)

### 2.8.1 Standard higher-order functions on lists

Function `filter` takes a predicate `p?` and a list `xs`, and it returns a new list consisting of only those elements of `xs` that satisfy `p?`:

**129a.** *(transcript 97a)* +≡ ▷ 128e 129b ▷  
→ (define even? (x) (= (mod x 2) 0))  
→ (filter even? '(1 2 3 4 5 6 7 8 9 10))  
(2 4 6 8 10)

As requested in Exercise 27, function `filter` can be used to define a concise version of the `remove-multiples` function from Section 2.3.4.

Function `filter` must receive a predicate as its first argument, but `map` works with any unary function; (`map f xs`) returns the list of results formed by applying function `f` to every element of list `xs`.

**129b.** *(transcript 97a)* +≡ ▷ 129a 130a ▷  
→ (map add1 '(3 4 5))  
(4 5 6)  
→ (map ((curry +) 5) '(3 4 5))  
(8 9 10)  
→ (map (lambda (x) (\* x x)) '(1 2 3 4 5 6 7 8 9 10))  
(1 4 9 16 25 36 49 64 81 100)  
→ (primes≤= 20)  
(2 3 5 7 11 13 17 19)  
→ (map ((curry <) 10) (primes≤= 20))  
(# #f #f #f #t #t #t #t)

add1      124  
primes≤=    104d

Functions `filter` and `map` build new lists from the elements of old ones—a common pattern of computation. Another common pattern is linear search. In  $\mu$ Scheme, linear search is implemented by two functions, `exists?` and `all?`.

Each takes a predicate, and as you might expect, `exists?` tells whether there is an element of the list satisfying the predicate; `all?` tells whether they all do.

**130a.** *(transcript 97a)*  $\equiv$   $\triangleleft$  129b 130b  $\triangleright$

```

-> (exists? even? '(1 2 3 4 5 6 7 8 9 10))
#t
-> (all? even? '(1 2 3 4 5 6 7 8 9 10))
#f
-> (all? even? (filter even? '(1 2 3 4 5 6 7 8 9 10)))
#t
-> (exists? even? (filter (o not even?) '(1 2 3 4 5 6 7 8 9 10)))
#f

```

When called on the empty list, an important “corner case,” `exists?` and `all?` act like the mathematical  $\exists$  and  $\forall$ .

**130b.** *(transcript 97a)*  $\equiv$   $\triangleleft$  130a 130c  $\triangleright$

```

-> (exists? even? '())
#f
-> (all? even? '())
#t

```

Higher-order functions on lists can make programs wonderfully concise, because they relieve you from writing the same patterns of recursion over and over again. Writing common recursions in a concise, standard way helps not just the person who writes the code but also the people who read it. A reader who sees a recursive function has to figure out what is called recursively and how it works. But one who sees `map`, `filter`, or `exists?` already knows how the code works.

A very general pattern of recursion is to combine the `car` of a list with the results of a recursive call. This pattern is embodied in the higher-order function `foldr`. Function `foldr` expects a combining operator (a two-argument function), which I'll call  $\oplus$ ; a starting value, which I'll call  $N$ ; and a list of values, which I'll call  $vs$ . Using  $\oplus$ , `foldr` combines (“folds”) all the values from  $vs$  into  $N$ . For example, if  $\oplus$  is  $+$ ,  $N$  is 0, and  $vs$  is a list of numbers, then  $(\text{foldr } \oplus N \text{ } vs)$  is the sum of the list of numbers.

The general case of `foldr` is most easily understood when written using infix notation. I'll write  $(\oplus x y)$  as  $x \oplus y$  and  $(\text{cons } v vs)$  as  $v :: vs$ . In this notation, if  $vs$  has the form  $v_1 :: v_2 :: \dots :: v_n :: '()$ , then  $(\text{foldr } \oplus N \text{ } vs)$  is  $v_1 \oplus v_2 \oplus \dots \oplus v_n \oplus N$ . In effect, every `cons` is replaced with  $\oplus$ , and  $'()$  is replaced with  $N$ .

What if  $\oplus$  is not associative? The `r` in `foldr` means that  $\oplus$  associates to the right. The right operand of  $\oplus$  is always either  $N$  or the result of applying a previous  $\oplus$ :

$$(\text{foldr } \oplus N \text{ } '(v_1 v_2 \dots v_n)) = v_1 \oplus (v_2 \oplus (\dots \oplus (v_n \oplus N))).$$

Function `foldr` has a companion function, `foldl`, in which  $\oplus$  associates to the left, and  $N$  is combined with  $v_1$ , not with  $v_n$ . Because each result becomes a right operand of  $\oplus$ , `foldl` is effectively `foldr` operating on a reversed list:

$$(\text{foldl } \oplus N \text{ } '(v_1 v_2 \dots v_n)) = v_n \oplus (v_{n-1} \oplus (\dots \oplus (v_1 \oplus N))).$$

For example,  $(\text{foldl } - 0 \text{ } '(1 2 3 4))$  is  $(4 - (3 - (2 - (1 - 0)))) = 2$ . On the same list,  $(\text{foldr } - 0 \text{ } '(1 2 3 4))$  is  $(1 - (2 - (3 - (4 - 0)))) = -2$ .

**130c.** *(transcript 97a)*  $\equiv$   $\triangleleft$  130b 133a  $\triangleright$

```

-> (foldl - 0 '(1 2 3 4))
2
-> (foldr - 0 '(1 2 3 4))
-2

```

More applications of `foldr` and `foldl` are suggested in Exercises 8, 29, and 30.

## 2.8.2 Visualizing the standard list functions

Which list functions should be used when? Functions `exists?` and `all?` are not hard to figure out, but `map`, `filter`, and `foldr` can be more mysterious. They can be demystified a bit using pictures, as inspired by [Harvey and Wright \(1994\)](#).

A generic list `xs` can be depicted as a list of circles:

$$xs = \circ \circ \circ \circ \circ \cdots \circ$$

If `f` is a function that turns one circle into one triangle, as in  $(f \circ) = \triangle$ , then  $(\text{map } f \text{ xs})$  turns a list of circles into a list of triangles.

$$\begin{array}{ll} xs & = \circ \circ \circ \circ \circ \cdots \circ \\ & \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ (\text{map } f \text{ xs}) & = \triangle \triangle \triangle \triangle \triangle \cdots \triangle \end{array}$$

If `p?` is a function that takes a circle and returns a Boolean, as in  $(p? \circ) = b$ , then  $(\text{filter } p? \text{ xs})$  selects just some of the circles:

$$\begin{array}{ll} xs & = \circ \circ \circ \circ \circ \cdots \circ \\ & \downarrow \quad \times \quad \downarrow \quad \times \quad \downarrow \\ (\text{filter } p? \text{ xs}) & = \circ \quad \circ \quad \circ \quad \cdots \quad \circ \end{array}$$

Finally, if `f` is a function that takes a circle and a box and produces another box, as in  $(f \circ \square) = \square$ , then  $(\text{foldr } f \square \text{ xs})$  folds all of the circles into a single box:

$$\begin{array}{ll} xs & = \circ \circ \circ \circ \circ \cdots \circ \\ & \overbrace{\hspace{10em}}^{} \\ & \downarrow \\ (\text{foldr } f \square \text{ xs}) & = \square \end{array}$$

The single box can contain any species of value—even another sequence of circles.

## 2.8.3 Implementing the standard list functions

Most of the higher-order list functions are easy to implement and easy to understand. Each is a recursive function with one base case (which consumes the empty list) and one induction step (which consumes a cons cell). All are part of the initial basis of  $\mu$ Scheme. Except for `app`, they are described by the algebraic laws in Figure 2.4 on the following page.

Function `filter` is structured in the same way as function `remove-multiples` from chunk 104b; the only difference is in the test. Filtering the empty list produces the empty list. In the induction step, depending on whether the car satisfies `p?`, `filter` may or may not cons.

**131.** *(predefined  $\mu$ Scheme functions 98a) +≡*

◀ 128c 132a ▷

even? 129a

```
(define filter (p? xs)
  (if (null? xs)
      '()
      (if (p? (car xs))
          (cons (car xs) (filter p? (cdr xs)))
          (filter p? (cdr xs)))))
```

(filter p? '())	= '()
(filter p? (cons y ys))	= (cons y (filter p? ys)), when (p? y)
(filter p? (cons y ys))	= (filter p? ys), when (not (p? y))
(map f '())	= '()
(map f (cons y ys))	= (cons (f y) (map f ys))
(exists? p? '())	= #f
(exists? p? (cons y ys))	= #t, when (p? y)
(exists? p? (cons y ys))	= (exists? p? ys), when (not (p? y))
(all? p? '())	= #t
(all? p? (cons y ys))	= (all? p? ys), when (p? y)
(all? p? (cons y ys))	= #f, when (not (p? y))
(foldr combine zero '())	= zero
(foldr combine zero (cons y ys))	= (combine y (foldr combine zero ys))
(foldl combine zero '())	= zero
(foldl combine zero (cons y ys))	= (foldl combine (combine y zero) ys)

Figure 2.4: Algebraic laws of pure higher-order functions on lists

Function `map` is even simpler. There is no conditional test; the induction step just applies `f` to the car, then conses.

**132a.** *(predefined μScheme functions 98a)* +≡  
`(define map (f xs)  
 (if (null? xs)  
 '()  
 (cons (f (car xs)) (map f (cdr xs)))))`

△131 132b▷

Function `app` is like `map`, except its argument is applied only for side effect. Function `app` is typically used with `printu`. Because `app` is executed for side effects, its behavior cannot be expressed using simple algebraic laws.

**132b.** *(predefined μScheme functions 98a)* +≡  
`(define app (f xs)  
 (if (null? xs)  
 #f  
 (begin (f (car xs)) (app f (cdr xs)))))`

△132a 132c▷

Each of the preceding functions processes every element of its list argument. Functions `exists?` and `all?` don't necessarily do so. Function `exists?` stops the moment it finds a satisfying element; `all?` stops the moment it finds a *non*-satisfying element.

**132c.** *(predefined μScheme functions 98a)* +≡  
`(define exists? (p? xs)  
 (if (null? xs)  
 #f  
 (if (p? (car xs))  
 #t  
 (exists? p? (cdr xs)))))  
  
(define all? (p? xs)  
 (if (null? xs)  
 #t  
 (if (p? (car xs))  
 (all? p? (cdr xs))  
 #f)))`

△132b 133b▷

Function `all?` could also be defined using De Morgan's law, which says that  $\neg\forall x.P(x) = \exists x.\neg P(x)$ . Negating both sides gives this definition:

**133a.** *(transcript 97a)* +≡

```
> (define alt-all? (p? xs) (not (exists? (o not p?) xs)))
-> (alt-all? even? '(1 2 3 4 5 6 7 8 9 10))
#f
-> (alt-all? even? '())
#t
-> (alt-all? even? (filter even? '(1 2 3 4 5 6 7 8 9 10)))
#t
```

△130c 133c▷

§2.9

Practice IV:  
Higher-order  
functions for  
polymorphism

133

Finally, `foldr` and `foldl`, although simple, are not necessarily easy to understand. Study their algebraic laws, and remember that `(car xs)` is always a first argument to `combine`, and `zero` is always a second argument.

**133b.** *(predefined μScheme functions 98a)* +≡

```
(define foldr (combine zero xs)
  (if (null? xs)
    zero
    (combine (car xs) (foldr combine zero (cdr xs)))))
(define foldl (combine zero xs)
  (if (null? xs)
    zero
    (foldl combine (combine (car xs) zero) (cdr xs))))
```

△132c

## 2.9 PRACTICE IV: HIGHER-ORDER FUNCTIONS FOR POLYMORPHISM

A function like `filter` doesn't need to know what sort of value predicate `p?` is expecting. For example, `filter` can be used with function `even?` to select elements of a list of numbers; or it can be used with function `(o even? alist-pair-attribute)` to select elements of an association list that contains symbol-number pairs; or it can be used with infinitely many other combinations of predicates and lists. The ability to be used with arguments of many different types makes `filter` *polymorphic* (see sidebar on the next page). Functions `exists?`, `all?`, `map`, `foldl`, and `foldr` are also polymorphic. By contrast, a function that works with only one type of argument, like `<`, is *monomorphic*. Polymorphic functions are especially easy to reuse. In this section, polymorphism is demonstrated in examples that implement set operations and sorting.

As shown in chunks 107a and 107c above, set operations can be implemented using recursive functions. But they can be made more compact and (eventually) easier to understand by using the higher-order functions `exists?`, `curry`, and `foldl`.

**133c.** *(transcript 97a)* +≡

```
> (val emptyset '())
-> (define member? (x s) (exists? ((curry equal?) x) s))
-> (define add-element (x s) (if (member? x s) s (cons x s)))
-> (define union (s1 s2) (foldl add-element s1 s2))
-> (define set-of-list (xs) (foldl add-element '() xs))
-> (set-of-list '(a b c x y a))
(y x c b a)
-> (union '(1 2 3 4) '(2 4 6 8))
(8 6 1 2 3 4)
```

△133a 135a▷

cons	P S307e
even?	129a

These set functions work on sets of atoms, and because `member?` calls `equal?`, they also work on sets of lists of atoms. But they won't work on other kinds of sets, like sets of sets or sets of association lists; `equal?` is too pessimistic.

When programmers can say “code reuse” with a fancy Greek name, they sound smart. What’s less smart is that at least three different programming techniques are all called “polymorphism.”

- The sort of polymorphism we find in the standard list functions is called *parametric polymorphism*.<sup>a</sup> In parametric polymorphism, the polymorphic code always executes the same algorithm in the same way, regardless of the types of the arguments. Parametric polymorphism is the simplest kind of polymorphism, and although it is most useful when combined with higher-order functions, it can be implemented without special mechanisms at run time. Parametric polymorphism is found in some form in every functional language.
- Object-oriented languages such as Smalltalk (Chapter 10) enjoy another kind of polymorphism, which is called *subtype polymorphism*. In subtype polymorphism, code might execute *different* algorithms when operating on values of different types. For example, in a language with subtype polymorphism, squares and circles might be different types of geometric shapes, but they might both implement draw functions. However, the circle would draw differently from the square, using different code. Subtype polymorphism typically requires some sort of object or class system to create the subtypes; examples can be found in Chapter 10.
- Finally, in some languages, a single symbol can stand for unrelated functions. For example, in Python, the symbol + is used not only to add numbers but also to concatenate strings. A number is not a kind of string, and a string is not a kind of number, and the algorithms are unrelated. Nonetheless, + works on more than one type of argument, so it is considered polymorphic. This kind of polymorphism is called *ad-hoc polymorphism*, but Anglo-Saxon people tend to call it *overloading*. It is described, in passing, in Chapter 9.

In object-oriented parts of the world, “polymorphism” by itself usually means subtype polymorphism, and parametric polymorphism is often called *generics*. But to a dyed-in-the-wool functional programmer, *generic programming* means writing recursive functions that consume *types*.<sup>b</sup> If you stick with one crowd or the other, you’ll quickly learn the local lingo, but if your interests become eclectic, you’ll want to watch out for that word “generic.”

<sup>a</sup>The word “parametric” might look like it comes from a function’s parameter or from passing functions as parameters. But the polymorphism is called “parametric” because of *type parameters*, which are defined formally, as part of the language Typed  $\mu$ Scheme, in Chapter 6.

<sup>b</sup>You are not expected to understand this.

To address the issue, let’s focus on sets of association lists. Two association lists are considered equal if they each have the same keys and attributes, *regardless of how the key-attribute pairs are ordered*. In other words, although an association list is *represented* in the world of code as a *sequence* of key-attribute pairs, what it stands for in the world of ideas is a *set* of key-value pairs. Therefore, two association lists are equal if and only if each contains all the key-value pairs found in the other:

**135a.** *(transcript 97a)* +≡

```

-> (define sub-alist? (al1 al2)
  ; all of al1's pairs are found in al2
  (all? (lambda (pair)
    (equal? (alist-pair-attribute pair)
      (find (alist-pair-key pair) al2)))
  al1))
-> (define =alist? (al1 al2)
  (and (sub-alist? al1 al2) (sub-alist? al2 al1)))
-> (=alist? '() '())
#t
-> (=alist? '((E coli) (I Magnin) (U Thant))
  '((E coli) (I Ching) (U Thant)))
#f
-> (=alist? '((U Thant) (I Ching) (E coli))
  '((E coli) (I Ching) (U Thant)))
#t

```

◁133c 136c▷

§2.9

Practice IV:  
Higher-order  
functions for  
polymorphism

135

Function `=alist?` makes it possible to implement sets of association lists, but it doesn't dictate *how*. The wrong way to do it is to write a new version of `member?` which uses `=alist?` instead of `equal?`; it could be called `al-member?`. Then because `add-element` calls `member?`, we would need a new version of `add-element`, which would use `al-member?` instead of `member?`. And so on. The wrong path leads to a destination at which everything except `emptyset` is reimplemented. And then if anyone wants sets of sets, all the operations have to be reimplemented again. The destination is a maintainer's hell, where there are several different implementations of sets, all using nearly identical code and all broken in the same way. For example, the implementation above performs badly on large sets, and if better performance is needed, any improvement has to be reimplemented  $N$  times. Instead of collecting monomorphic implementations of sets, a better way is to use higher-order functions to write one implementation that's polymorphic.

### 2.9.1 Approaches to polymorphism in Scheme

In Scheme, polymorphic set functions can be implemented in three styles. All three use higher-order functions; instead of using `equal?`, the set functions use an equality predicate that is stored in a data structure or passed as a parameter. The style is identified by the location in which the predicate is stored:

- In the simplest style, a new parameter, the equality predicate `my-equal?`, is added to *every* function. The modified functions look like this:

**135b.** *(polymorphic-set transcript 135b)* ≡

(S310d) 136a▷

```

-> (define member? (x s my-equal?)
  (exists? ((curry my-equal?) x) s))
member?
-> (define add-element (x s my-equal?)
  (if (member? x s my-equal?) s (cons x s)))
add-element

```

cons P S307e

Passing an equality predicate to every operation, as in `(member? x s equal?)` or `(add-element x s =alist?)`, is the responsibility of the client code. That's a heavy burden. Client code must keep track of sets *and* their equality predicates, and it must ensure the same equality predicate is always passed with the same set, consistently, every time.

- This burden can be relieved by storing a set's equality predicate with its elements. In the second style, a set is represented by a record with two fields: an equality predicate and a list of elements.

**136a.** *(polymorphic-set transcript 135b)*  $\equiv$  (S310d)  $\triangleleft$  135b 136b  $\triangleright$

```

-> (record aset [eq? elements])
-> (val emptyset (lambda (my-equal?) (make-aset my-equal? '())))
-> (define member? (x s)
      (exists? ((curry (aset-eq? s)) x) (aset-elements s)))
-> (define add-element (x s)
      (if (member? x s)
          s
          (make-aset (aset-eq? s) (cons x (aset-elements s)))))
```

For sets of association lists, these operations are used as follows:

**136b.** *(polymorphic-set transcript 135b)*  $\equiv$  (S310d)  $\triangleleft$  136a  $\triangleright$

```

-> (val alist-empty (emptyset =alist?))
-> (val s (add-element '((U Thant) (I Ching) (E coli)) alist-empty))
  (make-aset <function> (((U Thant) (I Ching) (E coli))))
-> (val s (add-element '((Hello Dolly) (Goodnight Irene)) s))
  (make-aset <function> (((Hello Dolly) (Goodnight Irene)) ((U Thant)...))
-> (val s (add-element '((E coli) (I Ching) (U Thant)) s))
  (make-aset <function> (((Hello Dolly) (Goodnight Irene)) ((U Thant)...))
-> (member? '((Goodnight Irene) (Hello Dolly)) s)
#t
```

The best feature of this style is that a predicate is supplied only when constructing an empty set.

- The second style imposes embarrassing run-time costs. Each set must contain the equality predicate, which adds extra memory to each set, and which requires a level of indirection to gain access either to the equality predicate or to the elements. The second style also imposes a lesser but still nontrivial burden on client code, which has to propagate the equality predicate to each point where an empty set might be created.

In the third style, these issues are addressed by storing the equality predicate in the *operations*. Each operation is represented by a closure, and the equality function is placed the environment of that closure. In this style, no extra memory is added to any set, a set's elements are accessed without indirection, and the equality predicate is cheaper to fetch from a closure than it would be from a set. And client code has to supply each equality predicate only once, to create the specialized operations that work with that predicate.

In a third-style implementation of sets, the equality predicate is placed into closures by function `set-ops-with`, which returns a record of set operations. Each operation is specialized to use the given equality.

**136c.** *(transcript 97a)*  $\equiv$  (S315a 137a)  $\triangleright$

```

-> (record set-ops [member? add-element])
-> (define set-ops-with (my-equal?)
      (make-set-ops
        (lambda (x s) (exists? ((curry my-equal?) x) s)) ; member?
        (lambda (x s) ; add-element
          (if (exists? ((curry my-equal?) x) s) s (cons x s))))
```

Specialized operations for a set of association lists are created by applying `set-ops-with` to predicate `=alist?`.

**137a.** *(transcript 97a)* +≡

```
-> (val alist-set-ops (set-ops-with =alist?))
-> (val al-member?      (set-ops-member?      alist-set-ops))
-> (val al-add-element (set-ops-add-element alist-set-ops))
```

◀136c 137b▶

These operations can be used without mentioning the equality predicate.

**137b.** *(transcript 97a)* +≡

```
-> (val emptyset '())
-> (val s (al-add-element '((U Thant) (I Ching) (E coli)) emptyset))
  (((U Thant) (I Ching) (E coli)))
-> (val s (al-add-element '((Hello Dolly) (Goodnight Irene)) s))
  (((Hello Dolly) (Goodnight Irene)) ((U Thant) (I Ching) (E coli)))
-> (val s (al-add-element '((E coli) (I Ching) (U Thant)) s))
  (((Hello Dolly) (Goodnight Irene)) ((U Thant) (I Ching) (E coli)))
-> (al-member? '((Goodnight Irene) (Hello Dolly)) s)
#t
```

◀137a 137c▶

§2.9  
Practice IV:  
Higher-order  
functions for  
polymorphism

137

### 2.9.2 A polymorphic, higher-order sort

Another widely used, polymorphic algorithm is sorting. Good sorts are tuned for performance, and when code is tuned for performance, it should be reused. And a sort function can be reused in the same way as the set functions: just as sets require an equality function that operates on set elements, sorting requires a comparison function that operates on list elements. So like the set operations, a sort function should take the comparison function as a parameter. For example, the polymorphic, higher-order function `mk-insertion-sort` takes a comparison function `lt?`, and returns a function that sorts a list of elements into nondecreasing order (according to function `lt?`). The algorithm is from chunk 103a.

**137c.** *(transcript 97a)* +≡

```
-> (define mk-insertion-sort (lt?)
  (letrec ([insert (lambda (x xs)
    (if (null? xs)
        (list1 x)
        (if (lt? x (car xs))
            (cons x xs)
            (cons (car xs) (insert x (cdr xs))))))]
    [sort   (lambda (xs)
      (if (null? xs)
          '()
          (insert (car xs) (sort (cdr xs)))))])
  sort))
```

◀137b 137d▶

=alist?	135a
car	¶ 164a
cdr	¶ 164a
cons	¶ S307e
emptyset	107a
null?	¶ 164a

This definition includes our first example of `letrec`, which is like `let`, but which makes each bound name visible to all the others. Internal functions `sort` and `insert` are both recursive, and because both are defined in same `letrec`, either one can also call the other.

Function `mk-insertion-sort` makes sorting flexible and easy to reuse. For example, when used with appropriate comparison functions, it can make both increasing and decreasing sorts.

**137d.** *(transcript 97a)* +≡

```
-> (val sort-increasing (mk-insertion-sort <))
-> (val sort-decreasing (mk-insertion-sort >))
-> (sort-increasing '(6 9 1 7 4 3 8 5 2 10))
  (1 2 3 4 5 6 7 8 9 10)
-> (sort-decreasing '(6 9 1 7 4 3 8 5 2 10))
  (10 9 8 7 6 5 4 3 2 1)
```

◀137c 139a▶

Function `mk-insertion-sort` implements the same pattern of computation as `mk-rand`, only more useful: “if you tell me when one element should come before another, I can sort a list of elements.” It can equally well sort lists by length, sort pairs lexicographically, and whatever else you need. That’s the power of higher-order, polymorphic functions.

## 2.10 PRACTICE V: CONTINUATION-PASSING STYLE

In the examples above, higher-order functions use `lambda` to capture *data* in a closure. Such data include counters, predicates, equality tests, comparison functions, and more general functions. Using these functions as data doesn’t really affect the *control flow* of our computations; for example, no matter what comparison function is passed to insertion sort, the insertion-sort code executes in the same way. But higher-order functions can also be used for control flow. A function call is used in the same way that a C programmer or an assembly-language programmer might use a `goto`: to transfer control and never come back. A function used in this way is called a *continuation*. And a continuation is even more powerful than a `goto`, because a continuation can take arguments!<sup>10</sup>

As an initial example, continuations can be used to represent the exits from a computation. To develop the example, let’s return to association lists. As noted in Section 2.3.8, `find` has a little problem: it cannot distinguish between a key that is bound to `'()` and a key that, like `milk` in the `aisle`, is not bound at all. Solving this problem requires a change in the interface. One possibility is to change only the value returned. For example, as in full Scheme, `find` could return a key-value pair if it finds the key and `#f` if it finds nothing. Or like Lua, it could always return a pair, one element of which would say if it found anything and the other element of which would be what it found. But the conditional logic required to tests these kinds of results can be annoying.

A more flexible alternative is to pass two *continuations* to the `find` routine: one to tell the `find` routine what to do next (“how to continue”) if it succeeds, and one in case of failure. That is, when `(find-c k alist succ fail)` is called, one of two things must happen. If key `k` is found in `al`, then `find-c` calls `succ` with the associated value. If `k` is not found, `find-c` calls `fail` with no arguments:

```
(find-c k alist succ fail) = (succ v), when (find k alist) = v  
(find-c k alist succ fail) = (fail), otherwise
```

The code looks like this:

**138.** *(definition of find-c 138)≡* (139a)

```
(define find-c (key alist success-cont failure-cont)  
  (letrec  
    ([search (lambda (alist)  
              (if (null? alist)  
                  (failure-cont)  
                  (if (equal? key (alist-first-key alist))  
                      (success-cont (alist-first-attribute alist))  
                      (search (cdr alist)))))]  
     (search alist)))
```

<sup>10</sup>Your teachers probably hated `goto`. Maybe they considered it harmful. Or maybe they didn’t even tell you about `goto`—trying to find `goto` in an introductory programming book can be like trying to find Trotsky in a picture of early Soviet leaders. If you’ve been unfairly deprived of `goto`, Knuth (1974) can remedy the injustice.

As an example, `find-c` can be used to code a “pair-or-symbol” response:

139a. *(transcript 97a)* +≡  
-> *(definition of find-c 138)*  
-> *(find-c 'Hello '((Hello Dolly) (Goodnight Irene))*  
    *(lambda (v) (list2 'the-answer-is v))*  
    *(lambda () 'the-key-was-not-found))*  
*(the-answer-is Dolly)*  
-> *(find-c 'Goodbye '((Hello Dolly) (Goodnight Irene))*  
    *(lambda (v) (list2 'the-answer-is v))*  
    *(lambda () 'the-key-was-not-found))*  
*the-key-was-not-found*

◀137d 139b▶

§2.10

Practice V:  
Continuation-  
passing  
style

139

More usefully, `find-c` can implement a table with a default element.

139b. *(transcript 97a)* +≡  
-> *(define find-default (key table default)*  
    *(find-c key table (lambda (x) x) (lambda () default)))*

◀139a 139c▶

Before going deeper into continuations, let’s exercise `find-default`.

Function `find-default` can be used to count frequencies of words, using a table with default value zero. The table maps each word to the number of times it occurs, and it’s used in function `freq`, which finds the frequencies of all the words in a list and lists the most frequent first. In `freq`, words are visited by `foldr`, which is passed `add`, which looks up a word’s count in the table and increases the count by 1.

139c. *(transcript 97a)* +≡  
-> *(define freq (words)*  
    *(let*  
        *([add (lambda (word table)*  
            *(bind word (+ 1 (find-default word table 0)) table)])*  
        *[sort (mk-insertion-sort*  
            *(lambda (p1 p2) (> (cadr p1) (cadr p2))))])*  
        *(sort (foldr add '() words))))*  
-> *(freq '(it was the best of times , it was the worst of times ! ))*  
*((it 2) (was 2) (the 2) (of 2) (times 2) (best 1) (, 1) (worst 1) (! 1))*

◀139b 139d▶

As another example, `find-default` can help find out what words follow another word. (Statistics about sequences of words are sometimes used to characterize authorship.) This time, the table maps each word to the set of words that follow it; the default element is the empty set. Because the follow-set function operates on more than one word at a time, a `foldr` would be a bit hard to read; instead, the code defines an auxiliary recursive function, `walk`:

139d. *(transcript 97a)* +≡  
-> *(define followers (words)*  
    *(letrec*  
        *([add (lambda (word follower table)*  
            *(bind word*  
                *(add-element follower (find-default word table '()))*  
                *table)])*  
        *[walk (lambda (first rest table)*  
            *(if (null? rest)*  
                *table*  
                *(walk (car rest)*  
                    *(cdr rest)*  
                    *(add first (car rest) table))))])*  
        *(walk (car words) (cdr words) '())))*  
-> *(followers '(it was the best of times , it was the worst of times ! ))*  
*((it (was)) (was (the)) (the (worst best)) (best (of)) (of (times)) ...*

add-element 133c  
car P 164a  
cdr P 164a  
mk-insertion-sort 137c  
null? P 164a

The answer doesn't fit on the page, so let's show just the words that are followed by more than one word. They can be selected by using the curried form of `filter` to accept only elements that have more than one word in their `cadr`.

140. *<transcript 97a>* +≡ ◀ 139d 144b ▶

```
-> (define more-than-one? (xs)
  (if (null? xs) #f (not (null? (cdr xs)))))
-> (val multi-followers
  (o
   ((curry filter) (lambda (p) (more-than-one? (cadr p))))
   followers))
-> (multi-followers
  '(it was the best of times , it was the worst of times ! ))
((the (worst best)) (times (! ,)))
-> (multi-followers
  '(now is the time for all good men to come to the aid of the party))
((the (party aid time)) (to (the come)))
```

### 2.10.1 Continuation-passing style and direct style

A function like `find-c` is said to be written in *continuation-passing style*: not only does it take continuations as arguments, but it can return only what the continuations return.<sup>11</sup> Continuation-passing style is easily identified by looking at a function's algebraic laws: if every right-hand side is a call to a parameter, the function is written in continuation-passing style. Continuation-passing style can also be identified by looking at the code: a function written in continuation-passing style ends either by calling a continuation or by calling another function written in continuation-passing style (including itself).

A function written in continuation-passing style is inherently polymorphic: the type of its result is whatever the continuations return, and if it's given different continuations, it can return values of different types. When the function is defined but not yet used, the continuations are not yet known, so the type of the result isn't yet known. But that type is still useful to refer to, so it is referred to abstractly as the *answer type*. The answer type can help you design appropriate continuations; for example, in `find-default`, the answer type is the type of `default`, but it is also the type of `(lambda (x) x)` applied to a value in `table`—so the type of `default` must be the same as the types of the values in `table`.

Unlike `find-c`, `find-default` is *not* written in continuation-passing style: it does not take any continuations, so it cannot return the result of calling one. A function like `find-default` is written in *direct style*. Direct style is usually easier to understand, but continuation-passing style is more flexible and powerful.

### 2.10.2 Continuation-passing for backtracking

When it succeeds or fails, `find-c` exits. But continuations can also more sophisticated control flow, like the ability to backtrack on failure. In direct-style imperative code, backtracking is hard to get right, but using continuations and purely functional data structures, it's easy. And a backtracking problem can demonstrate the virtues of leaving the answer type unknown.

One classic problem that can be solved with backtracking is *Boolean satisfaction*. The problem is to find an assignment to a collection of variables such that a Boolean formula is satisfied, that is, a *satisfying assignment*. This problem has many

<sup>11</sup>Properly speaking, continuation-passing style is a representation of programs in which *all* functions, even primitive functions, end by transferring control to a continuation (Appel 1992).

applications, often in verifying the correctness of a hardware or software system. For example, on a railroad, if the signals are obeyed, no two trains should ever be on the same track at the same time, and this property can be checked by solving a very large Boolean-satisfaction problem.

The satisfaction problem can be simplified by considering only formulas in *conjunctive normal form*. (General Boolean formulas are the subject of Exercise 41.) A conjunctive normal form is a conjunction of disjunctions of literals:

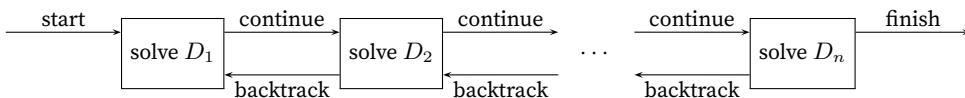
$$\begin{aligned} \textit{CNF} &::= D_1 \wedge D_2 \wedge \cdots \wedge D_n && \text{conjunction} \\ D &::= l_1 \vee l_2 \vee \cdots \vee l_m && \text{disjunction} \\ l &::= x \mid \neg x && \text{literal} \end{aligned}$$

The  $x$  is a metavariable that may stand for any Boolean variable.

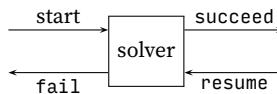
A *CNF* formula is satisfied if *all* of its disjunctions  $D_i$  are satisfied, and a disjunction is satisfied if *any* of its literals  $l_i$  is satisfied. A literal  $x$  is satisfied if  $x$  is true; a literal  $\neg x$  is satisfied if  $x$  is false. A formula is satisfied by more than one assignment; for example, the formula  $x \vee y \vee z$  is satisfied by 7 of the 8 possible assignments to  $x, y, z$ . Satisfying assignments can be found by a backtracking search. Because the satisfaction problem is NP-hard, a search might take exponential time.

The search algorithm presented below incrementally improves an *incomplete* assignment. An incomplete assignment associates values with some variables—all, none, or some number in between. An incomplete assignment is represented by an association list in which each key is the name of a variable and each value is #t or #f. A variable that doesn't appear in the list is *unassigned*.

Given disjunction  $D_i$  and an incomplete assignment  $\text{cur}$ , the search algorithm tries to *extend*  $\text{cur}$  by adding variables in such a way that  $D_i$  is satisfied. If that works, the algorithm continues by trying to extend the assignment to satisfy  $D_{i+1}$ . But if it can't satisfy  $D_i$ , the algorithm doesn't give up; instead it *backtracks* to  $D_{i-1}$ . Maybe  $D_{i-1}$  can be satisfied in a *different* way, such that it becomes possible to satisfy  $D_i$ . As an example, suppose  $D_1 = x \vee y \vee z$ , and the search algorithm finds the assignment  $\{x \mapsto \#t, y \mapsto \#f, z \mapsto \#f\}$ , which satisfies it. If  $D_2 = \neg x \vee y \vee z$ , the search has to go back and find a different assignment to satisfy  $D_1$ . The search can be viewed as a process of going back and forth over the  $D_i$ 's, tracing a path through this graph:



The graph is composed of solvers that look like this:



cdr	$\mathcal{P}$ 164a
followers	139d
null?	$\mathcal{P}$ 164a

Thanks to the graph structure, a solver can just tackle an individual  $D_i$ , without keeping track of what it's supposed to do afterward. Its future obligations are stored in two continuations, *succeed* and *fail*, which it receives as parameters. The third continuation in the diagram, written as *resume*, is the failure continuation for the next solver: if the next solver can't succeed, it calls *resume* to ask the current solver to try something else.

The graph structure and its continuations are used to solve formulas in all three forms: *CNF*,  $D$ , and  $l$ . Each form is solved by its own function. For example, a *CNF* formula  $D_1 \wedge \cdots \wedge D_n$  is solved by a function that receives the formula,

an incomplete assignment `cur`, and continuations `succeed` and `fail`. When the solver is called, as represented above by the arrow labeled “start,” it acts as follows:

- If `cur` cannot possibly satisfy the first disjunction  $D_1$ , call `fail` with no parameters.
- If `cur` can satisfy  $D_1$ , or if it can be extended to satisfy  $D_1$ , then compute the (possibly extended) satisfying solution, and pass it to `succeed`. In addition, because  $D_1$  might have *more than one* solution, also pass `resume`, which `succeed` can call if it needs to backtrack and try an alternative solution.

To make this story precise requires precise specifications of the solver functions and of a representation of formulas.

The representation of *CNF*, the set of formulas in conjunctive normal form, is described by the following equations, which resemble the recursion equations in Section 2.5.8:

$$\begin{aligned} \textit{CNF} &= \textit{LIST}(D) \\ D &= \textit{LIST}(\textit{LIT}_B) \\ \textit{LIT}_B &= \textit{SYM} \cup \{ (\textit{list2} \textit{'not} x) \mid x \in \textit{SYM} \} \end{aligned}$$

In conjunctive normal form, the symbols  $\wedge$  and  $\vee$  are implicit. The  $\neg$  symbol is represented explicitly using the Scheme symbol `not`.

In the code, the forms are referred to by names `ds`, `ls`, and `lit`. Assignments and continuations are also referred to by conventional names:

<code>ds</code>	A list of disjunctions in <i>CNF</i>
<code>ls</code>	A list of literals in $D$
<code>lit</code>	A single literal in $\textit{LIT}_B$
<code>cur</code>	A current assignment (association list)
<code>fail</code>	A failure continuation
<code>succeed</code>	A success continuation

A failure continuation takes no arguments and returns an answer. A success continuation takes two arguments—an assignment and a failure continuation—and returns an answer.

Each of the three solver functions is written in continuation-passing style.

- Calling `(find-cnf-true-assignment ds cur fail succeed)` tries to extend `cur` to an assignment that satisfies the list of disjunctions `ds`
- Calling `(find-D-true-assignment ls cur fail succeed)` tries to extend `cur` to an assignment that satisfies any literal in the list `ls`.
- Calling `(find-lit-true-assignment lit cur fail succeed)` tries to extend `cur` to an assignment that satisfies literal `lit`.

Each function calls `succeed` on success and `fail` on failure. The specifications can be made precise using algebraic laws.

The laws for any given solver function depend on what species of formula the function solves. Each species is solved by a different algorithm, which depends on how a formula in the species is satisfied. For example, a *CNF* formula is satisfied if all its disjunctions are satisfied. There are two cases:

- If there are no disjunctions, then `cur` trivially satisfies them all. And when `cur` is passed to `succeed`, nothing else can be done with an empty list of disjunctions, so the resumption continuation is `fail`.

- If there is a disjunction, it should be solved by calling the solver for disjunctions: `find-D-true-assignment`. If that call fails, nothing more can be done, so its failure continuation should be `fail`. But if that call succeeds, the search needs to continue by solving the remaining disjunctions. So the *CNF* solver creates a new continuation to be passed as the success continuation. The new continuation calls `find-cnf-true-assignment` recursively, and if the recursive call fails, it backtracks by transferring control to `resume`.

Each case is described by a law:

```
(find-cnf-true-assignment '()      cur fail succeed)) = (succeed cur fail)
(find-cnf-true-assignment (cons d ds) cur fail succeed)) =
  (find-D-true-assignment d cur fail
    (lambda (cur' resume) (find-cnf-true-assignment ds cur' resume succeed)))
```

The flow of control is characteristic of continuation-passing style: in the nonempty case, only the disjunction solver knows whether `d` can be solved, but it's the *CNF* solver that decides what to do—and that decision is embodied in the continuations that are passed to the disjunction solver.

The disjunction solver works in almost the same way, except that a disjunction is satisfied if any one of its literals is satisfied. So in the nonempty case, the roles of the success and failure continuations are reversed: if a literal succeeds, the solver succeeds, but if a literal fails, the solver must try the other literals.

```
(find-D-true-assignment '()      cur fail succeed)) = (fail)
(find-D-true-assignment (cons lit ls) cur fail succeed)) =
  (find-lit-true-assignment lit cur
    (lambda () (find-D-true-assignment ls cur fail succeed)))
  succeed)
```

You might be concerned that if the first literal is solved successfully, the later literals will never be examined, and some possible solutions might be missed. But those later literals `ls` are examined by the failure continuation that is passed to `find-lit-true-assignment`, which is eventually passed to `succeed` as a resumption continuation. If anything goes wrong downstream, that resumption continuation will eventually be invoked, and it will call `(find-D-true-assignment ls ...)`.

The last of the three solver functions solves a literal. It needs to know what variable appears in a literal and what value would satisfy the literal.

**143a.** *(utility functions for solving literals 143a)*≡

143b▷

```
-> (define variable-of (lit)
  (if (symbol? lit)
    lit
    (cadr lit)))
-> (define satisfying-value (lit)
  (symbol? lit)) ; #t satisfies 'x; #f satisfies '(not x)
```

find-c      138  
 symbol?      P 164a

Because an ordinary literal is a symbol and a negated literal is a list like `(not x)`, the value that satisfies a literal `lit` is always equal to `(symbol? lit)`.

A literal is satisfied if and only if the current assignment binds the literal's variable to a satisfying value. The test uses `find-c` with a Boolean answer type.

**143b.** *(utility functions for solving literals 143a)*+≡

△143a 144a▷

```
-> (define satisfies? (alist lit)
  (find-c (variable-of lit) alist
    (lambda (b) (= b (satisfying-value lit)))
    (lambda () #f)))
```

The last solver function, `find-lit-true-assignment`, succeeds either when the current assignment already satisfies its literal, or when it can be made to satisfy the literal by adding a binding. There is at most one way to succeed, so the resumption continuation passed to succeed is always fail.

```
(find-lit-true-assignment lit cur fail succeed) = (succeed cur fail),
  when cur satisfies lit
(find-lit-true-assignment lit cur fail succeed) = (succeed (bind x v cur) fail),
  when x is lit's variable, cur does not bind x, and lit is satisfied by {x ↦ v}
(find-lit-true-assignment lit cur fail succeed) = (fail), otherwise
```

144

To determine that cur does not bind  $x$ , the solver function uses `find-c` with yet another pair of continuations:

**144a.** *(utility functions for solving literals 143a)* +≡ △143b

```
-> (define binds? (alist lit)
      (find-c (variable-of lit) alist (lambda (_) #t) (lambda () #f)))
```

All three solver functions are coded in Figure 2.5 on the facing page. And they can be used with different continuations that produce answers of different types. For example, the *CNF* solver can be used to tell whether a formula is satisfiable, to produce one solution, or to produce all solutions. To see if a formula has a solution, use `find-cnf-true-assignment` with a success continuation that returns `#t` and a failure continuation that returns false:

**144b.** *(transcript 97a)* +≡ △140 144c ▷

```
-> (define satisfiable? (formula)
      (find-cnf-true-assignment formula '()
                                (lambda () #f)
                                (lambda (cur resume) #t)))
```

As an example, the formula

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee y \vee \neg z).$$

has a solution.

**144c.** *(transcript 97a)* +≡ △144b 144d ▷

```
-> (val sample-formula '((x y z) ((not x) (not y) (not z)) (x y (not z))))
-> (satisfiable? sample-formula)
#t
```

To actually solve a formula, use a success continuation that returns `cur`:

**144d.** *(transcript 97a)* +≡ △144c 144e ▷

```
-> (define one-solution (formula)
      (find-cnf-true-assignment formula '()
                                (lambda () 'no-solution)
                                (lambda (cur resume) cur)))
-> (one-solution sample-formula)
((x #t) (y #f))
```

This formula is satisfied if  $x$  is true and  $y$  is false; the value of  $z$  doesn't matter.

To find all solutions, use a success continuation that adds the current solution to the set returned by `resume`. (To avoid adding duplicate solutions, `cur` is added using `al-add-element`, not `cons`.) The failure continuation returns the empty set.

**144e.** *(transcript 97a)* +≡ △144d 145b ▷

```
-> (define all-solutions (formula)
      (find-cnf-true-assignment
        formula
        '()
        (lambda () emptyset)
        (lambda (cur resume) (al-add-element cur (resume)))))
```

**145a.** *(solver functions for CNF formulas 145a)*  $\equiv$

```
-> (define find-lit-true-assignment (lit cur fail succeed)
  (if (satisfies? cur lit)
    (succeed cur fail)
    (if (binds? cur lit)
      (fail)
      (let ([new (bind (variable-of lit) (satisfying-value lit) cur)])
        (succeed new fail))))
  -> (define find-D-true-assignment (literals cur fail succeed)
  (if (null? literals)
    (fail)
    (find-lit-true-assignment (car literals) cur
      (lambda () (find-D-true-assignment
        (cdr literals) cur fail succeed)))
    succeed)))
  -> (define find-cnf-true-assignment (disjunctions cur fail succeed)
  (if (null? disjunctions)
    (succeed cur fail)
    (find-D-true-assignment (car disjunctions) cur fail
      (lambda (cur' resume)
        (find-cnf-true-assignment
          (cdr disjunctions) cur' resume succeed))))))
```

§2.10

Practice V:  
Continuation-  
passing  
style

145

Figure 2.5: The three solver functions

There are 9 distinct solutions, which is too many to show.

**145b.** *(transcript 97a)*  $\equiv$

```
-> (val answers (all-solutions sample-formula))
-> (length answers)
9
```

$\triangleleft$  144e 145c  $\triangleright$

Is there a solution in which  $x$  and  $y$  are both false? Both true?

**145c.** *(transcript 97a)*  $\equiv$

```
-> (exists?
  (lambda (cur) (and (= #f (find 'x cur)) (= #f (find 'y cur))))
  answers)
#f
-> (exists?
  (lambda (cur) (and (= #t (find 'x cur)) (= #t (find 'y cur))))
  answers)
#t
```

$\triangleleft$  145b 145d  $\triangleright$

It appears that  $x$  and  $y$  can be true at the same time, but not false at the same time.

Not all formulas have solutions. Most obviously, formula  $x \wedge \neg x$  has no solution.

**145d.** *(transcript 97a)*  $\equiv$

```
-> (one-solution '((x) ((not x))))
no-solution
```

$\triangleleft$  145c 166c  $\triangleright$

Both the solver and `find-c` demonstrate that a function written in continuation-passing style can often be used in different ways, simply by passing it different continuations with different answer types.

al-add-element	137a
car	$\mathcal{P}$ 164a
cdr	$\mathcal{P}$ 164a
emptyset	107a
find-c	138
length	100
null?	$\mathcal{P}$ 164a
satisfies?	143b
satisfying-value	143a
variable-of	143a

## 2.11 OPERATIONAL SEMANTICS

The operational semantics of  $\mu$ Scheme resembles the operational semantics of Impcore. And it uses the same techniques—an abstract machine, a big-step evaluation judgment, inference rules, and so on. But Impcore and  $\mu$ Scheme treat environments differently:

- Impcore uses three environments, which bind functions, global variables, and formal parameters respectively.  $\mu$ Scheme uses a single environment  $\rho$ , which binds all names.
- In Impcore, environments bind names to values. In  $\mu$ Scheme, environments bind names to *mutable locations*, and SET changes the contents of a location, not a binding in an environment. The contents of locations are modeled by an explicit store  $\sigma$ , which is a function from locations to values; think of it as the machine's memory.
- In Impcore, environments are never copied, so when the semantics says a name is rebound, the implementation can mutate the relevant environment data structure. In  $\mu$ Scheme, an environment is copied every time a lambda is evaluated, so it isn't safe to mutate an environment data structure. Instead, the implementation builds an extended environment with a new binding.

$\mu$ Scheme's semantics assumes there is an infinite supply of fresh locations; a fresh location  $\ell$  is allocated by writing  $\ell \notin \text{dom } \sigma$ . In Section 2.12, this allocation judgment is implemented using `malloc`, which has only a finite supply of locations. In Chapter 4, the allocation judgment is implemented using a more sophisticated allocator, which creates the illusion of an infinite supply by reusing old locations.

In  $\mu$ Scheme, an abstract machine evaluating an expression  $e$  in environment  $\rho$  is in state  $\langle e, \rho, \sigma \rangle$ . The evaluation judgment takes the form  $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$ . This judgment says two things:

- The result of evaluating expression  $e$  in environment  $\rho$ , when the state of the store is  $\sigma$ , is the value  $v$ .
- The evaluation may update the store to produce a new store  $\sigma'$ .

Evaluating an expression never changes an environment; at most, evaluating an expression can change the store. Therefore, the right-hand side of an evaluation judgment does not need a new environment  $\rho'$ .

Evaluation never copies a store  $\sigma$ , and when a rule makes a new store  $\sigma'$ , it discards the old store. Therefore each store is used exactly once; such a store is called *single-threaded* or *linear*. The store is *designed* that way because a single-threaded store can be implemented efficiently: when a rule calls for a new store, the implementation does not have to build one; instead, it mutates the previous store.

### 2.11.1 Abstract syntax and values

$\mu$ Scheme's abstract syntax is straightforward. It has the expressions of Impcore, plus LETX and LAMBDA expressions. And APPLY can apply arbitrary expressions, not just names.

**146.**  $\langle \text{ast.t 146} \rangle \equiv$

```
Def* = VAL    (Name name, Exp exp)
      | EXP    (Exp)
      | DEFINE (Name name, Lambda lambda)
```

147a▷

147a. *(ast.t 146)* +≡

△146

```

Exp* = LITERAL (Value)
| VAR      (Name)
| SET      (Name name, Exp exp)
| IFX      (Exp cond, Exp truex, Exp falsex)
| WHILEX   (Exp cond, Exp body)
| BEGIN    (Explist)
| APPLY    (Exp fn, Explist actuals)
| LETX     (Letkeyword let, Namelist xs, Explist es, Exp body)
| LAMBDA(X (Lambda)

```

Definitions of Value and Lambda are shown with the interpreter (Section 2.12); type Letkeyword is defined here.

147b. *(type definitions for μScheme 147b)* ≡

(S303d) 154b ▷

```
typedef enum Letkeyword { LET, LETSTAR, LETREC } Letkeyword;
```

A μScheme value is a symbol, number, Boolean, empty list, cons cell, closure, or primitive function. These values are written SYMBOL(*s*), NUMBER(*n*), BOOLV(*b*), NIL, PAIR⟨*l*<sub>1</sub>, *l*<sub>2</sub>⟩, ⟨LAMBDA⟨⟨*x*<sub>1</sub>, …, *x*<sub>*n*</sub>⟩, *e*⟩, *ρ*⟩, and PRIMITIVE(*p*). The corresponding data definition is shown with the interpreter (Section 2.12).

### 2.11.2 Variables and functions

The most interesting parts of the semantics are the rules for lambda, let, and function application. The key ideas involve mutable locations:

- Each variable refers to a location. To examine or change a variable, we must first look up its name in *ρ* (to find its location), then look at or change the contents of the location (which means looking at *σ* or producing a new *σ'*).<sup>12</sup>
- Let-bindings allocate fresh locations, bind them to variables, and initialize them.
- Function application also allocates fresh locations, which hold the values of actual parameters. These locations are then bound to the names of the formal parameters of the function being applied.

#### Variables

The single environment makes it easy to look up the value of a variable. Lookup requires two steps: *ρ(x)* to find the location in which *x* is stored, and *σ(ρ(x))* to fetch its value. In a compiled system, these two steps are implemented at different times. At compile time, the compiler decides in what location to keep *x*. At run time, a machine instruction fetches a value from that location.

Looking up a variable doesn't change the store *σ*.

$$\frac{x \in \text{dom } \rho \quad \rho(x) \in \text{dom } \sigma}{\langle \text{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle} \quad (\text{VAR})$$

Assignment translates the name into a location, then changes the value in that location, producing a new store. The evaluation of the right-hand side may also change the store, from *σ* to *σ'*.

$$\frac{x \in \text{dom } \rho \quad \rho(x) = \ell \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \{ \ell \mapsto v \} \rangle} \quad (\text{ASSIGN})$$

<sup>12</sup>In μScheme, two variables never refer to the same location. If *x* and *y* did refer to the same location, then when *x* was mutated, *y* would change. This behavior is called “aliasing,” and it makes compiler writers miserable. μScheme variables cannot alias (Exercise 51).

The threading of the store is subtle; the conclusion must extend  $\sigma'$ , not  $\sigma$ . If the conclusion were to extend  $\sigma$ , the semantics would be saying that changes made by  $e$  should be undone. This behavior would surprise the programmer, and it would force the implementor to save a copy of the old  $\sigma$  and then reinstate it after the evaluation of  $e$ . At considerable expense.

*Let, let\*, and letrec*

A LET expression evaluates all the right-hand sides, then binds the names to the resulting values, and finally evaluates its body in the resulting environment. Nothing is added to the environment  $\rho$  until all the right-hand sides have been evaluated.

$$\frac{\begin{array}{c} x_1, \dots, x_n \text{ all distinct} \\ \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \hline \langle e, \rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \\ \text{(LET)} \end{array}}{\langle \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Locations  $\ell_1, \dots, \ell_n$  are not only fresh but also mutually distinct.

By contrast, a LETSTAR expression binds the result of each evaluation into the environment immediately. The action is revealed by the subscripts on  $\rho$ .

$$\frac{\begin{array}{c} \rho_0 = \rho \quad \sigma_0 = \sigma \\ \langle e_1, \rho_0, \sigma_0 \rangle \Downarrow \langle v_1, \sigma'_0 \rangle \quad \ell_1 \notin \text{dom } \sigma'_0 \quad \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} \quad \sigma_1 = \sigma'_0\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma'_{n-1} \rangle \\ \ell_n \notin \text{dom } \sigma'_{n-1} \quad \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} \quad \sigma_n = \sigma'_{n-1}\{\ell_n \mapsto v_n\} \\ \hline \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle \\ \text{(LETSTAR)} \end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Finally, LETREC binds the locations into the environment *before* evaluating the expressions, so that references to the new names are valid in every right-hand side, and the names stand for the new locations, not for any old ones.

$$\frac{\begin{array}{c} \ell_1, \dots, \ell_n \notin \text{dom } \sigma \text{ (and all distinct)} \\ x_1, \dots, x_n \text{ all distinct} \\ e_i \text{ has the form LAMBDA}(\dots), 1 \leq i \leq n \\ \rho' = \rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\ \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \dots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \hline \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \\ \text{(LETREC)} \end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

The contents of  $\ell_1, \dots, \ell_n$  are not specified until *after* all the  $e_1, \dots, e_n$  have been evaluated; informally, the  $e_i$ 's are evaluated with the  $x_i$ 's bound to new, uninitialized locations. Until the locations are initialized, using them is unsafe; using a location that contains “unspecified” is an *unchecked* run-time error. For safety, as in full Scheme, the evaluation of every  $e_i$  must be independent of the value of any  $x_j$ ,

and it also must not mutate any  $x_j$ . In other words, evaluating  $e_i$  must neither read nor write the contents of any location  $\ell_j$ . To guarantee that evaluating  $e_i$  does not read or write *any* locations,  $\mu$ Scheme insists that each  $e_i$  be a LAMBDA expression.

*Function abstraction; function application; statically scoped closures*

Functional abstraction wraps the current environment, along with a lambda expression, in a closure. LAMBDA *copies* the current environment. It is because environments can be copied that they map names to locations, not values; otherwise different closures could not share a mutable location. (One example of such sharing is the “resettable counter” in Section 2.7.1.)

$$\frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle (\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho \rangle, \sigma) \rangle} \quad (\text{MKCLOSURE})$$

When a closure  $\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \rho_c \rangle$  is applied, the body of the function,  $e_c$ , is evaluated using the environment in the *closure*,  $\rho_c$ , which is extended by binding the formal parameters to *fresh* locations  $\ell_1, \dots, \ell_n$ . These locations are initialized with the values of the actual parameters; afterward, the body can change them.

$$\frac{\begin{array}{c} \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\ \langle e, \rho, \sigma \rangle \Downarrow \langle (\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \rho_c \rangle, \sigma_0) \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \hline \langle e_c, \rho_c \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma_n \{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{APPLY}(e, e_1, \dots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{APPLYCLOSURE})$$

The APPLYCLOSURE rule closely resembles the APPLYUSER rule used in Impcore. The crucial differences are as follows:

- In  $\mu$ Scheme, the function to be applied is designated by an arbitrary expression, which must evaluate to a closure. In Impcore, the function is designated by a name, which is looked up in  $\phi$  to find a user-defined function. Both a  $\mu$ Scheme closure and an Impcore user-defined function contain formal parameters and a function body, but only the closure contains the all-important environment  $\rho_c$ , which stores the locations of the variables mentioned in the body.<sup>13</sup> In Impcore, no such environment is needed; every function is defined at top level, so every variable is either a formal parameter or can be found in the global environment  $\xi$ .
- In  $\mu$ Scheme, the parameters are added not to the empty environment but to the environment  $\rho_c$  stored in the closure. And like let-bound names, the formal parameters are bound to fresh locations, not directly to values.

As in Impcore, the evaluation of  $e_c$  is independent of the environment  $\rho$  of the calling function, so a  $\mu$ Scheme function behaves the same way no matter where it is called from. (Also as in Impcore, the evaluation is *not* independent of the store  $\sigma$  at the point of the call, so when mutable state is involved, a  $\mu$ Scheme function may

<sup>13</sup>In my implementation, the environment binds all variables that are in scope, not just those that are mentioned in the body of the lambda expression. Such promiscuous binding could use memory unnecessarily, and it could add to the expense of applying the closure. For details, see Exercise 19 on page 304 in Chapter 4 and Exercise 10 on page 332 in Chapter 5.

behave differently when called at different *times*, even when called from the same *place* in the source code. A good example is the counter-stepping function from Section 2.7.1.)

The **APPLYCLOSURE** rule describes an implementation of first-class, nested functions called *statically scoped closures*. The “closure” part you know about; “statically scoped” means that the meanings of the variables (scope) are determined by the compile-time (static) context in which the code appears—those variables are either bound in the global environment or are bound by statically enclosing lets and lambdas. Because static scoping is determined by properties of the text, it is sometimes also called *lexical scoping*. As an example of local reasoning about a static context, a programmer (or an implementation of Scheme) who encounters an expression like `(lambda (x) (+ x y))` can discover what `y` is simply by searching “outward” for a `let` or `lambda` that binds `y`. The search stops at the enclosing definition, which is at top level; if the search reaches top level without finding a `let` binding or `lambda` binding for `y`, then `y` must be bound in the global environment by a `val` or `define`.

*Dynamic scoping: an alternative to static scoping* Closures are an innovation of Scheme; they were not part of original Lisp. Original Lisp represented every function as a bare **LAMBDA** expression with formal parameters and a body, in much the same way as Impcore represents a user-defined function. This representation admits of a very simple implementation, which never has to copy an environment. Original Lisp treated **LAMBDA** as a value, using the following rule for function abstraction.

$$\overline{\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \sigma \rangle} \\ (\text{LAMBDA for original Lisp})$$

Original Lisp applied a function by evaluating its body in the environment of its *caller*, extended with bindings of formal parameters:

$$\begin{aligned} & \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\ & \langle e, \rho, \sigma \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \sigma_0 \rangle \\ & \quad \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ & \quad \vdots \\ & \quad \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ & \langle e_c, \rho \{ x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n \}, \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n \} \rangle \Downarrow \langle v, \sigma' \rangle \\ & \qquad \qquad \qquad (\text{APPLYLAMBDA for original Lisp}) \end{aligned}$$

This rule is much easier to implement than Scheme’s rule, but it makes `lambda` much less useful; in original Lisp, higher-order functions that contain inner `lambda`s, like `o` and `curry`, don’t work as expected.

To find the location of a free variable, an implementation of original Lisp looks at the calling function, and its caller, and so on. The context in which a function is called is a dynamic property, and the original rule is a form of *dynamic scoping*. Dynamic scoping came about because McCarthy didn’t realize that closures were needed; when the original behavior was called to his attention, he characterized the semantics as a bug in the Lisp interpreter. Scheme’s static scoping was widely seen as an improvement, and in the 1980s, static scoping was adopted in Common Lisp. If you want to experiment with a language that has dynamic scoping, try Emacs Lisp. Some interesting examples are presented by Abelson and Sussman (1985, pp. 321–325).

Function application takes the environment in a closure and extends it with bindings for formal parameters:

$$\rho_c \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}.$$

The idea is that formal parameters hide outer variables of the same name. To express that idea formally, formal parameters can be placed in an environment of their own:  $\rho_f = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ . The two environments can then be combined using a + symbol:  $\rho_c + \rho_f$ . The combined environment, in which the body of the function is evaluated, obeys the following laws:

$$\begin{aligned}\text{dom}(\rho_c + \rho_f) &= \text{dom } \rho_c \cup \text{dom } \rho_f \\ (\rho_c + \rho_f)(x) &= \begin{cases} \rho_f(x), & \text{when } x \in \text{dom } \rho_f \\ \rho_c(x), & \text{when } x \notin \text{dom } \rho_f \end{cases}\end{aligned}$$

To look for  $x$  in a combined environment, we look first in  $\rho_f$ , and only if we don't find it do we look in  $\rho_c$ . That is,  $\rho_f$  hides  $\rho_c$ . Combined environments start to shine when there are more of them; for example, in Smalltalk-80, the body of a method is evaluated in an environment that combines class variables, instance variables, formal parameters, and local variables.

### 2.11.3 Rules for other expressions

The rules for evaluating the other expressions of  $\mu$ Scheme are very similar to the corresponding rules of Impcore.

As in Impcore, a literal value evaluates to itself without changing the store.

$$\frac{}{\langle \text{LITERAL}(v), \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle} \quad (\text{LITERAL})$$

Conditionals, loops, and sequences are evaluated as in Impcore, except that falsehood is represented by `BOOLV(#f)`, not 0.

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 \neq \text{BOOLV}(\#f) \quad \langle e_2, \rho, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma \rangle \Downarrow \langle v_2, \sigma'' \rangle} \quad (\text{IFTRUE})$$

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 = \text{BOOLV}(\#f) \quad \langle e_3, \rho, \sigma' \rangle \Downarrow \langle v_3, \sigma'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma \rangle \Downarrow \langle v_3, \sigma'' \rangle} \quad (\text{IFFALSE})$$

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 \neq \text{BOOLV}(\#f) \quad \langle e_2, \rho, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle \quad \langle \text{WHILE}(e_1, e_2), \rho, \sigma'' \rangle \Downarrow \langle v_3, \sigma''' \rangle}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma \rangle \Downarrow \langle v_3, \sigma''' \rangle} \quad (\text{WHILEITERATE})$$

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 = \text{BOOLV}(\#f)}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma \rangle \Downarrow \langle \text{BOOLV}(\#f), \sigma' \rangle} \quad (\text{WHILEEND})$$

$$\frac{}{\langle \text{BEGIN}(), \rho, \sigma \rangle \Downarrow \langle \text{BOOLV}(\#f), \sigma \rangle} \quad (\text{EMPTYBEGIN})$$

$$\begin{aligned}\langle e_1, \rho, \sigma_0 \rangle &\Downarrow \langle v_1, \sigma_1 \rangle \\ \langle e_2, \rho, \sigma_1 \rangle &\Downarrow \langle v_2, \sigma_2 \rangle\end{aligned}$$

⋮

$$\frac{\langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \rho, \sigma_0 \rangle \Downarrow \langle v_n, \sigma_n \rangle} \quad (\text{BEGIN})$$

Of  $\mu$ Scheme's many primitives, only a few are presented here.

*Arithmetic* Primitive arithmetic operations obey the same rules as in Impcore, but they have to be notated differently: in Impcore, every value is a number, but in  $\mu$ Scheme, only a value of the form  $\text{NUMBER}(n)$  is a number. As in Chapter 1, addition serves as a model for all the arithmetic primitives.

Scheme,

S-expressions, and  
first-class functions

$$\frac{\begin{array}{c} \langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(+), \sigma_1 \rangle \\ \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2 \rangle \\ \langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle \text{NUMBER}(m), \sigma_3 \rangle \end{array}}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{NUMBER}(n + m), \sigma_3 \rangle} \quad (\text{APPLYADD})$$

*Equality* Testing for equality is a bit tricky. Primitive function  $=$  implements a relation  $v \equiv v'$ , which is defined by these rules:

$$\frac{m = n \text{ (identity of numbers)}}{\text{NUMBER}(n) \equiv \text{NUMBER}(m)} \quad (\text{EQNUMBER})$$

$$\frac{s = s' \text{ (identity of symbols)}}{\text{SYMBOL}(s) \equiv \text{SYMBOL}(s')} \quad (\text{EQSYMBOL})$$

$$\frac{b = b' \text{ (identity of Booleans)}}{\text{BOOLV}(b) \equiv \text{BOOLV}(b')} \quad (\text{EQBOOL})$$

$$\frac{}{\text{NIL} \equiv \text{NIL}} \quad (\text{EQNIL})$$

The complementary relation is written  $v \not\equiv v'$ ; it holds if and only if there is no proof of  $v \equiv v'$ . It is worth noting that  $\text{PAIR}(\ell_1, \ell_2) \not\equiv \text{PAIR}(\ell_1, \ell_2)$ . That is, cons cells never compare equal, even when they are identical.

The behavior of primitive equality is defined using relations  $\equiv$  and  $\not\equiv$ :

$$\frac{\begin{array}{c} \langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(=), \sigma_1 \rangle \\ \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v_1, \sigma_2 \rangle \\ \langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle v_2, \sigma_3 \rangle \\ v_1 \equiv v_2 \end{array}}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{BOOLV}(\#t), \sigma_3 \rangle} \quad (\text{APPLYEQTRUE})$$

$$\frac{\begin{array}{c} \langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(=), \sigma_1 \rangle \\ \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v_1, \sigma_2 \rangle \\ \langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle v_2, \sigma_3 \rangle \\ v_1 \not\equiv v_2 \text{ (i.e., no proof of } v_1 \equiv v_2) \end{array}}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{BOOLV}(\#f), \sigma_3 \rangle} \quad (\text{APPLYEQFALSE})$$

*Printing* As in Impcore, the operational semantics takes no formal notice of printing, so the semantics of `println` are those of the identity function.

$$\frac{\begin{array}{c} \langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{println}), \sigma_1 \rangle \\ \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v, \sigma_2 \rangle \end{array}}{\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle v, \sigma_2 \rangle \quad \text{while printing } v} \quad (\text{APPLYPRINTLN})$$

Primitive `println` has the same semantics as `print`, but `printu` has a more restrictive semantics: it is defined only when the code point is a suitable number.

$$\frac{\begin{array}{c} \langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{printu}), \sigma_1 \rangle \\ \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2 \rangle \\ 0 \leq n < 2^{16} \end{array}}{\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2 \rangle \quad \text{while printing the UTF-8 coding of } n} \quad (\text{APPLYPRINTU})$$

*List operations* The primitive CONS builds a new cons cell. The cons cell is a pair of *locations* that hold the values of the car and cdr.

$$\frac{\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{cons}), \sigma_1 \rangle \quad \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v_1, \sigma_2 \rangle \quad \langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle v_2, \sigma_3 \rangle}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{PAIR}(\ell_1, \ell_2), \sigma_3 \{ \ell_1 \mapsto v_1, \ell_2 \mapsto v_2 \} \rangle} \quad (\text{CONS})$$

Primitives car and cdr observe cons cells.

$$\frac{\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{car}), \sigma_1 \rangle \quad \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{PAIR}(\ell_1, \ell_2), \sigma_2 \rangle}{\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle \sigma_2(\ell_1), \sigma_2 \rangle} \quad (\text{CAR})$$

$$\frac{\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{cdr}), \sigma_1 \rangle \quad \langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{PAIR}(\ell_1, \ell_2), \sigma_2 \rangle}{\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle \sigma_2(\ell_2), \sigma_2 \rangle} \quad (\text{CDR})$$

If the result of evaluating  $e_1$  is not a PAIR, rules CAR and CDR do not apply, and the abstract machine gets stuck. In such a case, my interpreter issues a run-time error message.

#### 2.11.4 Rules for evaluating definitions

A definition typically adds a new binding to the environment and changes the store. Its evaluation judgment is  $\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$ , which says that evaluating definition  $d$  in environment  $\rho$  with store  $\sigma$  produces a new environment  $\rho'$  and a new store  $\sigma'$ .

##### Global variables

When a global variable  $x$  is bound to an expression, what happens depends on whether  $x$  is already bound in  $\rho$ . If  $x$  is bound, then VAL is equivalent to SET: the environment is unchanged, but the store is updated.

$$\frac{x \in \text{dom } \rho \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{VAL}(x, e), \rho, \sigma \rangle \rightarrow \langle \rho, \sigma' \{ \rho(x) \mapsto v \} \rangle} \quad (\text{DEFINEOLDGLOBAL})$$

If  $x$  is *not* already bound, VAL allocates a fresh location  $\ell$ , extends the environment to bind  $x$  to  $\ell$ , evaluates the expression in the new environment, and stores the result in  $\ell$ :

$$\frac{x \notin \text{dom } \rho \quad \ell \notin \text{dom } \sigma \quad \langle e, \rho \{ x \mapsto \ell \}, \sigma \{ \ell \mapsto \text{unspecified} \} \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{VAL}(x, e), \rho, \sigma \rangle \rightarrow \langle \rho \{ x \mapsto \ell \}, \sigma' \{ \ell \mapsto v \} \rangle} \quad (\text{DEFINENEWGLOBAL})$$

The “unspecified” value stored in  $\ell$  effectively means that an adversary gets to look at your code and choose the least convenient value. Code that depends on an unspecified value invites disaster, or at least an unchecked run-time error.

Why does VAL add the binding to  $\rho$  before a value is available to store in  $\ell$ ? To enable a recursive function to refer to itself. To see the need, consider what would

happen in the following definition of factorial if the binding were not added until after the lambda expression was evaluated:

```
(val fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

If the binding isn't added before the closure is created, then when the body of the lambda expression is evaluated, it won't be able to call `fact`—because the environment saved in the closure won't contain a binding for `fact`. That's why the semantics for `VAL` in  $\mu$ Scheme is different from the semantics for `VAL` in Impcore.

### Top-level functions

`DEFINE` is syntactic sugar for a `VAL` binding to a `LAMBDA` expression.

$$\frac{\langle \text{VAL}(f, \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e)), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}{\langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle} \quad (\text{DEFINEFUNCTION})$$

### Top-level expressions

A top-level expression is syntactic sugar for a binding to the global variable `it`.

$$\frac{\langle \text{VAL}(\text{it}, e), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}{\langle \text{EXP}(e), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle} \quad (\text{EVALEXP})$$

## 2.12 THE INTERPRETER

$\mu$ Scheme's interpreter, like Impcore's interpreter, is decomposed into modules that have well-defined interfaces. Modules for names and printing are reused from Chapter 1, as is. Because Scheme's environments and values are different from Impcore's, environment and value modules are new. And the abstract-syntax module is generated automatically; its representations and code are derived from the data definition shown in Section 2.11.1.

### 2.12.1 Representation of values

The representation of values is generated from the data definition shown here. The empty list is called `NIL`, which is its name in Common Lisp. A function is represented by a `CLOSURE`, which is a `Lambda` with an environment; the `Lambda` structure contains the function's formal-parameter names and body.

**154a.** `<value.t 154a>` ≡

```
Lambda  = (Namelist formals, Exp body)
Value   = SYM  (Name)
        | NUM  (int32_t)
        | BOOLV (bool)
        | NIL
        | PAIR (Value *car, Value *cdr)
        | CLOSURE (Lambda lambda, Env env)
        | PRIMITIVE (int tag, Primitive *function)
```

**154b.** `<type definitions for μScheme 147b> +≡`

(S303d) ◁ 147b 155a ▷

```
typedef Value (Primitive)(Exp e, int tag, Valuelist vs);
```

Semantics	Concept	Interpreter	
$d$	Definition	Def (page 146)	
$e$	Expression	Exp (page 146)	
$\ell$	Location	Value *	
$x$	Name	Name (page 43)	
$v$	Value	Value (page 154)	
$\rho$	Environment	Env (page 155)	§2.12 The interpreter
$\sigma$	Store	Machine memory (the C heap)	
$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	Expression evaluation	$\text{eval}(e, \rho) = v$ , with $\sigma$ updated to $\sigma'$ (page 157)	155
$\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$	Definition evaluation	$\text{evaldef}(d, \rho, \text{echo}) = \rho'$ , with $\sigma$ updated to $\sigma'$ (page 161)	
$x \in \text{dom } \rho$	Definedness	$\text{find}(x, \rho) \neq \text{NULL}$ (page 155)	
$\rho(x)$	Location lookup	$\text{find}(x, \rho)$ (page 155)	
$\sigma(\rho(x))$	Value lookup	$*\text{find}(x, \rho)$ (page 155)	
$\rho\{x \mapsto \ell\}$	Binding	$\text{bindalloc}$ (page 155)	
$\ell \notin \text{dom } \sigma$	Allocation	$\text{bindalloc}$ (page 155)	
$\sigma\{\ell \mapsto v\}$	Store update	$*\ell = v$	

Table 2.6: Correspondence between  $\mu$ Scheme semantics and code

Why does a Primitive take so many arguments? Shouldn't a primitive function just take a Value list and return a Value? No. If a primitive fails, it needs to show where the failure occurred; that's the Exp  $e$ . And by using an integer tag, a single Primitive function can implement multiple  $\mu$ Scheme primitives. The C function that implements  $\mu$ Scheme's arithmetic primitives, for example, makes it easy for those primitives to share the code that ensures both arguments are numbers. Implementations of the primitives appear in Section 2.12.5 and in Appendix L.

### 2.12.2 Interfaces

#### The Environment and the Store

In the operational semantics, the store  $\sigma$  models the memory of the abstract machine. In the implementation, the store is represented by the memory of the real machine; a location is represented by a C pointer of type Value \*. An environment Env maps names to pointers;  $\text{find}(x, \rho)$  returns  $\rho(x)$  whenever  $x \in \text{dom } \rho$ ; when  $x \notin \text{dom } \rho$ , it returns NULL.

155a. *(type definitions for  $\mu$ Scheme 147b)* +≡  
 $\text{typedef struct Env } *Env;$

(S303d) ▷ 154b

155b. *(function prototypes for  $\mu$ Scheme 155b)* ≡  
 $\text{Value } *\text{find}(\text{Name name}, \text{Env env});$

(S303d) 155c▷

```

bindalloc,           S312c
  in μScheme S312c
  in μScheme (in
    GC?!)          S359a
bindalloclist,      S312d
  in μScheme S312d
  in μScheme (in
    GC?!)          S359b
type Exp   A          S312b
type Name  43b        43b
type Namelist      43b
type Value  A          S303c
type Valuelist      S303c

```

A location is allocated, initialized, and bound to a name in one step, by function bindalloc. Formally, when called with store  $\sigma$ ,  $\text{bindalloc}(x, v, \rho)$  chooses an  $\ell \notin \text{dom } \sigma$ , updates the store to be  $\sigma\{\ell \mapsto v\}$ , and returns the extended environment  $\rho\{x \mapsto \ell\}$ .

155c. *(function prototypes for  $\mu$ Scheme 155b)* +≡  
 $\text{Env bindalloc } (\text{Name name}, \text{Value v}, \text{Env env});$   
 $\text{Env bindalloclist}(\text{Namelist xs}, \text{Valuelist vs}, \text{Env env});$

(S303d) ▷ 155b 156a▷

*Programming Languages: Build, Prove, and Compare* © 2020 by Norman Ramsey.

Calling `bindalloclist`( $\langle x_1, \dots, x_n \rangle, \langle v_1, \dots, v_n \rangle, \rho$ ) does the same job for a list of values, returning  $\rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}$ , where  $\ell_1, \dots, \ell_n$  are fresh locations, which `bindalloclist` initializes to values  $v_1, \dots, v_n$ .

Although environments use a different interface from Impcore, their representation is unchanged. In both interpreters, an environment is represented using mutable state; that is, the environment associates each name with a pointer to a location that can be assigned to. Impcore's interface hides the use of locations, but  $\mu$ Scheme's interface exposes them. As a result, `find` can do the work of both `isvalbound` and `fetchval`.

2  
156

<i>Impcore</i>	<i><math>\mu</math>Scheme</i>
<code>isvalbound</code> ( $x, \rho$ )	$\text{find}(x, \rho) \neq \text{NULL}$
<code>fetchval</code> ( $x, \rho$ )	$*\text{find}(x, \rho)$

Using this interface, a  $\mu$ Scheme variable can be mutated without using a function like Impcore's `bindval`; code just assigns to the location returned by `find`. As long as  $x \in \text{dom } \rho$ , `bindval`( $x, v, \rho$ ) is replaced by  $*\text{find}(x, \rho) = v$ . Leveraging C pointers and mutable locations makes  $\mu$ Scheme's interface simpler than Impcore's.

### Allocation

The fresh locations created by `bindalloc` and `bindalloclist` come from `allocate`. Calling `allocate(v)` finds a location  $\ell \notin \text{dom } \sigma$ , stores  $v$  in  $\ell$  (thereby updating  $\sigma$ ), and returns  $\ell$ .

**156a.** *(function prototypes for  $\mu$ Scheme 155b)* +≡  
Value \*allocate(Value v);

(S303d) ◁ 155c 156b ▷

Allocation is described in great detail in Chapter 4.

### Values

Values are represented as specified by the data definition in chunk 154a. For convenience, values `#t` and `#f` are always available in C variables `truev` and `falsev`.

**156b.** *(function prototypes for  $\mu$ Scheme 155b)* +≡  
Value truev, falsev;

(S303d) ◁ 156a 156c ▷

Values can be tested for truth or falsehood by function `istru`; any value different from `#f` is regarded as true.

**156c.** *(function prototypes for  $\mu$ Scheme 155b)* +≡  
bool istru(Value v);

(S303d) ◁ 156b 156d ▷

When an unspecified value is called for by the semantics, one can be obtained by calling function `unspecified`.

**156d.** *(function prototypes for  $\mu$ Scheme 155b)* +≡  
Value unspecified(void);

(S303d) ◁ 156c 157a ▷

If you get the  $\mu$ Scheme interpreter to crash, your  $\mu$ Scheme code is probably looking at a value returned by `unspecified`. That's an unchecked run-time error.

### Evaluation

As in Impcore, the  $\rightarrow$  relation in the operational semantics is implemented by `evaldef`, and the  $\Downarrow$  relation is implemented by `eval`. The store  $\sigma$  is not passed or returned explicitly; it is represented by the C store, i.e., by the contents of memory. Because  $\sigma$  is *single-threaded*—every store is used exactly once and then discarded—the semantics can be implemented by updating memory in place.

%%	Print a percent sign
%d	Print an integer in decimal
%e	Print an Exp
%E	Print an Explist (list of Exp)
%\	Print a Lambda (in ASCII, the \ character is a common proxy for $\lambda$ )
%n	Print a Name
%N	Print a Namelist (list of Name)
%p	Print a Par
%P	Print a Parlist (list of Par)
%r	Print an Env
%s	Print a char* (string)
%t	Print a Def
%v	Print a Value
%V	Print a Valuelist (list of Value)

§2.12  
*The interpreter*  
157

Table 2.7: Specifications used in `print` and `fprint`

For example,  $\text{eval}(e, \rho)$ , when evaluated with store  $\sigma$ , finds a  $v$  and a  $\sigma'$  such that  $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$ , updates the store to be  $\sigma'$ , and returns  $v$ .

**157a.** *(function prototypes for  $\mu$ Scheme 155b)*  $\equiv$

(S303d)  $\triangleleft$  156d

```
Value eval (Exp e, Env rho);
Env evaldef(Def d, Env rho, Echo echo);
```

Similarly,  $\text{evaldef}(e, \rho, \text{echo})$ , when evaluated with store  $\sigma$ , finds a  $\rho'$  and a  $\sigma'$  such that  $\langle e, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$ , updates the store to be  $\sigma'$ , and returns  $\rho'$ . If `echo` is ECHOING, `evaldef` also prints the name or value of whatever expression is evaluated or added to  $\rho$ .

### Printing

Just like the Impcore interpreter, the  $\mu$ Scheme interpreter uses functions `print` and `fprint`, but the  $\mu$ Scheme interpreter knows how to print more kinds of things. The alternatives are shown in Table 2.7. Most of these specifications are used only to debug the interpreter.

#### 2.12.3 Implementation of the evaluator

As in Impcore, the evaluator starts with `switch`, which chooses how to evaluate `e` based on its syntactic form:

**157b.** *(eval.c 157b)*  $\equiv$

159c  $\triangleright$

```
Value eval(Exp e, Env env) {
    switch (e->alt) {
        case LITERAL: {evaluate e->literal and return the result 158a}
        case VAR: {evaluate e->var and return the result 158b}
        case SET: {evaluate e->set and return the result 158c}
        case IFX: {evaluate e->ifx and return the result 161b}
        case WHILEX: {evaluate e->whilex and return the result 161c}
        case BEGIN: {evaluate e->begin and return the result 161d}
        case APPLY: {evaluate e->apply and return the result 158e}
        case LETX: {evaluate e->letx and return the result 159d}
        case LAMBDA: {evaluate e->lambdax and return the result 158d}
    }
    assert(0);
}
```

allocate,	
in $\mu$ Scheme	164b
in $\mu$ Scheme (in	
GC?!)	
	270e
checkoverflow	
	S197a
type Def	$\mathcal{A}$
type Echo	S289b
type Env	155a
eval	229a
evaldef	161e
type Exp	$\mathcal{A}$
istrue	S318b
unspecified	S318c
type Value	$\mathcal{A}$

## Literals

As in Impcore, a literal evaluates to itself.

**158a.** *(evaluate e->literal and return the result 158a)≡  
return e->literal;*

(157b 48d)

## Scheme, S-expressions, and first-class functions

---

### Variables and assignment

Variable lookup and assignment are simpler than in Impcore, because  $\mu$ Scheme has only one rule for each form.

2  
158

$$\frac{x \in \text{dom } \rho \quad \rho(x) \in \text{dom } \sigma}{\langle \text{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle} \quad (\text{VAR})$$

$$\frac{x \in \text{dom } \rho \quad \rho(x) = \ell \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \{ \ell \mapsto v \} \rangle} \quad (\text{ASSIGN})$$

In the code,  $\rho(x)$  is implemented by `find(x, ρ)`, and  $\sigma(\ell)$  is implemented by `*ℓ`.

**158b.** *(evaluate e->var and return the result 158b)≡*

```
if (find(e->var, env) == NULL)
    runerror("name %n not found", e->var);
return *find(e->var, env);
```

(157b 48d)

In an assignment, the store is set to  $\sigma' \{ \ell \mapsto v \}$  by assigning to  $*\ell$ .

**158c.** *(evaluate e->set and return the result 158c)≡*

```
if (find(e->set.name, env) == NULL)
    runerror("set unbound variable %n in %e", e->set.name, e);
return *find(e->set.name, env) = eval(e->set.exp, env);
```

(157b 48d)

## Closures and function application

Wrapping a closure is simple:

$$\frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho \rangle, \sigma \rangle} \quad (\text{MKCLOSURE})$$

**158d.** *(evaluate e->lambdax and return the result 158d)≡*

```
return mkClosure(e->lambdax, env);
```

(157b)

Formal parameters  $x_1, \dots, x_n$  are confirmed to be distinct when  $e$  is parsed, by function `check_exp_duplicates` in chunk S326c. Checking at parse time exposes problems right away, without waiting for bad code to be evaluated.

When a function is applied, its actual parameters are evaluated and stored in `vs`. The next step depends on whether the function is a primitive or a closure.

**158e.** *(evaluate e->apply and return the result 158e)≡*

(157b 48d)

```
{
    Value      f  = eval  (e->apply.fn,           env);
    Valuelist vs = evallist(e->apply.actuals, env);

    switch (f.alt) {
        case PRIMITIVE:
            ⟨apply f.primitive to vs and return the result 159a⟩
        case CLOSURE:
            ⟨apply f.closure to vs and return the result 159b⟩
        default:
            runerror("%e evaluates to non-function %v in %e", e->apply.fn, f, e);
    }
}
```

Because a primitive is represented by a pair containing a function pointer and a tag, its application is simpler than in Impcore. Function `f.primitive.function` gets the tag, the arguments `vs`, and the abstract syntax `e`. (The syntax is used in error messages.)

**159a.** *(apply f.primitive to vs and return the result 159a)*  $\equiv$  (158e)  
`return f.primitive.function(e, f.primitive.tag, vs);`

A closure is applied by extending its stored environment ( $\rho_c$  in the operational semantics) with the bindings for the formal variables, then evaluating the body in that environment.

**§2.12**  
*The interpreter*

159

$$\begin{array}{c}
 \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\
 \langle e, \rho, \sigma \rangle \Downarrow \langle (\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \rho_c), \sigma_0 \rangle, \sigma_0 \rangle \\
 \quad \quad \quad \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\
 \quad \quad \quad \vdots \\
 \quad \quad \quad \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\
 \hline
 \langle e_c, \rho_c \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma_n \{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma' \rangle \\
 \langle \text{APPLY}(e, e_1, \dots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \\
 \qquad \qquad \qquad \text{(APPLYCLOSURE)}
 \end{array}$$

**159b.** *(apply f.closure to vs and return the result 159b)*  $\equiv$  (158e)

```
{
    Namelist xs = f.closure.lambda.formals;
    checkargc(e, lengthNL(xs), lengthVL(vs));
    return eval(f.closure.lambda.body,
                bindalloclist(xs, vs, f.closure.env));
}
```

As in Impcore's interpreter, `evallist` evaluates a list of arguments in turn, returning a list of values.

**159c.** *(eval.c 157b) +≡* △157b

```

static Valuelist evallist(Explist es, Env env) {
    if (es == NULL) {
        return NULL;
    } else {
        Value v = eval(es->hd, env); // enforce uScheme's order of evaluation
        return mkVL(v, evallist(es->tl, env));
    }
}
```

*Let, let\*, and letrec*

Each expression in the `let` family uses its internal name-expression pairs to create a new environment, then evaluates the body in that environment. Each form creates the new environment in a different way.

**159d.** *(evaluate e->letx and return the result 159d)*  $\equiv$  (157b)

```

switch (e->letx.let) {
    case LET:    <extend env by simultaneously binding es to xs 160a>
        break;
    case LETSTAR: <extend env by sequentially binding es to xs 160b>
        break;
    case LETREC: <extend env by recursively binding es to xs 161a>
        break;
    default:    assert(0);
}
return eval(e->letx.body, env);
```

checkargc	48b
type Env	155a
env	157b
eval	157a
evallist	S305b
type Explist	S303b
find	155b
lengthNL	A
lengthVL	A
mkClosure	A
mkVL	A
type Namelist	
	43b
runerror	47
type Value	A
type Valuelist	
	S303c

As with `lambda`, all names are confirmed to be distinct at parse time.

A LET expression evaluates its right-hand sides, then binds them all at once. All the work is done by functions `evalist` and `bindallocist`.

$$\frac{\begin{array}{c} x_1, \dots, x_n \text{ all distinct} \\ \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \\ (\text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma) \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle v, \sigma' \rangle} \quad (\text{LET})$$

**160a.** *extend env by simultaneously binding es to xs* 160a  $\equiv$  (159d)  
`env = bindallocist(e->letx.xs, evalist(e->letx.es, env), env);`

A LETSTAR expression binds a new name as each expression is evaluated.

$$\frac{\begin{array}{c} \rho_0 = \rho \quad \sigma_0 = \sigma \\ \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} \quad \sigma_1 = \sigma'_0\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma'_{n-1} \rangle \\ \ell_n \notin \text{dom } \sigma'_{n-1} \quad \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} \quad \sigma_n = \sigma'_{n-1}\{\ell_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma) \Downarrow \langle v, \sigma' \rangle} \quad (\text{LETSTAR})$$

**160b.** *extend env by sequentially binding es to xs* 160b  $\equiv$  (159d)

```

{
  Namelist xs;
  Explist es;

  for (xs = e->letx.xs, es = e->letx.es;
       xs && es;
       xs = xs->tl, es = es->tl)
    env = bindalloc(xs->hd, eval(es->hd, env), env);
    assert(xs == NULL && es == NULL);
}
  
```

Finally, before evaluating any expressions, LETREC binds each name to a fresh location.

$$\frac{\begin{array}{c} \ell_1, \dots, \ell_n \notin \text{dom } \sigma \text{ (and all distinct)} \\ x_1, \dots, x_n \text{ all distinct} \\ e_i \text{ has the form LAMBDA}(\dots), 1 \leq i \leq n \\ \rho' = \rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\ \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \dots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \\ (\text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma) \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle v, \sigma' \rangle} \quad (\text{LETREC})$$

The locations' initial contents are unspecified, and they remain unspecified until all the values are computed. The right-hand sides are confirmed to be LAMBDA's at parse time, by the same function that confirms the  $x_i$ 's are distinct.

**161a.** *(extend env by recursively binding es to xs 161a) ≡*

(159d)

```

{
    Namelist xs;

    for (xs = e->letx.xs; xs; xs = xs->t1)
        env = bindalloc(xs->hd, unspecified(), env);
    Valuelist vs = evalist(e->letx.es, env);
    for (xs = e->letx.xs;
        xs && vs;
        xs = xs->t1, vs = vs->t1)
        *find(xs->hd, env) = vs->hd;
    assert(xs == NULL && vs == NULL);
}

```

§2.12

The interpreter

161

*Conditional, iteration, and sequence*

The control-flow operations are implemented much as they are in Impcore. The semantic rules are not worth repeating.

**161b.** *(evaluate e->ifx and return the result 161b) ≡*

(157b 48d)

```

if (istru(e->ifx.cond, env))
    return eval(e->ifx.true, env);
else
    return eval(e->ifx.false, env);

```

**161c.** *(evaluate e->whilex and return the result 161c) ≡*

(157b 48d)

```

while (istru(e->whilex.cond, env))
    eval(e->whilex.body, env);
return falsev;

```

**161d.** *(evaluate e->begin and return the result 161d) ≡*

(157b 48d)

```

{
    Value lastval = falsev;
    for (Explist es = e->begin; es; es = es->t1)
        lastval = eval(es->hd, env);
    return lastval;
}

```

#### 2.12.4 Evaluating true definitions

Each true definition is evaluated by function evaldef, which updates the store and returns a new environment. If echo is ECHOES, evaldef also prints. Function evaldef doesn't handle record definitions; the record form is syntactic sugar, not a true definition.

**161e.** *(evaldef.c 161e) ≡*

```

Env evaldef(Def d, Env env, Echo echo) {
    switch (d->alt) {
        case VAL:   (evaluate val binding and return new environment 162a)
        case EXP:   (evaluate expression, assign to it, and return new environment 162b)
        case DEFINE: (evaluate function definition and return new environment 162c)
    }
    assert(0);
}

```

bindalloc	155c
bindallocist	155c
type Def	A
type Echo	S289b
type Env	155a
env	157b
eval	157a
evallist	S305b
type Explist	S303b
falsev	156b
find	155b
type Namelist	43b
unspecified	156d
type Value	A
type Valuelist	S303c

According to the operational semantics, the right-hand side of a val binding must be evaluated in an environment in which the name d->val.name is bound.

If no binding is present, one is added, with an unspecified value.

$$\frac{x \in \text{dom } \rho \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{VAL}(x, e), \rho, \sigma \rangle \rightarrow \langle \rho, \sigma' \{ \rho(x) \mapsto v \} \rangle} \quad (\text{DEFINEOLDGLOBAL})$$

*Scheme,  
S-expressions, and  
first-class functions*

2  
162

$$\frac{x \notin \text{dom } \rho \quad \ell \notin \text{dom } \sigma \quad \langle e, \rho \{ x \mapsto \ell \}, \sigma \{ \ell \mapsto \text{unspecified} \} \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{VAL}(x, e), \rho, \sigma \rangle \rightarrow \langle \rho \{ x \mapsto \ell \}, \sigma' \{ \ell \mapsto v \} \rangle} \quad (\text{DEFINENEWGLOBAL})$$

**162a.** *(evaluate val binding and return new environment 162a)*  $\equiv$  (161e)

```
{
    if (find(d->val.name, env) == NULL)
        env = bindalloc(d->val.name, unspecified(), env);
    Value v = eval(d->val.exp, env);
    *find(d->val.name, env) = v;
    ⟨if echo calls for printing, print either v or the bound name S305c⟩
    return env;
}
```

As in Impcore, evaluating a top-level expression has the same effect on the environment as evaluating a definition of it, except that the interpreter always prints the value, never the name “it.”

**162b.** *(evaluate expression, assign to it, and return new environment 162b)*  $\equiv$  (161e)

```
{
    Value v = eval(d->exp, env);
    Value *itloc = find(strtoname("it"), env);
    ⟨if echo calls for printing, print v S305d⟩
    if (itloc == NULL) {
        return bindalloc(strtoname("it"), v, env);
    } else {
        *itloc = v;
        return env;
    }
}
```

A DEFINE is rewritten to VAL.

**162c.** *(evaluate function definition and return new environment 162c)*  $\equiv$  (161e)

```
return evaldef(mkVal(d->define.name, mkLambdax(d->define.lambda)),
              env, echo);
```

### 2.12.5 Implementations of the primitives

Each primitive is associated with a unique tag, which identifies the primitive, and with a function, which implements the primitive. The tags enable one function to implement multiple primitives, which makes it easy for similar primitives to share code. The primitives are implemented by these functions:

arith	Arithmetic primitives, which expect integers as arguments
binary	Non-arithmetic primitives that expect two arguments
unary	Primitives that expect one argument

Each arithmetic primitive expects two integer arguments, which are obtained by projecting  $\mu$ Scheme values. The projection function `projectint32` takes not only a value but also an expression, so if its argument is not an integer, it can issue an informative error message.

**163a.** *(prim.c 163a)* $\equiv$ 

```
static int32_t projectint32(Exp e, Value v) {
    if (v.alt != NUM)
        runerror("in %e, expected an integer, but got %v", e, v);
    return v.num;
}
```

163b▷

§2.12

The interpreter

163

Function `arith` first converts its arguments to integers, then consults the tag to decide what to do. In each case, it computes a number or a Boolean, which is converted a  $\mu$ Scheme value by either `mkNum` or `mkBool`, both of which are generated automatically from the definition of `Value` in code chunk 154a. Checks for arithmetic overflow are not shown.

**163b.** *(prim.c 163a)* $+ \equiv$ 

```
Value arith(Exp e, int tag, Valuelist args) {
    checkargc(e, 2, lengthVL(args));
    int32_t n = projectint32(e, nthVL(args, 0));
    int32_t m = projectint32(e, nthVL(args, 1));

    switch (tag) {
        case PLUS: return mkNum(n + m);
        case MINUS: return mkNum(n - m);
        case TIMES: return mkNum(n * m);
        case DIV:   if (m==0) runerror("division by zero");
                    else return mkNum(divide(n, m)); // round to minus infinity
        case LT:   return mkBoolv(n < m);
        case GT:   return mkBoolv(n > m);
        default: assert(0);
    }
}
```

163a 163c▷

### Other binary primitives

$\mu$ Scheme has two other binary primitives, which don't require integer arguments: `cons` and `=`. The implementation of `=` is relegated to the Supplement, but the implementation of `cons` is shown here. Because S-expressions are a recursive type, a `cons` cell must contain pointers to S-expressions, not S-expressions themselves. Every `cons` must therefore allocate fresh locations for the pointers. This behavior makes `cons` a major source of allocation in  $\mu$ Scheme programs.<sup>14</sup>

**163c.** *(prim.c 163a)* $+ \equiv$ 

```
Value cons(Value v, Value w) {
    return mkPair(allocate(v), allocate(w));
}
```

163b 164a▷

allocate	156a
bindalloc	155c
checkargc	48b
checkarith	S198b
echo	161e
env	161e
eval	157a
evaldef	157a
type Exp	A
find	155b
lengthVL	A
mkLambdax	A
mkNum	A
mkPair	A
mkVal	A
nthVL	A
runerror	47
strtoname	43c
unspecified	156d
type Value	A
type Valuelist	

S303c

<sup>14</sup>In full Scheme, a `cons` cell is typically represented by a pointer to an object allocated on the heap, so `cons` requires only one allocation, not two.

Unary primitives are implemented here. Most of the cases are relegated to the Supplement.

**164a.** *⟨prim.c 163a⟩* +≡

△163c

```

Value unary(Exp e, int tag, Valuelist args) {
    checkargc(e, 1, lengthVL(args));
    Value v = nthVL(args, 0);
    switch (tag) {
        case NULLP:
            return mkBoolv(v.alt == NIL);
        case CAR:
            if (v.alt == NIL)
                runerror("in %e, car applied to empty list", e);
            else if (v.alt != PAIR)
                runerror("car applied to non-pair %v in %e", v, e);
            return *v.pair.car;
        case PRINTU:
            if (v.alt != NUM)
                runerror("printu applied to non-number %v in %e", v, e);
            print_utf8(v.num);
            return v;
        case ERROR:
            runerror("%v", v);
            return v;
        <other cases for unary primitives S308c>
        default:
            assert(0);
    }
}

```

### 2.12.6 Implementation of memory allocation

In this chapter, a new location is allocated with `malloc`.

**164b.** *⟨loc.c 164b⟩* ≡

```

Value* allocate(Value v) {
    Value *loc = malloc(sizeof(*loc));
    assert(loc != NULL);
    *loc = v;
    return loc;
}

```

A much more interesting and efficient allocator is described in Chapter 4.

## 2.13 EXTENDING $\mu$ SCHEME WITH SYNTACTIC SUGAR

Like Impcore,  $\mu$ Scheme is stratified into two layers: a *core language* and *syntactic sugar* (page 68). The core language is defined by the operational semantics and is implemented by functions `eval` and `evaldef`. The syntactic sugar is defined and implemented by translating it into the core language. In Scheme, the core language can be very small indeed: even the `LET` and `BEGIN` forms can be implemented as syntactic sugar. (But in  $\mu$ Scheme, they are part of the core.) The translations of `LET` and `BEGIN` are shown below, as are the translations used to implement short-circuit conditionals, `cond`, and the record form. These translations introduce two key programming-language concepts: *capture-avoiding substitution* and *hygiene*.

### 2.13.1 Syntactic sugar for LET forms

A let expression can be desugared into an application of a lambda:

$$(\text{let} ([x_1 e_1] \cdots [x_n e_n]) e) \triangleq ((\text{lambda} (x_1 \cdots x_n) e) e_1 \cdots e_n).$$

A let\* expression can be desugared into a sequence of nested let expressions. The desugaring is defined by structural induction on the list of bindings in the let\*, which calls for two equations: one for the base case (no bindings) and one for the induction step (a nonempty sequence of bindings). An empty let\* is desugared into its body. A nonempty let\* is desugared into a nested sequence of let expressions: a let expression for the first binding, followed by the desugaring of the let\* expression for the remaining bindings.

$$\begin{aligned} (\text{let*} () & e) \triangleq e \\ (\text{let*} ([x_1 e_1] \cdots [x_n e_n]) e) & \triangleq \\ & (\text{let} ([x_1 e_1]) (\text{let*} ([x_2 e_2] \cdots [x_n e_n]) e)) \end{aligned}$$

This translation works just like any other recursive function—but the recursive function is applied to *syntax*, not to values. It looks like this:

**165. *parse.c* 165** ≡

```
Exp desugarLetStar(Namelist xs, Explist es, Exp body) {
    if (xs == NULL || es == NULL) {
        assert(xs == NULL && es == NULL);
        return body;
    } else {
        return desugarLet(mkNL(xs->hd, NULL), mkEL(es->hd, NULL),
                           desugarLetStar(xs->tl, es->tl, body));
    }
}
```

The desugared code works just as well as  $\mu$ Scheme’s core code—and you can prove it (Exercises 44 and 45).

Finally, a letrec can be desugared into a let expression that introduces all the variables, which is followed by a sequence of assignments:

$$\begin{aligned} (\text{letrec} ([x_1 e_1] \cdots [x_n e_n]) e) & \triangleq \\ & (\text{let} ([x_1 \text{unspecified}] \cdots [x_n \text{unspecified}]) \\ & \quad (\text{begin} (\text{set} x_1 e_1) \cdots (\text{set} x_n e_n) \\ & \quad e)) \end{aligned}$$

This translation works only when each  $e_i$  is a lambda expression, as required by the operational semantics.

### 2.13.2 Syntactic sugar for Lisp’s original conditional form

$\mu$ Scheme’s conditional expression, written using if, allows for only two alternatives. But real programs often choose among three or more alternatives. For some such choices, C and the Algol-like languages offer a switch statement, but it can choose only among integer values that are known at compile time—typically enumeration literals. A more flexible multi-way choice is offered by a syntactic form from McCarthy’s original Lisp: the cond expression. A cond expression contains an arbitrarily long sequence of question-answer pairs: one for each choice. In my opin-

checkargc	48b
type Exp	$\mathcal{A}$
type Explist	S303b
lengthVL	$\mathcal{A}$
type Namelist	
	43b
nthVL	$\mathcal{A}$
runerror	47
type Value	$\mathcal{A}$
type Valuelist	
	S303c

ion, cond is better than if, because cond makes it obvious how many alternatives there are and what each one is doing.

**166a.** *(transcript for extended μScheme 166a)*  $\equiv$

166b ▷

```
-> (define compare-numbers (n m)
  (cond
    [(< n m) 'less]
    [(= n m) 'equal]
    [(> n m) 'greater]))
-> (compare-numbers 3 2)
greater
-> (compare-numbers 3 3)
equal
-> (compare-numbers 3 4)
less
```

As another example, cond can be used to implement remove-multiples, from chunk 104b:

**166b.** *(transcript for extended μScheme 166a)*  $+ \equiv$

△ 166a 169a ▷

```
-> (define divides? (p n) (= (mod n p) 0))
-> (define remove-multiples (p ns)
  (cond [(null? ns) '()]
        [((divides? p (car ns)) (remove-multiples p (cdr ns)))
         [#t (cons (car ns) (remove-multiples p (cdr ns))))])
-> (remove-multiples 3 '(1 2 3 4 5 6))
(1 2 4 5)
```

Although cond is superior, it is found in very few languages—almost exclusively the Lisp-like languages—and if is found everywhere. For μScheme, therefore, I chose the familiar if over the superior cond. Fortunately, now that you know about cond, you can use it—it's syntactic sugar:

$$\begin{aligned} (\text{cond}) &\stackrel{\Delta}{=} (\text{error } 'cond:-all-question-results-were-false) \\ (\text{cond } [e_q\ e_a] \dots) &\stackrel{\Delta}{=} (\text{if } e_q\ e_a\ (\text{cond } \dots)) \end{aligned}$$

### 2.13.3 Syntactic sugar for conditional operators: avoiding variable capture

As a replacement for the primitive function and, which can be called only after both its arguments are evaluated, μScheme provides a syntactic form &&, which evaluates its second argument only when necessary. And actually, && can accept more than two arguments; it is desugared to if expressions as follows:

$$\begin{aligned} (&& e) &\stackrel{\Delta}{=} e \\ (&& e_1 \dots e_n) &\stackrel{\Delta}{=} (\text{if } e_1\ (&& e_2 \dots e_n)\ \#\text{f}) \end{aligned}$$

μScheme also provides a || form, which is also desugared into if expressions, but there's a challenge. The following desugaring doesn't always work:

$$(\text{|| } e_1\ e_2) \stackrel{\Delta}{=} (\text{if } e_1\ \#\text{t}\ e_2)$$

The problem with that right-hand side is that if  $e_1$  is not #f,  $(\text{|| } e_1\ e_2)$  should return the value of  $e_1$ , just as the predefined function or does. But the desugaring into if returns #t:

**166c.** *(transcript 97a)*  $+ \equiv$

△ 145d 167a ▷

```
-> (or 7 'seven)
7
-> (if 7 \#t 'seven)
\#t
```

When  $e_1$  is not `#f`, a desugaring could return  $e_1$ , but this desugaring doesn't always work either:

$$(\parallel e_1 e_2) \stackrel{\Delta}{=} (\text{if } e_1 e_1 e_2)$$

This one fails because it could evaluate  $e_1$  twice. And if  $e_1$  has a side effect, the desugaring performs the side effect twice, leading to wrong answers.

**167a.** *(transcript 97a)*  $\equiv$

$\triangleleft 166c$   $167b \triangleright$

```
-> (val n 2)
-> (or (< 0 (set n (- n 1))) 'finished)
#t
-> (val n 2)
-> (if (< 0 (set n (- n 1))) (< 0 (set n (- n 1))) 'finished)
#f
```

$\mu$ Scheme with  
syntactic sugar

167

The `or` function works because it is a function call: both  $e_1$  and  $e_2$  are evaluated, and their results are bound to the formal parameters of the `or` function. A desugaring could achieve the almost same effect with a `let` binding:

$$(\parallel e_1 e_2) \text{ is almost } (\text{let } ([x e_1]) (\text{if } x x e_2))$$

This one works with the examples given so far:

**167b.** *(transcript 97a)*  $\equiv$

$\triangleleft 167a$   $167c \triangleright$

```
-> (val n 2)
-> (or (< 0 (set n (- n 1))) 'finished)
#t
-> (val n 2)
-> (let ([x (< 0 (set n (- n 1)))] (if x x 'finished))
#t
```

But binding  $x$  doesn't always work; if  $x$  is used in  $e_2$ , the desugaring can fail:

**167c.** *(transcript 97a)*  $\equiv$

$\triangleleft 167b$   $187b \triangleright$

```
-> (val n 0)
-> (val x 'finished)
-> (or (< 0 n) x)
finished
-> (val n 0)
-> (val x 'finished)
-> (let ([x (< 0 n)] (if x x x))
#f
```

This failure has a name: the global variable  $x$  is said to be *captured* by the desugaring. To avoid capturing  $x$ , it is sufficient to choose *some* name  $x$  that doesn't appear in  $e_2$ . Such an  $x$  is called *fresh*. So finally, the following desugaring always works:

$$(\parallel e_1 e_2) \stackrel{\Delta}{=} (\text{let } ([x e_1]) (\text{if } x x e_2)), \text{ provided } x \text{ does not appear in } e_2$$

Using this idea, the general rules for desugaring  $\parallel$  are as follows:

$$\begin{aligned} (\parallel e) &\stackrel{\Delta}{=} e \\ (\parallel e_1 \cdots e_n) &\stackrel{\Delta}{=} (\text{let } ([x e_1]) (\text{if } x x (\parallel e_2 \cdots e_n))), \\ &\quad \text{where } x \text{ does not appear in any } e_i \end{aligned}$$

car	$\mathcal{P}$ 164a
cdr	$\mathcal{P}$ 164a
cons	$\mathcal{P}$ S307e
null?	$\mathcal{P}$ 164a

Variable capture and fresh names are perennial issues in programming languages.

#### 2.13.4 Making syntactic sugar precise: hygienic substitution

Desugaring replaces one expression with another. To define replacement precisely, programming-language theorists have developed the concept of *substitution*.

- Substitution is the computational engine driving the simplest, most foundational accounts of what it means to evaluate expressions (or run programs). If you ever study the lambda calculus, you'll see that substitution is its only computational mechanism.
- Substitution is used to check the types of polymorphic functions.  $\mu$ Scheme's functions aren't typechecked, and C's functions aren't polymorphic, but many programming languages do support both polymorphism and type checking (Chapter 6). And when a polymorphic function is used, the correctness of the use is checked using substitution. Such substitutions must avoid capturing *type variables* (Section 6.6.7). Even when types are inferred, as in ML and Haskell they are inferred using substitution (Chapter 7).
- In Prolog, when the system is trying to answer a query or prove a goal, subgoals are spawned using substitution.

Substitution algorithms that avoid variable capture are called *hygienic*. They are hard to get right. But if you are interested in programming-language foundations, you need to understand them. And if you are interested in language design, you need to understand that using a language based on substitution is not fun. (As examples, I submit TeX and Tcl.) The more different contexts in which you see substitution, the better you will understand it.

Let's use substitution to take a second look at  $\mu$ Scheme's conditional expressions. The expression  $(\&\& e_1 e_2)$  is desugared into an *if* expression in three steps:

1. The replacement will be derived from the *template expression* (*if*  $\square_1 \square_2 \#f$ ). Mathematically, the symbols  $\square_1$  and  $\square_2$  are ordinary program variables; the  $\square$  notation telegraphs an intention to substitute for them.
2. Subexpressions  $e_1$  and  $e_2$  are extracted from the  $\&\&$  expression.
3. In the template,  $e_1$  is substituted for  $\square_1$  and  $e_2$  is substituted for  $\square_2$ , simultaneously. The resulting expression is an *instance* of the template—"instance" being the word for a thing obtained by substitution. The original  $\&\&$  expression is replaced by the instance of the template, which is (*if*  $e_1 e_2 \#f$ ).

The original expression and its replacement have the same semantics:

- If the original  $\&\&$  expression is evaluated in environment  $\rho$ , both  $e_1$  and  $e_2$  are supposed to be evaluated in environment  $\rho$ . In addition, expression  $e_2$  is supposed to be evaluated only if  $e_1$  evaluates to  $\#f$ .
- If the instantiated template (the *if* expression) is evaluated in environment  $\rho$ , the semantics confirms that both  $e_1$  and  $e_2$  are evaluated in environment  $\rho$ . In addition, expression  $e_2$  is evaluated only if  $e_1$  evaluates to  $\#f$ .

The  $\|$  expression can't be desugared so easily.

- If the original  $\|$  expression is evaluated in environment  $\rho$ , both  $e_1$  and  $e_2$  are supposed to be evaluated in environment  $\rho$ .
- If the template for  $\|$  is

```
(let ([x  $\square_1$ ]) (if x x  $\square_2$ ))
```

and the instance is evaluated in environment  $\rho$ , then  $e_1$  is evaluated in environment  $\rho$ , but if  $e_2$  is evaluated, it is evaluated in environment  $\rho\{x \mapsto v_1\}$ , where  $v_1$  is the value of  $e_1$ .

Using this template,  $e_2$  is evaluated in an extended environment, and variable  $x$  can be captured. For example, if  $\langle e_1, \rho, \sigma \rangle \Downarrow \langle \text{BOOLV}(\#f), \sigma' \rangle$  and if  $e_2$  is  $x$ , the value of the  $\|$  expression should be  $\sigma'(\rho(x))$ . But the value of the instantiated template is always  $\text{BOOLV}(\#f)$ . If, however,  $x$  is not mentioned in  $e_2$ , then  $x$  is not captured by the substitution, and it is a theorem that

If  $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle$ , if  $x$  is not mentioned in  $e$ , and if  $\ell \notin \text{dom } \sigma$ , then  
 $\langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto v'\} \rangle \Downarrow \langle v, \sigma \rangle$ .

This theorem follows either from Exercise 52 in this chapter or from Exercise 9 in Chapter 5.

Therefore,  $\|$  can be desugared correctly if we choose a good  $x$ . Instead of a single template for  $\|$ , think of a whole *family* of templates

$\{( \text{let} ([x \square_1]) (\text{if } x x \square_2) ) \mid x \text{ is a variable}\}$ .

Every choice of  $x$  determines a template, and any given  $e_1$  and  $e_2$  can be desugared using any template in which  $x$  does not appear in  $e_2$ .

### 2.13.5 Syntactic sugar for BEGIN

The same idea of hygiene—choosing a variable in a template that does not interfere with what is substituted—is used in the rules for desugaring BEGIN:

$$\begin{aligned} (\text{begin } e) &\stackrel{\triangle}{=} e \\ (\text{begin } e_1 \dots e_n) &\stackrel{\triangle}{=} (\text{let} ([x \ e_1]) (\text{begin } e_2 \dots e_n)), \\ &\quad \text{where } x \text{ does not appear in any } e_i \end{aligned}$$

### 2.13.6 Syntactic sugar for record definitions

A Scheme record contains a fixed collection of named fields. In modern implementations of full Scheme, records are provided natively, as a primitive data structure. In  $\mu$ Scheme, records are simulated using cons cells. The record-definition form in Section 2.4.1 is desugared into a sequence of function definitions:

**169a.** *(transcript for extended  $\mu$ Scheme 166a)*  $\equiv$   $\triangleleft$  166b 169b  $\triangleright$   
 $\rightarrow (\text{record frozen-dinner [protein starch vegetable dessert]})$   
 $\text{make-frozen-dinner}$   
 $\text{frozen-dinner?}$   
 $\text{frozen-dinner-protein}$   
 $\text{frozen-dinner-starch}$   
 $\text{frozen-dinner-vegetable}$   
 $\text{frozen-dinner-dessert}$

A `frozen-dinner` is represented by a list with five elements: the symbol `make-frozen-dinner`, followed by the values of the fields. This representation, when printed, resembles code that might be executed to reconstruct the record.

**169b.** *(transcript for extended  $\mu$ Scheme 166a)*  $\equiv$   $\triangleleft$  169a  $\triangleright$   
 $\rightarrow (\text{make-frozen-dinner 'steak 'potato 'green-beans 'pie})$   
 $(\text{make-frozen-dinner steak potato green-beans pie})$

Using this representation, the record definition is desugared into a sequence of definitions that looks something like this:

**169c.** *(selected equivalent definitions for record frozen-dinner 169c)*  $\equiv$  170  $\triangleright$   
 $(\text{define make-frozen-dinner } (x \ y \ z \ w))$   
 $\quad (\text{cons 'make-frozen-dinner } (\text{cons } x (\text{cons } y (\text{cons } z (\text{cons } w '())))))$   
 $\quad ; \dots \text{ continued next page} \dots$

**170.** *(selected equivalent definitions for record frozen-dinner 169c) +≡*

```
(define frozen-dinner? (v)
  (&& (pair? v)
    (= 'make-frozen-dinner (car v))
    (pair? (cdr v))
    (pair? (cddr v))
    (pair? (cdddr v))
    (pair? (cdr (cdddr v)))
    (null? (cddr (cdddr v)))))

(define frozen-dinner-protein (r)
  (if (frozen-dinner? r)
    (car (cdr r))
    (error (list2 r 'is-not-a-frozen-dinner-record)))))

(define frozen-dinner-starch (r)
  (if (frozen-dinner? r)
    (car (cdr (cdr r)))
    (error (list2 r 'is-not-a-frozen-dinner-record)))))
```

In general, a record definition with record name  $r$  and field names  $f_1, \dots, f_n$  is desugared as follows:<sup>15</sup>

$$\begin{aligned} (\text{record } r (f_1 \dots f_n)) &\stackrel{\Delta}{=} \\ &\text{(define make-}r\text{ (x}_1 \dots x_n\text{)} \\ &\quad (\text{cons 'make-}r\text{ (cons x}_1\text{ (cons \dots (cons x}_n\text{ '()) \dots)))}) \\ &\text{(define r? (x) (&& (pair? x) (= (car x) 'make-}r\text{) \dots)))} \\ &\text{(define r-}f_1\text{ (x) (if (r? x) (car (cdr x)) (error \dots)))} \\ &\text{(define r-}f_2\text{ (x) (if (r? x) (car (cdr (cdr x))) (error \dots)))} \\ &\vdots \\ &\text{(define r-}f_n\text{ (x) (if (r? x)} \\ &\quad (\text{car (cdr (cdr \dots (cdr x))))}) \\ &\quad (\text{error \dots}))}) \end{aligned}$$

In `make-r`, hygiene requires that  $x_1, \dots, x_n$  be mutually distinct and different from `cons`; otherwise `cons` could mistakenly refer to one of the arguments. Hygiene is discussed more deeply in Section 2.14.4.

## 2.14 SCHEME AS IT REALLY IS

Like Lisp before it, Scheme has split into dialects. The model for  $\mu$ Scheme is the 1998 R<sup>5</sup>RS standard, which embodies the minimalist design philosophy of the original Scheme. The 2007 R<sup>6</sup>RS standard defines a bigger, more complicated Scheme, which most R<sup>5</sup>RS implementations chose not to adopt. The subsequent R<sup>7</sup>RS standard splits Scheme into two languages: a small one, finalized in 2013, that more closely resembles the original, and a big one, still incomplete as of 2020, that is believed to be better suited to mainstream software development. And while standards have their advantages, many Schemers prefer Racket, a nonstandard dialect that benefits from a talented team of developers and contributors. But most dialects share aspects that I consider interesting, impressive, or relevant for someone making a transition from  $\mu$ Scheme—which is the topic of this section.

### 2.14.1 Language differences

Let's begin with some minor lexical and syntactic differences. Through 2007, identifiers and symbols in full Scheme were not case-sensitive; for example, '`Foo`' was

<sup>15</sup>The real story is more complicated: instead of using `names`, `cons`, `pair?`, `car`, and so on, the desugaring creates *literals* that refer directly to those primitives.

the same as 'foo. But as of the 2007 R<sup>6</sup>RS standard—to give it its full name, the Revised<sup>6</sup> Report on Scheme (Sperber et al. 2009)—identifiers and symbols are case-sensitive, as in  $\mu$ Scheme.

Full Scheme uses `define` to introduce all top-level bindings, with slightly different concrete syntax from  $\mu$ Scheme:

```
(define var exp)
(define (fun args) exp)
```

The `define` form can also be used within the body of a `let` or a `lambda`. Using definition forms within `let` and `lambda` makes programs easier to read and refactor than in  $\mu$ Scheme, and it's cheap: these forms are desugared to a new `letrec*` form, which uses a single environment like `letrec` but is evaluated sequentially like `let*`.

Full Scheme operates on association lists with a function called `assoc`, not with `find` and `bind`. (In full Scheme, `find` is a function that takes a predicate and a list and returns the first element of the list that satisfies the predicate.) Calling `(assoc obj alist)` finds the first pair in `alist` whose `car` field is `equal?` to `obj`, and it returns that pair. If no pair in `alist` has `obj` as its `car`, `assoc` returns `#f`.

**171a.** *(R<sup>6</sup>RS Scheme transcript 171a)≡* 171b▷  
> (define e '((a 1) (b 2) (c 3))) ;; r6rs Scheme implemented in Racket,  
> (assoc 'a e) ;; not  $\mu$ Scheme!  
'(a 1)  
> (assoc 'b e)  
'(b 2)  
> (assoc 'd e)  
#f  
> (assoc (list 'a) '(((a)) ((b)) ((c))))  
'((a))

A result from `assoc` can be tested directly in an `if` expression, and if not `#f`, it can be updated in place by primitive function `set-cdr!`.

Full Scheme includes the other list functions found in  $\mu$ Scheme, but often under slightly different names, such as `for-all`, `exists`, `fold-left`, and `fold-right`. And the full Scheme functions are more general: they can operate on any number of lists simultaneously.

Full Scheme has an additional quoting mechanism: using *quasiquotation*, you can splice computed values or lists into quoted S-expressions (Exercise 55).

In full Scheme, a function can take a variable number of arguments. Either the function takes one formal parameter, which is the whole list of arguments, or a formal parameter separated by a trailing period is bound to any “extra” arguments:

**171b.** *(R<sup>6</sup>RS Scheme transcript 171a)+≡* △ 171a  
> ((lambda (x y . zs) zs) 3 4 5 6) ;; Racket's r6rs Scheme  
'(5 6)  
> ((lambda xs xs) 3 4 5 6)  
'(3 4 5 6)

In addition, many primitive functions and macros, such as `+`, `<`, and `max`, etc., accept an arbitrary number of arguments—even zero.

In full Scheme, the syntactic form `set` is called `set!`. And full Scheme can mutate the contents of cons cells, not just variables, using primitive functions `set-car!` and `set-cdr!` (Exercises 50 and 58).

Finally, in full Scheme, the order in which expressions are evaluated is usually unspecified. To enforce a particular order of evaluation, use `let*` or `letrec*`.

### 2.14.2 Proper tail calls (upcoming in Chapter 3)

Implementations of full Scheme must be *properly tail-recursive*. Informally, an implementation is properly tail-recursive if every *tail call* is optimized to act as a “*goto with arguments*,” that is, it does not push anything on the call stack (Steele 1977). An arbitrarily long sequence of tail calls takes no more space than one ordinary call.

Intuitively, a call is a tail call if it is the last thing a function does, i.e., the result of the tail call is also the result of the calling function. As an important special case, proper tail recursion requires that if the last thing a full Scheme function does is make a recursive call to itself, the implementation makes that recursive call as efficient as a *goto*.

Many tail calls are easy to identify; for example, in a C program, the last call before a *return* or before the end of a function is a tail call. And in a C statement `return f(args)`, the call to `f` is a tail call. To identify all tail calls, however, we need a more precise definition.

In Scheme, a tail call is a function call that occurs in a *tail context*. Tail contexts are defined by induction over abstract syntax. The full story is told by in Kelsey, Clinger, and Rees (1998, Section 3.5), from whom I have adapted this account, but the key rules look like this:

- The body of a `lambda` occurs in a tail context.
- When `(if e1 e2 e3)` occurs in a tail context, `e2` and `e3` occur in a tail context.
- When `(begin e1 e2 ... en)` occurs in a tail context, `en` occurs in a tail context.
- When a `let`, `let*`, or `letrec` occurs in a tail context, the body of the `let` occurs in a tail context.

The following example shows one tail call, to `f`. The calls to `g` and `h` are not tail calls. The reference to `x` is in a tail context, but it is not a call and so is not a tail call.

```
(lambda ()
  (if (g)
      (let ([x (h)])
        x)
      (if (g) (f) #f)))
```

The definition of tail context gives us a precise way to identify a function in continuation-passing style (Section 2.10.1): a function is written in continuation-passing style if and only if every expression in a tail context is either a call to a parameter or is a call to a function that is written in continuation-passing style.

The interpreter in this chapter does not optimize tail calls. But the interpreter in Chapter 3 does, and Chapter 3’s exercises can help you understand how proper tail calls work and how they affect the use of stack space.

### 2.14.3 Data types

Full Scheme has more primitive data types and functions than  $\mu$ Scheme. These types include mutable vectors (arrays), which can be written literally using the `#(...)` notation, as well as characters, mutable strings, and “I/O ports.” And the R<sup>6</sup>RS and R<sup>7</sup>RS standards include records with named fields;  $\mu$ Scheme’s record form (Section 2.13.6) is a scaled-down version of R<sup>6</sup>RS records.

Full Scheme supports lazy computations with `delay` and `force`.

Full Scheme provides many types of numbers, the meanings and representations of which are carefully specified in the standard. Numeric types can be arranged in a tower, in which each level contains all the levels below it:

```
number
complex
real
rational
integer
```

Numbers may also be *exact* or *inexact*. Most Scheme implementations, for example, automatically do exact arithmetic on arbitrarily large integers (“bignums”). If you are curious about bignums, you can implement them yourself; do Exercises 49 and 50 in Chapter 9 or Exercises 37 to 39 in Chapter 10.

#### 2.14.4 From syntactic sugar to syntactic abstraction: Macros

Full Scheme includes not only the syntactic sugar described in Section 2.13 but also the ability for programmers to define new forms of syntactic sugar, called *macros* (or *syntactic abstractions*). Macros have the same status as any other Scheme code: they can be included in user code and in libraries, and anybody can define one by writing a *macro transformer*. And Scheme’s macros are *hygienic*:

- If a macro transformer inserts a binding for an identifier (variable or keyword) not appearing in the macro use, the identifier is in effect renamed throughout its scope to avoid conflicts with other identifiers. For example, just as prescribed by the rules in Section 2.13.3, if a macro transformer expands  $(\lambda e_1 e_2)$  into  $(\text{let } ([t e_1]) (\text{if } t e_2))$ , then  $t$  is automatically renamed to avoid conflicting with identifiers in  $e_1$  and  $e_2$ . Standard Scheme does something similar with `or`; the expression  $(\text{or } e_1 e_2)$  is defined to expand to  $(\text{let } ([x e_1]) (\text{if } x e_2))$ . Scheme’s macro facility renames  $x$  as needed to avoid capture.
- Hygiene also protects each macro’s unbound names. For example,  $\mu$ Scheme’s `record` definition desugars into definitions that use `cons`, `car`, `cdr`, and `pair?` (Section 2.13.6). These names must not take their meanings from the context in which the `record` definition appears, as shown by this contrived example, which mixes  $\mu$ Scheme with full Scheme:

```
(let ([z 5])
  (define (cons y ys) 'you-lose)
  (define (pair? x) #f)
  (record point (x y))
  (assert (point? (make-point 3 4))))
```

If `record` were desugared naïvely, the code for `point?` would use the version of `pair?` that’s in scope, and the assertion would fail. But the assertion  $(\text{point? } (\text{make-point } e_1 e_2))$  should always succeed. And in a truly hygienic macro system, it does: the hygienic macro system guarantees that when a macro is desugared, its free names refer to the bindings visible where the macro was defined, regardless of how those names may be bound where the macro is used.

Hygiene makes Scheme macros remarkably powerful. Macro transformers can define new language features that in most settings that would require new abstract syntax. A good implementation of Scheme is more than just a programming language; it is a system for crafting programming languages. Details, examples, and ideas can be found in some of the readings mentioned in Section 2.15.2.

## 2.14.5 *call/cc* (also upcoming in Chapter 3)

Full Scheme includes a primitive function which can capture continuations that are defined implicitly within the interpreter, as if a program had been written in continuation-passing style. Primitive *call-with-current-continuation*, also called *call/cc*, can help you implement a staggering variety of control structures, including backtracking, multiple threads, exceptions, and more. Such *control operators* are a major topic of Chapter 3.

## 2.15 SUMMARY

Functional programming is a style of programming that relies on first-class, nested functions—in other words, on *lambda*. Functional programmers compose functions aggressively; they use functions to make other functions. Typical patterns of composition and computation are embodied in standard higher-order functions like maps, filters, and folds, where they can easily be reused. Such compositions emphasize “wholemeal programming”: code is composed of operations on whole lists or whole S-expressions, not of operations on one element at a time. Functional programming also emphasizes the use of immutable data: rather than mutate a variable, the experienced functional programmer usually binds a new one, as discussed on page 121. Immutability simplifies unit testing and enables more function composition. And thanks to very fast allocators (Chapter 4), immutability can be implemented efficiently.

### 2.15.1 Key words and phrases

**ATOM** A Scheme value that can be compared for equality in constant time: a symbol, a number, a Boolean, or the empty list. Atoms are the base case in the definition of S-EXPRESSIONS.

**CLOSURE** The run-time representation of a function, produced by evaluating a LAMBDA ABSTRACTION. Includes a representation of the function’s code and references to its FREE VARIABLES—or their locations. Closures are the simplest way to implement FIRST-CLASS, NESTED functions. A closure can be optimized to refer only to those free variables that are not also global variables—that is, the LET-bound and LAMBDA-bound variables of enclosing functions.

**CONS, CAR, CDR** The basic primitives that operate on lists, or more generally, S-EXPRESSIONS. By rights, `null?` should have equal status with `cons`, `car`, and `cdr`, but for some reason it is rarely mentioned in the same breath.

**FILTER** A HIGHER-ORDER FUNCTION that selects just some elements from a container structure—usually a list—according to a predicate function that is passed in.

**FIRST-CLASS FUNCTION** A function value that enjoys the same privileges as simple scalar values like machine integers: it may be passed to other functions, returned from other functions, stored in local and global variables, and stored in objects allocated on the heap, such as CONS CELLS. Most useful when NESTED. Functions in Scheme, Icon, and C are first-class values, but functions in Impcore are not.

**FIRST-ORDER FUNCTION** A function whose arguments and results are not functions and do not contain functions. Compare with HIGHER-ORDER FUNCTION.

**FOLD** A HIGHER-ORDER form of “reduction” that uses a given function to combine all elements of a container structure—usually a list. Folds can be used to implement sum, product, map, filter, and a zillion other functions that combine information two elements at a time. Like MAPS, folds can be found on lists, strings, mutable and immutable arrays, sets, trees, and so on.

**FREE VARIABLE** A name appearing in the body of a function that is bound neither by the LAMBDA ABSTRACTION that introduces the function nor by any LET BINDING within the body of the function. Names that refer to primitives like + and cons are typically free variables. LOCATIONS of free variables are captured in CLOSURES.

**HIGHER-ORDER FUNCTION** A function that either takes one or more functions as arguments, or more interestingly, that returns one or more functions as results. Classic examples include MAP, FILTER, and FOLD. More interesting higher-order functions can be defined only in a language in which functions are both FIRST-CLASS and NESTED. Compare with FIRST-ORDER FUNCTION.

**LAMBDA ABSTRACTION** The syntactic form by which a function is introduced. Scheme functions need not be named.

**LOCATION** A reference to MUTABLE storage. In both the semantics and implementation of Scheme, a variable stands for a MUTABLE location, not for a value, and what is captured in a CLOSURE are the *locations* of the FREE VARIABLES.

**LOCATION SEMANTICS** A semantics in which a variable stands for a mutable LOCATION in a store, not for a value. Describes languages like C and Scheme. Contrast with VALUE SEMANTICS.

**MAP** Any of a family of HIGHER-ORDER FUNCTIONS that operate on a container—like a list—and apply a function to each value contained.  $\mu$ Scheme provides maps only on lists, but functional languages that provide more data structures provide more maps. For example, ML compilers typically ship with maps on lists, strings, mutable and immutable arrays, optional values, sets, and trees.

**MONOMORPHIC** A property of a function or value that may be used only with values of a single type. The word is Greek for “one shape.” As an example, the + primitive is monomorphic because it works only with integers. By contrast, the car primitive, because it works with lists whose elements are of any type, is POLYMORPHIC .

**MUTABLE** Said of an entity whose state can change during the execution of a program, like a  $\mu$ Scheme variable. The very word “variable” suggests that the value can vary, which is to say it is mutable. Mutability is ultimately implemented by reference to a machine LOCATION. In  $\mu$ Scheme, variables are mutable but values are immutable. In full Scheme, records and CONS cells are mutable, using functions like set-car! and set-cdr!.

**NESTED FUNCTION** A function defined within another function. Using STATIC SCOPING, a nested function has access to the formal parameters and local variables of the enclosing function. Most useful when FIRST CLASS (and implemented by CLOSURES). Functions in Scheme and Lua may nest and are first-class; functions in Pascal and Ada may nest but are not first-class.

**POLYMORPHIC** A property of a function or value that may be used with more than one set of values. The word is Greek for “many shapes.” As an example, the length function is polymorphic because it works with any list, regardless of what sorts of values are stored in the list. By contrast, the sum function (Exercise 29), because it works only on lists of integers, is **MONOMORPHIC**.

**S-EXPRESSION** The basic datatype of Scheme. Also, the concrete, parenthesized-prefix syntax used to write Scheme. A fully general S-expression is either an ATOM or a CONS cell containing two S-expressions. An “ordinary” S-expression is either an atom or a list of ordinary S-expressions.

**SHARED MUTABLE STATE** A means of communication at a distance, not evident in a function’s arguments and results. In Scheme, any variable is mutable and any variable can be captured in a CLOSURE, so any variable can contain shared mutable state. In C, only global variables can contain shared mutable state—and they can be shared by any two functions, which is why your instructors may have warned you against them. Mutable state can be protected from over-sharing by hiding it under a LAMBDA ABSTRACTION, as in the example of the resettable counter or the random-number seed (Section 2.7.1 on page 126). When a function relies on mutable state, its abilities to be unit tested or to be composed with other functions may be compromised.

**SHORT-CIRCUIT EVALUATION** Describes a syntactic form that looks like an ordinary function call or operator, but that does not evaluate all of its operations. Short-circuit evaluation is common for conjunction and disjunction (Boolean “and” and “or”), which in C and  $\mu$ Scheme are written `&&` and `||`.

**STORE** In semantics, the set of meaningful LOCATIONS. Typically implemented by a combination of machine memory and machine registers. May include memory locations on the call stack as well as on the managed heap.

**SYNTACTIC SUGAR** Concrete syntax that extends a language by being translated into existing syntax. Examples in  $\mu$ Scheme include SHORT-CIRCUIT conditionals and record definitions, among other forms (Section 2.13).

**VALUE SEMANTICS** A semantics in which a variable stands for a value, not a mutable LOCATION. Describes languages like Impcore, ML, and Haskell. Contrast with LOCATION SEMANTICS.

### 2.15.2 Further reading

For insight into how a language is born, John McCarthy’s original paper (1960) and book (1962) about Lisp are well worth reading. But some more recent treatments of Lisp are clearer and more complete; these include books by Touretzky (1984), Wilensky (1986), Winston and Horn (1984), Graham (1993), and Friedman and Felleisen (1996). For the serious Lisper, the Common Lisp manual (Steele 1984) is an invaluable reference.

For Scheme, the closest analog is the “Lambda: The Ultimate —” series by Steele and Sussman (Sussman and Steele 1975; Steele and Sussman 1976, 1978); I especially recommend the 1978 article. For reference, Dybvig’s (1987) book is clear and well organized, and there are always the official standards (Kelsey, Clinger, and Rees 1998; Sperber et al. 2009; Shinn, Cowan, and Gleckler 2013).

Recursion isn’t just for functional programmers; it is also highly regarded in procedural languages, as taught by Rohl (1984), Roberts (1986) and Reingold and

Reingold (1988). Proper tail recursion is precisely defined by Clinger (1998), who also explores some of the implications.

To argue that functional programming matters, Hughes (1989) shows that it provides superior ways of putting together code: tools like `map`, `filter`, and `foldr` enable us to combine small, simple functions into big, powerful functions. By Hughes's standard,  $\mu$ Scheme is only half a functional language: although  $\mu$ Scheme provides higher-order functions, it does not provide *lazy evaluation*, a technique now strongly associated with Haskell.

Algebraic laws are explored in depth by Bird and Wadler (1988), who include many more list laws than I present. Algebraic laws are also a great tool for specifying the behavior of abstract data types (Liskov and Guttag 1986). The algebraic approach can also be used on procedural programs, albeit with some difficulty (Hoare et al. 1987). And algebraic laws support a systematic, effective, *property-based* approach to software testing (Claessen and Hughes 2000).

Full Scheme can express a shocking range of programming idioms, algorithms, data structures, and other computer-science ideas; the demonstration by Abelson and Sussman (1985) is likely to impress you. Beginners will get more out of Harvey and Wright (1994), who aim at students with little programming experience. Another approach for beginners uses five subsets of Scheme, which are carefully crafted to help raw beginners evolve into successful Schemers (Felleisen et al. 2018).

The idea of using statically scoped closures to implement first-class, nested functions did not originate with Scheme. The idea had been developed in a number of earlier languages, mostly in Europe. Examples include Iswim (Landin 1966), Pop-2 (Burstall, Collins, and Popplestone 1971), and Hope (Burstall, MacQueen, and Sannella 1980). The book by Henderson (1980) is from this school. Also highly recommended is the short, but very interesting, book by Burge (1975).

Continuation-passing style was used by Reynolds (1972) to make the meanings of “definitional” interpreters independent of their implementation language. The continuation-passing backtracking search in Section 2.10 is based on a “Byrd box,” which was used to understand Prolog programs (Byrd 1980); my terminology is that of Proebsting (1997), who describes an implementation of Icon, which has backtracking built in (Griswold and Griswold 1996).

Macros are nicely demonstrated by Flatt (2012), who creates new languages using Scheme’s syntactic abstraction together with other tools unique to Racket. If you like the ideas and you want to define your own macros, continue with Hendershott’s (2014) tutorial.

Macros have a long history. Kohlbecker et al. (1986) first addressed the problem of variable capture; they introduce a *hygiene condition* sufficient to avoid variable capture, and they define a hygienic macro expander. Dybvig, Hieb, and Brugeman (1992) build on this work, reducing the cost of macro expansion and enabling macros to track source-code locations; a key element of their macro expander is the *syntax object*, which encapsulates not only a fragment of abstract-syntax tree but also some information about its environment, so that variable capture can be avoided. Moving from full Scheme to Racket, Flatt et al. (2012) further increase the power of the macro system by enabling macro expanders to share information. But the complexity of the system is acknowledged as a drawback; in particular, it is no longer so obvious that variable capture is always avoided. Flatt (2016) proposes a new, simpler model wherein a fragment of syntax is associated with a *set* of environments, which together determine the meaning of each name mentioned within the fragment.

## 2.16 EXERCISES

The exercises are summarized in Table 2.8. Some of the highlights are as follows:

- Exercise 13 guides you to an *efficient*, purely functional representation of queues, without using mutation.
- Exercises 22 and 24 ask you to prove some classic algebraic laws of pure functional programming: append is associative, and the composition of maps is the map of the composition. The laws can be used to improve programs, sometimes even by optimizing compilers. The proofs combine equational reasoning with induction.
- Exercise 38 asks you to implement a “data structure” whose values are represented as functions.
- Exercise 41 invites you to work toward mastery of continuation-passing style by implementing a solver for general Boolean formulas.
- Exercise 60 on page 200 asks you to add a trace facility to the  $\mu$ Scheme interpreter; it will help you master the C code for the evaluator.

As you tackle the exercises, you can refer to Table 2.3 on page 99, which lists all the functions in  $\mu$ Scheme’s initial basis.

### 2.16.1 Retrieval practice and other short questions

- A. What laws relate primitives car, cdr, null?, and cons?
- B. What laws describe the behavior of the map function?
- C. Given the definition (record sundae [ice-cream sauce topping]), what code do you write to find out if a sundae is topped with 'cherry?
- D. Given the same definition of sundae, how do you make an ice-cream sundae with vanilla ice cream, chocolate sauce, topped with sprinkles?
- E. Given a food value that might be a frozen-dinner or a sundae, how do you interrogate it to find out which one it is?
- F. To select all even numbers from a list of numbers, do you use map, filter, or a fold?
- G. To double all the numbers in a list of numbers, do you use map, filter, or a fold?
- H. To find the largest of a nonempty list of numbers, do you use map, filter, or a fold?
- I. What’s the cost of a naïve list reversal?
- J. What’s the cost of a list reversal that uses accumulating parameters?
- K. According to function equal?, when are two S-expressions considered equal?
- L. According to primitive function =, when are two S-expressions considered equal?
- M. If you want to write procedural code in  $\mu$ Scheme, are you better off using let or let\*?

Exercises	Section	Notes	
1 to 5	2.3.4	Functions that consume lists, including lists that represent sets (§2.3.7).	
6 and 7	2.3.2	Functions that take accumulating parameters.	
8 and 9	2.2, 2.3.5	Lists all the way down—that is, S-expressions. A little coding and one proof (see §2.5.8).	
10 to 12	2.4	Functions involving records and trees.	
13 to 15	2.3, 2.5	Implement and specify purely functional data structures. I recommend using <code>let</code> to define local variables (§2.6).	§2.16. Exercises 179
16 to 23	2.5	Equational reasoning with first-order functions.	
24 to 26	2.5	Equational reasoning with higher-order functions.	
27 to 30	2.8	Standard higher-order functions on lists.	
31 to 36	2.7, 2.8	Functions that take other functions as arguments.	
37 to 40	2.7	Functions that return new functions made with <code>lambda</code> .	
41	2.10	Continuations.	
42 to 45	2.11	Using the operational semantics to reason about algebraic laws (§2.5) and syntactic sugar (§2.13).	
46 to 50	2.11	Using the operational semantics to explore alternatives to the existing design and the semantics of possible new features.	
51 and 52	2.11, 1.7.3	Metatheory: absence of aliasing, safety of extended environments.	
53 to 55	2.12, 2.13	Implementing syntactic sugar, and adding <i>quasiquotation</i> syntax for writing S-expressions.	
56 to 59	2.12.5	New primitives: <code>list</code> , <code>apply</code> , <code>set-car!</code> , <code>set-cdr!</code> , and <code>read</code> .	
60	2.12	Enhancement to the interpreter: tracing calls.	

Table 2.8: Synopsis of all the exercises, with most relevant sections

- N. What test is performed by the function `(o not ((curry =) 0))`?
- O. Which of the following list functions are naturally polymorphic in the list element? `length`, `sum`, `reverse`, `append`, `minimum`, `member?`, `sort`
- P. Which of the following list functions are not naturally polymorphic but can be made so by passing an additional (function-valued) parameter? `length`, `sum`, `reverse`, `append`, `minimum`, `member?`, `sort`
- Q. A search function takes two continuations: one for success and one for failure. Which continuation expects a parameter? Why?
- R. In Impcore, a variable stands for a value  $v$ , but in  $\mu$ Scheme, a variable stands for a location  $\ell$ . What example in the chapter exploits this aspect of  $\mu$ Scheme’s semantics?
- S. The equation  $(|| e_1 e_2) = (\text{if } e_1 e_1 e_2)$  is not quite a valid algebraic law. What could go wrong?
- T. The equation  $(|| e_1 e_2) = (\text{let } ([x \ e_1]) (\text{if } x \times e_2))$  is not quite a valid algebraic law. What could go wrong?

## 2.16.2 Functions that consume lists

1. Comparing elements of two lists. Implement both of the following functions:

(a) When both `xs` and `ys` are in set  $LIST(ATOM)$ , `contig-sublist?` determines whether the first list is a contiguous subsequence of the second. That is, `(contig-sublist? xs ys)` returns `#t` if and only if there are two other lists `front` and `back`, such that `ys` is equal to `(append (append front xs) back)`.

*(exercise transcripts 180a)≡*

```
-> (contig-sublist? '(a b c) '(x a y b z c))  
#f  
-> (contig-sublist? '(a y b) '(x a y b z c))  
#t  
-> (contig-sublist? '(x) '(x a y b z c))  
#t
```

(b) When both `xs` and `ys` are in set  $LIST(ATOM)$ , `sublist?` determines whether the first list is a mathematical subsequence of the second. That is, `(sublist? xs ys)` returns `#t` if and only if the list `ys` contains the elements of `xs`, in the same order, but possibly with other values in between.

*(exercise transcripts 180a)++≡*

```
-> (sublist? '(a b c) '(x a y b z c))  
#t  
-> (sublist? '(a y b) '(x a y b z c))  
#t  
-> (sublist? '(a z b) '(x a y b z c))  
#f  
-> (sublist? '(x y z) '(x a y b z c))  
#t
```

2. Numbers and lists of digits. Function `explode-digits` converts a nonnegative number to a list of its decimal digits, and function `implode-digits` converts back:

*(exercise transcripts 180a)++≡*

```
-> (explode-digits 1856)  
(1 8 5 6)  
-> (explode-digits 0)  
(0)  
-> (implode-digits '(2 4 6 8))  
2468
```

Implement `explode-digits` and `implode-digits`.

*Hint:* A list of digits is much easier to work with if the least significant digit is first. Work with such lists, and at need, use `reverse`.

3. Sets represented as lists. Implement the following set functions:

- (`remove x s`) returns a set having the same elements as set `s` with element `x` removed.
- (`subset? s1 s2`) determines if `s1` is a subset of `s2`.
- (`=set? s1 s2`) determines if lists `s1` and `s2` represent the same set.

4. *Sets as lists: understanding equality.* Chunks 107a and 107c use lists to represent sets, and chunk 107d shows an example in which a set's *element* may also be a list. In the text, I claim that if `member?` uses `=` instead of `equal?`, the example in chunk 107d doesn't work.

- (a) In the example, which set functions go wrong? Is it `add-element`, `member?`, or both?
- (b) What goes wrong exactly, and why should the fault be attributed to using the `=` primitive instead of the `equal?` function?

---

## §2.16. Exercises

181

5. *Synchronized access to two lists.* Implement function `dot-product`, which computes the dot product of two vectors, represented as lists. (If the vectors are  $u_1, \dots, u_n$  and  $v_1, \dots, v_n$ , the dot product is  $u_1 \cdot v_1 + \dots + u_n \cdot v_n$ , where  $u_i \cdot v_i$  means multiplication.)

*(exercise transcripts 180a)*  $\equiv$

```
-> (dot-product '(1 2 3) '(10 5 2))  
26
```

### 2.16.3 Programming with accumulating parameters

6. *Accumulating parameters and reversal.* A list of things has an inductive structure; it is either empty or is a cons cell containing a thing and a list of things. A numeral has a similar inductive structure; it is either a digit or is a numeral followed by a digit. If you've done Exercise 7 in Chapter 1, you've already implemented several functions that manipulate the digits of a decimal numeral. Here's one to which you can apply the method of *accumulating parameters* described in Section 2.3.2:

- (a) Define a function `reverse-digits`, which is given a nonnegative number and returns a number whose decimal representation is the decimal representation of the original number, but with the digits reversed.

*(exercise transcripts 180a)*  $\equiv$

```
-> (reverse-digits 123)  
321  
-> (reverse-digits 1066)  
6601  
-> (reverse-digits 100)  
1  
-> (reverse-digits 77)  
77
```

Function `reverse-digits` could simply convert a number to a list of digits, call `reverse`, and convert the list back to a number. But look again at Section 2.3.2, and use similar ideas to define `reverse-digits` without ever materializing a list of digits.<sup>16</sup> By materializing only numbers and Booleans, you will avoid allocating space on the heap.

- (b) Is the following algebraic law valid?

$$(\text{reverse-digits} (\text{reverse-digits } n)) = n$$

Justify your answer.

---

<sup>16</sup>To *materialize* a value is to represent it explicitly during computation. In Scheme, a materialized value is stored in a location that is either referred to by name or is part of a cons cell.

7. *Accumulating parameters and cost savings.* Function `preorder` in Section 2.4.2 may allocate unnecessary cons cells. Using the method of accumulating parameters from Section 2.3.2, rewrite `preorder` so that the number of cons cells allocated is exactly equal to the number of cons cells in the final answer.

*Scheme,*

*S-expressions, and  
first-class functions*

182

#### 2.16.4 Generalizing lists to S-expressions

8. *Inspection and manipulation of ordinary S-expressions.* As described in Section 2.5.8, an ordinary S-expression is either an atom or a list of ordinary S-expressions—and of course a list of ordinary S-expressions is either empty or formed with `cons`. Using your understanding of these possible forms, implement functions `count`, `countall`, `mirror`, and `flatten`:

- (a) When  $x$  is an atom and  $xs$  is in  $LIST(SEXP_O)$ ,  $(\text{count } x \ xs)$  returns the number of (top-level) elements of  $xs$  that are equal to  $x$ . Assume  $x$  is not  $'()$ .

```
exercise transcripts 180a +≡
-> (count 'a '(1 b a (c a)))
1
```

- (b) When  $x$  is an atom and  $xs$  is in  $LIST(SEXP_O)$ ,  $(\text{countall } x \ xs)$  returns the number of times  $x$  occurs *anywhere* in  $xs$ , not only at top level. Assume  $x$  is not  $'()$ .

```
exercise transcripts 180a +≡
-> (countall 'a '(1 b a (c a)))
2
```

- (c) When  $xs$  is a list of S-expressions,  $(\text{mirror } xs)$  returns a list in which every list in  $xs$  is recursively mirrored, and the resulting lists are in reverse order.

```
exercise transcripts 180a +≡
-> (mirror '(1 2 3 4 5))
(5 4 3 2 1)
-> (mirror '((a (b 5)) (c d) e))
(e (d c) ((5 b) a))
```

Informally, `mirror` returns the S-expression you would get if you looked at its argument in a vertically oriented mirror, except that the individual atoms are not reversed. (Try putting a mirror to the right of the example, facing left.) More precisely, `mirror` consumes an S-expression and returns the S-expression that you would get if you wrote the brackets and atoms of the original S-expression in reverse order, exchanging open brackets for close brackets and vice versa.

- (d) Function `flatten` consumes a list of S-expressions and erases internal brackets. That is, when  $xs$  is a list of S-expressions,  $(\text{flatten } xs)$  constructs a list having the same atoms as  $xs$  in the same order, but in a flat list. For purposes of this exercise,  $'()$  should be considered not as an atom but as an empty list of atoms.

```
exercise transcripts 180a +≡
-> (flatten '((I Ching) (U Thant) (E Coli)))
(I Ching U Thant E Coli)
-> (flatten '((((a))))))
(a)
```

```

⟨exercise transcripts 180a⟩ +≡
  -> (flatten '())
  ()
  -> (flatten '((a) () ((b c) d e)))
  (a b c d e)

```

9. *Proof about S-expressions.* Not only is a list of ordinary S-expressions itself an ordinary S-expression, a list of fully general S-expressions is also a fully general S-expression. Prove it. That is, using the inductive definitions of  $LIST(A)$  and  $SEXP_{FG}$  in Section 2.5.8, prove that

$$LIST(SEXP_{FG}) \subseteq SEXP_{FG}.$$

The theorem implies that if you are writing a recursive function that consumes a list of S-expressions, you could instead try to write a more general function that consumes any S-expression. The more general function is sometimes simpler.

#### 2.16.5 Functions involving records and trees

10. *Inspecting the contents of records.*

- (a) Define a function `desserts` that takes a list of frozen dinners and returns a list of the desserts.
  - (b) Using `frozen-dinner?`, define a function `#dinners` that takes a list of values and returns the number of values that are frozen dinners.
  - (c) Define a function `steak-dinners` that takes a list of frozen dinners and returns a list containing only those frozen dinners that offer 'steak as a protein.
11. *Using node records to implement tree traversals.* Using the representation defined in Section 2.4, define functions that implement `postorder` and `inorder` traversal for binary trees.
12. *Traversals of rose trees.* A *rose tree* is a tree in which each node can have arbitrarily many children. We can define an abstract rose tree as a member of the smallest set that satisfies this recursion equation:

$$ROSE_{\mathcal{A}}(A) = \{ (\text{make-rose } a \text{ } ts) \mid a \in A \wedge ts \in LIST(ROSE_{\mathcal{A}}(A)) \}$$

If we prefer to write rose trees as S-expression literals, we can define

$$ROSE(A) = A \cup \{ (\text{cons } a \text{ } ts) \mid a \in A \wedge ts \in LIST(ROSE(A)) \}$$

Extend the preorder and level-order traversals of Sections 2.4.2 and 2.6 to rose trees. Note that in the non-abstract  $ROSE$  representation, a leaf node labeled `a` can be represented in two ways: either as '`a`' or as '`(a)`'.

#### 2.16.6 Functional data structures

13. *Efficient queues.* The implementation of queues in Section 2.6 on page 121 is simple but inefficient; `enqueue` can take time and space proportional to the number of elements in the queue. The problem is the representation: the queue operations have to get to both ends of the queue, and getting to

the back of a list requires linear time. A better representation uses *two* lists stored in a record:

```
(record queue [front back])
```

In this record,

- The queue-front list represents the front of the queue. It stores older elements, and they are ordered with the oldest element at the beginning, so functions `front` and `without-front` can be implemented using `car` and `cdr`.
- The queue-back list represents the back of the queue. It stores young elements, and they are ordered with the youngest element at the beginning, so function `enqueue` can be implemented using `cons`.

The only trick here is that when the queue-front elements are exhausted, the queue-back elements must somehow be transferred to the front. Using the two lists, implement value `emptyqueue` and functions `empty?`, `front`, `without-front`, and `enqueue`. Each operation should take constant time on average. (Proofs of average-case time over a sequence of operations use *amortized analysis*.)

14. *Algebraic laws for queues.* The inefficient queue operations in Section 2.6 and the efficient queue operations in Exercise 13 can both be described by a single set of algebraic laws. As explained in Section 2.5.2, such laws must specify the result of applying any acceptable observer to any combination of constructors. (The queues in this chapter are immutable, so mutators do not come into play.)

- Any queue can be constructed by applying `enqueue` zero or more times to `emptyqueue`.
- Only three operations get information from (observe) queues: `empty?`, `front`, and `without-front`.

*Write algebraic laws* sufficient to specify the behavior of all meaningful combinations of constructors and observers. Don't try to specify erroneous combinations like `(front emptyqueue)`.

*Hint:* This exercise is harder than it may appear—if you're not careful, you may find yourself specifying a stack, not a queue. Consider turning each algebraic law into a function, so you can test it as shown in Exercise 36.

15. *Graphs represented as S-expressions: topological sort.* Many directed graphs can be represented as ordinary S-expressions. For example, if each node is labeled with a distinct symbol, a graph can be represented as a list of edges, where each edge is a list containing the labels of that edge's source and destination. Write a function that topologically sorts a graph specified by this *edge-list* representation. The function `(tsort edges)` should return a list that contains the labels in `edges`, in topological order.

In topological sorting, the two symbols in an edge introduce a *precedence constraint*: for example, if '(`a b`) is an edge, then in the final sorted list of labels, `a` must precede `b`. As an example, `(tsort '((a b) (a c) (c b) (d b)))` can return either '(`a d c b`) or '(`a c d b`).

Not every graph can be topologically sorted; if the graph has a cycle, function `tsort` should call `error`.

For details on topological sorting, see Sedgewick (1988, Chapter 32), or Knuth (1973, Section 2.2.3).

*(exercise transcripts 180a)* +≡

```
-> (tsort '((duke commoner) (king duke) (queen duke) (country king)))
  (queen country king duke commoner)
```

### 2.16.7 Equational reasoning with first-order functions

16. Prove that `(member? x emptyset) = #f.`
17. Prove that for any predicate `p?`, `(exists? p? '()) = #f.`
18. Prove that for any list of values `xs`, `(all? (lambda (_ ) #t) xs) = #t.`
19. Prove that `(member? x (add-element x s)) = #t.`
20. Prove that `(length (reverse xs)) = (length xs).`
21. Prove that `(reverse (reverse xs)) = xs.`
22. Prove that `(append (append xs ys) zs) = (append xs (append ys zs)).`
23. Prove that `(flatten (mirror xs)) = (reverse (flatten xs))`, where functions `flatten` and `mirror` are defined as in Exercise 8.

### §2.16. Exercises

185

### 2.16.8 Equational reasoning with higher-order functions

24. Prove that the composition of `maps` is the `map` of the composition:

$$(\text{map } f (\text{map } g \text{ xs})) = (\text{map } (o f g) \text{ xs})$$

25. Prove again that the composition of `maps` is the `map` of the composition, but this time, prove equality of two functions, not just equality of two lists:

$$(o ((\text{curry map}) f) ((\text{curry map}) g)) = ((\text{curry map}) (o f g))$$

To prove that two functions are equal, show that when applied to equal arguments, they always return equal results.

26. Prove that if `takewhile` and `dropwhile` are defined as in Exercise 31, then for any list `xs`, `(append (takewhile p? xs) (dropwhile p? xs)) = xs.`

### 2.16.9 Using standard higher-order functions on lists

27. *Select a subsequence of elements.* Define function `remove-multiples-too`, which does what `remove-multiples` does, but works by using the higher-order functions in the initial basis. Copy function `divides?` from chunk 104a, and use whatever you like from the initial basis, but do not otherwise use `lambda`, `if`, or recursion.

*(exercise transcripts 180a)* +≡

```
-> (remove-multiples-too 2 '(2 3 4 5 6 7))
  (3 5 7)
```

28. *Maps and folds with scalar results.* Use `map`, `curry`, `foldl`, and `foldr` to define the following functions:

- (a) `cdr*`, which lists the `cdr`'s of each element of a list of lists:

```
exercise transcripts 180a +≡  
-> (cdr* '(a b c) (d e) (f))  
((b c) (e) ())
```

- (b) `max*`, which finds the maximum of a non-empty list of integers

- (c) `gcd*`, which finds the greatest common divisor of a non-empty list of nonnegative integers

- (d) `lcm*`, which finds the least common multiple of a non-empty list of non-negative integers.

- (e) `sum`, which finds the sum of a non-empty list of integers

- (f) `product`, which finds the product of a non-empty list of integers

```
exercise transcripts 180a +≡  
-> (sum '(1 2 3 4))  
10  
-> (product '(1 2 3 4))  
24  
-> (max* '(1 2 3 4))  
4  
-> (lcm* '(1 2 3 4))  
12
```

- (g) `mkaipairfn`, which when applied to an argument  $v$ , returns a function that when applied to a list of lists, places  $v$  in front of each element:

```
exercise transcripts 180a +≡  
-> ((mkaipairfn 'a) '((() (b c) (d) ((e f))))  
((a) (a b c) (a d) (a (e f))))
```

All seven functions can be defined by a sequence of definitions that includes only one `define`, one `lambda`, and seven `val` bindings. Try it.

29. *Maps and folds with list results.* Use `map`, `curry`, `foldl`, and `foldr` to define the following functions:

- (a) `append`, which appends two lists

- (b) `snoc`, which is `cons` spelled backwards, and which adds an S-expression to the *end* of a list:

```
exercise transcripts 180a +≡  
-> (snoc 'a '(b c d))  
(b c d a)
```

- (c) `reverse`, which reverses a list

- (d) `insertion-sort`, which sorts a list of numbers

When implementing `insertion-sort`, take `insert` as given in chunk 103a.

30. *Folds for everything.* Use `foldr` or `foldl` to implement `map`, `filter`, `exists?`, and `a11?`. It is OK if some of these functions do more work than their official versions, as long as they produce the same answers.

For the best possible solution, define functions that no civilized  $\mu$ Scheme program can distinguish from the originals. (A civilized program may execute any code, including `set`, but it may not change the functions in the initial basis.)

### 2.16.10 Functions that take functions as arguments

31. *Selections of sublists.* Function `takewhile` takes a predicate and a list and returns the longest prefix of the list in which every element satisfies the predicate. Function `dropwhile` removes the longest prefix and returns whatever is left over.

```
exercise transcripts 180a +≡
-> (define even? (x) (= (mod x 2) 0))
-> (takewhile even? '(2 4 6 7 8 10 12))
(2 4 6)
-> (dropwhile even? '(2 4 6 7 8 10 12))
(7 8 10 12)
```

### §2.16. Exercises

187

Implement `takewhile` and `dropwhile`.

(You might also look at Exercise 26, which asks you to prove a basic law about `takewhile` and `dropwhile`.)

32. *Lexicographic comparison.*

FIX ME!

Sorting pairs of integers lexicographically is only slightly more difficult:

```
transcript 97a +≡
-> (define pair< (p1 p2)
  (if (!= (car p1) (car p2))
    (< (car p1) (car p2))
    (< (cadr p1) (cadr p2))))
-> ((mk-insertion-sort pair<) '((4 5) (2 9) (3 3) (8 1) (2 7)))
((2 7) (2 9) (3 3) (4 5) (8 1))
```

Chunk 187b shows `pair<`, a function that compares pairs of integers lexicographically. Write a higher-order function `lex-<*` to compare lists of differing lengths lexicographically. It should take as a parameter an ordering on the elements of the list. Be careful not to make any assumptions about the elements, other than that they can be ordered by the parameter.

```
exercise transcripts 180a +≡
-> (val alpha-< (lex-<* <))
-> (alpha-< '(4 15 7) '(4 15 7 13 1))
#t
-> (alpha-< '(4 15 7) '(4 15 5))
#f
```

This lexicographic ordering of lists is closely related to alphabetical ordering; to see the relationship, translate the numbers to letters: the two results above say `DOG < DOGMA` and `DOG < DOE`.

<code>car</code>	$\mathcal{P}$ 164a
<code>mk-insertion-sort</code>	137c

33. *Binary search trees, polymorphically.* Write a higher-order implementation of binary-search trees. Use the third style of polymorphism described in Section 2.9.1. That is, write a function `specialized-search` that takes a comparison function as argument and returns an association list which associates symbols `lookup` and `insert` to the corresponding functions.
34. *Generalized dot product.* Exercise 5 asks for a `dot-product` function. Generalize this pattern of computation into a “fold-like” function `foldr-pair`, which operates on two lists of the same length. (This function is primitive in APL.)

35. *Generalized preorder traversal.* Function `preorder` in Section 2.4.2 is not so useful if, for example, one wants to perform some other computation on a tree, like finding its height. By analogy with `foldl` and `foldr`, define `fold-preorder`. In addition to the tree, it should take two arguments: a function to be applied to internal nodes, and a value to be used in place of the empty tree. Function `fold-preorder` can be used to get the labels and the height of the example tree on page 111:

```
(exercise transcripts 180a) +≡
  -> (fold-preorder
        (lambda (tag left right) (cons tag (append left right)))
        '()
        example-sym-tree)
        (A B C D E F G H I)
  -> (define tree-height (t)
        (fold-preorder (lambda (tag left right) (+ 1 (max left right)))
                      0
                      t))
  -> (tree-height example-sym-tree)
  4
```

36. *Generalized all?, with predicates of two or three arguments.* In this problem you generalize function `all?` so it can work with all pairs from two lists or all triples from three lists. For example, `all-pairs?` might be used with this `length-append-law` function to confirm that the law holds for 25 combinations of inputs:

```
(exercise transcripts 180a) +≡
  -> (define length-append-law (xs ys)
        (= (length (append xs ys))
           (+ (length xs) (length ys))))
  -> (all-pairs? length-append-law
        '((a b c) (singleton) () (3 1 4 1 5 9) (2 7 1 8 2 8))
        '((1 2 3) () (elephants got big feet) (z) (w x)))
  #t
```

Generalize `all?` to combinations of values drawn from two or three lists:

- (a) Define function `all-pairs?`, which tests its first argument on all pairs of values taken from its second two arguments. Here are some more example calls:

```
(exercise transcripts 180a) +≡
  -> (all-pairs? < '(1 2 3) '(4 5 6 7))
  #t
  -> (all-pairs? < '(1 2 3) '(4 5 6 2))
  #f
```

When the second test fails, the fault could be in the implementation or the specification. To simplify diagnosis, I expand the `<` function:

```
(exercise transcripts 180a) +≡
  -> (all-pairs? (lambda (n m) (< m n)) '(1 2 3) '(4 5 6 2))
  #f
```

It is definitely *not* an algebraic law that for all  $m$  and  $n$ ,  $m < n$ , so the fault here lies in the specification, not in the implementation of `<`.

- (b) Define function `all-triples?`, which works like `all-pairs?` but can test laws like the associativity of `append`.

```
(exercise transcripts 180a) +≡
```

```

-> (define a-a-law (xs ys zs)    ;; append/append law
    (equal? (append xs (append ys zs))
            (append (append xs ys) zs)))
-> (all-triples? a-a-law '((a) () (b c)) '((1 2) (3)) '((4 5 6)))
#t

```

### 2.16.11 Functions that return functions

37. *Using lambda: creation and combination of fault-detection functions.* This problem models a real-life fault detector for Web input. An *input* to a Web form is represented as an association list, and a *detector* is a function that takes an *input* as argument, then returns a (possibly empty) list of *faults*. A fault is represented as a symbol—typically the name of an input field that is unacceptable.

Define the following functions, each of which is either a detector or a detector builder:

- (a) Function `faults/none` is a detector that always returns the empty list of faults.
  - (b) Function `faults/always` takes one argument (a fault  $F$ ), and it returns a detector that finds fault  $F$  in every input.
  - (c) Function `faults/equal` takes two arguments, a key  $k$  and a value  $v$ , and it returns a detector that finds fault  $k$  if the input binds  $k$  to  $v$ . Otherwise it finds no faults.
  - (d) Function `faults/union` takes two detectors  $d_1$  and  $d_2$  as arguments. It returns a detector that, when applied to an input, returns all the faults found by detector  $d_1$  and also all the faults found by detector  $d_2$ .
38. *Sets represented as functions.* In just about any language, sets can be represented as lists, but in Scheme, because functions are first class, a set can be represented as a function. This function, called the *characteristic function*, is the function that returns #t when given an element of the set and #f otherwise. For example, the empty set is represented by a function that always returns #f, and membership test is by function application:

*exercise transcripts 180a* +≡  
 -> (val emptyset (lambda (x) #f))  
 -> (define member? (x s) (s x))

Representing each set by its characteristic function, solve the following problems:

- (a) Define set `evens`, which contains all the even integers.
  - (b) Define set `two-digits`, which contains all two-digit (positive) numbers.
  - (c) Implement `add-element`, `union`, `inter`, and `diff`. The set (`diff s1 s2`) is the set that contains every element of  $s_1$  that is not also in  $s_2$ .
  - (d) Implement the third style of polymorphism (page 136).
39. *Lists reimplemented using just lambda.* This exercise explores the power of `lambda`, which can do more than you might have expected.
- (a) I claim above that *any* implementation of `cons`, `car`, and `cdr` is acceptable provided it satisfies the list laws in Section 2.5.1. Define `cons`, `car`, `cdr`, and `empty-list` using only `if`, `lambda`, function application, the

primitive `=`, and the literals `#t` and `#f`. Your implementations should pass this test:

```
(exercise transcripts 180a) +≡
  -> (define nth (n xs)
        (if (= n 1)
            (car xs)
            (nth (- n 1) (cdr xs))))
    nth
  -> (val ordinals (cons '1st (cons '2nd (cons '3rd empty-list))))
  -> (nth 2 ordinals)
  2nd
  -> (nth 3 ordinals)
  3rd
```

- (b) Under the same restrictions, define `null?`.
- (c) Now, using only `lambda` and function application, Define `cons`, `car`, `cdr`, `null?`, and `empty-list`. Don't use `if`, `=`, or any literal values.

The hard part is implementing `null?`, because you can't use the standard representation of Booleans. Instead, you have to invent a new representation of truth and falsehood. This new representation won't support the standard `if`, so you have to develop an alternative, and you have to use that alternative in place of `if`. For inspiration, you might look at the definition of `binds?` in chunk 143b, which uses `find-c`. If there were no such thing as a Boolean, how would you adapt the `binds?` function?

Equipped with your alternative to `if` and a new representation of Booleans, implement `null?` using only `lambda` and function application. And explain what a caller of `null?` should write instead of `(if (null? x) e1 e2)`.

40. *Mutable reference cells implemented using lambda.* Exercise 58 invites you to add mutation to the  $\mu$ Scheme interpreter by adding primitives `set-car!` and `set-cdr!` But if you want to program with mutation, you don't need new primitives—`lambda` is enough.

A mutable container that holds one value is called a *mutable reference cell*. Design a representation of mutable reference cells in  $\mu$ Scheme, and implement in  $\mu$ Scheme, without modifying the interpreter, these new functions:

- (a) Function `make-ref` takes one argument  $v$  and returns a fresh, mutable reference cell that initially contains  $v$ . The mutable reference cell returned by `make-ref` is distinct from all other mutable locations.
- (b) Function `ref-get` takes one argument, which is a mutable reference cell, and returns its current contents.
- (c) Function `ref-set!` takes two arguments, a mutable reference cell and a value, and it updates the mutable reference cell so it holds the value. It also returns the value.

```
(exercise transcripts 180a) +≡
  -> (val r (make-ref 3))
  -> (ref-get r)
  3
  -> (ref-set! r 99)
  99
  -> (ref-get r)
  99
```

```

⟨exercise transcripts 180a⟩ +≡
-> (define inc (r)
      (ref-set! r (+ 1 (ref-get r))))
-> (inc r)
100

```

### 2.16.12 Continuations

41. *Continuation-passing style for a Boolean-formula solver.* Generalize the solver in Section 2.10.2 to handle *any* formula, where a formula is one of the following:

§2.16. Exercises

191

- A symbol, which stands for a variable
- The list (`not f`), where  $f$  is a formula
- The list (`and f1 ... fn`), where  $f_1, \dots, f_n$  are formulas
- The list (`or f1 ... fn`), where  $f_1, \dots, f_n$  are formulas

Mathematically, the set of formulas  $F$  is the smallest set satisfying this equation:

$$\begin{aligned} F = & \text{SYM} \\ & \cup \{ (\text{list2 } 'not f) \mid f \in F \} \\ & \cup \{ (\text{cons } 'and fs) \mid fs \in \text{LIST}(F) \} \\ & \cup \{ (\text{cons } 'or fs) \mid fs \in \text{LIST}(F) \} \end{aligned}$$

Define function `find-formula-true-asst`, which, given a satisfiable formula in this form, finds a *satisfying assignment*—that is, a mapping of variables to Booleans that makes the formula true. Remember De Morgan’s laws, one of which is mentioned on page 133.

Function `find-formula-true-asst` should expect three arguments: a formula, a failure continuation, and a success continuation. When it is called, as in `(find-formula-true-asst f fail succ)`, it should try to find a satisfying assignment for formula  $f$ . If it finds a satisfying assignment, it should call `succ`, passing both the satisfying assignment (as an association list) and a resume continuation. If it fails to find a satisfying assignment, it should call `(fail)`.

You’ll be able to use the ideas in Section 2.10.2, but probably not the code. Instead, try using `letrec` to define the following mutually recursive functions:

- `(find-formula-asst formula bool cur fail succeed)` extends current assignment `cur` to find an assignment that makes the single formula equal to `bool`.
- `(find-all-asst formulas bool cur fail succeed)` extends `cur` to find an assignment that makes every formula in the list `formulas` equal to `bool`.
- `(find-any-asst formulas bool cur fail succeed)` extends `cur` to find an assignment that makes any one of the `formulas` equal to `bool`.

<code>car</code>	P 164a
<code>cdr</code>	P 164a
<code>cons</code>	P S307e

In all the functions above, `bool` is `#t` or `#f`.

Solve the problem in two parts:

- Write algebraic laws for all four recommended functions.
- Write the code.

<p><b>MK CLOSURE</b></p> $\frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho \rangle, \sigma \rangle}$	<p><b>VAR</b></p> $\frac{x \in \text{dom } \rho \quad \rho(x) \in \text{dom } \sigma}{\langle \text{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle}$
<p><b>APPLY CLOSURE</b></p> $\frac{\ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \quad \langle e, \rho, \sigma \rangle \Downarrow \langle \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \rho_c \rangle, \sigma_0 \rangle \quad \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \quad \vdots \quad \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle}{\langle e_c, \rho_c \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma_n \{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle}$	$\frac{x \in \text{dom } \rho \quad \rho(x) = \ell \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \{ \ell \mapsto v \} \rangle}$
<p><b>LITERAL</b></p> $\frac{}{\langle \text{LITERAL}(v), \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$	
<p><b>IF TRUE</b></p> $\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 \neq \text{BOOLV}(\#f) \quad \langle e_2, \rho, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma \rangle \Downarrow \langle v_2, \sigma'' \rangle}$	
<p><b>IF FALSE</b></p> $\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 = \text{BOOLV}(\#f) \quad \langle e_3, \rho, \sigma' \rangle \Downarrow \langle v_3, \sigma'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma \rangle \Downarrow \langle v_3, \sigma'' \rangle}$	
<p><b>WHILE ITERATE</b></p> $\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 \neq \text{BOOLV}(\#f) \quad \langle e_2, \rho, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle \quad \langle \text{WHILE}(e_1, e_2), \rho, \sigma'' \rangle \Downarrow \langle v_3, \sigma''' \rangle}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma \rangle \Downarrow \langle v_3, \sigma''' \rangle}$	
<p><b>WHILE END</b></p> $\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v_1 = \text{BOOLV}(\#f)}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma \rangle \Downarrow \langle \text{BOOLV}(\#f), \sigma' \rangle}$	

Figure 2.9: Summary of operational semantics (expressions, except LET forms, BEGIN, and primitives)

LET

$$\frac{\begin{array}{c} x_1, \dots, x_n \text{ all distinct} \\ \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma\{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \\ \hline \langle \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

LETSTAR

$$\frac{\begin{array}{c} \rho_0 = \rho & \sigma_0 = \sigma \\ \langle e_1, \rho_0, \sigma_0 \rangle \Downarrow \langle v_1, \sigma'_0 \rangle & \ell_1 \notin \text{dom } \sigma'_0 \\ \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} & \sigma_1 = \sigma'_0\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma'_{n-1} \rangle \\ \ell_n \notin \text{dom } \sigma'_{n-1} & \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} & \sigma_n = \sigma'_{n-1}\{\ell_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle & & \end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

LETREC

$$\frac{\begin{array}{c} \ell_1, \dots, \ell_n \notin \text{dom } \sigma \text{ (and all distinct)} \\ x_1, \dots, x_n \text{ all distinct} \\ e_i \text{ has the form LAMBDA}(\dots), 1 \leq i \leq n \\ \rho' = \rho\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\ \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \dots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \\ \hline \langle \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

BEGIN

$$\frac{\begin{array}{c} \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \langle e_2, \rho, \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \hline \langle \text{BEGIN}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v_n, \sigma_n \rangle \end{array}}{\langle \text{BEGIN}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v_n, \sigma_n \rangle}$$

EMPTYBEGIN

$$\frac{\langle \text{BEGIN}(), \rho, \sigma \rangle \Downarrow \langle \text{BOOLV}(\#f), \sigma \rangle}{\langle \text{BEGIN}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v_n, \sigma_n \rangle}$$

Figure 2.10: Summary of operational semantics (LET forms and BEGIN)

	<b>APPLYADD</b>	
$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle$	$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(+), \sigma_1 \rangle$ $\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2 \rangle$ $\langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle \text{NUMBER}(m), \sigma_3 \rangle$ $\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{NUMBER}(n + m), \sigma_3 \rangle$	
	<b>EQNUMBER</b>	<b>EQSYMBOL</b>
$m = n$ (identity of numbers)	$s = s'$ (identity of symbols)	$\text{NUMBER}(n) \equiv \text{NUMBER}(m)$
$\text{NUMBER}(n) \equiv \text{NUMBER}(m)$	$\text{SYMBOL}(s) \equiv \text{SYMBOL}(s')$	
	<b>EQBOOL</b>	<b>EQNIL</b>
$b = b'$ (identity of Booleans)	$\text{BOOLV}(b) \equiv \text{BOOLV}(b')$	$\text{NIL} \equiv \text{NIL}$
	<b>APPLYEQTRUE</b>	
$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(=), \sigma_1 \rangle$	$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v_1, \sigma_2 \rangle$	
$\langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle v_2, \sigma_3 \rangle$	$v_1 \equiv v_2$	
$\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{BOOLV}(\#\text{t}), \sigma_3 \rangle$		
	<b>APPLYEQFALSE</b>	
$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(=), \sigma_1 \rangle$	$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v_1, \sigma_2 \rangle$	
$\langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle v_2, \sigma_3 \rangle$	$v_1 \not\equiv v_2$ (i.e., no proof of $v_1 \equiv v_2$ )	
$\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{BOOLV}(\#\text{f}), \sigma_3 \rangle$		
	<b>APPLYPRINTLN</b>	
$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{println}), \sigma_1 \rangle$	$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v, \sigma_2 \rangle$	
$\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle v, \sigma_2 \rangle$		while printing $v$
	<b>APPLYPRINTU</b>	
$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{printu}), \sigma_1 \rangle$	$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2 \rangle$	
$0 \leq n < 2^{16}$		
$\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2 \rangle$		while printing the UTF-8 coding of $n$
	<b>CONS</b>	
$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{cons}), \sigma_1 \rangle$	$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle v_1, \sigma_2 \rangle$	
$\langle e_2, \rho, \sigma_2 \rangle \Downarrow \langle v_2, \sigma_3 \rangle$	$\ell_1 \notin \text{dom } \sigma_3 \quad \ell_2 \notin \text{dom } \sigma_3 \quad \ell_1 \neq \ell_2$	
$\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle \Downarrow \langle \text{PAIR}(\ell_1, \ell_2), \sigma_3 \{ \ell_1 \mapsto v_1, \ell_2 \mapsto v_2 \} \rangle$		
	<b>CAR</b>	<b>CDR</b>
$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{car}), \sigma_1 \rangle$	$\langle e, \rho, \sigma_0 \rangle \Downarrow \langle \text{PRIMITIVE}(\text{cdr}), \sigma_1 \rangle$	
$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{PAIR}(\ell_1, \ell_2), \sigma_2 \rangle$	$\langle e_1, \rho, \sigma_1 \rangle \Downarrow \langle \text{PAIR}(\ell_1, \ell_2), \sigma_2 \rangle$	
$\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle \sigma_2(\ell_1), \sigma_2 \rangle$	$\langle \text{APPLY}(e, e_1), \rho, \sigma_0 \rangle \Downarrow \langle \sigma_2(\ell_2), \sigma_2 \rangle$	

Figure 2.11: Summary of operational semantics (primitives)

### 2.16.13 Using the operational semantics with laws and syntactic sugar

42. *Proof or refutation of algebraic laws for cdr.* Algebraic laws can often be proven by appeal to other algebraic laws, but eventually some proofs have to appeal to the operational semantics. This exercise explores the algebraic law for `cdr`.

- (a) The operational semantics for  $\mu$ Scheme includes rules for `cons`, `car`, and `cdr`. Assuming that  $x$  and  $xs$  are variables and are defined in  $\rho$ , use the operational semantics to prove that

$$(\text{cdr } (\text{cons } x \text{ } xs)) = xs$$

### §2.16. Exercises

195

- (b) Use the operational semantics to prove or disprove the following conjecture: if  $e_1$  and  $e_2$  are arbitrary expressions, then in any context in which the evaluation of  $e_1$  terminates and the evaluation of  $e_2$  terminates, the evaluation of  $(\text{cdr } (\text{cons } e_1 \text{ } e_2))$  terminates, and

$$(\text{cdr } (\text{cons } e_1 \text{ } e_2)) = e_2$$

The conjecture says that in any state, for any  $e_1$  and  $e_2$ , evaluating  $(\text{cdr } (\text{cons } e_1 \text{ } e_2))$  produces the same value as evaluating  $e_2$  would have.

43. *Proof of an algebraic law for if.*  $\mu$ Scheme's `if` expressions participate in many algebraic laws. Use the operational semantics to prove that

$$(\text{if } e_1 (\text{if } e_2 e_3 e_4) (\text{if } e'_2 e_3 e_4)) = (\text{if } (\text{if } e_1 e_2 e'_2) e_3 e_4)$$

44. *Proof of validity of desugaring for let.* Section 2.13.1 claims that `let` can be desugared into `lambda`:

$$(\text{let } ([x_1 \text{ } e_1] \cdots [x_n \text{ } e_n]) e) \triangleq ((\text{lambda } (x_1 \cdots x_n) e) e_1 \cdots e_n).$$

Using the operational semantics, prove that the claim is a good one for the case where  $n = 1$ . That is, prove that for any  $x_1, e_1, e, \rho$ , and  $\sigma$ , if

$$\langle \text{LET}(\langle x_1, e_1 \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle,$$

then there exists a  $\sigma''$  such that

$$\langle \text{APPLY}(\text{LAMBDA}(\langle x_1 \rangle, e), e_1), \rho, \sigma \rangle \Downarrow \langle v, \sigma'' \rangle.$$

Furthermore, show that if the choice of  $\ell_1 \notin \text{dom } \sigma$  is made in same way in both derivations,  $\sigma'$  and  $\sigma''$  are the same.

45. *Proof of validity of desugaring for let\*.* Section 2.13.1 claims that `let*` can be desugared as follows:

$$\begin{aligned} (\text{let* } () & e) \triangleq e \\ (\text{let* } ([x_1 \text{ } e_1] \cdots [x_n \text{ } e_n]) e) & \triangleq \\ & (\text{let } ([x_1 \text{ } e_1]) (\text{let* } ([x_2 \text{ } e_2] \cdots [x_n \text{ } e_n]) e)) \end{aligned}$$

Using the same technique as in Exercise 44, prove that these rules are a good desugaring of `let*` into `let`.

$$\begin{array}{c}
 \text{DEFINEOLDGLOBAL} \\
 \frac{x \in \text{dom } \rho}{\boxed{\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}}
 \end{array}
 \quad
 \begin{array}{c}
 \text{DEFINENEWGLOBAL} \\
 \frac{x \notin \text{dom } \rho \quad \ell \notin \text{dom } \sigma}{\langle \text{VAL}(x, e), \rho, \sigma \rangle \rightarrow \langle \rho, \sigma' \{ \rho(x) \mapsto v \} \rangle}
 \end{array}$$
  

$$\begin{array}{c}
 \text{DEFINEFUNCTION} \\
 \frac{\langle \text{VAL}(f, \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e)), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}{\langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}
 \end{array}$$
  

$$\begin{array}{c}
 \text{EVALEXP} \\
 \frac{\langle \text{VAL}(\text{it}, e), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}{\langle \text{EXP}(e), \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle}
 \end{array}$$

Figure 2.12: Summary of operational semantics (definitions)

#### 2.16.14 Using the operational semantics to explore language design

46. *Alternate semantics for val.* In both Scheme and  $\mu$ Scheme, when `val`'s left-hand side is already bound, `val` behaves like `set`. If `val` instead *always* created a new binding, the semantics would be simpler.
- (a) Express the operational semantics of such a `val` by writing a `DEFINE-GLOBAL` rule.
  - (b) Write a  $\mu$ Scheme program that detects whether `val` uses the Scheme semantics or the new semantics. Explain how it works.
  - (c) Compare the two ways of defining `val`. Think about how they affect code and coding style. Which design do you prefer, and why?

47. *Sensible restrictions on recursive val.* The behavior of the `DEFINENEWGLOBAL` rule may strike you as rather odd, as it permits a “definition” like

*⟨transcript 97a⟩* +≡  
 -> (`val` u u)

for a previously undefined `u`. The definition is valid, and `u` has a value, although the value is not specified. Similar behavior is typical of a number of dynamically typed languages, such as Awk, Icon, and Perl, in which a new variable—with a well-specified value, even—can be called into existence just by referring to it.

In  $\mu$ Scheme, the `DEFINENEWGLOBAL` rule makes it easy to define recursive functions, as explained on page 153. But you might prefer a semantics in which whenever  $u \notin \text{dom } \rho$ ,  $(\text{val } u u)$  is rejected.

*⟨imaginary transcript 196b⟩* ≡  
 -> (`val` u u)  
 Run-time error: variable `u` not found

Rewrite the semantics of  $\mu$ Scheme so that  $(\text{val } u u)$  and similarly disturbing expressions are rejected, but it is still possible to define recursive functions.

48. *Semantics in which every variable is always defined.*  $\mu$ Scheme's `val` definition distinguishes an undefined global variable from a global variable that is bound to the empty list. Suppose this distinction is eliminated, and that an undefined global variable behaves exactly as if it were bound to the empty list. Is it possible to write a short  $\mu$ Scheme program that gives a different answer under the new treatment? If so, write such a program. If not, explain why not.

49. *Operational semantics for short-circuit `&&`.* Section 2.13.3 proposes syntactic sugar for short-circuit conditionals. To know if the syntactic sugar is any good, we have to have a semantics in mind. Use rules of operational semantics to specify how `&&` should behave. That is, pretending that *binary* short-circuit `&&` is actual abstract syntax and that  $(\&\& e_1 e_2)$  is a valid expression of  $\mu$ Scheme, write rules for the evaluation of `&&` expressions.

50. *Operational semantics of mutation.* Write rules of operational semantics for full Scheme primitives `set-car!` and `set-cdr!`, which mutate locations in `cons` cells. To get started, revisit the rules for `CONS`, `CAR`, and `CDR` on page 153.

### 2.16.15 Metatheory

51. *Proof that variables don't alias.* Use the operational semantics to prove that variables in  $\mu$ Scheme cannot alias. That is, prove that the evaluation of a  $\mu$ Scheme program never constructs an environment  $\rho$  such that  $x$  and  $y$  are both defined in  $\rho$ ,  $x \neq y$ , and  $\rho(x) = \rho(y)$ .

*Hint:* it will help to prove that any location in the range of  $\rho$  is also in the domain of  $\sigma$ .

52. *Safe extension of environments.* Show that an environment can be extended with fresh variables without changing the results of evaluating an expression. In more detail,

- You are given  $e$  and  $\rho$  such that  $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma_1 \rangle$ .
- You are given  $\rho'$  such that  $\text{dom } \rho \subseteq \text{dom } \rho'$ , and for any  $x$  in  $e$ ,  $\rho'(x) = \rho(x)$ .
- You are given  $\sigma'$  such that  $\text{dom } \sigma \subseteq \text{dom } \sigma'$ , and for any  $\ell \in \text{dom } \sigma$ ,  $\sigma'(\ell) = \sigma(\ell)$ .
- Prove that there exists a  $\sigma'_1$  such that  $\langle e, \rho', \sigma' \rangle \Downarrow \langle v, \sigma'_1 \rangle$ , and that for any  $\ell \in \text{dom } \sigma$ ,  $\sigma'_1(\ell) = \sigma_1(\ell)$ .

Use structural induction on the derivation of  $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma_1 \rangle$ .

(This exercise is related to Exercise 9 on page 331 in Chapter 5.)

### 2.16.16 Implementing new syntax and syntactic sugar

53. *Syntactic sugar for short-circuit conditionals.* In full Scheme, `and` and `or` are macros that behave like the variadic `&&` and `||` operators defined in Section 2.13.3.

$$\begin{array}{ll} (\text{and}) & \equiv \#t \\ (\text{and } p) & \equiv p \\ (\text{and } p_1 \ p_2 \ \dots \ p_n) & \equiv (\text{if } p_1 \ (\text{and } p_2 \ \dots \ p_n) \ \#\text{f}) \end{array}$$

Implementing `or` requires *hygiene*: you must find a fresh variable  $x$  that does

not appear in any  $e_i$ .

$$\begin{aligned}
 (\text{or}) &\equiv \#f \\
 (\text{or } e) &\equiv e \\
 (\text{or } e_1 \dots e_n) &\equiv (\text{let } ([x \ e_1]) \ (\text{if } x \ x \ (\text{or } e_2 \dots e_n))), \\
 &\quad \text{where } x \text{ does not appear in any } e_i
 \end{aligned}$$

Reimplement  $\mu$ Scheme's and and or as follows:

- Remove and from the initial basis of  $\mu$ Scheme. Using the laws above, and emulating the example in Section G.7 on page S217, add a variadic, short-circuit and to  $\mu$ Scheme as syntactic sugar.
- Write an auxiliary function that is given a list of expressions and returns a variable  $x$  that does not appear in any of the expressions.
- Remove or from the initial basis of  $\mu$ Scheme, and using the laws above and your auxiliary function, add a variadic, short-circuit or to  $\mu$ Scheme as syntactic sugar.

To develop your understanding, and also to test your work, add two more parts:

- Write an expression of  $\mu$ Scheme that evaluates without error using both the original and and the new and, but produces different values depending on which version of and is used.
- Write an expression of  $\mu$ Scheme that evaluates without error using both the original or and the new or, but produces different values depending on which version of or is used.

54. *Syntactic sugar for records.* Implement the syntactic sugar for record described in Section 2.13.6, according to these rules:

```

(record r (f1 … fn)) ≡
  (define make-r (x1 … xn)
    (cons 'make-r (cons x1 (cons … (cons xn '())))))
  (define r? (x) (and (pair? x) (= (car x) 'make-r) …))
  (define r-f1 (x) (if (r? x) (car (cdr x)) (error …)))
  (define r-f2 (x) (if (r? x) (car (cdr (cdr x))) (error …)))
  :
  (define r-fn (x) (if (r? x)
    (car (cdr (cdr … (cdr x))))
    (error …)))

```

This exercise requires a lot of code. To organize it, I use these tricks:

- The record definition desugars into a *list* of definitions. It's not shown in the chapter, but  $\mu$ Scheme has a hidden, internal `mkDefs` function that turns a list of definitions into a single definition. I build the list of definitions like this:

```

<functions for desugaring record definitions 198>≡
  Deflist desugarRecord(Name recname, Namelist fieldnames) {
    return mkDL(recordConstructor(recname, fieldnames),
               mkDL(recordPredicate(recname, fieldnames),
                     recordAccessors(recname, 0, fieldnames)));
  }

```

- I build syntax for calls to the primitives `cons`, `car`, `cdr`, and `pair?`. For each of these  $\mu$ Scheme primitives, I define a C function, and it calls the literal primitive directly, like this:

```
(functions for desugaring record definitions 198) +≡
    static Exp carexp(Exp e) {
        return mkApply(mkLiteral(mkPrimitive(CAR, unary)), mkEL(e, NULL));
    }
```

- I define an auxiliary C function that generates  $\mu$ Scheme code that applies `cdr` to a list a given number of times.
- My C code builds syntax for a constructor function, a type predicate, and accessor functions. For each kind of function, I first build an expression that represents the body of the function, which I put in a local variable called `body`. I then use `body` in the `Def` that I return.

55. *Quasiquotation*. In Section 2.7.1, I put counter operations into a record. An alternative is to put them into an association list:

```
(transcript 97a) +≡
-> (val resettable-counter-from
  (lambda (x) ; create a counter
    (list2
      (list2 'step (lambda () (set x (+ x 1))))
      (list2 'reset (lambda () (set x 0))))))
```

Full Scheme offers a nicer way to write association lists:

```
(fantasy transcript 199c) ≡
-> (val resettable-counter-from
  (lambda (x) ; create a counter
    (quasiquote ((step (unquote (lambda () (set x (+ x 1))))))
      (reset (unquote (lambda () (set x 0)))))))
```

The `quasiquote` form works like the `quote` form—which is normally written using the tick mark '`'`—except that it recognizes `unquote`, and the unquoted expression is evaluated.

It might not be obvious that using `quasiquote` and `unquote` is any nicer than calling `list2` (or full Scheme's `list`). But full Scheme provides nice abbreviations: just as `quote` is normally written with a tick mark, `quasiquote` and `unquote` are normally written with a backtick and a comma, respectively:

```
(fantasy transcript 199c) +≡
-> (val resettable-counter-from
  (lambda (x) ; create a counter
    `((step ,(lambda () (set x (+ x 1))))
      (reset ,(lambda () (set x 0)))))
```

- Look at the implementation of `quote` in Section L.4.1 on page S314, which relies on functions `sSexp` and `parsesx` on pages S315 and S316. Emulating that code, write new functions `sQuasi` and `parsequasi` and use them to implement `quasiquote` and `unquote`.
- Look at the implementation of `getpar_in_context` in chunk S182d. Extend the function so that when `read_tick_as_quote` is set, it not only reads '`'` as `quote` but also reads '`,`' as `quasiquote` and `,` as `unquote`.

### 2.16.17 Implementing new primitives

- Scheme,  
S-expressions, and  
first-class functions
- 
- 200
56. *List construction.* Add the new primitive `list`, which should accept any number of arguments.
  57. *Application to a list of arguments constructed dynamically.* Add the new primitive `apply`, which takes as arguments a function and a list of values, and returns the results of applying the function to the values:

$$\begin{aligned} \langle e, \rho, \sigma_0 \rangle &\Downarrow \langle \text{PRIMITIVE}(\text{apply}), \sigma_1 \rangle \\ \langle e_1, \rho, \sigma_1 \rangle &\Downarrow \langle v, \sigma_2 \rangle \\ \langle e_2, \rho, \sigma_2 \rangle &\Downarrow \langle \text{PAIR}(\ell_1, \text{PAIR}(\ell_2, \dots, \text{PAIR}(\ell_n, \ell))), \sigma_3 \rangle \\ \sigma_3(\ell) &= \text{NIL} \quad \sigma_3(\ell_i) = v_i, \quad 1 \leq i \leq n \\ \underline{\langle \text{APPLY}(\text{LITERAL}(v), \text{LITERAL}(v_1), \text{LITERAL}(v_2), \dots, \text{LITERAL}(v_n)), \rho, \sigma_3 \rangle} &\Downarrow \langle v', \sigma_4 \rangle \\ \underline{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0 \rangle} &\Downarrow \langle v', \sigma_4 \rangle \\ &\quad (\text{APPLY-AS-PRIMITIVE}) \end{aligned}$$

58. *Implementation of mutation.* Add primitives `set-car!` and `set-cdr!`. Remember that `set-car!` does *not* change the value the `car` field of a cons cell; it replaces the contents of the location that the `car` field points to. You'll have it right if your mutations are visible through different variables:

```
<mutation transcript 200>≡  
-> (val q '())  
-> (val p '(a b c))  
-> (set q p)  
-> (set-car! p 'x)  
-> (car q)  
x
```

59. *Primitive to read S-expressions.* Add a read primitive (as in Exercise 33 of Chapter 1). You may find it helpful to call `p = getpar(...)`, followed by `mkPL(mkAtom(strtoname("quote")), mkPL(p, NULL))`.

Use the `read` primitive to build an interactive version of the metacircular interpreter in the Supplement.

### 2.16.18 Interpreter improvement

60. *Call tracing.* Instrument the interpreter so it traces calls and returns. Whenever a traced function is called, print its name (if any) and arguments. (If the name of a function is not known, print a representation of its abstract syntax.) When a traced function returns, print the function and its result. To help users match calls with returns, indent each call and return should by an amount proportional to the number of pending calls not yet returned. Choose one of the following two methods to indicate which functions to trace:

- (a) Provide primitives to turn tracing of individual functions on and off.
- (b) Use variable `&trace` as a “trace count.” (Simply look it up in the current environment.) While `&trace` is bound to a location containing a nonzero number, each call and return should decrement the trace count and print a line. If the trace count is negative, tracing runs indefinitely.

Test your work by tracing `length` (as shown on page 101). Also trace `sieve` and `remove-multiples`.

Printing the abstract syntax for a function provides good intuition, but it may be more helpful to print non-global functions in closure form.

- (c) When calling a closure, print it in closure form instead of printing its name. Which of the two methods is better?

## CHAPTER CONTENTS

---

5.1	NAMES AND ENVIRON- MENTS	310	5.7	SUMMARY	324
5.2	ABSTRACT SYNTAX AND VALUES	312	5.7.1	Key words and phrases	324
5.3	EVALUATION	315	5.7.2	Further reading	327
5.3.1	Evaluating definitions	318	5.8	EXERCISES	327
5.4	DEFINING AND EMBED- DING PRIMITIVES	319	5.8.1	Retrieval practice	328
5.5	NOTABLE DIFFERENCES BE- TWEEN ML AND C	321	5.8.2	Working with syntax and semantics	329
5.6	FREE AND BOUND VARI- ABLES	322	5.8.3	Higher-order functions, embedding, and projec- tion	330
			5.8.4	Proofs as data	331
			5.8.5	Free variables	331

## *Interlude: $\mu$ Scheme in ML*

**STUDENT:** You and Chris have spent ten years working on your compiler, and now you have this highly polished literate program, published as a book. What did you learn?

**HANSON:** C is a lousy language to write a compiler in.

Conversation with David R. Hanson, coauthor of  
*A Retargetable C Compiler: Design and Implementation*  
 (Fraser and Hanson 1995).

The interpreters in Chapters 1 to 4 are written in C, which has much to recommend it: C is relatively small and simple; it is widely known and widely supported; it has a perspicuous cost model in which it is easy to discover what is happening at the machine level; and it provides pointer arithmetic, which makes it a fine language in which to write a garbage collector. But as we move to more complicated and more ambitious bridge languages, C is less than ideal. In this and succeeding chapters, I therefore present interpreters written in the functional language Standard ML.

Standard ML is particularly well suited to symbolic computing, especially functions that operate on abstract-syntax trees. And an ML program can illustrate connections between language design, formal semantics, and implementations more clearly than a C program can. Some of the advantages of ML are detailed in the sidebar on page 308. In this chapter, and also in Appendices I and J, I develop infrastructure used to write interpreters in ML. So that you can focus on the new infrastructure, I apply it to a familiar language: I present another interpreter for  $\mu$ Scheme.

The  $\mu$ Scheme interpreter in this chapter has the same structure as the interpreter in Chapter 2. Like that interpreter, it has environments, abstract syntax, primitives, an evaluator for expressions, and an evaluator for definitions. Many details are as similar as I can make them, but many are not: even more than wanting the interpreters to look alike, I want my ML code to look like ML and my C code to look like C.

The ML code will be easier to read if you know my programming conventions.

- My naming conventions are the ones recommended by the SML'97 Standard Basis Library (Gansner and Reppy 2002). Names of types are written in lowercase letters with words separated by underscores, like `exp`, `def`, or `unit_test`. Names of functions and variables begin with lowercase letters, like `eval` or `evaldef`, but long names may be written in “camel case” with a mix of uppercase and lowercase letters, like `processTests` instead of the

*Helpful properties of the ML family of languages*

- ML is *safe*: there are no unchecked run-time errors, which means there are no bad behaviors that are entirely up to the programmer to avoid.
- Like Scheme, ML is naturally *polymorphic*. Polymorphism simplifies everything. For example, unlike the C code in Chapters 1 to 4, our ML code uses one representation of lists and one length function. The C code in Appendix F defines three different types of streams, each with its own get function; the ML code in Appendix I defines one type of stream and one streamGet function. As a final example, Impcore and  $\mu$ Scheme have different process\_tests functions for running unit tests. The ML version in Section I.3 is polymorphic in the environment or environments, and it is shared among all interpreters.

Polymorphic functions don't just save code. When you see one, polymorphic function, you know that it always does the same thing—polymorphism makes it easier for you to see what is the same and what is different.

- Unlike Scheme, ML uses a static type system to guarantee the internal consistency of polymorphic data structures. For example, if one element of a list is a function, every element of that list is a function. Such a list can be passed to a polymorphic function like length, and the same list can be used in a context where only a list of values is acceptable, like the implementation of function application in eval. And all of this happens without requiring any variable declarations or type annotations to be written in the code.

If this talk of polymorphism mystifies you, don't worry; polymorphism in programming languages is an important topic in its own right. Chapter 6 introduces and defines polymorphism; Chapter 7 shows how ML provides polymorphism without type annotations; and Chapter 9 shows a form of polymorphism that makes type annotations practical.

- Like Scheme, ML provides first-class, nested functions, and its initial basis contains useful higher-order functions. These functions help me simplify and clarify the code. For example, in C, running a list of unit tests back-to-front requires two special-purpose functions: one for Impcore and one for  $\mu$ Scheme. In ML, I just use foldr.
- To detect and signal errors, ML provides *exception handlers* and *exceptions*, which are more flexible and easier to use than C's setjmp and longjmp.
- Finally, least familiar but most important, ML provides native support for *algebraic data types*, which I use to represent both abstract syntax and values. These types provide *value constructors* like the IFX or APPLY used in previous chapters, but instead of switch statements, ML provides *pattern matching*. Using pattern matching enables me to write function definitions that look a lot like algebraic laws. Such definitions are easier to follow than C code. A detailed explanation accompanies the definition of function valueString on page 314.

Algebraic data types are explained in detail in Chapter 8, which presents the bridge language  $\mu$ ML. In essence,  $\mu$ ML is  $\mu$ Scheme plus type inference and algebraic data types. Like all chapters in this book, Chapter 8 presents both the use and the implementation of algebraic data types. To understand the implementation, you will want to understand implementations in Chapters 6 and 7, but to understand how algebraic data types are used, you could jump ahead and look at Section 8.1 now.

C-style `process_tests`. (Rarely, I may use an underscore in the name of a local variable.)

Names of exceptions are capitalized, like `NotFound` or `RuntimeError`, and they use camel case. Names of *value constructors*, which identify alternatives in algebraic data types, are written in all capitals, possibly with underscores, like `IFX`, `APPLY`, or `CHECK_EXPECT` (just like enumeration literals in C).

- If you happen to be a seasoned ML programmer, you'll notice something missing: the interpreter is not decomposed into modules. Modules are a chapter in their own right (Chapter 9), but compared to what's in Chapter 9, Standard ML's module system is complicated and hard to understand. To avoid having to explain it, I define no modules—although I do use “dot notation” to select functions that are defined in Standard ML's predefined modules. Avoiding module definitions gives you a good chance to digest this chapter even if your only previous experience with functional languages is your work with  $\mu$ Scheme in Chapter 2.

*Interlude:  
 $\mu$ Scheme in ML*

309

Because I don't use ML modules, I have no formal way to talk about interfaces and to distinguish interfaces from implementations. I work around this problem using a literate-programming trick: I put the types of functions and values, which is mostly what ML interfaces describe, in boxes preceding the implementations. This technique makes it possible to present an interface formally just before I present its implementation. The Noweb processor ensures that the material in the boxes is checked by the ML compiler.

A final aspect of my ML code is less a matter of convention than of necessity: ML is persnickety about the order in which definitions appear, and it has miserable support for mutually recursive data definitions. These properties are limitations that I have to work around.

The relevant differences between ML and C start with syntactic categories: at top level, C has both *declarations* and *definitions*, but ML has only definitions. C's syntactic structure makes it possible to be relatively careless about the order in which things appear: declare all your structures (probably in `typedefs`) in any order you like, and you can define them in just about any order you like. Then declare all your functions in any order you like, and you can define them in any order you like. Even if your data structures and functions are mutually recursive. Of course there are drawbacks: not all variables are guaranteed to be initialized, and global variables can be initialized only in limited ways. And it's too easy to define mutually recursive data structures that allow you to chase pointers forever.

ML's syntactic structure requires you to be careful about the order in which things appear: the definition of a name may appear only *after* the definitions of the other names it refers to. Of course there are benefits: every definition initializes its name, and initialization may use any valid expression, including `let` expressions, which in ML can contain nested definitions. And unless your code contains assignments to mutable reference cells, it is impossible to define circular data structures that allow you to chase pointers forever. As a consequence, unless a structurally recursive function fetches the contents of mutable reference cells, it is guaranteed to terminate. ML's designers thought this guarantee was more important than the convenience of writing data definitions in many orders. (And to be fair, using ML modules makes it relatively convenient to get things in the right order.) In this book, groups of related definitions are put into Noweb code chunks like `<support for names and environments 310a>`. I carefully stick together larger and larger chunks until eventually I wind up with a complete interpreter in a chunk like `<mlscheme.sml S373a>`, which appears in the Supplement.

What about mutually recursive data? Suppose for example, that type `exp` refers to value and type `value` refers to `exp`? Mutually recursive definitions like `exp` and `value` are written together, adjacent in the source code, connected with the keyword `and`. (You won't see `and` often, but when you do, please remember this: it means mutual recursion, never a Boolean operation.)

Mutually recursive function definitions provide more options: you can join them with `and`, but it is usually more convenient and more idiomatic to nest one inside the other using a `let` binding—you would use `and` only when both mutually recursive functions need to be called by some third, client function. When I use mutual recursion, I identify the technique I use. Now, on to the code!

## 5.1 NAMES AND ENVIRONMENTS, WITH INTRODUCTION TO ML

In my C code, names are an abstract type, and two names are equal if any only if they are the same pointer, so I can compare them using C's built-in `==` operator. In ML, strings are immutable and can be meaningfully compared using ML's built-in `=` operator, so I choose to represent names as strings.

**310a.** *(support for names and environments 310a)*  $\equiv$

(S237a) 310b▷

```
type name = string
```

The type syntax here is like C's `typedef`; it defines a type by *type abbreviation*.

In our C implementation of  $\mu$ Scheme, an environment  $\rho$  binds names to locations containing values; each location is represented by a pointer of C type `Value *`. In ML, such a pointer has type `value ref`. Like a C pointer, an ML `ref` can be read from and written to, but unlike a C pointer, it doesn't support arithmetic.

In C, the code that looks up or binds a name has to know what kind of thing a name stands for; that's why in the Impcore interpreter, we needed one set of environment functions for value environments  $\xi$  and  $\rho$  but a different set for function environment  $\phi$ . In ML, code for lookup and binding is independent of what a name stands for; the code is *polymorphic*. In the rest of this book, the *same* polymorphic environment code is used to implement environments that hold locations, values, and types.

In ML, as in  $\mu$ Scheme, polymorphism happens naturally, but unlike  $\mu$ Scheme, ML has a static type system, and polymorphism is reflected in the types. An ML environment has type `'a env`, where `'a` is called a *type parameter* or *type variable* and stands for an unknown type. (You'll find much more about type parameters, including an entire language devoted to them, in Section 6.6.) Such an environment is a polymorphic data structure, like a Scheme association list; it binds each name to a value of type `'a`. The environment type, like any type that takes a type parameter, can be *instantiated* at any type; instantiation substitutes a known type for every occurrence of the type parameter. To get an environment binding names to mutable locations, we instantiate type `'a env` using `'a = value ref`; the resulting type is called `value ref env`.

My code for environments doesn't just take advantage of polymorphism; it also takes advantage of ML's native support for lists and pairs. While in C, I represent an environment as a pair of lists, in ML, it's easier and simpler to represent an environment as a list of pairs: the ML expression `(e1, e2)` is a pair containing the value of `e1` and the value of `e2`. The type of the list is written `(name * 'a) list`. The type of a single pair is written `name * 'a`; a pair `(e1, e2)` has that type if `e1` has type `name` and `e2` has type `'a`.

**310b.** *(support for names and environments 310a)*  $+ \equiv$

(S237a) ▷ 310a 311a▷

```
type 'a env = (name * 'a) list
```

Semantics	Concept	Interpreter	
$d$	Definition	<code>def</code> (page 313)	
$e$	Expression	<code>exp</code> (page 313)	
$x$	Name	<code>name</code> (page 310)	
$v$	Value	<code>value</code> (page 313)	
$\ell$	Location	<code>value ref</code> ( <code>ref</code> is built into ML)	§5.1
$\rho$	Environment	<code>value ref env</code> (page 310)	<i>Names and environments</i>
$\sigma$	Store	Machine memory (the ML heap)	311
$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$	Expression evaluation	$\text{eval}(e, \rho) = v$ , with $\sigma$ updated to $\sigma'$ (page 316)	
$\langle d, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$	Definition evaluation	$\text{evaldef}(d, \rho) = (\rho', s)$ , with $\sigma$ updated to $\sigma'$ (page 318)	
$x \in \text{dom } \rho$	Definedness	<code>find</code> ( $x, \rho$ ) terminates without raising an exception (page 311)	
$\rho(x)$	Location lookup	<code>find</code> ( $x, \rho$ ) (page 311)	
$\sigma(\rho(x))$	Value lookup	<code>!(find (x, \rho))</code> (page 311)	
$\rho\{x \mapsto \ell\}$	Binding	<code>bind</code> ( $x, \ell, \rho$ ) (page 312)	
$\sigma\{\ell \mapsto v\}$ , where $\ell \notin \text{dom } \sigma$	Allocation	call <code>ref</code> $v$ ; the result is $\ell$	
$\sigma\{\ell \mapsto v\}$ , where $\ell \in \text{dom } \sigma$	Store update	$\ell := v$	

Table 5.1: Correspondence between  $\mu$ Scheme semantics and ML code

The 'a on the left of the = sign is called a *type parameter*; in use, it can be replaced with the type of whatever a name stands for.

The empty environment is represented by the empty list. In ML, that's written using square brackets. The `val` form is like  $\mu$ Scheme's `val` form.

**311a.** *(support for names and environments 310a)*  $\vdash \equiv$   
`val emptyEnv = []`

(S237a)  $\triangleleft$  310b 311b  $\triangleright$

`emptyEnv : 'a env`

(The phrase in the box is adapted from declarations that appear in interfaces to ML modules; through some Noweb hackery, such phrases are checked by the ML compiler.)

To look up a name in an environment, I define a function `find`, which is closely related to the `find` from Chapter 2: it returns whatever is in the environment, which has type 'a. If nothing is found, it *raises an exception*. Raising an exception is a lot like the `throw` operator in Chapter 3; it is roughly analogous to `longjmp`. The exceptions I use are listed in Table 5.2.

**311b.** *(support for names and environments 310a)*  $\vdash \equiv$   
`exception NotFound of name`  
`fun find (name, []) = raise NotFound name`  
`| find (name, (x, v)::tail) = if name = x then v else find (name, tail)`

(S237a)  $\triangleleft$  311a 312a  $\triangleright$

`find : name * 'a env -> 'a`

The `fun` definition form is ML's analog to `define`, but unlike  $\mu$ Scheme's `define`, it uses multiple clauses with *pattern matching*. Each clause is like an algebraic law. The first clause says that calling `find` with an empty environment raises an exception; The second clause handles a nonempty environment. The infix :: is ML's way of writing `cons`, and it is pronounced "cons."

NotFound	A name was looked up in an environment but not found there.
BindListLength	A call to bindList tried to extend an environment, but it passed two lists (names and values) of different lengths.
RuntimeError	Something else went wrong during evaluation, i.e., during the execution of eval.

---

Table 5.2: Exceptions defined especially for this interpreter

If I want to check  $x \in \text{dom } \rho$ , I use function isbound.

312a. *(support for names and environments 310a)*  $\equiv$  (S237a) ◁ 311b 312b ▷

```
fun isbound (name, []) = false
| isbound (name, (x, v)::tail) = name = x orelse isbound (name, tail)
```

Again using ::, function bind adds a new binding to an existing environment. Unlike Chapter 2's bind, it does not allocate.

312b. *(support for names and environments 310a)*  $\equiv$  (S237a) ◁ 312a 312c ▷

```
fun bind (name, v, rho) =
  (name, v) :: rho
```

bind : name \* 'a \* 'a env -> 'a env

While the list-of-pairs representation is cleaner than the pair of lists used in our C code, functions that operate on two lists are still useful. Function bindList adds a sequence of bindings to an environment; it is used to implement  $\mu$ Scheme's let and lambda. If the lists aren't the same length, it raises another exception. Function bindList resembles Chapter 2's bindallocList, but it does not allocate. Related function mkEnv manufactures a new environment given just a list of names and 'a's.

312c. *(support for names and environments 310a)*  $\equiv$  (S237a) ◁ 312b 312d ▷

```
bindList : name list * 'a list * 'a env -> 'a env
exception BindListLength
mkEnv      : name list * 'a list -> 'a env
```

bindList : name list \* 'a list \* 'a env -> 'a env

mkEnv : name list \* 'a list -> 'a env

exception BindListLength

fun bindList (x::vars, v::vals, rho) = bindList (vars, vals, bind (x, v, rho))
| bindList ([], [], rho) = rho
| bindList \_ = raise BindListLength

fun mkEnv (xs, vs) = bindList (xs, vs, emptyEnv)

Finally, environments can be composed using the + operator. To avoid conflicts with ML's addition function, I define a new infix function called <+>. The @ function is ML's way of writing append.

312d. *(support for names and environments 310a)*  $\equiv$  (S237a) ◁ 312c

```
(* composition *)
infix 6 <+>
fun pairs <+> pairs' = pairs' @ pairs
```

<+> : 'a env \* 'a env -> 'a env

Function <+> obeys the algebraic law  $\text{bindList}(xs, vs, \rho) = \rho <+> \text{mkEnv}(xs, vs)$ .

## 5.2 ABSTRACT SYNTAX AND VALUES

An abstract-syntax tree can contain a literal value. A value, if it is a closure, can contain an abstract-syntax tree. These two types are therefore mutually recursive, so I define them together, using and.

These particular types use as complicated a nest of definitions as you'll ever see. The keyword datatype defines a new algebraic datatype; the keyword withtype

introduces a new type abbreviation that is mutually recursive with the datatype. The first group of and keywords define additional algebraic datatypes, and the second group of and keywords define additional type abbreviations. Everything in the whole nest is mutually recursive.

**313a.** *(definitions of exp and value for  $\mu$ Scheme 313a)≡*

(S365c)

```

datatype exp = LITERAL of value
  | VAR      of name
  | SET      of name * exp
  | IFX      of exp * exp * exp
  | WHILEX   of exp * exp
  | BEGIN    of exp list
  | APPLY    of exp * exp list
  | LETX     of let_kind * (name * exp) list * exp
  | LAMBDA   of lambda

and let_kind = LET | LETREC | LETSTAR

and value = SYM      of name
  | NUM      of int
  | BOOLV   of bool
  | NIL
  | PAIR    of value * value
  | CLOSURE of lambda * value ref env
  | PRIMITIVE of primitive

withtype primitive = exp * value list -> value (* raises RuntimeError *)
and lambda   = name list * exp

```

§5.2  
Abstract syntax  
and values

313

The representations are the same as in C, with these exceptions:

- In a LETX expression, the bindings are represented by a list of pairs, not a pair of lists—just like environments.
- In the representation of a primitive function, there’s no need for an integer tag. As shown in Section 5.4 below, ML’s higher-order functions makes it easy to create groups of primitives that share code. Tags would be useful only if we wanted to distinguish one primitive from another when printing.
- None of the fields of exp, value, or lambda is named. Instead of being referred to by name, those fields are referred to by pattern matching.

A primitive function that goes wrong raises the `RuntimError` exception, which is the ML equivalent of calling `runerror`.

True definitions, unit tests, and extended definitions are all as in the C code, except again, fields are not named. These true definitions are used only in  $\mu$ Scheme; the unit tests are shared with the untyped language  $\mu$ Smalltalk (Chapter 10), and the extended definitions are shared with all other languages.

**313b.** *(definition of def for  $\mu$ Scheme 313b)≡*

(S365c)

```

datatype def = VAL      of name * exp
  | EXP      of exp
  | DEFINE   of name * lambda

```

emptyEnv	311a
type env	310b
type name	310a

The rest of this section defines utility functions on values.

### *String conversion*

ML does not provide a function like `printf`, but it provides plenty of primitives for creating, manipulating, and combining strings. The role of Chapter 1’s extensible print function is therefore played by a group of string-conversion functions. Function `valueString`, which converts an ML value to a string, is shown here. The other string-conversion functions are relegated to the Supplement.

A  $\mu$ Scheme value is an S-expression, and converting it to a string is mostly straightforward. The only tricky bit is printing a list made up of cons cells (PAIRs); function tail is mutually recursive with valueString, by being defined inside valueString, and it implements the same list-printing algorithm as the C code. (The algorithm, which goes back to McCarthy, is implemented by C function printtail on page S322.)

**314.** *(definition of valueString for  $\mu$ Scheme, Typed  $\mu$ Scheme, and nano-ML 314) ≡ (S365c)*

```
fun valueString (SYM v)    = v
| valueString (NUM n)   = intString n
| valueString (BOOLV b) = if b then "#t" else "#f"
| valueString (NIL)     = "()"
| valueString (PAIR (car, cdr)) =
  let fun tail (PAIR (car, cdr)) = " " ^ valueString car ^ tail cdr
      | tail NIL = ")"
      | tail v = " ." ^ valueString v ^ ")"
  in "(" ^ valueString car ^ tail cdr
  end
| valueString (CLOSURE _) = "<function>"
| valueString (PRIMITIVE _) = "<function>"
```

Function valueString provides our first comprehensive demonstration of pattern matching over the algebraic data type value. Function valueString takes one argument and is implemented using a case analysis on that argument, but the case analysis is defined by pattern matching. There is a case for each datatype constructor of the value type; the left-hand side of each case contains a *pattern match* that applies the constructor to a variable, to a pair of variables, or to the special “wildcard” pattern \_ (the underscore). All the variables in the pattern match, except the wildcard, are introduced into the environment and are available for use on the right-hand side of the =, just as if they had been bound by a  $\mu$ Scheme let or ML val.

From the point of view of a C programmer, it’s as if the pattern match combines a switch statement with assignment to local variables. In the matches for BOOLV and PAIR, for example, I like writing b and car a lot better than the v.u.boolv and v.u.pair.car that I have to write in C. And I really like that the variables b and car can be used only where they are meaningful—in C, I can write v.u.pair.car when referring to *any* value v, but if v isn’t a pair, the reference is meaningless (and to write it is an *unchecked* run-time error).

### Embedding and projection

A  $\mu$ Scheme S-expression can represent a symbol, number, Boolean, list, or function, and each of these values has to be represented by a value inside the interpreter. And just as in Chapter 2, every  $\mu$ Scheme value has the same ML type, which is different from what native ML code expects for lists, Booleans, and so on. So it is sometimes necessary to convert between them:

- When it sees a quote mark and brackets, the parser defined in the Supplement produces a native ML list of S-expressions. This list needs to be converted to a  $\mu$ Scheme list represented by a combination of PAIR and NIL.
- To implement a  $\mu$ Scheme conditional, the interpreter wants to use a native ML if expression, just as the interpreter in Chapter 1 uses a native C if statement. But ML’s if wants to test a condition of type bool, and a  $\mu$ Scheme condition has type value.

These issues are resolved by converting values from one language to another. Similar issues arise whenever one language is used to implement or describe another,

and such situations are so common that to keep the languages straight, we typically use a bit of jargon:

- The language being implemented or described—in our case,  $\mu$ Scheme—is called the *object language*.
  - The language doing the describing or implementation—in our case, ML—is called the *metalanguage*. (The name ML actually stands for “metalanguage.”)

To convert, say, an integer between object language and metalanguage, we use a pair of functions called *embedding* and *projection*. The embedding puts a metalanguage integer into the object language, converting an `int` into a value. The projection extracts a metalanguage integer from the object language, converting a value into an `int`. If the value can't be interpreted as an integer, the projection fails.<sup>1</sup> The embedding/projection pair for integers is defined as follows:

**315a.** ⟨utility functions on  $\mu$ Scheme, Typed  $\mu$ Scheme, and nano-ML values 315a⟩ $\equiv$  (S373a) 315b>

```

    fun embedInt n = NUM n
    fun projectInt (NUM n) = n
    | projectInt v =
        raise RuntimeError ("value " ^ valueString v ^ " is not an integer")

```

embedInt : int -> value projectInt : value -> int
--

Embedding and projection for Booleans is a little different; unlike some projection functions, `projectBool` is *total*: it always succeeds. The operational semantics of  $\mu$ Scheme treats any value other than `#f` as a true value, so by projecting every non-Boolean  $\mu$ Scheme value to `true`, `projectBool` reflects the semantics.<sup>2</sup>

**315b.** *(utility functions on  $\mu$ Scheme, Typed  $\mu$ Scheme, and nano-ML values 315a) +≡ (S373a) ≣ 315a 315c*

<pre>fun embedBool b = BOOLV b fun projectBool (BOOLV false) = false   projectBool _ = true</pre>	<pre>embedBool : bool -&gt; value projectBool : value -&gt; bool</pre>
---	--

Chapter 2 also contains a Boolean projection function, but without the jargon; there, the projection function is called `istru`.

A list of values can be embedded as a single value by converting ML's :: and [] to  $\mu$ Scheme's PAIR and NIL. The corresponding projection is left as Exercise 4.

**315c.** ⟨utility functions on  $\mu\text{Scheme}$ , Typed  $\mu\text{Scheme}$ , and nano-ML values 315a⟩ +≡ (S373a)  $\triangleleft$  315b

```
fun embedList []      = NIL          embedList : value list -> value
| embedList (h::t) = PAIR (h, embedList t)
```

### 5.3 EVALUATION

The machinery above is enough to write an evaluator, which takes an expression and an environment and produces a value. To make the evaluator easy to write, I do most of the work of evaluation in the nested function `ev`, which inherits the environment `rho` from the outer function `eval`. Most AST nodes are evaluated in the same environment as their parents, and each such node is evaluated by passing

<sup>1</sup>In general, we embed a smaller set into a larger set. Embeddings don't fail, but projections might. A mathematician would say that an embedding  $e$  of  $S$  into  $S'$  is an injection from  $S \rightarrow S'$ . The corresponding projection  $\pi_e$  is a left inverse of the embedding; that is  $\pi_e \circ e$  is the identity function on  $S$ . There is no corresponding guarantee for  $e \circ \pi_e$ ; for example,  $\pi_e$  may be undefined ( $\perp$ ) on some elements of  $S'$ , or  $e(\pi_e(x))$  may not equal  $x$ .

<sup>2</sup>A Boolean projection function formalizes the concepts of “truthy” and “falsy” found in languages like JavaScript: a value is truthy if it projects to `true` and falsy if it projects to `false`.

**BOOLV**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a  
**CLOSURE**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a  
**intString** S238f  
**NIL**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a  
**NUM**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a  
**PAIR**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a  
**PRIMITIVE**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a  
**RuntimeError** S366c  
**SYM**,  
 in nano-ML 415b  
 in Typed  $\mu$ Scheme 370b  
 in  $\mu$ Scheme 313a

it to `ev`, which lets `rho` be implicit. The first case of `ev` is the evaluation of a literal value `v`, which evaluates to itself.

**316a.** *(definitions of eval and evaldef for  $\mu$ Scheme 316a)*  $\equiv$

```
fun eval (e, rho) =
  let fun ev (LITERAL v) = v
      (more alternatives for ev for  $\mu$ Scheme 316b)
    in ev e
  end
```

eval : exp * value ref env -> value
ev : exp -> value

318b▷

Functions `eval` and `ev` are mutually recursive.

To evaluate VAR and SET, we need environment lookup. The environment `rho` binds each name to a mutable reference cell, which is ML's version of a pointer to a location allocated on the heap. In C, locations are read and written using special pointer syntax (the `*` syntax), but in ML, locations are read and written using functions `!` and `:=`, which are in the initial basis of Standard ML. (The `:=` symbol, like the `+` symbol, is an ordinary ML function which is declared to be *infix*.)

**316b.** *(more alternatives for ev for  $\mu$ Scheme 316b)*  $\equiv$

```
| ev (VAR x) = !(find (x, rho))
| ev (SET (x, e)) =
  let val v = ev e
  in find (x, rho) := v;
  v
end
```

(316a) 316c▷

The right-hand side of SET, here called `e`, is evaluated in the same environment as the SET, so I evaluate it using `ev`.

To evaluate IF and WHILE, we interpret a  $\mu$ Scheme value as a Boolean. That is the job of the projection function.

**316c.** *(more alternatives for ev for  $\mu$ Scheme 316b)*  $+ \equiv$

```
| ev (IFX (e1, e2, e3)) = ev (if projectBool (ev e1) then e2 else e3)
| ev (WHILEX (guard, body)) =
  if projectBool (ev guard) then
    (ev body; ev (WHILEX (guard, body)))
  else
    BOOLV false
```

(316a) ▷ 316b 316d▷

The code used to evaluate a while loop is nearly identical to the rule for lowering while loops in Chapter 3 (page 216).

To evaluate BEGIN, I define an auxiliary function `b`, which uses an accumulating parameter `lastval` to remember the value of the last expression. To get the right result for an empty BEGIN, I initialize `lastval` to false.

**316d.** *(more alternatives for ev for  $\mu$ Scheme 316b)*  $+ \equiv$

```
| ev (BEGIN es) =
  let fun b (e:::es, lastval) = b (es, ev e)
      | b ([], lastval) = lastval
    in b (es, BOOLV false)
  end
```

(316a) ▷ 316c 316e▷

To evaluate LAMBDA, I capture a closure, which is as simple as in C.

**316e.** *(more alternatives for ev for  $\mu$ Scheme 316b)*  $+ \equiv$

```
| ev (LAMBDA (xs, e)) = CLOSURE ((xs, e), rho)
```

(316a) ▷ 316d 317a▷

To evaluate an application, I begin by evaluating the expression `f` that is in the function position. What's next depends on whether `f` evaluates to a primitive, a closure, or something else. To apply a primitive function, as in C, I apply it to the syntax `e` and to the values of the arguments.

**317a.** *(more alternatives for ev for μScheme 316b)* +≡  
 | ev (e as APPLY (f, args)) =  
   (case ev f  
     of PRIMITIVE prim => prim (e, map ev args)  
     | CLOSURE clo   => *⟨apply closure clo to args 317b⟩*  
     | v => raise RuntimeError ("Applied non-function " ^ valueString v)  
   )  
 (316a) ◁ 316e 317c ▷

### §5.3. Evaluation

317

This code uses a nice feature of ML pattern matching: in the pattern match, phrase `e as` names the whole value being matched. The pattern `e as APPLY (f, args)` means “match this pattern if the given argument is an `APPLY` node, call the children of the node `f` and `args`, and call the entire node `e`.”

To apply a  $\mu$ Scheme closure, I create fresh locations to hold the values of the actual parameters. In C, I use function `allocate`; in ML, the built-in function `ref` does the same thing: create a new location and initialize its contents with a value. The ML expression `map ref` actuals does half the work of C’s `bindallocList`; function `bindList` does the other half.

**317b.** *(apply closure clo to args 317b)* ≡  
 let val ((formals, body), savedrho) = clo  
   val actuals = map ev args  
 in eval (body, bindList (formals, map ref actuals, savedrho))  
   handle BindListLength =>  
     raise RuntimeError ("Wrong number of arguments to closure; "  
                       "expected (" ^ spaceSep formals ^ ")")  
 end  
 (317a)

If the number of actual parameters doesn’t match the number of formal parameters, `bindList` raises the `BindListLength` exception, which `eval` catches using `handle`. The handler then raises `RuntimeError`. To give the programmer some clue what formal parameters were expected, I convert the list of formals to a string using function `spaceSep`.

To evaluate `LET`, it is easiest to unzip the list of pairs `bs` into a pair of lists (`names`, `values`); I use function `ListPair.unzip`, which is from the `ListPair` module in the Standard Basis Library. I then use `map` to apply both `ev` and then `ref` to each value. Rather than call `map` twice with `ev` and `ref` separately, as above, I apply `map` once with the composition `ref o ev`. The single `map` is more idiomatic.

**317c.** *(more alternatives for ev for μScheme 316b)* +≡  
 | ev (LETX (LET, bs, body)) =  
   let val (names, values) = ListPair.unzip bs  
   in eval (body, bindList (names, map (ref o ev) values, rho))  
   end  
 (316a) ◁ 317a 317d ▷

To evaluate `LETSTAR`, by contrast, it is easier to walk the bindings one pair at a time.

**317d.** *(more alternatives for ev for μScheme 316b)* +≡  
 | ev (LETX (LETSTAR, bs, body)) =  
   let fun step ((n, e), rho) = bind (n, ref (eval (e, rho)), rho)  
   in foldl step rho bs  
   end  
 (316a) ◁ 317c 318a ▷

APPLY	313a
applyChecking-Overflow	S242b
args,	
in molecule	S503d
in Typed μScheme	S398e
BEGIN	313a
bind	312b
bindList	312c
BindListLength	312c
BOOLV	313a
clo,	
in molecule	S503d
in Typed μScheme	S398e
CLOSURE	313a
ev,	
in molecule	S502b
in Typed μScheme	S398a
eval,	
in molecule	S502b
in Typed μScheme	S398a
find	311b
id	S263d
IFX	313a
LAMBDA	313a
LET	313a
LETSTAR	313a
LETX	313a
LITERAL	313a
PRIMITIVE	313a
projectBool	315b
RuntimeError	S366c
SET	313a
spaceSep	S239a
valueString	314
VAR	313a
WHILEX	313a

**318a.** *(more alternatives for ev for μScheme 316b) +≡* (316a) ◁ 317d

List.app : ('a -> unit) -> 'a list -> unit
--

```
| ev (LETX (LETREC, bs, body)) =
  let val (names, values) = ListPair.unzip bs
    (if any expression in values is not a lambda, reject the letrec S367a)
    val rho' =
      bindlist (names, map (fn _ => ref (unspecified())) values, rho)
    val updates = map (fn (n, e) => (n, eval (e, rho'))) bs
  in List.app (fn (n, v) => find (n, rho') := v) updates;
    eval (body, rho')
  end
```

### 5.3.1 Evaluating definitions

As in Chapter 2, the implementation of the rules for definitions is straightforward. Function evaldef takes a definition and an environment, and it returns a new environment and the interpreter's (string) response. (The C versions of evaldef in Chapters 1 and 2 *print* the response, but the ML code in this chapter is used not only for *μScheme*, but for statically typed languages in Chapters 6 to 8. For those languages, it is better to *return* the response from evaluation, so the response from evaluation can be combined with the response from type checking.)

When a definition introduces a new name, that definition is evaluated in an environment that *already* includes the name being defined. If the name *x* is not already bound, the *NotFound* exception is handled, and *x* is bound to a fresh location that is initialized with an unspecified value.

**318b.** *(definitions of eval and evaldef for μScheme 316a) +≡*

◁ 316a 318c ▷

withNameBound : name * value ref env -> value ref env
---

```
fun withNameBound (x, rho) =
  (find (x, rho); rho)
  handle NotFound _ => bind (x, ref (unspecified()), rho)
```

Given a *val* binding to name *x*, evaldef first uses withNameBound to make sure *x* is bound to a location in the environment. It then evaluates the right-hand side and stores the new value in *x*'s location. The interpreter's response is usually the value, but if the definition binds a lambda expression, the interpreter instead responds with the name *x*. As in Chapter 2, define is syntactic sugar for *val* with lambda.

The EXP form doesn't bind a name; evaldef just evaluates the expression, binds the result to *it*, and responds with the value.

**318c.** *(definitions of eval and evaldef for μScheme 316a) +≡*

◁ 318b 319 ▷

evaldef : def * value ref env -> value ref env * string
---

```
fun evaldef (VAL (x, e), rho) =
  let val rho = withNameBound (x, rho)
    val v   = eval (e, rho)
    val _   = find (x, rho) := v
    val response = case e of LAMBDA _ => x
                      | _ => valueString v
  in (rho, response)
  end
```

```

| evaldef (EXP e, rho) =
|   let val v = eval (e, rho)
|     val rho = withNameBound ("it", rho)
|     val _ = find ("it", rho) := v
|   in (rho, valueString v)
| end
| evaldef (DEFINE (f, lambda), rho) =
|   let val (xs, e) = lambda
|   in evaldef (VAL (f, LAMBDA lambda), rho)
| end

```

The differences between VAL and EXP are subtle: for VAL, the semantics demands that we add the name to environment rho *before* evaluating expression e. For EXP, we don't bind the name it until *after* evaluating the first top-level expression.

#### 5.4 DEFINING AND EMBEDDING PRIMITIVES

Most of  $\mu$ Scheme's primitives can be implemented using simple ML functions like + or PAIR. But functions like + and PAIR operate on ML values, and  $\mu$ Scheme primitives have to operate on  $\mu$ Scheme values. This issue, just like the issues of different forms of Booleans and lists, can be addressed by embedding and projection. Instead of writing code for each primitive (or each group of primitives) to convert arguments and results back and forth, which is the sensible solution when using C, we can write the primitives using native ML representations, then *embed* those primitives as  $\mu$ Scheme functions. Each primitive function is written with its own most natural ML type, then is embedded into the initial basis as a primitive function of ML type  $\text{exp} * \text{value list} \rightarrow \text{value}$ . (Because we never have to project a  $\mu$ Scheme function to acts as an ML function, there is no need to consider the corresponding projection.)

I want to embed functions of many different types, so instead of writing a single embedding function, I embed each primitive using a composition of functions, starting at its native ML type, and ending with  $\text{exp} * \text{value list} \rightarrow \text{value}$ . I think of the process as building a tower of types.<sup>3</sup> For the primitive function +, here's what it looks like:

- ML primitive + is given to me; it has type  $\text{int} * \text{int} \rightarrow \text{int}$ .
- I can embed that into a function that operates on  $\mu$ Scheme values; its type is now “lifted” to  $\text{value} * \text{value} \rightarrow \text{value}$ .
- I can embed that into a function that can be applied to a list of values; its type is now lifted to  $\text{value list} \rightarrow \text{value}$ .
- Finally, I can embed that into a function that is given the expression in which the primitive appears; its type is now lifted to  $\text{exp} * \text{value list} \rightarrow \text{value}$ , and I've reached the top of my tower.

bind	312b
bindList	312c
DEFINE	313b
eval	316a
EXP	313b
find	311b
LAMBDA	313a
LETREC	313a
LETX	313a
NotFound	311b
rho	316a
unspecified	S379
VAL	313b
valueString	314

That last type  $\text{exp} * \text{value list} \rightarrow \text{value}$  is the same as type primitive, which is used to represent a  $\mu$ Scheme primitive function.

<sup>3</sup>In the tower, every embedding function is higher-order. An alternative technique is to compose each ML primitive with a first-order projection function on the right and a first-order embedding function on the left; that technique is left for you to explore in Exercise 6 on page 330.

Each step between the bullets of the list above is implemented by an embedding function. In the last step, up to top of the type tower, I'm given a function  $f$  of type  $\text{value list} \rightarrow \text{value}$ , and I produce a function  $\text{inExp } f$  of type  $\text{exp * value list} \rightarrow \text{value}$ . The new function applies  $f$ , and if applying  $f$  raises the `RuntimeError` exception, it handles the exception, adds the expression to the error message, and re-raises the exception.

**320a.** *(utility functions for building primitives in  $\mu\text{Scheme}$  320a)*  $\equiv$

(S372a) 320b▷

Interlude:  
 $\mu\text{Scheme}$  in ML

5  
320

```
fun inExp f = inExp : (value list -> value) -> (exp * value list -> value)
  fn (e, vs) => f vs
    handle RuntimeError msg =>
      raise RuntimeError ("in " ^ expString e ^ ", " ^ msg)
```

In the middle step, I'm given a function  $f$  that takes one or two arguments of type  $\text{value}$  and returns a result of type  $\text{value}$ , and I produce a function  $\text{unaryOp } f$  or  $\text{binaryOp } f$  of type  $\text{value list} \rightarrow \text{value}$ . The function produced uses pattern matching to check the number of arguments and to extract the arguments and pass them to  $f$ . If a check fails, function `arityError` raises the `RuntimeError` exception.

**320b.** *(utility functions for building primitives in  $\mu\text{Scheme}$  320a)*  $+ \equiv$

(S372a) ▷ 320a 320c▷

```
unaryOp : (value -> value) -> (value list -> value)
binaryOp : (value * value -> value) -> (value list -> value)

fun arityError n args =
  raise RuntimeError ("expected " ^ intString n ^
    " but got " ^ intString (length args) ^ " arguments")
fun unaryOp f = (fn [a] => f a | args => arityError 1 args)
fun binaryOp f = (fn [a, b] => f (a, b) | args => arityError 2 args)
```

Functions  $\text{unaryOp}$  and  $\text{binaryOp}$  help implement any  $\mu\text{Scheme}$  primitive that is a “unary operator” or “binary operator.”

In the first step, which is combined with the middle step, I'm given a function like  $+$  that expects and returns ML integers, and I produce a function  $\text{arithOp } +$  of type  $\text{value list} \rightarrow \text{value}$ . The anonymous function on the right-hand side has type  $\text{value * value} \rightarrow \text{value}$ .

**320c.** *(utility functions for building primitives in  $\mu\text{Scheme}$  320a)*  $+ \equiv$

(S372a) ▷ 320b 321a▷

```
arithOp: (int * int -> int) -> (value list -> value)

fun arithOp f = binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
  | (NUM n, v) => ⟨report v is not an integer 321b⟩
  | (v, _) => ⟨report v is not an integer 321b⟩
  )
```

I can now define primitives by applying first  $\text{arithOp}$  and then  $\text{inExp}$  to ML functions like  $+$  and  $*$ . First steps for other types of ML primitives are shown below.

I organize my primitives into a list of (name, function) pairs. To write the list, I use Noweb code chunk *(primitives for  $\mu\text{Scheme}$  :: 320d)*. Each primitive on the list has type  $\text{value list} \rightarrow \text{value}$ . In chunk S372a, I apply  $\text{inExp}$  to every primitive on the list, then use the result to build  $\mu\text{Scheme}$ 's initial environment.<sup>4</sup> Here are the first four elements:

**320d.** *(primitives for  $\mu\text{Scheme}$  :: 320d)*  $\equiv$

(S372a) 321c▷

```
("+", arithOp op +) ::  
("-", arithOp op -) ::  
("*", arithOp op *) ::  
("/", arithOp op div) ::
```

The ML keyword `op` makes it possible to use an infix identifier as an ordinary value, so  $\text{arithOp op +}$  passes the value  $+$  (a binary function) to the function  $\text{arithOp}$ .

<sup>4</sup>Actually, the list contains all the primitives except one. The exception is `error`, which should not be wrapped in  $\text{inExp}$ .

Here are first and middle steps for predicates. To embed a predicates result of type `bool` into a result of type `value`, I compose each predicate with `embedBool`.

**321a.** *(utility functions for building primitives in  $\mu$ Scheme 320a)*  $\equiv$  (S372a)  $\triangleleft$  320c

<code>predOp : (value → bool) → (value list → value)</code>
<code>comparison : (value * value → bool) → (value list → value)</code>
<code>intcompare : (int * int → bool) → (value list → value)</code>

```
fun predOp f = unaryOp (embedBool o f)
fun comparison f = binaryOp (embedBool o f)
fun intcompare f = comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                                | (NUM n, v) => (report v is not an integer 321b)
                                | (v, _)      => (report v is not an integer 321b)
                                )

```

**321b.** *(report v is not an integer 321b)*  $\equiv$  (320c 321a)

```
raise RuntimeError ("expected an integer, but got " ^ valueString v)
```

Here are some of the predicates. Equality comparison uses `equalatoms`, which, as required by the semantics of  $\mu$ Scheme, succeeds only on symbols, numbers, Booleans, and the empty list. It is defined in the Supplement. The type predicates are anonymous functions.

**321c.** *(primitives for  $\mu$ Scheme :: 320d)*  $\equiv$  (S372a)  $\triangleleft$  320d

```
("<", intcompare op <) :: 
(">", intcompare op >) :: 
("=?", comparison equalatoms) :: 
("null?", predOp (fn (NIL _) => true | _ => false)) :: 
("boolean?", predOp (fn (BOOLV _) => true | _ => false)) ::
```

The remaining type predicates, the list primitives, and the printing primitives are defined in the Supplement.

## 5.5 NOTABLE DIFFERENCES BETWEEN ML AND C

The interpreter presented in Sections 5.3 and 5.4 uses the same overall design as the interpreters of Chapters 1 and 2. But the many small differences in the two languages add up to a different programming experience; the ML version is more compact and more reliable. The two experiences compare as follows:

- C code and ML code both allocate mutable locations on the heap, which they operate on with pointer syntax (\*) or primitive functions (! and :=), respectively. The C code leaks memory like crazy; plugging *all* the leaks would require a garbage collector considerably more elaborate than the one in Chapter 4. ML ships with a comprehensive garbage collector built in.
- Both interpreters use the same abstraction to represent abstract syntax trees: a tagged sum of products. Thanks to the little data-description language of Chapter 1, the representations are even specified similarly. But C's representation is unsafe—making sure the alt tag is consistent with the payload is up to the programmer. ML's algebraic data types guarantee consistency. The C code does offer one advantage, however: in the source code, the definitions of `struct Value` and `struct Exp` are decoupled. In the ML code, the definitions `value` and `exp`, because they are mutually recursive, must appear adjacent in the source code so they can be connected with `and`.
- Both interpreters use the same design to deal with run-time errors; errors may be detected anywhere and signaled with `runerror`, which calls `longjmp` (in C) or with `raise` (in ML). And in both interpreters, an error once detected

BOOLV	313a
embedBool	315b
equalatoms	S365d
expString	S378c
intString	S238f
NIL	313a
NUM	313a
RuntimeError	S366c
valueString	314

is handled in a central place, using `setjmp` or `handle`, as described in the Supplement.

- Both interpreters use functions, like “length of a list” and “find a name in an environment,” that could in principle be polymorphic. But only ML can define a function that is actually polymorphic. The C code in Chapters 1 and 2 must define a new length function for every type of list and a new find function for every type of environment.

*Interlude:*

5

$\mu$ Scheme in ML

---

322

- In C, we can use `printf`, and we can even define new functions that resemble `printf`, like `print`. In ML, the type checker won’t check the types of the arguments based on a format string, so we are forced to write string-conversion functions and to print only strings.
- In defining primitives, both interpreters use first-order embedding and projection functions to embed and project numbers (`projectInt32` and `mkNum` or `projectInt` and `embedInt`) and Booleans. Only in ML can we embed *functions*, using `binaryOp`, `intcompare`, and so on. And in ML, operations like `/` and `div` are functions, not syntax, so they can be embedded into  $\mu$ Scheme directly. To embed primitives, C requires significant “glue code.”

## 5.6 DEEPER INTO $\mu$ SCHEME: FREE AND BOUND VARIABLES

The ML implementation of  $\mu$ Scheme is easier to modify than the C version. It’s especially helpful that no matter what mistakes you make, the ML code cannot dump core or fail with inexplicable pointer errors. To take advantage of ML’s good properties, I’ve created exercises that enable you to approach little more closely the techniques that are actually used to implement functional languages (Exercises 9 and 10 on pages 331 and 332). These exercises build on a crucial concept in programming languages: the distinction between *free* and *bound* variables.

When an expression  $e$  refers to a name  $y$  that is introduced outside of  $e$  proper, we say that  $y$  is *free* in  $e$ . We often refer to the set of such names as “free variables,” even though “free name” would be more accurate. A variable in  $e$  that is introduced within  $e$  is a *bound* variable. For example, in the expression

```
(lambda (n) (+ 1 n))
```

the name `+` is a free variable, but `n` is a bound variable. Every variable that appears in an expression is either free or bound.

Each variable that appears in a definition is also free or bound. For example, in

```
(define map (f xs)
  (if (null? xs)
      '()
      (cons (f (car xs)) (map f (cdr xs)))))
```

the names `null?`, `cons`, `car`, and `cdr` are free, and the names `map`, `f`, and `xs` are bound.

Free variables play a key role in implementing closures efficiently. The operational semantics for  $\mu$ Scheme say that evaluating a `lambda` expression captures the *entire* environment  $\rho_c$ :

$$\frac{x_1, \dots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho_c, \sigma \rangle \Downarrow \langle \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho_c \rangle, \sigma \rangle \\ (\text{MKCLOSURE})}$$

Do we really need *all* the information in  $\rho_c$ ? If we look at the application rule to see how  $\rho_c$  is used, we can guess maybe not:

$$\begin{array}{c}
 \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \text{ (and all distinct)} \\
 \langle e, \rho, \sigma \rangle \Downarrow \langle (\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \rho_c), \sigma_0 \rangle \\
 \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\
 \vdots \\
 \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\
 \hline
 \langle e_c, \rho_c \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, \sigma_n \{\ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n\}\} \rangle \Downarrow \langle v, \sigma' \rangle \\
 \langle \text{APPLY}(e, e_1, \dots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle
 \end{array} \quad (\text{APPLYCLOSURE})$$

§5.6  
Free and bound  
variables  
323

How is  $\rho_c$  used? Only to evaluate the body of the LAMBDA. It turns out that the MKCLOSURE rule need not store all of  $\rho_c$ —it is enough to store only those bindings in  $\rho_c$  which refer to variables that are free in the LAMBDA expression. Exercise 9 asks you to prove this fact, and Exercise 10 asks you to use it to make the interpreter faster. To do these exercises, you need a precise definition of what a free variable is. We can write such a definition using a proof system.

I write a proof system just for expressions; the judgment form is  $y \in \text{fv}(e)$ . The notation  $\text{fv}(e)$  refers to the set of all variables that appear free in  $e$ , but we'd rather not construct such a set, so we'll pronounce the judgment  $y \in \text{fv}(e)$  as “ $y$  appears free in  $e$ .” To create the proof system, we consider each syntactic form.

A literal expression has no free variables. Formally speaking, no judgment of the form  $y \in \text{fv}(\text{LITERAL}(v))$  can ever be proved, and we express that fact by not having a rule for literals.

An expression consisting of a single variable  $x$  has just one free variable, which is  $x$  itself:

$$\overline{x \in \text{fv}(\text{VAR}(x))}$$

A variable is free in a SET expression if it is assigned to or if it is free in the right-hand side. So for a SET expression, we have two proof rules:

$$\frac{}{x \in \text{fv}(\text{SET}(x, e))} \qquad \frac{y \in \text{fv}(e)}{y \in \text{fv}(\text{SET}(x, e))}$$

A variable is free in an IF expression if and only if it is free in one of the subexpressions:

$$\frac{y \in \text{fv}(e_1)}{y \in \text{fv}(\text{IF}(e_1, e_2, e_3))} \qquad \frac{y \in \text{fv}(e_2)}{y \in \text{fv}(\text{IF}(e_1, e_2, e_3))} \qquad \frac{y \in \text{fv}(e_3)}{y \in \text{fv}(\text{IF}(e_1, e_2, e_3))}$$

A variable is also free in a WHILE expression if and only if it is free in one of the subexpressions:

$$\frac{y \in \text{fv}(e_1)}{y \in \text{fv}(\text{WHILE}(e_1, e_2))} \qquad \frac{y \in \text{fv}(e_2)}{y \in \text{fv}(\text{WHILE}(e_1, e_2))}$$

And the same for BEGIN:

$$\frac{y \in \text{fv}(e_i)}{y \in \text{fv}(\text{BEGIN}(e_1, \dots, e_n))}$$

A variable is free in an application if and only if it is free in the function or in one of the arguments:

$$\frac{y \in \text{fv}(e)}{y \in \text{fv}(\text{APPLY}(e, e_1, \dots, e_n))} \qquad \frac{y \in \text{fv}(e_i)}{y \in \text{fv}(\text{APPLY}(e, e_1, \dots, e_n))}$$

Finally, we get to an interesting case! A variable is free in a LAMBDA expression if it is free in the body and it is *not* one of the arguments:

*Interlude:*

5

*μScheme in ML*

324

$$\frac{y \in \text{fv}(e) \quad y \notin \{x_1, \dots, x_n\}}{y \in \text{fv}(\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e))}$$

The rules for the various LET forms require care. A variable is free in an ordinary LET if it is free in the right-hand side of any binding, or if it is both free in the body and not bound by the LET.

$$\frac{y \in \text{fv}(e_i)}{y \in \text{fv}(\text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e))} \qquad \frac{y \in \text{fv}(e) \quad y \notin \{x_1, \dots, x_n\}}{y \in \text{fv}(\text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e))}$$

The similarity between the second LET rule and the LAMBDA rule shows a kinship between LET and LAMBDA.

The rules for LETREC are almost identical to the rules for LET, except that in a LETREC, the bound names  $x_i$  are never free:

$$\frac{y \in \text{fv}(e_i) \quad y \notin \{x_1, \dots, x_n\}}{y \in \text{fv}(\text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e))}$$

$$\frac{y \in \text{fv}(e) \quad y \notin \{x_1, \dots, x_n\}}{y \in \text{fv}(\text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e))}$$

As usual, it is a nuisance to write the rule for LETSTAR directly. Instead, we treat a LETSTAR expression as a set of nested LET expressions, each containing just one binding. And an empty LETSTAR behaves just like its body.

$$\frac{y \in \text{fv}(\text{LET}(\langle x_1, e_1 \rangle, \text{LETSTAR}(\langle x_2, e_2, \dots, x_n, e_n \rangle, e)))}{y \in \text{fv}(\text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e))}$$

$$\frac{y \in \text{fv}(e)}{y \in \text{fv}(\text{LETSTAR}(\langle \rangle, e))}$$

## 5.7 SUMMARY

By exploiting higher-order functions, algebraic data types, pattern matching, and exceptions, we can make interpreters that are simpler, smaller, easier to read, and more flexible than interpreters we can write in C.

### 5.7.1 Key words and phrases

*The ML language*

**EXCEPTION** A way of signalling a named error condition. Exceptions replace C's `longjmp`. An exception acts like a VALUE CONSTRUCTOR: it can be a name by itself, or it can also carry one or more values. In ML, evaluating an expression may *raise* an exception, produce a value, or cause a checked run-time error.

**EXCEPTION HANDLER** Code that is executed when an exception is raised. Exception handlers replace C's `setjmp`. An exception handler may include a **PATTERN MATCH** that determines which exceptions are handled. Our interpreter's primary exception handlers are associated with the **GENERIC READ-EVAL-PRINT LOOP**.

**ALGEBRAIC DATA TYPE** A representation that is defined by a set of **VALUE CONSTRUCTORS**. Every value of the type is made by using or applying one of the type's value constructors. An algebraic data type is defined with the keyword `datatype`. Mutually recursive algebraic data types are defined with an initial `datatype`, and individual definitions are separated by keyword `and`.

§5.7. *Summary*

325

**VALUE CONSTRUCTOR** A name that either constitutes a value of **ALGEBRAIC DATA TYPE**, like `nil` or `NONE`, or that produces a value of algebraic data type when applied to a value or a tuple of values, like `::` or `SOME`.

**PATTERN MATCHING** The computational process by which a value of **ALGEBRAIC DATA TYPE** is observed. A pattern match comprises an expression being observed, called the **SCRUTINEE**, and a list of *arms*, each of which has a **PATTERN** on the left and an expression on the right. The *first* pattern that matches the scrutinee is chosen, and the corresponding right-hand side is evaluated. Pattern matching may be used in a `case` expression, in an **EXCEPTION HANDLER**, or in a **CLAUSAL DEFINITION**. Pattern matching is explained at length in Chapter 8.

**PATTERN** A variable, which matches anything, or a value constructor applied to zero or more patterns, which matches only a value created with that constructor.

**EXHAUSTIVE PATTERN MATCH** A pattern match that is guaranteed to match every possible value. Pattern matches should be exhaustive. If you write one that is not exhaustive, the ML compiler is required to warn you. You should deploy compiler options which turn that warning into an error.

**REDUNDANT PATTERN** An arm in a pattern match that is guaranteed never to be evaluated, because any values it might match are matched by preceding patterns. A redundant pattern match is a sign of a bug in your code—perhaps a misspelling of a **VALUE CONSTRUCTOR**. Most ML compilers warn you of redundant patterns. You should deploy compiler options which turn that warning into an error.

**CLAUSAL DEFINITION** A syntactic form of definition that combines a function definition and a pattern match. It is introduced with the keyword `fun`, and arms of the pattern are separated by vertical bars. Unlike an ordinary pattern match that is used with `case` or `handle`, the pattern match in a clausal definition begins with the name of the function being defined.

A clausal definition is the idiomatic way to define an ML function that begins with a pattern match. It is preferred over a function whose body is a `case` expression.

**LIST CONSTRUCTOR** Special syntax for writing lists and list patterns in ML. Instead of using `cons` (written `::`) and `nil`, a list constructor uses square brackets containing zero or more elements separated by commas. If the list constructor appears as an expression, each element is an expression. If the list constructor appears as a pattern, each element is a pattern.

**MUTABLE REFERENCE CELL** A location allocated on the heap. In ML, variables stand for values, not for locations, so the only way to get a location is to allocate a mutable reference cell. A reference cell containing a value of type  $\tau$  has type  $\tau$  ref. It is created by primitive function ref, which acts like  $\mu$ Scheme's allocate function (page 156). It is dereferenced by primitive function !, which acts like C's dereferencing operator \*. It is mutated by the infix primitive function :=; the ML expression  $p := e$  is equivalent to the C expression  $*p = e$ .

*Interlude:*

5

$\mu$ Scheme in ML

---

326

**TYPE ABBREVIATION** An abbreviation for a type, defined with keyword type. May take one or more TYPE VARIABLES as parameters. When there are no type parameters, a type abbreviation acts just like C's typedef.

**TYPE VARIABLE** In ML, a name that begins with a quote mark, like 'a. Stands for an unknown type. When used in an ML type, a type variable makes the type POLYMORPHIC.

**POLYMORPHIC TYPE** A type with one or more TYPE VARIABLES. A value or function with a polymorphic type may be used with any type replacing the type variable. For example in type 'a env, the 'a may stand for value ref, which gives us environments that store MUTABLE REFERENCE CELLS.

**SHORT-CIRCUIT CONDITIONALS** A conditional operator that evaluates its second operand only when necessary. In Standard ML, the short-circuit conditionals are andalso and orelse. Keyword and, which looks like it should be a conditional, actually means mutual recursion. Thanks, Professor Milner.

**MUTUAL RECURSION (DATA)** Two or more ALGEBRAIC DATA TYPES, each of which can contain a value of another. Defined using datatype and keyword and; in ML, the and always signifies mutual recursion. In most languages in this book, types exp and value are mutually recursive: an exp can contain a literal value, and a value might be a closure, which contains an exp.

**MUTUAL RECURSION (CODE)** Two or more functions, each of which can call the other. In ML, many mutually recursive functions are defined by nesting the definition of one inside the definition of the other. When both have to be called from outside, they can be defined at the same level using keywords fun and and. Mutually recursive functions can also be defined in C style, using mutable reference cells.

*Interpreter concepts*

**OBJECT LANGUAGE** In an interpreter, the language being implemented. In a semantics, the language being described. In this chapter, the object language is  $\mu$ Scheme.

**METALANGUAGE** In an interpreter, the language in which the interpreter is implemented. In a semantics, the language used for semantic description. In this chapter, the metalanguage is Standard ML. The ML in Standard ML stands for “metalinguage.”

**EMBEDDING** A mapping from METALANGUAGE values to OBJECT-LANGUAGE values. When the metalanguage is typed, like Standard ML, the mapping always succeeds. For example, any ML Boolean can be mapped to a  $\mu$ Scheme value.

**PROJECTION** A mapping from OBJECT-LANGUAGE values to METALANGUAGE values. When the metalanguage is typed, like Standard ML, projection might fail. For example, it is not possible to project the  $\mu$ Scheme value 3 into an ML function. But some projections, like `bool` in chunk 315b, always succeed—in a Boolean context, every  $\mu$ Scheme value is meaningful.

**READ-EVAL-PRINT LOOP** The control center of an interactive interpreter. It *reads* concrete syntax and parses it into abstract syntax, it *evaluates* the abstract syntax, and it *prints* the result. If an EXCEPTION is raised during evaluation, it is handled in the read-eval-print loop, and looping continues. The read-eval-print loop in Appendix O is reused throughout this book; in every language, it handles extended definitions. True definitions are handled by function `processDef`, which is different in each interpreter.

§5.8. Exercises

327

**INTERACTIVITY** A term I coined to describe the behavior of the READ-EVAL-PRINT LOOP. Interactivity determines whether the loop *prompts* the user before reading input, and whether it *prints* after evaluating a definition.

### *Programming-language concepts*

**FREE VARIABLE** A variable whose definition lies outside the function in which the variable appears. The meaning of a free variable depends on context. The idea of free variable generalizes beyond function definitions to include any language construct that introduces new variables, like a `let` expression. Variables that aren't free are **BOUND**.

**BOUND VARIABLE** A variable introduced by a function definition or other construct and whose meaning is independent of any appearance of its name outside the construct. Formal parameters of functions are bound variables, as are variables introduced by `let` forms. Variables that aren't bound are free. The name of a bound variable can be changed without changing the meaning of the program, provided the new name does not conflict with any variable that is FREE in the scope of the binding.

#### 5.7.2 *Further reading*

To learn Standard ML, you have several good choices. The most comprehensive published book is by Paulson (1996), but it may be more than you need. The much shorter book by Felleisen and Friedman (1997) introduces ML using an idiosyncratic, dialectical style. If you can learn from that style, the information is good. If you are a proficient C programmer, you might like the book by Ullman (1997). This book has a good track record helping C programmers make a transition to ML, but it also has a problem: the ML you learn is far from idiomatic.

There are also several good unpublished resources. Harper's (1986) introduction is short, sweet, and easy to follow; but it is for an older version of Standard ML. More recently, Harper (2011) has released an unfinished textbook on programming in Standard ML; it is up to date with the language, but the style is less congenial to beginners. Tofte (2009) presents “tips” on Standard ML, which I characterize as a 20-page quick-reference card. You probably can't get by on the “tips” alone, but when you are working at the computer, they are useful.

## 5.8 EXERCISES

The exercises are summarized in Table 5.3. Here are some of the highlights:

<i>Exercises</i>	<i>Sections</i>	<i>Notes</i>
1 and 2	5.2, 5.3	Working with syntax and semantics: add <code>cond</code> , support variadic functions.
3 to 6	5.3, 5.4	$\mu$ Scheme closures represented as ML functions; embedding and projection functions.
7 and 8	1.7, 5.2	Derivations of operational semantics represented as ML data structures.
9 and 10	5.6	Prove that closures use only free variables, and use that proof to improve the implementation (§5.3).

Table 5.3: Synopsis of all the exercises, with most relevant sections

- In Exercise 2, you extend  $\mu$ Scheme so a function can take an unbounded number of arguments.
- In Exercise 6, you develop a different technique for using ML functions as  $\mu$ Scheme primitives: instead of applying functions to the ML functions, you *compose* each ML function with an embedding function and a projection function. It's a very type-oriented way of building an interpreter.
- In Exercise 10, you use facts about free variables to change the representation of closures, and you measure to see if the change matters.

### 5.8.1 Retrieval practice and other short questions

- A. What's an example way to produce an ML value of type `name * int`?
- B. What ML values inhabit the type `(name * int) list`?
- C. In type `'a env`, what can the `'a` stand for?
- D. What are `[]` and `::`? How are they pronounced?
- E. In ML, any two-argument function can be defined as infix. Functions `+` and `*` you know. What do functions `@` and `^` do?
- F. From function `bind`, what species of value is produced by the ML expression `(name, v) :: rho`?
- G. What  $\mu$ Scheme definition form corresponds to ML's `fun`?
- H. Why is ML's definition of `find` split into two clauses? How does the ML evaluator decide which clause to execute? What does each clause do?
- I. During evaluation, what would cause exception `BindListLength` to be raised?
- J. In this chapter, what's the object language and what's the metalanguage?
- K. Can an ML value of type `int` always be embedded into a  $\mu$ Scheme value? If so, how? If not, why not?
- L. Can a  $\mu$ Scheme value always be projected into an ML value of type `int`? If so, how? If not, why not?
- M. Can an ML function of type `int * int -> int` always be embedded into a  $\mu$ Scheme primitive of type `exp * value list -> value`? If so, how? If not, why not?
- N. Suppose you want to change the semantics of  $\mu$ Scheme so that in `if` expressions and `while` loops, the number zero is treated as falsehood, as in JavaScript. What interpreter code do you change and how?

- O. In chunk *(apply closure clo to args generated automatically)*, explain what `bindList` constructs and why.
- P. In expression `(lambda (n) (+ 1 n))`, what names are free?
- Q. In expression `(lambda (x) (f (g x)))`, what names are free?
- R. In expression `(lambda (f g) (lambda (x) (f (g x))))`, what names are free?
- S. Besides `lambda`, what other syntactic form of expression can introduce new names that are bound in that expression?

## §5.8. Exercises

### 5.8.2 Working with syntax and semantics

329

1. *Syntactic sugar for cond.* Section 2.13.2 on page 165 describes syntactic sugar for Lisp's original conditional expression: the `cond` form. Add a `cond` form to  $\mu$ Scheme. Here is a little code to get you started:

**329. (rows added to ML  $\mu$ Scheme's exptable in exercises [prototype] 329)≡**

```
, ("(cond ([q a] ...))", desugarCond : (exp * exp) list -> exp
  let fun desugarCond qas = raise LeftAsExercise "desugar cond"
        val qa = bracket ("[question answer]", pair <$> exp <*> exp)
        in desugarCond <$> many qa
        end
  )
```

2. *Variadic functions.* Extend  $\mu$ Scheme to support functions with a variable number of arguments. Do so by giving the name `...` (three dots) special significance when it appears as the last formal parameter in a `lambda`. For example:

```
-> (val f (lambda (x y ...) (+ x (+ y (foldl + 0 ...))))
-> (f 1 2 3 4 5) ; in f, rho = { x |-> 1, y |-> 2, ... |-> '(3 4 5) }
15
```

In this example, if `f` gets fewer than two arguments, it is a checked run-time error. If `f` gets at least two arguments, any additional arguments are placed into an ordinary list, and the list is used to initialize the location of the formal parameter associated with `....`.

- (a) Implement this new feature. I recommend that you begin by changing the definition of `lambda` on page 313 to

```
and lambda = name list * { varargs : bool } * exp
```

The type system will tell you what other code you have to change. For the parser, you may find the following function useful:

```
fun newLambda (formals, body) =
  case reverse formals
    of "..." :: fs' => LAMBDA (reverse fs', {varargs=true},
                                    body)
     | _              => LAMBDA (formals, {varargs=false}, body)
```

<\$>	S263b
<*>	S263a
bracket	S276b
exp	S376b
LeftAsExercise	S237a
many	S267b
pair	S263d

The type of this function is

```
name list * exp -> name list * {varargs : bool} * exp,
```

and it is designed for you to adapt old syntax to new syntax; just drop it into the parser wherever `LAMBDA` is used.

- (b) As a complement to the varargs lambda, write a new call primitive such that

```
(call f '(1 2 3))
```

is equivalent to

```
(f 1 2 3)
```

Sadly, you can't use PRIMITIVE for this; you'll have to invent a new kind of thing that has access to the internal eval.

- (c) Demonstrate these utilities by writing a higher-order μScheme function `cons-logger` that counts cons calls in a private variable. It should operate as follows:

```
-> (val cl (cons-logger))
-> (val log-cons (car cl))
-> (val conses-logged (cdr cl))
-> (conses-logged)
0
-> (log-cons f e1 e2 ... en) ; returns (f e1 e2 ... en),
; incrementing private counter
; whenever cons is called
-> (conses-logged)
99 ; or whatever else is the number of times cons is called
; during the call to log-cons
```

- (d) Rewrite the APPLY-CLOSURE rule to account for the new abstract syntax and behavior. To help you, simplified [LaTeX](#) for the original rule is online.

### 5.8.3 Higher-order functions, embedding, and projection

3. *μScheme closures represented as ML functions.* Change the evaluation of lambda expressions so that evaluating a LAMBDA produces an ML function of type `exp * value list -> value`. (Hint: the `exp` parameter should be ignored.) That's the same type as a primitive function, so with this change, evaluating a LAMBDA can produce a PRIMITIVE, not a CLOSURE. On the strength of this trick, rename PRIMITIVE to FUNCTION, and eliminate CLOSURE from the interpreter.

What are the advantages of this simplification? What are the drawbacks?

4. *Embedding and projection for lists.*

- (a) Define function `projectList` of type `value -> value list`. If projection fails, `projectList` should raise the `RuntimeError` exception.

- (b) Rewrite function `embedList` to use `foldr`.

5. *Reusable embedding for integer functions.* In Section 5.4, functions `arithOp` and `comparison` use the same code fragment for embedding functions that consume integers. Break this embedding out into its own function, `intBinary` of type `(int * int -> 'a) -> (value * value -> 'a)`. Rewrite `arithOp` and `comparison` to use `intBinary`.

6. *Embedding by composing first-order functions.* In Section 5.4 on page 319, the primitive functions are defined by applying higher-order embedding functions to ML primitives. An alternate strategy is to compose the ML primitives on the right with first-order projection functions and on the left with first-order embedding functions. Embedding functions should be total; that is,

application of an embedding function should always succeed. But projection functions can be partial; application of a projection function can fail, in which case it should raise `RuntimeError`.

- (a) Define a projection function of type `value list -> value * value` and another of type `value list -> value`.
- (b) Define a projection function of type `value * value -> int * int`
- (c) Find an embedding function of type `int -> value`, or if you can't find one, define one.
- (d) Find an embedding function of type `bool -> value`, or if you can't find one, define one.
- (e) Using your embedding and projection functions, redefine the  $\mu$ Scheme primitives using function composition. For example, you should be able to define the  $\mu$ Scheme `cons` primitive by composing `PAIR` on the right with projection functions and on the left with embedding functions. For each primitive, the projections used on the right should depend only the argument type of the primitive, and the embeddings used on the left should depend only on the result type of the primitive.

Continue to use `inExp` to build `initialBasis`.

## §5.8. Exercises

331

### 5.8.4 Proofs as data

#### 7. Representation of judgments and derivations in ML.

- (a) Devise a representation, in Standard ML, of the judgments of the operational semantics for  $\mu$ Scheme.
- (b) Devise a representation, in Standard ML, of derivations that use the operational semantics of  $\mu$ Scheme.
- (c) Change the `eval` function of the  $\mu$ Scheme interpreter to return a *derivation* instead of a value.

You may wish to revisit the material on proofs and derivations in Section 1.7 on page 56.

#### 8. Validity of derivations.

Using the representation of derivations in item 7(b), write a *proof checker* that tells whether a given tree represents a valid derivation.

### 5.8.5 Free variables

#### 9. Proof: A closure needs only the free variables of its code part.

In this exercise, you prove that the evaluation of an expression doesn't depend on arbitrary bindings in the environment, but only on the bindings of the expression's free variables.

If  $X$  is a set of variables, we can ask what happens to an environment  $\rho$  if we remove the bindings of all the names that are *not* in the set  $X$ . The modified environment is written  $\rho|_X$ , and it is called the *restriction* of  $\rho$  to  $X$ . The exercise is to prove, by structural induction on derivations, that if  $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle$ , then  $\langle e, \rho|_{fv(e)}, \sigma \rangle \Downarrow \langle v, \sigma \rangle$ . (This theorem also justifies the syntactic sugar for short-circuit `||` described in Section 2.13.3 on page 166. And it is related to a similar theorem from Exercise 52 on page 197 in Chapter 2.)

To structure the proof, I recommend you introduce a definition and a lemma.

- Define  $\rho \sqsubseteq \rho'$  to mean that  $\text{dom } \rho \subseteq \text{dom } \rho'$ , and  $\forall x \in \text{dom } \rho : \rho(x) = \rho'(x)$ . The domain of  $\rho'$  contains the domain of  $\rho$ , and on their common domain, they agree. We might say that  $\rho'$  extends or refines  $\rho$ .
- If  $X \subseteq X' \subseteq \text{dom } \rho$ , then  $\rho|_X \sqsubseteq \rho|_{X'} \sqsubseteq \rho$ .

These tools are useful, because except for `LET` forms and `LAMBDA` expressions, if an expression  $e$  has a subexpression  $e_i$ , then  $\text{fv}(e_i) \subseteq \text{fv}(e)$ .

10. *Smaller closures and their performance.* The payoff for the proof in Exercise 9 is that we can use it to optimize code. In chunk 316e, a `LAMBDA` expression is evaluated by capturing a full environment  $\rho$ .

- Modify the code to capture a restricted environment that contains only the free variables of the `LAMBDA` expression. That is, instead of allocating the closure  $(\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho)$ , allocate the smaller closure  $(\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho|_X)$ , where the set  $X$  is defined by  $X = \text{fv}(\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e))$ .
- Measure the modified interpreter to see if the optimization makes a difference. Try running the metacircular evaluator in Section E.1, or try sorting a long list.



## CHAPTER CONTENTS

---

6.1	TYPED IMPCORE: A STATICALLY TYPED IMPERATIVE CORE	337	6.6.5	Type rules for Typed $\mu$ Scheme	370
6.1.1	Concrete syntax of Typed Impcore	338	6.6.6	Type equivalence and type-variable renaming	376
6.1.2	Predefined functions of Typed Impcore	339	6.6.7	Instantiation and renaming by capture-avoiding substitution	380
6.1.3	Abstract syntax, types, and values of Typed Impcore	340	6.6.8	Subverting the type system through variable capture	385
6.1.4	Type system for Typed Impcore	342	6.6.9	Preventing capture with type-lambda	386
6.1.5	Type rules for Typed Impcore	343	6.6.10	Other building blocks of a type checker	387
6.2	A TYPE-CHECKING INTERPRETER FOR TYPED IMPCORE	346	6.7	TYPE SYSTEMS AS THEY REALLY ARE	392
6.2.1	Type checking	346	6.8	SUMMARY	393
6.2.2	Typechecking definitions	350	6.8.1	Key words and phrases	393
6.8.2	Further reading	394			
6.3	EXTENDING TYPED IMPCORE WITH ARRAYS	352	6.9	EXERCISES	395
6.3.1	Types for arrays	352	6.9.1	Retrieval practice	396
6.3.2	New syntax for arrays	352	6.9.2	Type-system fundamentals	397
6.3.3	Rules for type constructors: formation, introduction, and elimination	354	6.9.3	Extending a monomorphic language	400
6.4	INTERLUDE: COMMON TYPE CONSTRUCTORS	357	6.9.4	Coding in a polymorphic language	400
6.5	TYPE SOUNDNESS	359	6.9.5	Extending a polymorphic language	401
6.6	POLYMORPHIC TYPE SYSTEMS AND TYPED $\mu$ SCHEME	360	6.9.6	Typing derivations	403
6.6.1	Concrete syntax	361	6.9.7	Implementing type checking	403
6.6.2	A replacement for type-formation rules: kinds	361	6.9.8	Metatheory	404
6.6.3	The heart of polymorphism: quantified types	365	6.9.9	Metatheory about implementation	407
6.6.4	Abstract syntax, values, and evaluation of Typed $\mu$ Scheme	370	6.9.10	Capture-avoiding substitution	407
			6.9.11	Holes in type systems	408

# 6

## Type systems for Impcore and $\mu$ Scheme

*But in a typed language a separate sort function must be defined for each type, while in a typeless language syntactic checking is lost. We suggest that a solution to this problem is to permit types themselves to be passed as a special kind of parameter, whose usage is restricted in a way which permits the syntactic checking of type correctness.*

John Reynolds, *Towards a Theory of Type Structure*

*The real success of a formal technique is when it is used ubiquitously without the designer being aware of it (e.g., type systems).*

Arvind, IFIP Working Group 2.8, 17 June 2008

The languages of the preceding chapters, Impcore and  $\mu$ Scheme, are *dynamically typed*, which is to say that many faults, such as applying a function to the wrong number of arguments, adding non-numbers, or applying car to a symbol, are not detected until run time. Dynamically typed languages are very flexible, but on any given execution, a program written in a dynamically typed language might surprise you; even a simple mistake like typing cdr when you meant car won't be detected until you find a test case that exercises the bad code. Even using cdr instead of car doesn't lead to a fault right away: cdr simply returns a list in a context where you were expecting an element instead. But if, for example, you then try to add 1 to the result of applying cdr, you are likely to get a run-time error message saying you tried to add 1 to a list value, which is not a number. This kind of error is a *checked run-time error*. The idea of *static typing* is to rule out such errors at compile time, without having to run the faulty code.

Before it is run, a program in a statically typed language undergoes a type analysis, which decides if it is OK to run the program. Different languages use different analyses, but they all build on the same two approaches: *type checking* and *type inference*. In type checking, every variable and formal parameter is annotated with a type, which restricts the values that the variable or parameter may have at run time. Type checking is used in such languages as Ada, Algol, C, C++, Go, Java, Modula-3, Pascal, and Rust. In type inference, also called type reconstruction, variables and parameters need not be annotated; instead, an algorithm assigns a type to each variable or parameter, based on how the variable or parameter is used—if you like, the algorithm reconstructs the types from the code. Type inference is used in such languages as Haskell, Hope, Miranda, OCaml, and Standard ML. In both approaches, the decision about whether it is OK to run a program is made according to the rules of a language-dependent *type system*.

A good type system does much more than just rule out programs that might commit run-time errors; types also act as documentation. The name and type of a function often suffice to show what the function is supposed to do. And the more expressive the type system, the better the documentation.

Type systems, type checking, and type inference can be used in many ways. One of the most effective uses is to help guarantee *safety*. In a safe language, meaningless operations (adding non-numbers, dereferencing a null pointer, and so on) are either ruled out or are detected and reported. Although safety can be guaranteed by checking every operation at run time—as in Impcore,  $\mu$ Scheme, and full Scheme—a good static type system performs most checks at compile time. Here are some example properties that a simple static type system can guarantee:

- That numbers are added only to numbers
- That every function receives the correct number of arguments
- That only Booleans are used in `if` expressions
- That `car` and `cdr` are applied only to lists

When these properties are guaranteed, there is no need for the implementations of addition, function call, and `if` to check their operands at run time. And potentially meaningless operations are reported to the programmer right away, before a program is shipped to its users.

To guarantee safety, a type system must be crafted such that if a program is accepted by the type system, the rules of the type system guarantee that no meaningless operations can be executed. Technically the guarantee is provided by a *type-soundness theorem*, an idea so popular that it has its own slogan: “well-typed programs don’t go wrong.” It’s a wonderful slogan, but “going wrong” has a precise, technical meaning, which is usually narrow. For example, “going wrong” does not typically include unwanted behaviors like these:

- A number is divided by zero.
- The `car` or `cdr` primitive is applied to an *empty* list.
- A reference to an element of an array falls outside the bounds of that array.

These misbehaviors *can* be ruled out by a static type analysis, but because the types get complicated, they often aren’t. Most general-purpose languages stick with a simple type system, and they guarantee safety by supplementing the static type system with some run-time checks for errors like division by zero or array access out of bounds.

Safety is great, but not if it puts the programmer in a straitjacket. When a type system is overly restrictive, programmers complain, often using slang terms like “strong typing<sup>1</sup>.” And restrictive type systems can make it hard to reuse code. For example, Pascal’s type system was notorious for making it impossible to write a function capable of sorting arrays of different *lengths*. As another example, in Chapters 1 and 2, C’s type system requires a distinct set of list functions for each possible list type, even though the functions defined on different types have identical bodies. Such duplication can be avoided, while maintaining safety and type checking, by using a *polymorphic* type system, like the one described in Section 6.6 in this chapter. Polymorphic type systems provide abstractions that can be parameterized by types; one example of such an abstraction is the C++ template.

Another way to dodge restrictions is to use *unsafe* language features, like casts to and from `void *` in C. You might want unsafe features for reasons like these:

---

<sup>1</sup>Which is for weak minds.

- You want to write a program that manipulates memory directly, like a device driver or a garbage collector. People who build systems would like such programs to be safe, but how best to make them safe—say, by a combination of a sophisticated type system and a formal proof of correctness—is a topic of ongoing research.
- You want a relatively simple type system and you don’t want to pay overhead for run-time checks. For example, the C programming language has a simple type system that detects many errors, but if you cast a pointer from one type to another, as almost all C programs do, you’re on your own.

Even in a language with unsafe features, static typing is useful: types serve as documentation, and a type system can prevent many run-time errors, even if a few slip through. The ones that slip through are called *unchecked* run-time errors; the potential of an unchecked run-time error is what makes a language unsafe. If you’re designing an unsafe language, the best practice is to allow unsafe features only in limited contexts, not absolutely anywhere. If you want to see this done well, study Modula-3 (Nelson 1991).

Type systems are a highly developed technology, and to learn how statically typed languages work and how to use them effectively, you will need to master techniques that are presented in the next four chapters. This chapter describes type systems and type checking. Unlike most chapters in this book, it presents not one language but two: Typed Impcore and Typed  $\mu$ Scheme. Typed Impcore is straightforward; it models the restrictive type systems of such languages as Pascal and C. Typed Impcore introduces type systems and serves as an uncomfortably restrictive example. Typed  $\mu$ Scheme is more ambitious; although its design starts from  $\mu$ Scheme, it ends up requiring so many type annotations that the result has a very different feel from  $\mu$ Scheme. Typed  $\mu$ Scheme introduces polymorphism and serves as an eye-opening example.

The power of Typed  $\mu$ Scheme may be eye-opening, but the need for explicit type annotations makes Typed  $\mu$ Scheme programs unpleasant to write. This unpleasantness is relieved by the more advanced type systems described in Chapters 7 to 9. In Chapter 7, we move from Typed  $\mu$ Scheme to nano-ML. The key change is to restrict the type system: the *Hindley-Milner type system* provides most of the power of polymorphism while also supporting type inference. Using Hindley-Milner types with type inference, a programmer can benefit from polymorphism without writing types explicitly. The Hindley-Milner type system is such a “sweet spot” that it forms the core of many of today’s innovative, statically typed languages, including Standard ML, Haskell, OCaml, Agda, Idris, and many others. Truly effective use of the Hindley-Milner system requires user-defined algebraic data types, which are the main idea of Chapter 8.

In Chapter 9, we move from Typed  $\mu$ Scheme to Molecule. Molecule uses type annotations and type checking, but the annotations that direct polymorphism are applied to entire modules, not to individual functions. The resulting language provides all the explicit control you get with Typed  $\mu$ Scheme, as well as different expressive power. And Typed  $\mu$ Scheme’s crushing notational burden is lifted.

## 6.1 TYPED IMPCORE: A STATICALLY TYPED IMPERATIVE CORE

From a language designer’s point of view, the role of a type system is to find errors that would otherwise be hard to detect.<sup>2</sup> In Impcore, the language implementation

---

<sup>2</sup>A language implementor may view the type system in a different role. In most implementations of languages, the type of a value or variable determines how the value is represented and where a variable

has few opportunities to detect errors. All values are integers, so it is impossible to use “the wrong kind” of value in any given context. The only error we can detect automatically is a violation of rule **APPLYUSER**, which requires that the number of formal parameters must equal the number of actual parameters in a function call. To make Typed Impcore interesting enough to study, I add the following features:

- Values may be *integers*, *Booleans*, *arrays* or *unit* (page 342).

I use a single representation (the one from Chapter 1) for both integers and Booleans. In serious implementations of real languages, it is quite common for two or more high-level types (e.g., integer and Boolean) to share a single low-level representation (e.g., machine word). I use a new representation for arrays.

- I require that every variable and expression have a *type* that is known at compile time. The types of some variables, such as the formal parameters of functions, are written down explicitly, much as they are in C and Java. I present precise, formal rules that show whether an expression has a type and what the type is.
- I show how to turn the type rules into a *type checker*. The type checker permits a definition to be evaluated only if all the expressions in that item have types. The type checker I present in this chapter covers only integers and Booleans; extending the implementation to include arrays is left as Exercise 18.
- I discuss *type soundness* (Exercise 26), which means informally that in *every* execution, at *every* point in the program, when an expression produces a value, that value is consistent with the expression’s type. Type soundness ensures that if a program passes the type checker, it does not suffer from type errors at run time.

I present Typed Impcore in two stages. I begin with the integer and Boolean parts of Typed Impcore, and then in Section 6.3, I focus on arrays.

### 6.1.1 Concrete syntax of Typed Impcore

The concrete syntax of Typed Impcore is shown in Figure 6.1 on the facing page. The primary difference with plain Impcore is that in Typed Impcore we declare the argument and result types of functions. Declaring types requires a syntax in which to write them; in addition to Impcore’s definitions (*def*) and expressions (*exp*), Typed Impcore includes a syntactic category of types (*type*).

Explicit types are needed only in function definitions:

338. *(transcript 338)*≡

```
-> (define int add1 ([n : int]) (+ n 1))
add1 : (int -> int)
-> (add1 4)
5 : int
-> (define int double ([n : int]) (+ n n))
double : (int -> int)
-> (double 4)
8 : int
```

339a▷

---

can be stored. For example, an integer is best stored in an integer register, whereas a string may not even fit in a register.

```

def      ::= exp
| (use file-name)
| (val variable-name exp)
| (define type function-name (formals) exp)
| unit-test

unit-test ::= (check-expect exp exp)
| (check-assert exp)
| (check-error exp)
| (check-type-error def)
| (check-function-type function (type) -> type))

exp      ::= literal
| variable-name
| (set variable-name exp)
| (if exp exp exp)
| (while exp exp)
| (begin {exp})
| (function-name {exp})

formals  ::= {[variable-name : type]}

type     ::= int | bool | unit | (array type)

literal   ::= numeral

```

§6.1  
Typed Impcore: a  
statically typed  
imperative core

---

339

Figure 6.1: Concrete syntax of Typed Impcore

Types provide good documentation, and to document a function's type in a testable way, Typed Impcore adds a new unit-test form, `check-function-type`:

**339a.** *(transcript 338)* +≡ △338 339b ▷  
 -> (check-function-type add1 (int -> int))  
 -> (check-function-type double (int -> int))

Because unit tests are not run until the end of a file, it's possible—and helpful—to put a function's type test *before* its definition.

**339b.** *(transcript 338)* +≡ △339a 339c ▷  
 -> (check-function-type positive? (int -> bool))  
 -> (define bool positive? ([n : int]) (> n 0))

In Typed Impcore, an `if` expression requires a Boolean condition:

**339c.** *(transcript 338)* +≡ △339b 353a ▷  
 -> (if 1 77 88)  
 type error: Condition in if expression has type int, which should be bool  
 -> (if (positive? 1) 77 99)  
 77 : int

### 6.1.2 Predefined functions of Typed Impcore

The predefined functions of Typed Impcore do the same work at run time as their counterparts in Chapter 1, but because they include explicit types for arguments

and results, their definitions look different. And because Typed Impcore has no Boolean literals, I write  $(= 1 0)$  for falsehood and  $(= 0 0)$  for truth.

**340a.** *(predefined Typed Impcore functions 340a)≡* 340b▷  
(define bool and ([b : bool] [c : bool]) (if b c b))  
(define bool or ([b : bool] [c : bool]) (if b b c))  
(define bool not ([b : bool]) (if b (= 1 0) (= 0 0)))

The comparison functions accept integers and return Booleans.

**340b.** *(predefined Typed Impcore functions 340a) +≡* △340a  
(define bool <= ([m : int] [n : int]) (not (> m n)))  
(define bool >= ([m : int] [n : int]) (not (< m n)))  
(define bool != ([m : int] [n : int]) (not (= m n)))

Functions `mod` and negated are defined in the Supplement.

### 6.1.3 Abstract syntax, types, and values of Typed Impcore

As in Chapter 5, I define the abstract syntax of Typed Impcore by presenting the representation I use in the implementation. The type system is so simple that I use `ty` not only as the abstract syntax for types but also as the internal representation of types. The type `funty` represents the type of a function in Typed Impcore; it is the abstract syntax used in `check-function-type`.

**340c.** *(types for Typed Impcore 340c)≡* (S383a) 340d▷  
datatype ty = INTTY | BOOLTY | UNITTY | ARRAYTY of ty  
datatype funty = FUNTY of ty list \* ty

Any representation of type `funty` describes *one* type; for example, the comparison functions all have a type that says “two integer arguments and a Boolean result.” A language in which a name has at most one type is called *monomorphic*.

To print types, I use functions `typeString` and `funtyString`, which are defined in Appendix P. To tell when types are equal, I use functions `eqType` and `eqTypes`:

**340d.** *(types for Typed Impcore 340c) +≡* (S383a) △340c 340e▷  
eqType : ty \* ty -> bool  
eqTypes : ty list \* ty list -> bool

```
fun eqType (INTTY,      INTTY)      = true
| eqType (BOOLTY,     BOOLTY)      = true
| eqType (UNITTY,      UNITTY)      = true
| eqType (ARRAYTY t1, ARRAYTY t2) = eqType (t1, t2)
| eqType (_, _)           = false
and eqTypes (ts1, ts2) = ListPair.allEq eqType (ts1, ts2)
```

Always check types for equality using `eqType`, not the built-in `=` operator. As shown in Section 6.6.6 below, a single type can sometimes have multiple representations, which `=` reports as different but should actually be considered the same. Using `eqType` gets these cases right; if you use `=`, you risk introducing bugs that will be hard to find.

We also need to check function types for equality.

**340e.** *(types for Typed Impcore 340c) +≡* (S383a) △340d  
eqFunty : funty \* funty -> bool

```
fun eqFunty (FUNTY (args, result), FUNTY (args', result')) =
  eqTypes (args, args') andalso eqType (result, result')
```

There are two forms of value: `NUM`, which represents integers, Booleans, and the unit value; and `ARRAY`, which represents arrays.

**340f.** *(definitions of exp and value for Typed Impcore 340f)≡* (S381b) 341a▷  
datatype value = NUM of int
| ARRAY of value array

The abstract syntax of expressions includes every form that you would expect in Impcore, plus new forms for equality, printing, and array operations.

**341a.** *(definitions of exp and value for Typed Impcore 340f)*  $\equiv$  (S381b)  $\triangleleft$  340f

```
datatype exp = LITERAL of value
| VAR of name
| SET of name * exp
| IFX of exp * exp * exp
| WHILEX of exp * exp
| BEGIN of exp list
| APPLY of name * exp list
⟨Typed Impcore syntax for equality and printing 341b⟩
⟨array extensions to Typed Impcore's abstract syntax 353d⟩
```

§6.1  
Typed Impcore: a  
statically typed  
imperative core

341

In Typed Impcore, the primitives `=`, `print`, and `println` cannot be represented as functions, because they operate on values of more than one type: they are *polymorphic*. In a monomorphic language like Typed Impcore or C, a polymorphic primitive cannot be implemented as a function; it needs its own syntax.

**341b.** *(Typed Impcore syntax for equality and printing 341b)*  $\equiv$  (341a)

```
| EQ of exp * exp
| PRINTLN of exp
| PRINT of exp
```

Similarly, as described in Section 6.3.2 below, Typed Impcore's array operations also use special-purpose syntax. Such special-purpose syntax is typical; for example, to dereference a pointer in C, we use a syntactic form written with `*` or `->`.

In Typed Impcore, the syntax for a function definition requires that the result type and the types of the formal parameters be identified explicitly. As is customary in formal semantics and in Pascal-like and ML-like languages, but not in C, the syntax puts each formal parameter's type to the right of its name.

**341c.** *(definition of def for Typed Impcore 341c)*  $\equiv$  (S381b)

```
type userfun = { formals : (name * ty) list, body : exp, returns : ty }
datatype def = VAL of name * exp
| EXP of exp
| DEFINE of name * userfun
```

The unit tests include `check-expect` and `check-error` from untyped Impcore and  $\mu$ Scheme, plus two new unit tests related to types:

**341d.** *(definition of unit\_test for Typed Impcore 341d)*  $\equiv$  (S381b)

```
datatype unit_test = CHECK_EXPECT of exp * exp
| CHECK_ASSERT of exp
| CHECK_ERROR of exp
| CHECK_TYPE_ERROR of def
| CHECK_FUNCTION_TYPE of name * funty
```

In Typed Impcore, as in Typed Impcore, a function is not an ordinary value. Instead, there are two forms of function: `USERDEF`, which represents a user-defined function, and `PRIMITIVE`, which represents a primitive function. Because the type system rules out most errors in primitive operations, I use a simpler representation of primitives than the one in Chapter 5: in Typed Impcore, a primitive function does not take a parameter of type `exp`.

type name 310a

**341e.** *(definition of type func, to represent a Typed Impcore function 341e)*  $\equiv$  (S381b)

```
datatype func = USERDEF of name list * exp
| PRIMITIVE of value list -> value
```

A `func` contains no types; types are needed only during type checking, and the `func` representation is used at run time, after all types have been checked.

### 6.1.4 Type system for Typed Impcore

I've designed Typed Impcore's static type system to be more restrictive than Impcore's dynamic type system. For example, because the static type system distinguishes integers from Booleans, you may not use an integer to control an if expression. This restriction should not burden you overmuch; if you have an integer  $i$  that you wish to treat as a Boolean, you can simply write  $(!= i 0)$ . Similarly, if you have a Boolean  $b$  that you wish to treat as an integer, you can write  $(if b 1 0)$ . Another restriction is that you cannot assign the result of a while loop to an integer variable. The motivation is that it seems like a strange thing to do, because a while loop produces no interesting result.

Typed Impcore's type system, like any type system, determines which expressions (also called *terms*) have types. The implementation accepts a definition only if its subterms have types; otherwise, the implementation rejects it. Just as we use a formal proof system to specify precisely which terms have values, we use another formal proof system to specify precisely which terms have types. Before getting into the details of the proof rules, however, I discuss the elements of the system.

To talk about types, we use two new metavariables:  $\tau$  (pronounced "tau", rhymes with "wow") for types and  $\Gamma$  (pronounced "gamma") for type environments. Using these metavariables, I can describe the fundamental elements of the Typed Impcore type system: *simple types*, for which we write  $\tau$ ; three *base types*, for which we write INT, BOOL, and UNIT; one *type constructor*, for which we write ARRAY; many *function types*, for which we write  $\tau_f$ ; and three *type environments*, for which we write  $\Gamma_\xi$ ,  $\Gamma_\phi$ , and  $\Gamma_\rho$ . The type environments give the types of global variables, functions, and formal parameters respectively.

A simple type is a base type or the array constructor applied to a simple type.<sup>3</sup> Because each type in Typed Impcore requires its own special-purpose abstract syntax, I write the names using the **SMALL CAPS** font, which we conventionally use for abstract syntax.

$$\tau ::= \text{INT} \mid \text{BOOL} \mid \text{UNIT} \mid \text{ARRAY}(\tau)$$

The type of a function has some arguments and a result.

$$\tau_f ::= \tau_1 \times \cdots \times \tau_n \rightarrow \tau$$

Just as functions are not values, function types are not value types.

The integer, Boolean, and array types describe values that represent integers, Booleans, and arrays. The unit type plays a more subtle role; a flippant way to explain it is that the purpose of the unit type is to be different from all the others. The unit type answers this question: what type should we give to an expression that is executed purely for its side effect and does not produce an interesting value (like a while loop or print expression)? Most typed languages reserve a special type to represent the result of an operation that is executed only for side effects. In C, C++, and Java, this type is called void, because the type is *uninhabited*, i.e., there are no values of type void. In the functional language ML, the special type is called unit, because the unit type has exactly one inhabitant. For Typed Impcore, the unit type is more appropriate than void, because Typed Impcore is an expression-oriented language in which every terminating evaluation produces a value.

In ML, the inhabitant of the unit type is the empty tuple, which is written  $()$ . Its value is uninteresting: every expression of type unit is guaranteed to produce

<sup>3</sup>In other words, types are defined by induction, and the base types are the base cases. In Typed Impcore, the type of an array does not include the array's size.

the same empty tuple, unless its evaluation fails to terminate or raises an exception. In Typed Impcore, the inhabitant of the unit type is the value 0—but as long as the type system is designed and implemented correctly, no Typed Impcore program can tell what the inhabitant is.

### 6.1.5 Type rules for Typed Impcore

We write a type system using the same kind of formal rules we use to write operational semantics; only the forms of the judgments are different. The judgments in an operational semantics enable us to determine when an expression will be evaluated to produce a value. The judgments in a type system enable us to determine when an expression has a type. To explore the relationships between the judgments of a type system and the judgments of the corresponding operational semantics, try Exercise 26.

#### Type-formation rules

Types in Typed Impcore are so simple that we can tell what is and isn't a type without having to do a whole lot of reasoning. The tiny grammar for  $\tau$  on the preceding page tells the story. But in languages with more flexible type systems, the story usually begins with rules that say what is and isn't a type. The judgment form for Typed Impcore is  $\boxed{\tau \text{ is a type}}$ , and here are the rules:

$$\frac{}{\text{(UNITTYPE)}} \quad \frac{}{\text{(INTTYPE)}} \quad \frac{}{\text{(BOOLTYPE)}}$$


---


$$\frac{}{\text{UNIT is a type}} \quad \frac{}{\text{INT is a type}} \quad \frac{}{\text{BOOL is a type}}$$


---


$$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}} \quad \frac{}{\text{(ARRAYTYPE)}}$$

In the interpreter, every value of type  $\text{ty}$  (chunk 340c) is a type.

#### Typing judgment for expressions

The typing judgment for expressions is  $\boxed{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$ , meaning that given type environments  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ , expression  $e$  has type  $\tau$ . Judgments of this form are proved by the rules of Typed Impcore's type system. These rules are deterministic, and in a well-typed Typed Impcore program, every expression has exactly one type. In other words, given the environments  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  and the abstract-syntax tree  $e$ , there is at most one  $\tau$  such that  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$ . (For a proof of uniqueness, see Exercise 20 on page 404.) To find this  $\tau$ , or to report an error if no such  $\tau$  exists, is the job of a *type checker*.

#### Type rules for expressions

All literals are integers. This rule is sound because there are no Boolean literals in Typed Impcore: the parser creates only integer literals.

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LITERAL}(v) : \text{INT}} \quad \text{(LITERAL)}$$

The use of a variable is well typed if the variable is bound in the environment. As in Chapter 1, formals hide globals.

$$\frac{x \in \text{dom } \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\rho(x)} \quad (\text{FORMALVAR})$$

$$\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\xi(x)} \quad (\text{GLOBALVAR})$$

An assignment is well typed if the type of the right-hand side matches the type of the variable being assigned to. In other words, assigning to a variable mustn't change its type. And just as in the operational semantics, we look for the variable in two environments.

$$\frac{x \in \text{dom } \Gamma_\rho \quad \Gamma_\rho(x) = \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau} \quad (\text{FORMALASSIGN})$$

$$\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi \quad \Gamma_\xi(x) = \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau} \quad (\text{GLOBALASSIGN})$$

If the assignment is well typed, the type of the assignment is the type of the variable and the value assigned to it. I could have chosen type unit, but this choice would have ruled out such expressions as (`set x (set y 0)`). Such expressions are permitted by the rule I have chosen, which is copied from C.

A conditional expression is well typed if the condition is Boolean and the two branches have the same type. In that case, the type of the conditional expression is the type shared by the branches.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{IF}(e_1, e_2, e_3) : \tau} \quad (\text{IF})$$

Unlike the rules for literals, variables, and assignment, the IF rule is structured differently from the IF rules in the operational semantics. The operational semantics has two rules: one corresponds to  $e_1 \Downarrow \text{BOOLV}(\#t)$  and evaluates  $e_2$ , and one corresponds to  $e_1 \Downarrow \text{BOOLV}(\#f)$  and evaluates  $e_3$ . Each rule evaluates just two of the three subexpressions. The type system, which cares only about the *type* of  $e_1$ , not its value, needs only one rule, which checks the types of all three subexpressions.

A WHILE loop is well typed if the condition is Boolean and the body is well typed. The  $\tau$  that is the type of the body  $e_2$  is not used in the conclusion of the rule, because we care only that the type  $\tau$  exists, not what type it is.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}} \quad (\text{WHILE})$$

As explained above, a WHILE loop does not produce a useful result, so I give it the unit type.

Like the IF rule, the WHILE rule is structured differently from the WHILE rules in the operational semantics. The operational semantics has two rules: one corresponds to  $e_1 \Downarrow \text{BOOLV}(\#t)$  and iterates the loop, and one corresponds to  $e_1 \Downarrow \text{BOOLV}(\#f)$  and terminates the loop. The type system needs only one rule, which checks both types.

A BEGIN expression is well typed if all of its subexpressions are well typed. Because every subexpression except the last is executed only for its side effect,

we would be justified in requiring each such subexpression to have type `UNIT`. But I want to allow `SET` expressions inside `BEGIN` expressions, so I permit a subexpression in a `BEGIN` sequence to have any type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \quad \dots \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n} \quad (\text{BEGIN})$$

The premises mentioning  $e_1, \dots, e_{n-1}$  are necessary because although we don't care what the types of  $e_1, \dots, e_{n-1}$  are, we do insist that they be well typed.

An empty `BEGIN` is always well typed and has type `UNIT`.

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}() : \text{UNIT}} \quad (\text{EMPTYBEGIN})$$

A function application is well typed if the function is applied to the right number and types of arguments. A function's type is looked up in the function-type environment  $\Gamma_\phi$ . The type of the application is the result type of the function.

$$\frac{\Gamma_\phi(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(f, e_1, \dots, e_n) : \tau} \quad (\text{APPLY})$$

Next come the polymorphic operations—each of the special-purpose syntactic forms has its own typing rule. An equality test is well typed if it tests two values of the same type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{EQ}(e_1, e_2) : \text{BOOL}} \quad (\text{EQ})$$

A `print` or `println` expression is well typed if its subexpression is well typed.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{PRINTLN}(e) : \text{UNIT}} \quad (\text{PRINTLN})$$

### *Typing judgment and type rules for definitions*

By analogy with operational semantics, the type rule for a definition may produce a new type environment. The judgment has the form  $\boxed{\langle d, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma'_\xi, \Gamma'_\phi \rangle}$ , which says that when definition  $d$  is typed given type environments  $\Gamma_\xi$  and  $\Gamma_\phi$ , the new environments are  $\Gamma'_\xi$  and  $\Gamma'_\phi$ .

If a variable  $x$  has not been bound before, its `VAL` binding requires only that the right-hand side be well typed. The newly bound  $x$  takes the type of its right-hand side.

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \quad x \notin \text{dom } \Gamma_\xi}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle} \quad (\text{NEWVAL})$$

If  $x$  is already bound, the `VAL` binding acts like a `SET`, and just like a `SET`, the `VAL` must not change  $x$ 's type (see Exercise 29 on page 408).

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \quad \Gamma_\xi(x) = \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \quad (\text{OLDVAL})$$

Programmers need to use top-level expressions of many types, so to avoid any unsoundness, typechecking a top-level expression does not bind the variable `it`. Instead, a top-level expression is simply checked, and the type environments don't change.

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \quad (\text{EXP})$$

The definition of a new function  $f$  updates the function environment. The type of each argument is given in the function's definition, and in an environment where each  $x_i$  has type  $\tau_i$ , the function's body  $e$  must be well typed and have type  $\tau$ , which is the result type given in the function's definition. Assuming  $e$  has type  $\tau$ , function  $f$  is added to the type environment  $\Gamma_\phi$  with function type  $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ . Because  $f$  could be called recursively from  $e$ ,  $f$  also goes into the type environment used to typecheck  $e$ .

$$\frac{\begin{array}{c} \tau_1, \dots, \tau_n \text{ are types} \\ \tau_f = \tau_1 \times \cdots \times \tau_n \rightarrow \tau \\ f \notin \text{dom } \Gamma_\phi \\ \Gamma_\xi, \Gamma_\phi \{f \mapsto \tau_f\}, \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, ((x_1 : \tau_1, \dots, x_n : \tau_n), e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \{f \mapsto \tau_f\} \rangle}$$

(DEFINE)

In addition to the typing judgment for  $e$ , the body of the function, this rule also uses well-formedness judgments for types  $\tau_1$  to  $\tau_n$ . In general, when a type appears in syntax, the rule for that syntax may need a premise saying the type is well formed. (The result type  $\tau$  here does not need such a premise, because when the judgment form  $\cdots \vdash e : \tau$  is derivable,  $\tau$  is guaranteed to be well formed; see Exercise 22.)

A function can be redefined, but just as we mustn't change the type of a previously defined variable, we also mustn't change the type of a previously defined function (see Exercise 29 on page 408).

$$\frac{\begin{array}{c} \tau_1, \dots, \tau_n \text{ are types} \\ \Gamma_\phi(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau \\ \Gamma_\xi, \Gamma_\phi \{f \mapsto \tau_1 \times \cdots \times \tau_n \rightarrow \tau\}, \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, ((x_1 : \tau_1, \dots, x_n : \tau_n), e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle}$$

(REDEFINE)

## 6.2 A TYPE-CHECKING INTERPRETER FOR TYPED IMPCORE

Most of my interpreter for Typed Impcore is what you would expect from looking at Chapter 5's interpreter for  $\mu$ Scheme. But because Typed Impcore is a typed language, there are a few differences:

- In addition to abstract syntax for expressions and definitions, we also have abstract syntax for types, as shown in chunk *(types for Typed Impcore 340c)*. Because the type system is so simple, the abstract syntax also serves as our internal representation of types.
- To ensure that every expression in a Typed Impcore program is well typed, we need a type checker.
- The read-eval-print loop now also checks types.

My type checker handles only integers, Booleans, and the unit type; array types are left for the Exercises.

### 6.2.1 Type checking

Given an expression  $e$  and a collection of type environments  $\Gamma_\xi$ ,  $\Gamma_\phi$ , and  $\Gamma_\rho$ , calling  $\text{typeof}(e, \Gamma_\xi, \Gamma_\phi, \Gamma_\rho)$  returns a type  $\tau$  such that  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$ . If no such

Type system	Concept	Interpreter	
$d$	Definition	def, xdef, or unit_test (pages 341, S365 and 341)	
$e$	Expression	exp (page 341)	
$x, f$	Name	name (page 310)	
$\tau$	Type	ty (page 340)	§6.2
$\Gamma_\xi, \Gamma_\rho$	Type environment	ty env (pages 310 and 340)	A type-checking interpreter for Typed Impcore
$\Gamma_\phi$	Function-type environment	funty env (pages 310 and 340)	
$\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$	Typecheck $e$	typeof( $e, \Gamma_\xi, \Gamma_\phi, \Gamma_\rho = \tau$ , and often $ty e = \tau$ ) (page 347)	347
$\langle d, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma'_\xi, \Gamma'_\phi \rangle$	Typecheck $d$	typdef( $d, \Gamma_\xi, \Gamma_\phi = (\Gamma'_\xi, \Gamma'_\phi, s)$ (page 350)	
$x \in \text{dom } \Gamma$	Definedness	find( $x, \Gamma$ ) terminates without raising an exception (page 311)	
$\Gamma(x)$	Type lookup	find( $x, \Gamma$ ) (page 311)	
$\Gamma\{x \mapsto \tau\}$	Binding	bind( $x, \tau, \Gamma$ ) (page 312)	
$\Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$	Binding	bindlist([ $x_1, \dots, x_n$ ], [ $\tau_1, \dots, \tau_n$ ], $\Gamma$ ) (page 312)	

Table 6.2: Correspondence between Typed Impcore’s type system and code

type exists, typeof raises the `TypeError` exception (or possibly `NotFound`). Internal function `ty` computes the type of  $e$  using the environments passed to `typeof`.

347a. (type checking for Typed Impcore 347a)  $\equiv$

(S383a) 350c ▷

typeof : exp * ty env * funty env * ty env -> ty
ty : exp -> ty

```
fun typeof (e, globals, functions, formals) =
  let function ty, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b
  in ty e
  end
```

Just as we can derive an implementation of eval by examining the rules of operational semantics, we can derive an implementation of `ty` by examining the rules of the type system. To help show the connection between the type system and the type checker, I show the relevant rules before each case of the function `ty`.

All literals are integers.

INTTY	340c
LITERAL	341a

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LITERAL}(v) : \text{INT}} \quad (\text{LITERAL})$$

347b. (function `ty`, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b)  $\equiv$

(347a) 348a ▷

```
fun ty (LITERAL v) = INTTY
```

To type a variable, we must try two environments.

$$\frac{x \in \text{dom } \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\rho(x)} \quad (\text{FORMALVAR})$$

$$\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\xi(x)} \quad (\text{GLOBALVAR})$$

The code is shorter than the rules!

348a. *(function ty, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b) + \equiv* (347a)  $\triangleleft$  347b 348b  $\triangleright$   
| ty (VAR x) = (find (x, formals) handle NotFound \_ => find (x, globals))

If the variable is not found in either  $\Gamma_\rho$  or  $\Gamma_\xi$ , the type checker raises the NotFound exception.

To check an assignment, we must also try two environments.

$$\frac{x \in \text{dom } \Gamma_\rho \quad \Gamma_\rho(x) = \tau}{\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau}} \quad (\text{FORMALASSIGN})$$

$$\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi \quad \Gamma_\xi(x) = \tau}{\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau}} \quad (\text{GLOBALASSIGN})$$

Function ty calls itself recursively to determine the types of the variable and the expression, then compares them for equality.

348b. *(function ty, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b) + \equiv* (347a)  $\triangleleft$  348a 348d  $\triangleright$   
| ty (SET (x, e)) =

```
let val tau_x = ty (VAR x)
    val tau_e = ty e
    in if eqType (tau_x, tau_e) then tau_x
        else (raise TypeError for an assignment 348c)
    end
```

When the variable and the expression have different types—a case not covered by the specification—ty gives an explanatory error message. Creating this message takes more work than checking the types.

348c. *(raise TypeError for an assignment 348c) \equiv* (348b)  
raise TypeError ("Set variable " ^ x ^ " of type " ^ typeString tau\_x ^  
" to value of type " ^ typeString tau\_e)

To check a conditional, we check that the condition is Boolean and that both branches have the same type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{IF}(e_1, e_2, e_3) : \tau} \quad (\text{IF})$$

Again, most of the code is devoted to error messages.

348d. *(function ty, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b) + \equiv* (347a)  $\triangleleft$  348b 349a  $\triangleright$   
| ty (IFX (e1, e2, e3)) =

```
let val tau1 = ty e1
    val tau2 = ty e2
    val tau3 = ty e3
    in if eqType (tau1, BOOLTY) then
        if eqType (tau2, tau3) then
            tau2
        else
            raise TypeError
                ("In if expression, true branch has type " ^ typeString tau2 ^ " but false branch has type " ^ typeString tau3)
        else
            raise TypeError
                ("Condition in if expression has type " ^ typeString tau1 ^ ", which should be " ^ typeString BOOLTY)
    end
```

To check a `while` loop, we check that the condition is Boolean and that the body has a type—we don't care what type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}} \quad (\text{WHILE})$$

**349a.** *(function `ty`, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b) +≡ (347a) ▷ 348d 349b ▷*

```
| ty (WHILEX (e1, e2)) =
  let val tau1 = ty e1
  val tau2 = ty e2
  in if eqType (tau1, BOOLTY) then
      UNITTY
    else
      raise TypeError ("Condition in while expression has type " ^
                      typeString tau1 ^ ", which should be " ^
                      typeString BOOLTY)
  end
```

To check a `BEGIN`, we check the types of all its subexpressions.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \quad \dots \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n} \quad (\text{BEGIN})$$

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}() : \text{UNIT}} \quad (\text{EMPTYBEGIN})$$

The type of the `BEGIN` is the type of the last subexpression, or if there are no subexpressions, `UNITTY`. The implementation uses Standard ML basis function `List.last`.

**349b.** *(function `ty`, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b) +≡ (347a) ▷ 349a 349c ▷*

```
| ty (BEGIN es) =
  let val bodytypes = map ty es
  in List.last bodytypes handle Empty => UNITTY
  end
```

To check an equality test, we check that the expressions being tested have the same type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{EQ}(e_1, e_2) : \text{BOOL}} \quad (\text{EQ})$$

When computing the types of  $e_1$  and  $e_2$ , I use an ML trick: `val` binds the pair of names ( $\tau_1$ ,  $\tau_2$ ) to a pair of ML values. This trick has the same effect as the separate computations of  $\tau_1$  and  $\tau_2$  in the `WHILEX` case above, but it highlights the similarity of the two computations, and it uses scarce vertical space more effectively.

**349c.** *(function `ty`, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  347b) +≡ (347a) ▷ 349b 350a ▷*

```
| ty (EQ (e1, e2)) =
  let val (tau1, tau2) = (ty e1, ty e2)
  in if eqType (tau1, tau2) then
      BOOLTY
    else
      raise TypeError ("Equality sees values of different types " ^
                      typeString tau1 ^ " and " ^ typeString tau2)
  end
```

To check a print expression, we check that its subexpression is well typed.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{PRINTLN}(e) : \text{UNIT}} \quad (\text{PRINTLN})$$

**§6.2**  
*A type-checking interpreter for Typed Impcore*

349

BEGIN	341a
BOOLTY	340c
EQ	341b
eqType	340d
find	311b
formals	347a
globals	347a
IFX	341a
NotFound	311b
SET	341a
ty	347b
TypeError	S237b
typeString	S385d
UNITTY	340c
VAR	341a
WHILEX	341a

The type of the subexpression doesn't affect the type of the print expression.

$$350a. \langle \text{function ty, checks type of expression, given } \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \text{ 347b} \rangle + \equiv \quad (347a) \triangleleft 349c \ 350b \triangleright$$

$$\begin{aligned} | \ \text{ty} (\text{PRINTLN } e) &= (\text{ty } e; \text{UNITY}) \\ | \ \text{ty} (\text{PRINT } e) &= (\text{ty } e; \text{UNITY}) \end{aligned}$$

To check a function application, we look up the function in the environment  $\Gamma_\phi$ .

$$\frac{\Gamma_\phi(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(f, e_1, \dots, e_n) : \tau} \quad (\text{APPLY})$$

The types of the actual parameters must match the types of the formal parameters. If they don't, function `badParameter`, which finds an actual parameter whose type didn't match, and it builds an error message. It is defined in the Supplement.

$$350b. \langle \text{function ty, checks type of expression, given } \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \text{ 347b} \rangle + \equiv \quad (347a) \triangleleft 350a$$

$$\begin{aligned} | \ \text{ty} (\text{APPLY } (f, \text{actuals})) &= \boxed{\text{badParameter} : \text{int} * \text{ty list} * \text{ty list} \rightarrow 'a} \\ &\quad \text{let val actualtypes} &&= \text{map ty actuals} \\ &\quad \text{val FUNTY (formaltypes, resulttype)} &&= \text{find } (f, \text{functions}) \\ &\quad \langle \text{definition of badParameter S381c} \rangle \\ &\quad \text{in if eqTypes (actualtypes, formaltypes) then} \\ &\quad \quad \text{resulttype} \\ &\quad \text{else} \\ &\quad \quad \text{badParameter (1, actualtypes, formaltypes)} \\ &\quad \text{end} \end{aligned}$$

### 6.2.2 Typechecking definitions

The form of the typing judgment for a definition  $d$  is  $\langle d, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma'_\phi, \Gamma'_\xi \rangle$ . This judgment is implemented by a static analysis that I call *typing* the definition.

$$350c. \langle \text{type checking for Typed Impcore 347a} \rangle + \equiv \quad (S383a) \triangleleft 347a$$

$$\boxed{\text{typedef} : \text{def} * \text{ty env} * \text{funty env} \rightarrow \text{ty env} * \text{funty env} * \text{string}}$$

$$\text{fun typedef } (d, \text{globals}, \text{functions}) =$$

$$\begin{aligned} \text{case } d \\ \text{of } \langle \text{cases for typing definitions in Typed Impcore 350d} \rangle \end{aligned}$$

A `val` binding may change a variable's value, but not its type. Depending on whether the name is already defined, there are two rules.

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \quad x \notin \text{dom } \Gamma_\xi}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \{x \mapsto \tau\}, \Gamma_\phi \rangle} \quad (\text{NEWVAL})$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau \quad \Gamma_\xi(x) = \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \quad (\text{OLDVAL})$$

Which rule applies depends on whether  $x$  is already defined.

$$350d. \langle \text{cases for typing definitions in Typed Impcore 350d} \rangle \equiv \quad (350c) \ 351b \triangleright$$

$$\begin{aligned} \text{VAL } (x, e) &= \\ &\quad \text{if not (isbound } (x, \text{globals})) \text{ then} \\ &\quad \quad \text{let val tau} = \text{typeof } (e, \text{globals}, \text{functions}, \text{emptyEnv}) \\ &\quad \quad \text{in } (\text{bind } (x, \tau), \text{functions}, \text{typeString } \tau) \\ &\quad \quad \text{end} \\ &\quad \text{else} \\ &\quad \quad \text{let val tau}' = \text{find } (x, \text{globals}) \\ &\quad \quad \text{val tau} = \text{typeof } (e, \text{globals}, \text{functions}, \text{emptyEnv}) \\ &\quad \quad \text{in if eqType } (\tau, \tau') \text{ then} \\ &\quad \quad \quad (\text{globals}, \text{functions}, \text{typeString } \tau) \\ &\quad \quad \text{else} \\ &\quad \quad \quad \langle \text{raise TypeError with message about redefinition 351a} \rangle \\ &\quad \text{end} \end{aligned}$$

351a. *(raise TypeError with message about redefinition 351a)*  $\equiv$  (350d)

```
raise TypeError ("Global variable " ^ x ^ " of type " ^ typeString tau' ^
    " may not be redefined with type " ^ typeString tau)
```

A top-level expression has to have a type, but it leaves the environments unchanged.

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \quad (\text{EXP})$$

351b. *(cases for typing definitions in Typed Impcore 350d)*  $\equiv$  (350c)  $\triangleleft$  350d 351c

```
| EXP e =>
  let val tau = typeof (e, globals, functions, emptyEnv)
  in (globals, functions, typeString tau)
  end
```

§6.2  
A type-checking  
interpreter for  
Typed Impcore

351

Like a variable definition, a function definition has two rules, depending on whether the function is already defined.

$$\frac{\begin{array}{c} \tau_1, \dots, \tau_n \text{ are types} \\ \tau_f = \tau_1 \times \dots \times \tau_n \rightarrow \tau \\ f \notin \text{dom } \Gamma_\phi \\ \Gamma_\xi, \Gamma_\phi \{f \mapsto \tau_f\}, \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \{f \mapsto \tau_f\} \rangle} \quad (\text{DEFINE})$$

$$\frac{\begin{array}{c} \tau_1, \dots, \tau_n \text{ are types} \\ \Gamma_\phi(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau \\ \Gamma_\xi, \Gamma_\phi \{f \mapsto \tau_1 \times \dots \times \tau_n \rightarrow \tau\}, \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \quad (\text{REDEFINE})$$

Before checking whether  $f \in \text{dom } \Gamma_\phi$ , I build the function type, get the type of the body ( $\tau$ ), and confirm that  $\tau$  is equal to the returns type in the syntax. Only then do I check  $f \in \text{dom } \Gamma_\phi$  and finish accordingly.

351c. *(cases for typing definitions in Typed Impcore 350d)*  $\equiv$  (350c)  $\triangleleft$  351b

```
| DEFINE (f, {returns, formals, body}) =>
  let val (fnames, ftys) = ListPair.unzip formals
      val def's_type      = FUNTY (ftys, returns)
      val functions'      = bind (f, def's_type, functions)
      val formals         = bindList (fnames, ftys, emptyEnv)
      val tau             = typeof (body, globals, functions', formals)
  in if eqType (tau, returns) then
      if not (isbound (f, functions)) then
          (globals, functions', funtyString def's_type)
      else
          let val env's_type = find (f, functions)
              in if eqFunty (env's_type, def's_type) then
                  (globals, functions, funtyString def's_type)
              else
                  raise TypeError
                  ("Function " ^ f ^ " of type " ^ funtyString env's_type
                   ^ " may not be redefined with type " ^ funtyString def's_type)
          end
      else
          raise TypeError ("Body of function has type " ^ typeString tau ^ "
                           ", which does not match declared result type " ^ "
                           "of " ^ typeString returns)
  end
```

APPLY	341a
badParameter	S381c
bind	312b
bindList	312c
DEFINE	341c
emptyEnv	311a
eqFunty	340e
eqType	340d
eqTypes	340d
EXP	341c
find	311b
functions	347a
FUNTY	340c
funtyString	S386a
isbound	312a
PRINT	341b
PRINTLN	341b
ty	347b
TypeError	S237b
typeof	347a
typeString	S385d
UNITTY	340c
VAL	341c

### 6.3 EXTENDING TYPED IMPCORE WITH ARRAYS

Type systems are intimately connected with data structures, and a new data structure often requires a new type. To demonstrate the connection, I extend Typed Impcore by adding arrays. Arrays are found in almost every programming language and are frequently used data structures in such languages as Fortran, Java, and Pascal. This section presents array types, concrete syntax, abstract syntax, evaluation code, and type rules.

#### 6.3.1 Types for arrays

The array type is a different *kind* of thing from the integer, Boolean, and UNIT types. Properly speaking, “array” is not a “type” at all: it is a *type constructor*, i.e., a thing you use to build types. To name just a few possibilities, you can build arrays of integers, arrays of Booleans, and arrays of arrays of integers. In general, given any type  $\tau$ , you can build the type “array of  $\tau$ .” In our abstract syntax, the array type constructor is represented by ARRAY, and in the ML code, by ARRAYTY, as shown in chunk 340c. In the concrete syntax, it is represented by (array type), as shown on page 339. So for example, the type of arrays of Booleans is (array bool), and the type of arrays of arrays of integers is (array (array bool)).

“Type constructor” is such a useful idea that even INT, UNIT, and BOOL are often treated as type constructors, even though they take no arguments and so can be used to build only one type apiece. Such nullary type constructors are usually called *base types*, because the set of all types is defined by induction and the nullary type constructors are the base cases.

Whenever you add a new type to a language, whether it is a base type or a more interesting type constructor, you also add operations for values of that type. Let’s look at operations for arrays.

#### 6.3.2 New syntax for arrays

Like Typed Impcore’s equality and printing operations, each of its array operations gets its own syntactic form.

- To evaluate an expression of the form (make-array  $e_1 e_2$ ), we evaluate  $e_1$  to get an integer  $i$  and  $e_2$  to get a value  $v$ . We then allocate and return a new array containing  $i$  elements, each initialized to  $v$ ; if  $i$  is negative, the evaluation results in a checked run-time error. A make-array expression is analogous to a creator (Section 2.5.2) for the array type.
- To evaluate an expression of the form (array-at  $e_1 e_2$ ), we evaluate  $e_1$  to get an array  $a$  and  $e_2$  to get an integer  $i$ . We then return the  $i$ th element of  $a$ , where the first element is element number 0; if  $i$  is out of bounds, the evaluation results in a checked run-time error. An array-at expression is analogous to an observer for the array type.
- To evaluate an expression of the form (array-put  $e_1 e_2 e_3$ ), we evaluate  $e_1$  to get an array  $a$ ,  $e_2$  to get an integer  $i$ , and  $e_3$  to get a value  $v$ . We then modify  $a$  by storing  $v$  as its  $i$ th element; if  $i$  is out of bounds, the evaluation results in a checked run-time error. An array-put expression is analogous to a mutator for the array type.
- To evaluate an expression of the form (array-size  $e$ ), we evaluate  $e$  to get an array  $a$ . We then return the number of elements in  $a$ . An array-size expression is analogous to an observer for the array type.

Here's a very simple example of an array of Booleans.

353a. *(transcript 338)* +≡  
-> (val truth-vector (make-array 3 (= 0 1)))  
[0 0 0] : (array bool)  
-> (array-put truth-vector 1 (= 0 0))  
1 : bool  
-> truth-vector  
[0 1 0] : (array bool)

△339c 353b ▷

§6.3  
*Extending Typed  
Impcore with  
arrays*

353

Here's another example in which we create and initialize a matrix, which in Typed Impcore is represented as an array of arrays.

353b. *(transcript 338)* +≡  
-> (define (array (array int)) matrix-using-a-and-i  
 ; return square matrix of side length; a and i are for local use only  
 ([length : int] [a : (array (array int))] [i : int])  
 (begin  
 (set a (make-array length (make-array 0 0)))  
 (set i 0)  
 (while (< i length)  
 (begin  
 (array-put a i (make-array length 0))  
 (set i (+ i 1))))  
 a))  
-> (define (array (array int)) matrix ([length : int])  
 (matrix-using-a-and-i length (make-array 0 (make-array 0 0) 0)))

△353a 353c ▷

Function `matrix` fills a matrix with zeros; to update the values, we use syntactic forms `array-put` and `array-at`.

353c. *(transcript 338)* +≡  
-> (val a (matrix 3))  
[[0 0 0] [0 0 0] [0 0 0]] : (array (array int))  
-> (val i 0)  
-> (val j 0)  
-> (while (< i 3) (begin  
 (set j 0)  
 (while (< j 3) (begin  
 (array-put (array-at a i) j (+ i j))  
 (set j (+ j 1))))  
 (set i (+ i 1))))  
-> a  
[[0 1 2] [1 2 3] [2 3 4]] : (array (array int))  
-> (val a.1 (array-at a 1))  
[1 2 3] : (array int)  
-> (val a.1.1 (array-at a.1 1))  
2 : int

△353b

The `array-at` form operates on arrays of any type. As shown, it can index type `(array int)` and return a result of type `int`, and it can index into an array of type `(array (array int))` and return a result of type `(array int)`. This behavior is *polymorphic*. Typed Impcore is *monomorphic*, which means that a function can be used for arguments and results of one and only one type. So like `=` and `println`, `array-at` can't be implemented as a primitive function; it must be a syntactic form. In fact, *every* array operation is a syntactic form, defined as follows:

type exp 341a

353d. *(array extensions to Typed Impcore's abstract syntax 353d)* ≡

(341a)

| AMAKE of exp \* exp  
| AAT of exp \* exp  
| APUT of exp \* exp \* exp  
| ASIZE of exp

This example illustrates a general principle: *in a monomorphic language, polymorphic primitives require special-purpose abstract syntax*. This principle also applies to C and C++, for example, which denote array operations with syntax involving square brackets.

Each of the array operations is governed by rules that say how it is typechecked and evaluated. Type checking is presented in the next section; evaluation is presented here. Because the evaluation *rules* are not particularly interesting, they are omitted from this book; only the code is shown.

To evaluate an array operation, we need to project at least one argument from a value to an array or to an integer. If the program type checks, the projection should always succeed; if a projection fails, there is a bug in the type checker. The projections are implemented by functions `toArray` and `toInt`.

**354a.** *(definitions of functions `toArray` and `toInt` for Typed Impcore 354a)* (S381b)

```
fun toArray (ARRAY a) = a
| toArray _           = raise BugInTypeChecking "non-array value"
fun toInt  (NUM n)   = n
| toInt _            = raise BugInTypeChecking "non-integer value"
```

Given `toArray` and `toInt`, evaluating the array operations is easy. Everything we need is in the `Array` module from ML's Standard Basis Library. The library includes run-time checks for bad subscripts or array sizes; these checks are needed because Typed Impcore's type system is not powerful enough to preclude such errors.

**354b.** *(more alternatives for `ev` for Typed Impcore 354b)* (S388e)

```
| ev (AAT (a, i)) =
  Array.sub (toArray (ev a), toInt (ev i))
| ev (APUT (e1, e2, e3)) =
  let val (a, i, v) = (ev e1, ev e2, ev e3)
  in Array.update (toArray a, toInt i, v);
     v
  end
| ev (AMAKE (len, init)) = ARRAY (Array.array (toInt (ev len), ev init))
| ev (ASIZE a)          = NUM    (Array.length (toArray (ev a)))
```

### 6.3.3 Rules for type constructors: formation, introduction, and elimination

In a monomorphic language, a new type constructor needs new syntax, and new syntactic forms need new type rules. The design of the syntax is guided by the classification of operations in Section 2.5.2: syntax should include forms that create and observe, as well as forms that produce or mutate, or both. What about rules? First, rules should say how to use the new type constructor to make new types. Such rules are called *formation rules*. Second, rules should say what syntactic forms *create new values* that are described by the new type constructor. Such rules are called *introduction rules*. A syntactic form that appears in an introduction rule is analogous to a *creator* function as described in Section 2.5.2. Finally, rules should say what syntactic forms *use the values* that are described by the new type constructor. Such rules are called *elimination rules*. A syntactic form that appears in an elimination rule may be analogous to a mutator or observer function as described in Section 2.5.2.

To help you recognize whether a rule is for formation, introduction, or elimination, here's a trick: draw a box around the type of interest, and see if the rule matches any of these templates:

- A formation rule answers the question, “what types can I make?” Below the line, it has the judgment “ $\boxed{\quad}$  is a type,” where  $\boxed{\quad}$  is a type of interest:

$$\frac{\dots}{\boxed{\quad} \text{ is a type}} \quad (\text{FORMATION TEMPLATE})$$

- An introduction rule answers the question, “how do I *make* a value of the interesting type?” Below the line, it has a typing judgment that ascribes the type of interest to an expression whose form is somehow related to the type. To write that expression, I use a ? mark:

$$\frac{\dots}{\Gamma \vdash ?: \boxed{\quad}} \quad (\text{INTRODUCTION TEMPLATE})$$

- An elimination rule answers the question, “what can I *do* with a value of the interesting type?” Above the line, it has a typing judgment that ascribes the type of interest to an expression whose form is unknown. Such an expression will be written as  $e, e_1, e'$ , or something similar:

$$\frac{\dots \quad \Gamma \vdash e : \boxed{\quad} \quad \dots}{\dots} \quad (\text{ELIMINATION TEMPLATE})$$

These templates work perfectly with the formation, introduction, and elimination rules for arrays. To start, an array type is formed by supplying the type of its elements. In Typed Impcore, the length of an array is not part of its type.<sup>4</sup>

$$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}} \quad (\text{ARRAYFORMATION})$$

An array is made using `make-array`, which is described by an introduction rule.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{INT} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{MAKE-ARRAY}(e_1, e_2) : \text{ARRAY}(\tau)} \quad (\text{MAKEARRAY})$$

The `MAKE-ARRAY` form is definitely related to the array type, and this rule matches the introduction template.

An array is used by indexing it, updating it, or taking its length. Each operation is described by an elimination rule.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \text{INT}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-AT}(e_1, e_2) : \tau} \quad (\text{ARRAYAT})$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \text{INT} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-PUT}(e_1, e_2, e_3) : \tau} \quad (\text{ARRAYPUT})$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \text{ARRAY}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{ARRAY-SIZE}(e) : \text{INT}} \quad (\text{ARRAYSIZE})$$

In each of these rules, an expression of array type ( $e_1$  or  $e$ ) has an arbitrary syntactic form, which need not be related to arrays. These rules match the elimination template.

<sup>4</sup>Because the length of an array is not part of its type, Typed Impcore requires a run-time safety check for every array access. There are powerful type systems in which the length of an array *is* part of its type (Xi and Pfenning 1998), but they are beyond the scope of this book.

AAT	353d
AMAKE	353d
APUT	353d
ARRAY	340f
ASIZE	353d
BugInTypeChecking	S237b
ev	S388e
NUM	340f

*Understanding formation, introduction and elimination*

Not all syntactically correct types and expressions are acceptable in programs; acceptability is determined by formation, introduction, and elimination rules. Formation rules tell us what types are acceptable; for example, in Typed Impcore, `int`, `bool`, and `(array int)` are acceptable types, but `(int bool)`, `array`, and `(array array)` are not. In C, `unsigned` and `unsigned*` are acceptable, but `*` and `*unsigned` are not. Type-formation rules are usually easy to write.

Introduction and elimination rules tell us what terms (expressions) are acceptable. The words “introduction” and “elimination” come from formal logic; the ideas they represent have been adopted into programming languages via the principle of *propositions as types*. This principle says that a type constructor corresponds to a logical connective, a type corresponds to a proposition, and a term of the given type corresponds to a proof of the proposition. For example, logical implication corresponds to the function arrow; if type  $\tau$  corresponds to proposition  $P$ , then the type  $\tau \rightarrow \tau$  corresponds to the proposition “ $P$  implies  $P$ ”; and the identity function of type  $\tau \rightarrow \tau$  corresponds to a proof of “ $P$  implies  $P$ .”

If asked for a proof of a particular proposition or a term of a particular type, you can usually find both “direct” and “indirect” answers. Which is which depends on how the proof or term relates to the logical connective used to make the proposition or to the type constructor used to make the type. For example, if we’re asked for a term of type `ARRAY(bool)`, we can use many syntactic forms; for example, we might refer to the variable `truth-vector`, we might evaluate a conditional that returns an array, or we might apply a function that returns an array. All these forms are indirect: variable reference, conditionals, and function application can produce results of any type and are not specific to arrays. But if we choose the `make-array` form, it builds an array *directly*; a `make-array` term always produces an array.

The direct forms are the *introduction* forms, and their acceptable usage is described by introduction rules. The indirect forms are typically *elimination* forms, described by elimination rules. For example, the conditional is an elimination form for Booleans, and function application is an elimination form for functions. In general, an introduction form puts new information into a proof or a term, whereas an elimination form extracts information that was put there by an introduction form. For example, if I get a Boolean using `array-at`, I’m getting information that was in the array, so I’m using an elimination form for arrays, not an introduction form for Booleans.

An introduction or elimination form is a syntactic form of *expression*, and it is associated with a *type* (or type constructor) that appears in its typing rule. If we’re interested in type constructor  $\mu$ , the typing rule for an introduction form will have a  $\mu$  type in the conclusion; types in premises are often arbitrary types  $\tau$ . The typing rule for an elimination form will have a  $\mu$  type in a premise; the type in the conclusion is often an arbitrary type  $\tau$ , or sometimes a fixed type like `bool`. Identifying introduction and elimination forms is the topic of Exercise 2 on page 397.

In a good design, information created by any introduction form can be extracted by an elimination form, and vice versa. Knowing when we have enough typing rules is the same problem as knowing when we have enough algebraic laws, as discussed in Section 2.5.2 on page 113. Algebraic laws’ *creators* and *producers* relate to a type system’s introduction forms, and *observers* relate to elimination forms.

I don't show you how to turn these rules into code for the type checker; that problem is left for Exercise 18.

357. (function ty, checks type of expression, given  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  **【prototype】** 357)  $\equiv$   
| ty (AAT (a, i)) = raise LeftAsExercise "AAT"  
| ty (APUT (a, i, e)) = raise LeftAsExercise "APUT"  
| ty (AMAKE (len, init)) = raise LeftAsExercise "AMAKE"  
| ty (ASIZE a) = raise LeftAsExercise "ASIZE"

§6.4

Interlude: common  
type constructors

357

## 6.4 INTERLUDE: COMMON TYPE CONSTRUCTORS

Arrays are just one abstraction that is useful for structuring data. Others include functions, products, sums, and mutable references, all of which have proven their worth in many language designs. All these common abstractions have standard type rules, which are described in this section.

*Functions* If functions are first-class values, they should have first-class types. The function type constructor, which takes two arguments, is written using an infix  $\rightarrow$ . Its introduction and elimination forms are  $\lambda$ -abstraction and function application. I describe a  $\lambda$ -abstraction that, like a function definition in Typed Impcore, makes the types of the formal parameters explicit. And to keep the rules simple, I describe only one-argument functions.

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \rightarrow \tau_2 \text{ is a type}} \quad (\text{ARROWFORMATION})$$

$$\frac{\Gamma \{x \mapsto \tau\} \vdash e : \tau'}{\Gamma \vdash \text{LAMBDA}(x : \tau, e) : \tau \rightarrow \tau'} \quad (\text{ARROWINTRO})$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{APPLY}(e_1, e_2) : \tau'} \quad (\text{ARROWELIM})$$

*Products* A product, often called a *pair* or *tuple*, groups together values of different types. It corresponds to ML's “tuple” type, to C's “struct,” to Pascal's “record,” and to the “Cartesian product” you may remember from math class. It is written using an infix  $\times$ . (To motivate the word “product,” think about counting inhabitants: If two values inhabit type `bool` and five values inhabit type `lettergrade`, how many values inhabit product type `bool × lettergrade`?)

In addition to a formation rule, product types are supported by one introduction form, `PAIR`, and two elimination forms, `FST` and `SND`:

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \times \tau_2 \text{ is a type}} \quad (\text{PAIRFORMATION})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{PAIR}(e_1, e_2) : \tau_1 \times \tau_2} \quad (\text{PAIRINTRO})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{FST}(e) : \tau_1} \quad (\text{FST})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{SND}(e) : \tau_2} \quad (\text{SND})$$

AAT	353d
AMAKE	353d
APUT	353d
ASIZE	353d
LeftAsExercise	S237a

Like array operations, the pair operations are polymorphic, i.e., they can work with pairs of any types. We already know these operations, because Chapter 2 presents

them under other names: `cons`, `car`, and `cdr`. As noted in that chapter, their dynamic semantics is given by these algebraic laws:

$$\begin{aligned}\text{FST}(\text{PAIR}(v_1, v_2)) &= v_1 \\ \text{SND}(\text{PAIR}(v_1, v_2)) &= v_2\end{aligned}$$

In addition to `cons`, `car`, and `cdr`, there are other ways to write the concrete syntax of pair operations. Often  $\text{PAIR}(e_1, e_2)$  is written in concrete syntax as  $(e_1, e_2)$ . Syntactic forms `FST` and `SND` may be written to look like functions `fst` and `snd`. In ML, these forms are written `#1` and `#2`; in mathematical notation, projection functions are sometimes written  $\pi_1$  and  $\pi_2$ . Postfix notation is also used, so, e.g., `FST(e)` might be written as `e.1`.

Using `FST` and `SND` to get the elements of a pair can be awkward. In ML code, there is a better idiom: pattern matching, which is a combination of elimination and binding. We can model the ML expression `let val (x, y) = e' in e end` by the abstract syntax `LETPAIRIN(x, y, e', e)`.

$$\frac{\Gamma \vdash e' : \tau_1 \times \tau_2}{\frac{\Gamma \{x \mapsto \tau_1, y \mapsto \tau_2\} \vdash e : \tau}{\Gamma \vdash \text{LETPAIRIN}(x, y, e', e) : \tau}} \quad (\text{LETPAIR})$$

The pair rules can be generalized to give rules for tuples with any number of elements—even zero! The type of tuples with zero elements can serve as a `UNIT` type, since it has only one value: the empty tuple. The tuple rules can be further generalized to give a name to each element of a tuple, which makes it possible to refer to elements by name instead of by position (Exercise 5 on page 400). This is roughly how a C `struct` works.

*Sums* Where a product provides an ordered collection of values of different types, a sum provides a *choice* among values of different types. Almost every language provides obvious support for products, but support for sums is sometimes absent and often hard to recognize. C’s “union” types and Pascal’s “variant records” are examples of sum types. ML’s datatype is also a form of sum type. A value of type  $\tau_1 + \tau_2$  is either a value of type  $\tau_1$  or a value of type  $\tau_2$ , and you can always tell which. (To motivate the word “sum,” count the inhabitants of type `bool + lettergrade`. If you’re not sure about counting inhabitants, do Exercise 3 on page 399.)

In addition to a formation rule, sum types are supported by two introduction forms:

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 + \tau_2 \text{ is a type}} \quad (\text{SUMFORMATION})$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_2 \text{ is a type}}{\Gamma \vdash \text{LEFT}_{\tau_2}(e) : \tau_1 + \tau_2} \quad (\text{LEFT})$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau_1 \text{ is a type}}{\Gamma \vdash \text{RIGHT}_{\tau_1}(e) : \tau_1 + \tau_2} \quad (\text{RIGHT})$$

Sum types are supported by a single elimination form: `case`, which in some languages is called `switch`.<sup>5</sup> I don’t try to invent a readable abstract syntax for the `case` expression; it’s easier to use some ML-like concrete syntax.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2}{\frac{\Gamma \{x_1 \mapsto \tau_1\} \vdash e_1 : \tau}{\frac{\Gamma \{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \text{ of } \text{LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2 : \tau}}} \quad (\text{SUMELIMCASE})$$

<sup>5</sup>The `case` statement was invented by Tony Hoare, who said it was one of the three things in his career he was most proud of. The other two? The Quicksort algorithm and the fact that he started and finished his career in industry.

Just like products, sums can be generalized so that each alternative has a name (Exercise 6 on page 400).

The dynamic semantics of sums is given by the CASE rules of operational semantics:

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle \text{LEFT}(v_1), \sigma' \rangle \quad \ell_1 \notin \text{dom } \sigma'}{\langle e_1, \rho\{x_1 \mapsto \ell_1\}, \sigma'\{\ell_1 \mapsto v_1\} \rangle \Downarrow \langle v, \sigma'' \rangle} \quad (\text{CASELEFT})$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle \text{RIGHT}(v_2), \sigma' \rangle \quad \ell_2 \notin \text{dom } \sigma'}{\langle e_2, \rho\{x_2 \mapsto \ell_2\}, \sigma'\{\ell_2 \mapsto v_2\} \rangle \Downarrow \langle v, \sigma'' \rangle} \quad (\text{CASERIGHT})$$

If a language has first-class functions, there is an alternative to case: the either form. The expression (either  $e f g$ ) applies either  $f$  or  $g$  to the value “carried” inside  $e$ . It is equivalent to  $\text{case } e \text{ of } \text{LEFT}(x_1) \Rightarrow f(x_1) \mid \text{RIGHT}(x_2) \Rightarrow g(x_2)$ .

*Mutable cells* The types shown above are all *immutable*, meaning that once you create a value you can't change it. Since mutation is a valuable programming technique, essential for both procedural and object-oriented programming, it makes sense to have type-system support for mutable cells. Exercise 7 suggests that you design type rules for a type constructor that supports mutation. You can use ML's ref constructor, with its three operations ref, !, and :=, as a model.

## 6.5 TYPE SOUNDNESS

If a program is well typed, what does that imply? The answer depends on the type system. A static type system should guarantee some properties about programs—usually safety properties. In other words, if a program type checks at compile time, that should tell you something about the program's behavior at run time.

A serious professional picks a safety property, designs a type system around it, and proves that “well-typed programs don't go wrong,” i.e., that well-typed programs satisfy the safety property. The type systems in this book can guarantee such properties as “the program never attempts to take car of an integer” or “a function is always called with the correct number of arguments.” More advanced type systems can guarantee such properties as “no array access is ever out of bounds,” “no pointer ever refers to memory that has been deallocated,” or “no private information is ever stored in a public variable.” Whatever the property of interest, if you can prove that a type system guarantees it, you have a *soundness* result.

To express a claim about soundness, we need to know the intended meaning of each type. One common meaning is that a type prescribes a set of values, which I write as  $[\![\tau]\!]$ . Here are some examples:

$$\begin{aligned} [\![\text{int}]\!] &= \{\text{NUMBER}(n) \mid n \text{ is an integer}\} \\ [\![\text{bool}]\!] &= \{\text{BOOLV}(\#t), \text{BOOLV}(\#f)\} \\ [\![\text{sym}]\!] &= \{\text{SYMBOL}(s) \mid s \text{ is a string}\} \\ [\![\tau \rightarrow \tau']\!] &= \text{a set of functions taking values in } [\![\tau]\!] \text{ to values in } [\![\tau']\!] \end{aligned}$$

A simple soundness claim might say that an expression of type  $\tau$  evaluates to a value in the set  $[\![\tau]\!]$ . There are two details: evaluation must terminate, and evaluation and type checking must be done in compatible environments. Here is an example: If for any  $x$  in  $\text{dom } \Gamma$ ,  $x \in \text{dom } \rho \wedge \rho(x) \in [\![\Gamma(x)]\!]$ , and if  $\Gamma \vdash e : \tau$ ,

and if  $\langle \rho, e \rangle \Downarrow v$ , then  $v \in \llbracket \tau \rrbracket$ . That is, if the environments make sense, and if expression  $e$  has type  $\tau$ , and if evaluating  $e$  produces a value, then it produces a value in  $\llbracket \tau \rrbracket$ . To simplify the statement of the claim, I've simplified the theory: there's only one type environment, and I don't worry about a store. A claim like this could be proved by simultaneous induction on the structure of the proof of  $\Gamma \vdash e : \tau$  and the proof of  $\langle \rho, e \rangle \Downarrow v$ .

An even stronger soundness claim would add something to the effect that unless a primitive like / or car fails, or unless  $e$  loops forever, that the evaluation of  $e$  does indeed produce a value. Using a small-step semantics like the one described in Chapter 3, we might say that when all the expressions in an abstract machine are well typed, the machine has either terminated with a value and an empty stack, or it can step to a new state in which all its expressions are still well typed.

A type system with a strong soundness claim rules out most run-time errors. For example, in a sound, typed, Scheme-like language, then if evaluating  $e$  does not attempt to divide by zero or take car or cdr of the empty list, and if evaluating  $e$  doesn't get into an infinite loop, then the evaluation of  $e$  completes successfully.

## 6.6 POLYMORPHIC TYPE SYSTEMS AND TYPED $\mu$ SCHEME

Typed Impcore is both too complicated and not powerful enough. It is too complicated because of the multiple type environments  $\Gamma_\xi$ ,  $\Gamma_\phi$ , and  $\Gamma_\rho$ . It is not powerful enough because each operation that works with values of more than one type, like = or println, has to be built into Typed Impcore's abstract syntax. A function defined by a user can operate only on values of a single type, which is to say it is *monomorphic*. A user-defined, general-purpose array-reversal function, for example, is impossible. Limitations like these make Typed Impcore an accurate model of such languages as C and Pascal.

To restrict each function to a single type hurts programmers. Many useful data structures, including arrays, tables, pointers, lists, products, sums, and objects, inherently work with multiple types: they are *polymorphic*. But when user-defined functions are not also polymorphic, computations like the length of a list either have to be coded without functions, using only the special-purpose syntax needed to compute with lists, or a new length function has to be written for every list type used in the program.

Another problem with a monomorphic language is that adding polymorphic data structures is expensive. A language designer can do it, provided they are willing to add new rules to a type system and to revisit its proof of type soundness. But a programmer can't; unless there is some sort of template or macro system, it is not obvious how a programmer can add a user-defined, polymorphic type *constructor* such as the env type constructor we use for environments in Chapters 5 to 10. At best, a programmer can add a new base type, not a new type constructor.

These problems are solved by *polymorphic type systems*. Such type systems make it possible for a programmer to write polymorphic functions, and for anyone—a programmer or a language designer—to add new type constructors. Our first polymorphic type system is designed to make things as easy as possible for the language designer and implementor; it does not make things easy for programmers. The type system is part of a language called *Typed micro-Scheme*, or Typed  $\mu$ Scheme for short. Typed  $\mu$ Scheme is patterned after  $\mu$ Scheme: it uses the same *values* as  $\mu$ Scheme, similar abstract syntax, and a static, polymorphic type system.

We begin our study of Typed  $\mu$ Scheme with concrete syntax. We continue by examining *kinds* and *quantified types*, which are the two ideas at the core of the type system. Kinds are used to ensure that every type written in the source code is well

formed and meaningful; kinds classify types in much the same way that types classify terms. Quantified types are used to express polymorphism; quantified types make it possible to implement polymorphic operations using ordinary functions instead of special-purpose abstract syntax. Once these two ideas are presented and are integrated into  $\mu$ Scheme, the rest of the chapter presents technical details needed to make a polymorphic type system work: *type equivalence* and *substitution*. Type equivalence is a relation that shows when two types cannot be distinguished by any program, even if they don't look identical. And substitution is the mechanism by which a polymorphic value is *instantiated* so it can be used.

§6.6.2  
A replacement for  
type-formation  
rules: kinds

---

361

### 6.6.1 Concrete syntax of Typed $\mu$ Scheme

Typed  $\mu$ Scheme is much like  $\mu$ Scheme, except as follows.

- Function definitions and `lambda` abstractions require type annotations for parameters.
- Function definitions require explicit result types.
- All `letrec` expressions require type annotations for bound names—and each name may be bound only to a `lambda` abstraction.
- Instead of  $\mu$ Scheme's single `val` form, Typed  $\mu$ Scheme provides two forms: `val-rec`, which is recursive and defines only functions, and `val`, which is non-recursive and can define any type of value. Only the `val-rec` form requires a type annotation. Typed  $\mu$ Scheme's `val` and `val-rec` forms resemble the corresponding forms in Standard ML.

The type system of Typed  $\mu$ Scheme is more powerful than that of Typed Impcore:

- Typed  $\mu$ Scheme adds quantified types, which are written with `forall`. To introduce values of quantified type, Typed  $\mu$ Scheme adds a new syntactic form of expression: `type-lambda`. To eliminate values of quantified type, Typed  $\mu$ Scheme adds the syntactic form `@`.
- Syntactically, Typed  $\mu$ Scheme does not distinguish a “type” from a “type constructor”; both can be called “types,” and both are in the syntactic category *type-exp*. The category, which is called “type-level expression,” also includes ill-formed nonsense that is neither type nor type constructor, like `(int int)`.
- In Typed  $\mu$ Scheme, only the type constructor for functions needs special-purpose syntax; its introduction form is `lambda` and its elimination form is function application. Other type constructors, like pairs and arrays, can be added without adding new syntax or new typing rules; they just go into the initial basis. And their operations are implemented as ordinary (primitive) functions, without special-purpose syntax.

The syntax of Typed  $\mu$ Scheme is shown in Figure 6.3 on the following page.

### 6.6.2 A replacement for type-formation rules: kinds

Types in source code are written by programmers, and they can't be trusted. “Types” like `(int int)` are ill formed and must be rejected. In Typed Impcore, we decide which types are well formed or ill formed by consulting type-formation rules. And for a language with a fixed set of types and type constructors, that's fine. But in Typed  $\mu$ Scheme, we want to be able to add new type constructors without

```

def      ::= (val variable-name exp)
         | (val-rec [variable-name : type-exp] exp)
         | (define type-exp function-name (formals) exp)
         | exp
         | (use file-name)
         | unit-test

unit-test ::= (check-expect exp exp)
           | (check-assert exp)
           | (check-error exp)
           | (check-type exp type-exp)
           | (check-type-error def)

exp      ::= literal
         | variable-name
         | (set variable-name exp)
         | (if exp exp exp)
         | (while exp exp)
         | (begin {exp})
         | (exp {exp})
         | (let-keyword ({[variable-name exp]}) exp)
         | (letrec [{([variable-name : type-exp] exp)]} exp)
         | (lambda (formals) exp)
         | (type-lambda [type-formals] exp)
         | [@ exp {type-exp}]

let-keyword ::= let | let*

formals   ::= {[variable-name : type-exp]}

type-formals ::= {'type-variable-name'}

type-exp   ::= type-constructor-name
         | 'type-variable-name
         | (forall ({'type-variable-name}) type-exp)
         | ({type-exp} -> type-exp)
         | (type-exp {type-exp})

literal    ::= numeral | #t | #f | 'S-exp | (quote S-exp)

S-exp      ::= literal | symbol-name | ({S-exp})

numeral   ::= sequence of digits, possibly prefixed with a minus sign

*-name    ::= sequence of characters not a numeral and not containing
             (, ), [, ], {, }, ;, or whitespace

```

Figure 6.3: Concrete syntax of Typed  $\mu$ Scheme

adding new rules. A better approach uses just a few rules to encompass arbitrarily many type constructors. This approach assigns each type constructor a *kind*.

Kinds classify types (and type constructors) in much the same way that types classify terms. A kind shows how a type constructor may be used. For example, both `int` and `array` are type constructors of Typed  $\mu$ Scheme, but the kinding rules require that they be used differently. The `int` constructor is a kind of constructor that is a type all by itself; the `array` constructor is a kind of constructor that has to be applied to an element type in order to make another type. We write the first kind  $*$  and pronounce it “type”; we write the second kind  $* \Rightarrow *$  and pronounce it “type arrow type.” To use a kind, we make a formal judgment that a type constructor has a particular kind. Formally, we write  $\tau :: \kappa$  to say that type constructor  $\tau$  has kind  $\kappa$ . For example, the judgment “ $\tau :: *$ ” is equivalent to the judgment “ $\tau$  is a type” we use in Typed Impcore.

Like types, kinds are defined by a simple inductive structure: there is one base kind, “type” ( $*$ ), and other kinds are made using arrows ( $\Rightarrow$ ). As concrete notation, we write

$$\kappa ::= * \mid \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa$$

Types that are inhabited by values, like `int` or `(list bool)`, have kind  $*$ . Types of other kinds, like `list` and `array`, are ultimately used to make types of kind  $*$ .

Some common kinds, with example type constructors of those kinds, are as follows:

$*$	<code>int, bool, unit</code>
$* \Rightarrow *$	<code>list, array, option</code>
$* \times * \Rightarrow *$	<code>pair, sum, Standard ML's -&gt;</code>

More exotic kinds can be found in languages like Haskell, which includes not only “monads,” which are all types of kind  $* \Rightarrow *$ , but also “monad transformers,” which are types of kind  $(* \Rightarrow *) \Rightarrow (* \Rightarrow *)$ .

The syntax of kinds induces these kind-formation rules:

$$\frac{}{* \text{ is a kind}} \quad \text{(KINDFORMATIONTYPE)}$$

$$\frac{\kappa_1, \dots, \kappa_n \text{ are kinds} \quad \kappa \text{ is a kind}}{\kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \text{ is a kind}} \quad \text{(KINDFORMATIONARROW)}$$

How do we know which type constructors have which kinds? We put the kind of each type constructor in a *kind environment*, written  $\Delta$ . Here is an example environment that shows common type constructors and their kinds. A binding in a kind environment is written using the  $::$  symbol, not  $\mapsto$ . The  $::$  symbol is pronounced “has kind.”

$$\Delta_0 = \{ \text{int} :: *, \text{bool} :: *, \text{unit} :: *, \text{pair} :: * \times * \Rightarrow *, \\ \text{sum} :: * \times * \Rightarrow *, \text{array} :: * \Rightarrow *, \text{list} :: * \Rightarrow * \}$$

This environment shows how both `int` and `array` may be used. And we can add new type constructors just by adding them to  $\Delta$ .

A kind environment is used to tell what *types* are well formed. No matter how many type constructors are defined, Typed  $\mu$ Scheme handles them using just three type-formation rules:

- A type can be formed by writing a type constructor. In abstract syntax, I write  $\text{TYCON}(\mu)$ , where  $\mu$  is the name of the constructor. In concrete syntax, we write the name of the constructor, like `int`. A type constructor is well formed if and only if it is bound in  $\Delta$ .

- A type can be formed by applying a type to other types. In abstract syntax, I write  $\text{CONAPP}(\tau, [\tau_1, \dots, \tau_n])$ , where  $\tau$  and  $\tau_1, \dots, \tau_n$  are type-level expressions. In concrete syntax, we write application of a type constructor using the same concrete syntax as for application of a function. For example, we write  $(\text{list int})$  for the type “list of integer.” A constructor application is well formed if its arguments have the kinds it expects, as formalized in the  $\text{KINDAPP}$  rule below.

- A type can be a function type. In abstract syntax, I write  $\tau_1 \times \dots \times \tau_n \rightarrow \tau$ , where  $\tau_1, \dots, \tau_n$  are the argument types and  $\tau$  is the result type. In concrete syntax, we write  $(\tau_1 \dots \tau_n \rightarrow \tau)$ .<sup>6</sup> A function type is well formed if and only if types  $\tau_1$  to  $\tau_n$  and  $\tau$  all have kind  $*$ .

These rules are formalized using the *kinding judgment*  $\Delta \vdash \tau :: \kappa$ . This judgment says that in kind environment  $\Delta$ , type-level expression  $\tau$  has kind  $\kappa$ . Kinds classify types in much the same way that types classify expressions.

$$\frac{\mu \in \text{dom } \Delta}{\Delta \vdash \text{TYCON}(\mu) :: \Delta(\mu)} \quad (\text{KINDINTROCON})$$

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \dots \times \kappa_n \Rightarrow \kappa \quad \Delta \vdash \tau_i :: \kappa_i, \quad 1 \leq i \leq n}{\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \dots, \tau_n]) :: \kappa} \quad (\text{KINDAPP})$$

$$\frac{\Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \quad \Delta \vdash \tau :: *}{\Delta \vdash \tau_1 \times \dots \times \tau_n \rightarrow \tau :: *} \quad (\text{KINDFUNCTION})$$

No matter how many type constructors we may add to Typed  $\mu$ Scheme, these kinding rules tell us everything we ever need to know about the formation of types. Compare this situation with the situation in Typed Impcore. In Typed Impcore, we need the `BASETYPES` rule for `int` and `bool`. To add arrays we need the `ARRAYFORMATION` rule. To add lists we would need a list-formation rule (Exercise 4 on page 400). And so on. Unlike Typed Impcore’s type system, Typed  $\mu$ Scheme’s type system can easily be extended with new type constructors (Exercises 10 to 13 starting on page 401). Similar ideas are used in languages in which *programmers* can define new type constructors, including  $\mu$ ML (Chapter 8) and Molecule (Chapter 9).

### Implementing kinds

A kind is represented using the datatype `kind`.

**364a.** *(kinds for typed languages 364a) ≡* (S394b) 364b▷  
`datatype kind = TYPE` (\* kind of all types \*)  
`| ARROW of kind list * kind` (\* kind of many constructors \*)

Kinds are equal if and only if they are identical.

**364b.** *(kinds for typed languages 364a) +≡* (S394b) ▷364a  
`fun eqKind (TYPE, TYPE) = true`  
`| eqKind (ARROW (args, result), ARROW (args', result')) =`  
`eqKinds (args, args') andalso eqKind (result, result')`  
`| eqKind (_, _) = false`  
`and eqKinds (ks, ks') = ListPair.allEq eqKind (ks, ks')`

<sup>6</sup>The arrow that signifies a function occurs in the *middle* of the parentheses, between types. In other words, the function arrow  $\rightarrow$  is an *infix* operator. This infix syntax violates Lisp’s *prefix* convention, in which keywords, type constructors, and operators always come first, immediately after an open parenthesis. Prefix syntax might look like “ $(\text{function}(\tau_1 \dots \tau_n) \tau)$ .” But when functions take or return other functions, prefix syntax is too hard to read.

The kinds of the primitive type constructors, which populate the initial kind environment  $\Delta_0$ , are represented as follows.

365. *(primitive type constructors for Typed  $\mu$ Scheme :: 365)*  $\equiv$

(391e)

```
("int", TYPE) ::  
("bool", TYPE) ::  
("sym", TYPE) ::  
("unit", TYPE) ::  
("list", ARROW ([TYPE], TYPE)) ::
```

The kind system and the type-formation rules shown above replace the type-formation rules of Typed Impcore. To get to Typed  $\mu$ Scheme, however, we need something more: quantified types.

§6.6.3  
*The heart of polymorphism:  
quantified types*

365

### 6.6.3 *The heart of polymorphism: quantified types*

A quantified type describes a polymorphic function or value. Polymorphic functions aplenty can be found in Chapter 2; one of the simplest is `length`. Function `length` can be applied to any list, no matter what the types of its elements.

```
define length (xs)  
  (if (null? xs) 0 (+ 1 (length (cdr xs))))
```

If you're going to define `length` in a typed language, what type should you give it? In Typed Impcore, even if we add lists, there's no good answer. As in C, we would have to write a version for each list type:

```
define int lengthI ([xs : (list int)])  
  (if (null? xs) 0 (+ 1 (lengthI (cdr xs))))  
define int lengthB ([xs : (list bool)])  
  (if (null? xs) 0 (+ 1 (lengthB (cdr xs))))  
define int lengthS ([xs : (list sym)])  
  (if (null? xs) 0 (+ 1 (lengthS (cdr xs))))
```

Such duplication wastes effort; except for the types, the functions are identical. The problem is that using only Typed Impcore, we have no way to say that the behavior of `length` is independent of the type of the elements in the list. In a polymorphic type system, we solve this problem by introducing *type variables* and *quantified types*.

A type variable stands for an unknown type. A quantified type gives us permission to *substitute* any type for the unknown type. In text, I write type variables using the Greek letters  $\alpha$ ,  $\beta$ , and  $\gamma$ ; I write quantified types using  $\forall$ . For example, I write the type of a polymorphic `length` function as  $\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$ . In Typed  $\mu$ Scheme this type is written (`forall ['a] ((list 'a) -> int)`). To use this `length` function on a list of integers, we *instantiate* it for integers; this operation strips “ $\forall \alpha.$ ” from the front of the type, and in what remains, substitutes `int` for  $\alpha$ . The type of the resulting instance is `int list -> int`, or in Typed  $\mu$ Scheme, a function of type `((list int) -> int)`.

Like `lambda`, the  $\forall$  is a *binding construct*, and the variable  $\alpha$  is sometimes called a *type parameter*. Like the name of a formal parameter, the name of a type parameter doesn't matter; for example, the type of the `length` function could also be written  $\forall \beta. \beta \text{ list} \rightarrow \text{int}$ , and its meaning would be unchanged. That's because the meaning of a quantified type is determined by how it behaves when we strip the quantifier and substitute for the bound type variable.

To express type variables and quantified types, we need suitable abstract syntax. I use `TYVAR` and `FORALL`. And to know when quantified types are well formed, we need kinding rules for `TYVAR` and `FORALL`. (The kind system replaces the type-formation rules used in Typed Impcore; remember the slogan “just as types classify terms, kinds classify types.”)

<code>args</code>	S425b
<code>args'</code>	S425b
<code>ARROW</code>	S425a
<code>eqKind</code>	S425b
<code>eqKinds</code>	S425b
<code>type kind</code>	S425a
<code>ks'</code>	S425b
<code>result</code>	S425b
<code>result'</code>	S425b
<code>TYPE</code>	S425a

The kinding rule for a type variable says that, just like a type constructor, a type variable is looked up in the environment  $\Delta$ .

$$\frac{\alpha \in \text{dom } \Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)} \quad (\text{KINDINTROVAR})$$

The kind of a quantified type is always  $*$ , and the FORALL quantifier may be used only over types of kind  $*$ . Within the body of the FORALL, the quantified variables stand for types. We therefore introduce them into the kind environment with kind  $*$ .

$$\frac{\Delta \{ \alpha_1 :: *, \dots, \alpha_n :: * \} \vdash \tau :: *}{\Delta \vdash \text{FORALL}((\alpha_1, \dots, \alpha_n), \tau) :: *} \quad (\text{KINDALL})$$

In Typed  $\mu$ Scheme, as in Standard ML, all type variables have kind  $*$ . But in some polymorphic type systems, type variables may have any kind. For example, the functional language Haskell uses a type system that allows type variables to have other kinds.

To build types in Typed  $\mu$ Scheme, we use type-level expressions. These include not only type variables and the FORALL quantifier but also type constructors, constructor application, and function types. In my ML code, I represent the abstract syntax of type-level expressions using the type tyex.

**366a.** *(types for Typed  $\mu$ Scheme 366a)*  $\equiv$  (S394b) 390a▷

```
datatype tyex = TYCON of name          (* type constructor *)
              | CONAPP of tyex * tyex list (* type-level application *)
              | FUNTY of tyex list * tyex (* function type *)
              | FORALL of name list * tyex (* quantified type *)
              | TYVAR of name           (* type variable *)
```

Even though not every tyex represents a well-formed type, it's easier to call them all "types"—except when we have to be careful.

The abstract syntax defined by tyex corresponds to the *type-exp* nonterminal in the concrete syntax on page 362. As examples of this concrete syntax, here are the types of some polymorphic functions and values related to lists.

**366b.** *(transcript 366b)*  $\equiv$  367a▷

```
-> length
<function> : (forall ['a] ((list 'a) -> int))
-> cons
<function> : (forall ['a] ('a (list 'a) -> (list 'a)))
-> car
<function> : (forall ['a] ((list 'a) -> 'a))
-> cdr
<function> : (forall ['a] ((list 'a) -> (list 'a)))
-> '()
() : (forall ['a] (list 'a))
```

Polymorphism is not restricted to functions: even though it is not a function, the empty list has a quantified type.

The parenthesized-prefix syntax above is easy to parse and easy to understand, but not so easy to read. In the text, I also use an algebraic notation in which I write type constructors  $\mu$ , type variables  $\alpha$ , and types  $\tau$ . I follow the ML postfix convention for application of type constructors, and I use special notation for the function constructor:

$$\tau ::= \mu \mid \alpha \mid (\tau_1, \dots, \tau_n) \tau \mid \forall \alpha_1, \dots, \alpha_n . \tau \mid \tau_1 \times \dots \times \tau_n \rightarrow \tau$$

Using this notation, the types of `length`, `cons`, `car`, `cdr`, and `'()` are written as follows:

```

length : ∀α . α list → int
cons   : ∀α . α × α list → α list
car    : ∀α . α list → α
cdr    : ∀α . α list → α list
'()   : ∀α . α list

```

To use a value with a quantified type, we have to say what unknown type each bound type variable stands for. This process is called *instantiation*. To instantiate a value that has a quantified type, we use the @ form in the concrete syntax. This form gives the value to be instantiated, plus one type to be substituted for each bound type variable. For reasons that may be evident from the syntax, instantiation is sometimes also called *type application*.<sup>7</sup> An instantiation or application may be pronounced with the word “at,” as in “length at integer.”

§6.6.3  
The heart of  
polymorphism:  
quantified types

367

**367a.** *(transcript 366b)* +≡

△366b 367b ▷

```

-> (val length-at-int [@ length int])
length-at-int : ((list int) -> int)
-> (val cons-at-bool [@ cons bool])
cons-at-bool : (bool (list bool) -> (list bool))
-> pair
<function> : (forall ['a 'b] ('a 'b -> (pair 'a 'b)))
-> (val car-at-pair [@ car (pair sym int)])
car-at-pair : ((list (pair sym int)) -> (pair sym int))
-> (val cdr-at-sym [@ cdr sym])
cdr-at-sym : ((list sym) -> (list sym))
-> (val empty-at-int [@ '() int])
() : (list int)

```

In each case, we get the type of the instantiated function by *substituting* each type parameter for the corresponding type variable in the `forall`.

Getting the instantiations you want takes thought and practice. A common mistake is to instantiate by substituting the type you want the instantiated function to have. If you want a function of type  $((\text{list bool}) \rightarrow \text{int})$ , you should instantiate `length` at `bool`. If you try instead to instantiate `length` at the desired type  $((\text{list bool}) \rightarrow \text{int})$ , the results won’t be what you hope for:

**367b.** *(transcript 366b)* +≡

△367a 367c ▷

```

-> (val useless-length [@ length ((list bool) -> int)])
useless-length : ((list ((list bool) -> int)) -> int)

```

Once instantiated, a polymorphic function is monomorphic, and it can be applied to values. A function like `useless-length` has a good type but can’t be used to take the length of a list of Booleans.

**367c.** *(transcript 366b)* +≡

△367b 371 ▷

```

-> (length-at-int '(1 4 9 16 25))
5 : int
-> (cons-at-bool #t '#f '#f)
(#t #f #f) : (list bool)
-> (car-at-pair ([@ cons (pair sym int)]
                  ([@ pair sym int] 'Office 231)
                  [@ '() (pair sym int)]))
(Office . 231) : (pair sym int)
-> (cdr-at-sym '(a b c d))
(b c d) : (list sym)

```

type name 310a

<sup>7</sup>The similarity between instantiation and function application goes beyond mere syntactic resemblance. Instantiation is defined by substituting for type variables bound by  $\forall$ . And in Church’s lambda calculus, which is the most fundamental theoretical model of programming languages, function application is defined by substituting for term variables bound by  $\lambda$ . Instantiation is not the same thing as a function from types to terms, but they are related.

```
-> (useless-length '(#t #f #f))
type error: function useless-length of type ...
-> [@ length bool]
<function> : ((list bool) -> int)
-> ([@ length bool] '(#t #f #f))
3 : int
```

As the car-at-pair example and the final two length examples show, instantiated values don't have to be named; they can be used directly.

The instantiation operator @ lets you *use* a polymorphic value; it is the elimination form for a quantified type. But to enjoy the full power of polymorphism, you must also be able to *create* a polymorphic value; you need an introduction form. In Typed  $\mu$ Scheme, the introduction form is written using type-lambda; it is sometimes called *type abstraction*. As an example, I use type-lambda to define the polymorphic functions list1, list2, and list3.

**368a.** *<predefined Typed  $\mu$ Scheme functions 368a>* 368b ▷

```
(val list1 (type-lambda ['a] (lambda ([x : 'a])
                                     ([@ cons 'a] x [@ '() 'a]))))
```

**368b.** *<predefined Typed  $\mu$ Scheme functions 368a>*  $\doteqdot$  368a 368c ▷

```
(val list2 (type-lambda ['a] (lambda ([x : 'a] [y : 'a])
                                     ([@ cons 'a] x ([@ list1 'a] y))))))
```

```
(val list3 (type-lambda ['a] (lambda ([x : 'a] [y : 'a] [z : 'a])
                                     ([@ cons 'a] x ([@ list2 'a] y z))))))
```

Some of the higher-order functions in Chapter 2 are also easy to write using type-lambda. Their types are as follows:

$$\begin{aligned} \circ &: \forall \alpha, \beta, \gamma. (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \\ \text{curry} &: \forall \alpha, \beta, \gamma. (\alpha \times \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow (\beta \rightarrow \gamma)) \\ \text{uncurry} &: \forall \alpha, \beta, \gamma. (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow (\alpha \times \beta \rightarrow \gamma) \end{aligned}$$

Their implementations use type-lambda:

**368c.** *<predefined Typed  $\mu$ Scheme functions 368a>*  $\doteqdot$  368b 368d ▷

```
(val o (type-lambda ['a 'b 'c]
                      (lambda ([f : ('b -> 'c)])
                        (g : ('a -> 'b)))
                      (lambda ([x : 'a]) (f (g x))))))

(val curry (type-lambda ['a 'b 'c]
                         (lambda ([f : ('a 'b -> 'c)])
                           (lambda ([x : 'a]) (lambda ([y : 'b]) (f x y))))))

(val uncurry (type-lambda ['a 'b 'c]
                            (lambda ([f : ('a -> ('b -> 'c))])
                              (lambda ([x : 'a] [y : 'b]) ((f x) y)))))
```

Other higher-order functions are not only polymorphic but also recursive. To define such functions, we nest letrec (for recursion) inside type-lambda (for polymorphism).

**368d.** *<predefined Typed  $\mu$ Scheme functions 368a>*  $\doteqdot$  368c 369b ▷

```
(val length
    (type-lambda ['a]
      (letrec
        ([length-mono : ((list 'a) -> int)]
          (lambda ([xs : (list 'a)])
            (if ([@ null? 'a] xs)
                0
                (+ 1 (length-mono ([@ cdr 'a] xs)))))))
      length-mono)))
```

369a. *(predefined Typed  $\mu$ Scheme functions 368a)* +≡

△369b

```
(val filter
  (type-lambda ('a)
    (letrec
      [([filter-mono : (('a -> bool) (list 'a) -> (list 'a))]
       (lambda ([p? : ('a -> bool)] [xs : (list 'a)])
         (if ([@ null? 'a] xs)
             []
             [(@ '() 'a]
               (if (p? (@ car 'a] xs))
                   ([@ cons 'a] (@ car 'a] xs)
                     (filter-mono p? (@ cdr 'a] xs)))
                   (filter-mono p? (@ cdr 'a] xs)))))))
      (filter-mono)))))

(val map
  (type-lambda ('a 'b)
    (letrec
      [([map-mono : (('a -> 'b) (list 'a) -> (list 'b))]
       (lambda ([f : ('a -> 'b)] [xs : (list 'a)])
         (if ([@ null? 'a] xs)
             []
             [(@ '() 'b]
               ([@ cons 'b] (f (@ car 'a] xs))
                 (map-mono f (@ cdr 'a] xs)))))))
      (map-mono))))
```

§6.6.3

The heart of  
polymorphism:  
quantified types

369

Figure 6.4: Predefined recursive, polymorphic functions `filter` and `map`

I call the inner function `length-mono` because it—like any other value introduced with `lambda`—is monomorphic, operating only on lists of the given element type '`a`: the recursive call to `length-mono` does *not* require an instantiation.

Every polymorphic, recursive function is defined using the same pattern: `val` to `type-lambda` to `letrec`. As another example, here is an explicitly typed version of the reverse-append function:

369b. *(predefined Typed  $\mu$ Scheme functions 368a)* +≡

△368d 369a▷

```
(val revapp
  (type-lambda ['a]
    (letrec [([revapp-mono : ((list 'a) (list 'a) -> (list 'a))]
            (lambda ([xs : (list 'a)] [ys : (list 'a)])
              (if ([@ null? 'a] xs)
                  ys
                  (revapp-mono (@ cdr 'a] xs)
                    ([@ cons 'a] (@ car 'a] xs) ys))))])
    (revapp-mono)))
```

Two more examples appear in Figure 6.4. As shown by these examples, explicit types and polymorphism impose a heavy notational burden. At the cost of a little expressive power, that burden can be lifted by *type inference*, as in nano-ML (Chapter 7). Or the burden can be lightened by instantiating an entire module at once, as in Molecule (Chapter 9).

To summarize this section, the essential new idea in a polymorphic type system is the *quantified type*. It comes with its own special-purpose syntax: `forall` to form a quantified type, `type-lambda` to introduce a quantified type, and `@` to eliminate a quantified type. The rest of this chapter shows how a type system based on quantified types is combined with  $\mu$ Scheme to produce Typed  $\mu$ Scheme.

**370a.** *(definitions of exp and value for Typed  $\mu$ Scheme 370a)*  $\equiv$ (S393b) 370b  $\triangleright$ 

```

datatype exp = LITERAL of value
  | VAR      of name
  | SET      of name * exp
  | IFX      of exp * exp * exp
  | WHILEX   of exp * exp
  | BEGIN    of exp list
  | APPLY    of exp * exp list
  | LETX     of let_kind * (name * exp) list * exp
  | LETRECX  of ((name * tyex) * exp) list * exp
  | LAMBDA   of lambda_exp
  | TYLAMBDA of name list * exp
  | TYAPPLY  of exp * tyex list
and let_kind = LET | LETSTAR

```

The values of Typed  $\mu$ Scheme are the same as the values of  $\mu$ Scheme; adding a type system doesn't change the representation used at run time.

**370b.** *(definitions of exp and value for Typed  $\mu$ Scheme 370a)*  $+ \equiv$ (S393b)  $\triangleleft$  370a

```

and   value = NIL
  | BOOLV    of bool
  | NUM      of int
  | SYM      of name
  | PAIR     of value * value
  | CLOSURE  of lambda_value * value ref env
  | PRIMITIVE of primitive
withtype primitive = value list -> value (* raises RuntimeError *)
and lambda_exp = (name * tyex) list * exp
and lambda_value = name      list * exp

```

The definitions of Typed  $\mu$ Scheme are like those of Typed Impcore, plus the recursive binding form VALREC (see sidebar on the facing page).

**370c.** *(definition of def for Typed  $\mu$ Scheme 370c)*  $\equiv$ 

(S393b)

```

datatype def = VAL of name * exp
  | VALREC of name * tyex * exp
  | EXP    of exp
  | DEFINE of name * tyex * lambda_exp

```

### 6.6.5 Type rules for Typed $\mu$ Scheme

The type rules for Typed  $\mu$ Scheme are very like the rules for Typed Impcore. The important design differences are these:

- There is only one type environment,  $\Gamma$ . It maps values to types.
- There is a kind environment,  $\Delta$ , which keeps track of the kinds of type variables and type constructors.
- There are no special-purpose typing rules associated with individual type constructors; type formation is handled by the general-purpose kinding rules, and the rest is handled by the general-purpose typing rules for type abstraction, instantiation, lambda abstraction, and application.

Like  $\mu$ Scheme, Typed  $\mu$ Scheme has more definition forms that it really needs: given that it has `lambda` and `letrec` expressions, all it really needs is `val` (Exercise 21 on page 404). But Typed  $\mu$ Scheme is not meant to be as small as possible; it's meant to convey understanding and to facilitate comparisons. So it includes `define`.

In untyped  $\mu$ Scheme, `define` is just syntactic sugar for a `val` binding to a `lambda` (page 122). Because a  $\mu$ Scheme `val` makes its bound name visible on the right-hand side, such functions can even be recursive.  $\mu$ Scheme's operational semantics initializes the name to an unspecified value, then overwrites the name with a closure. This semantics extends to *all* `val` bindings; for example, in untyped  $\mu$ Scheme you can write `(val n (+ n 1))`, and on the right-hand side, the value of `n` is unspecified. But in Typed  $\mu$ Scheme, we can't afford to compute with unspecified values; if an expression like `(+ n 1)` typechecks, we have to *know* that `n` has an integer value.

Typed  $\mu$ Scheme works around this problem by changing the operational semantics of `val` back to the semantics used in Impcore: in Typed  $\mu$ Scheme, as in Impcore, a `val` binding is not recursive, and the name being defined is not visible on the right-hand side.

**371.** *<transcript 366b>*  $+ \equiv$

*<367c 377a>*

```
-> (val n (+ n 1))
Name n not found
```

For recursive bindings, Typed  $\mu$ Scheme introduces the new form `val-rec`, which has the same operational semantics as  $\mu$ Scheme's `val` form. To ensure that the right-hand side does not evaluate the name before it is initialized, Typed  $\mu$ Scheme restricts the right-hand side to be a `lambda` form. And to make it possible to typecheck recursive calls, Typed  $\mu$ Scheme requires a type annotation that gives the type of the bound name. With `val-rec` and `lambda`, it's easy to express `define` as syntactic sugar.

The distinction between `val` and `val-rec` can be found in other languages. Look for it! For example, in C, type definitions act like `val`, but in Modula-3, they act like `val-rec`. And in Haskell, *every* definition form acts like `val-rec`! (Haskell gets away with this because no definition form evaluates its right-hand side.)

There are also two important technical details:

- In a `forall` type, the names of quantified type variables are not supposed to matter. This detail affects any decision about whether two types are the same.
- Type application using `@` works by substituting a type for a type variable. And when `forall` types are nested, substitution is easy to get wrong.

type env	310b
type name	310a
type tyex	366a

As I present the rules, I take it for granted that the names of quantified type variables don't matter and that substitution is implemented correctly. To enable you to implement the rules, I then present detailed implementations of type-equivalence testing and substitution (Sections 6.6.6 and 6.6.7).

Type rules for expressions

The typing judgment for an expression is  $\boxed{\Delta, \Gamma \vdash e : \tau}$ , meaning that given kind environment  $\Delta$  and type environment  $\Gamma$ , expression  $e$  has type  $\tau$ . I start with literal expressions; the type of a literal depends only on its value. Assigning types to literal values requires quite a few rules, but they are what you would expect for homogeneous lists. There is one fine point: empty lists are polymorphic, but non-empty lists are monomorphic.

$$\frac{\Delta, \Gamma \vdash \text{LITERAL}(\text{NUM}(n)) : \text{int} \quad \Delta, \Gamma \vdash \text{LITERAL}(\text{BOOLV}(n)) : \text{bool}}{\Delta, \Gamma \vdash \text{LITERAL}(\text{SYM}(n)) : \text{sym}} \quad (\text{LITERALS1})$$

$$\frac{\Delta, \Gamma \vdash \text{LITERAL}(v) : \tau}{\Delta, \Gamma \vdash \text{LITERAL}(\text{NIL}) : \forall \alpha . \alpha \text{ list} \quad \Delta, \Gamma \vdash \text{LITERAL}(\text{PAIR}(v, \text{NIL})) : \tau \text{ list}} \quad (\text{LISTLITERALS1})$$

$$\frac{\Delta, \Gamma \vdash \text{LITERAL}(v) : \tau \quad \Delta, \Gamma \vdash \text{LITERAL}(v') : \tau \text{ list}}{\Delta, \Gamma \vdash \text{LITERAL}(\text{PAIR}(v, v')) : \tau \text{ list}} \quad (\text{LISTLITERALS2})$$

The use of a variable is well typed if the variable is bound in the type environment  $\Gamma$ .

$$\frac{x \in \text{dom } \Gamma \quad \Gamma(x) = \tau}{\Delta, \Gamma \vdash \text{VAR}(x) : \tau} \quad (\text{VAR})$$

$$\frac{\Delta, \Gamma \vdash e : \tau \quad x \in \text{dom } \Gamma \quad \Gamma(x) = \tau}{\Delta, \Gamma \vdash \text{SET}(x, e) : \tau} \quad (\text{SET})$$

As in Typed Impcore, a `while` loop is well typed if the condition is Boolean and the body is well typed. The result has type `unit`.

$$\frac{\Delta, \Gamma \vdash e_1 : \text{bool} \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash \text{WHILE}(e_1, e_2) : \text{unit}} \quad (\text{WHILE})$$

Also as in Typed Impcore, a conditional expression is well typed if the condition is Boolean and the two branches have the same type  $\tau$ . In that case, the type of the conditional expression is also  $\tau$ .

$$\frac{\Delta, \Gamma \vdash e_1 : \text{bool} \quad \Delta, \Gamma \vdash e_2 : \tau \quad \Delta, \Gamma \vdash e_3 : \tau}{\Delta, \Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau} \quad (\text{IF})$$

As in Typed Impcore, a sequence is well typed if its subexpressions are well typed.

$$\frac{\Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Delta, \Gamma \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n} \quad (\text{BEGIN})$$

Also as in Typed Impcore, the empty `BEGIN` has type `unit`.

$$\frac{}{\Delta, \Gamma \vdash \text{BEGIN}() : \text{unit}} \quad (\text{EMPTYBEGIN})$$

The rule for `LET` is straightforward: compute the types of the bound names, then check the body using those types.

$$\frac{\Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \quad \Delta, \Gamma \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LET})$$

The letrec form is intended for recursive functions. In a form like

$$\begin{aligned} & \text{(letrec } [[f_1 : \tau_1] e_1] \\ & \quad [[f_2 : \tau_2] e_2)] \\ & \quad e), \end{aligned}$$

each bound name  $f_i$  is visible during the evaluation of each right-hand side  $e_i$ . But at run time,  $f_1$  and  $f_2$  don't get their values until after  $e_1$  and  $e_2$  have been evaluated. Here's the operational semantics:

$$\begin{aligned} & \ell_1, \ell_2 \notin \text{dom } \sigma \text{ (and all distinct)} \\ & \rho' = \rho\{f_1 \mapsto \ell_1, f_2 \mapsto \ell_2\} \\ & \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ell_2 \mapsto \text{unspecified}\} \\ & \quad \langle e_1, \rho', \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ & \quad \langle e_2, \rho', \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle \\ & \quad \frac{\langle e, \rho', \sigma_2 \{ \ell_1 \mapsto v_1, \ell_2 \mapsto v_2 \} \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{LETREC}(\langle f_1 : \tau_1, e_1, f_2 : \tau_2, e_2 \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{LETREC2}) \end{aligned}$$

While  $e_1$  and  $e_2$  are being evaluated, the contents of  $\ell_1$  and  $\ell_2$  are unspecified, and therefore untrustworthy. In particular, the contents of  $\ell_1$  and  $\ell_2$  are independent of the types  $\tau_1$  and  $\tau_2$ . While  $e$  is being evaluated, by contrast, the contents of  $\ell_1$  and  $\ell_2$  do respect types  $\tau_1$  and  $\tau_2$ , and so it is safe to evaluate  $f_1$  and  $f_2$ . To preserve type safety, therefore, Typed  $\mu$ Scheme's type system must prevent  $f_1$  and  $f_2$  from being evaluated until after  $\ell_1$  and  $\ell_2$  have been updated to hold values  $v_1$  and  $v_2$ . And in  $\mu$ Scheme, the way to keep something from being evaluated is to protect it under a LAMBDA. (In a lazy language like Haskell, a right-hand side is never evaluated until its value is needed, so Haskell's letrec is not restricted in this way.) Typed  $\mu$ Scheme uses the same tactic as the ML family of languages: it requires that the right-hand sides  $e_1$  and  $e_2$  be LAMBDA forms. So letrec is useful only for defining recursive functions, including mutually recursive functions.

The rule applies equally well to the empty LET.

The rule for LETREC resembles the rule for LET, except that each right-hand side  $e_i$  can refer to any of the bound names  $x_j$ , so the right-hand sides are type-checked in the extended environment  $\Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$ . Each type must have kind \*.

$$\begin{aligned} & \Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \\ & \Delta, \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\ & \Delta, \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \\ & \quad \text{Every } e_i \text{ has the form LAMBDA}(\dots) \\ & \frac{}{\Delta, \Gamma \vdash \text{LETREC}(\langle x_1 : \tau_1, e_1, \dots, x_n : \tau_n, e_n \rangle, e) : \tau} \quad (\text{LETREC}) \end{aligned}$$

The LETREC rule also requires that each  $e_i$  be a LAMBDA expression. As described in the sidebar above, this requirement prevents any  $e_i$  from evaluating an uninitialized  $x_j$ .

A rule for LETSTAR would be annoying to write down directly—it would require a lot of bookkeeping for environments. Instead, I use syntactic sugar, rewriting

$$\frac{\Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1 \rangle, \text{LETSTAR}(\langle x_2, e_2, \dots, x_n, e_n \rangle, e)) : \tau \quad n > 0}{\Delta, \Gamma \vdash \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LETSTAR})$$

$$\frac{\Delta, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \text{LETSTAR}(\langle \rangle, e) : \tau} \quad (\text{EMPTYLETSTAR})$$

A function is well typed if its body is well typed, in an environment that gives the types of the arguments. These types must be well formed and have kind  $*$ . With these types added to the environment  $\Gamma$ , we compute the type of the body, which then determines the type of the function.

$$\frac{\Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \quad \Delta, \Gamma \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Delta, \Gamma \vdash \text{LAMBDA}(\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau} \quad (\text{LAMBDA})$$

An application is well typed if the function has arrow type, and if the types and number of actual parameters match the types and number of formal parameters on the left of the arrow.

$$\frac{\Delta, \Gamma \vdash e : \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Delta, \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \tau} \quad (\text{APPLY})$$

The interesting rules, which have no counterpart in Typed Impcore, are for type abstraction and application, which introduce and eliminate polymorphism. The elimination form is simpler. To use (“eliminate”) a polymorphic value, one chooses the types with which to instantiate the type variables. The notation  $[\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n]$  indicates the *simultaneous, capture-avoiding substitution* of  $\tau_1$  for  $\alpha_1$ ,  $\tau_2$  for  $\alpha_2$ , and so on.

$$\frac{\Delta, \Gamma \vdash e : \forall \alpha_1, \dots, \alpha_n . \tau \quad \Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n}{\Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \dots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n]} \quad (\text{TYAPPLY})$$

*Simultaneous* means “substitute for all the  $\alpha_i$ ’s at once,” not one at a time. Simultaneous substitution can differ from sequential substitution if some  $\tau_i$  contains an  $\alpha_j$ . *Capture-avoiding* means that the substitution doesn’t inadvertently change the meaning of a type variable; the details are explored at length in Section 6.6.7 on page 380.

The TYAPPLY rule justifies my informal claim that the names of quantified type variables don’t matter. The *only* way to use a quantified type is to substitute for those type variables, and once you substitute for them, they are gone. The names exist only to mark the correct locations for substitution—no more, and no less.

The introduction form for a quantified type is type abstraction. To create (“introduce”) a polymorphic value, one abstracts over new type variables using TYLAMBDA. The type variables go into the kind environment  $\Delta$ . In Typed  $\mu$ Scheme, type variables always stand for types, so they have kind  $*$ . (In related, more ambitious languages like Haskell or  $F_\omega$ , a type variable may have any kind.)

$$\frac{\alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n \quad \Delta \{ \alpha_1 :: *, \dots, \alpha_n :: * \}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \dots, \alpha_n, e) : \forall \alpha_1, \dots, \alpha_n . \tau} \quad (\text{TYLAMBDA})$$

Why the side condition  $\alpha_i \notin \text{ftv}(\Gamma)$ ? The set  $\text{ftv}(\Gamma)$  contains the *free type variables* of  $\Gamma$  (page 381), and the side condition is needed to avoid changing the meaning of  $\alpha_i$  in  $e$ . This need is illustrated in Section 6.6.9 on page 386.

Just as in the operational semantics, a definition can produce a new environment. The new environment is a type environment, not a value environment: it contains the types of the names introduced by the definition. As in Typed Impcore, the new environment is produced by *typing* the definition. The relevant judgment has the form  $\langle d, \Delta, \Gamma \rangle \rightarrow \Gamma'$ , which says that when definition  $d$  is typed in kind environment  $\Delta$  and type environment  $\Gamma$ , the new type environment is  $\Gamma'$ . In Typed  $\mu$ Scheme, a definition does not introduce any new types, so typing a definition leaves  $\Delta$  unchanged.

A VAL binding uses the current environment. It is not recursive, so the name being bound is not visible during the typechecking of the right-hand side.

$$\frac{\Delta, \Gamma \vdash e : \tau}{\langle \text{VAL}(x, e), \Delta, \Gamma \rangle \rightarrow \Gamma\{x \mapsto \tau\}} \quad (\text{VAL})$$

A VAL-REC binding, by contrast, is recursive, and it requires an explicit type  $\tau$ . Type  $\tau$  must be well formed and have kind  $*$ , and it must be the type of the right-hand side.

$$\frac{\begin{array}{c} \Delta \vdash \tau :: * \\ \Delta, \Gamma\{x \mapsto \tau\} \vdash e : \tau \\ e \text{ has the form LAMBDA}(\dots) \end{array}}{\langle \text{VAL-REC}(x, \tau, e), \Delta, \Gamma \rangle \rightarrow \Gamma\{x \mapsto \tau\}} \quad (\text{VALREC})$$

The bound name  $x$  is visible during the typechecking of the right-hand side  $e$ , but for safety,  $x$  must not be *evaluated* within  $e$  until *after*  $e$ 's value has been stored in  $x$ . In  $\mu$ Scheme, the way to keep something from being evaluated is to protect it under a LAMBDA.<sup>8</sup>

A top-level expression is syntactic sugar for a binding to it.

$$\frac{\langle \text{VAL(it, } e), \Delta, \Gamma \rangle \rightarrow \Gamma'}{\langle \text{EXP}(e), \Delta, \Gamma \rangle \rightarrow \Gamma'} \quad (\text{EXP})$$

A DEFINE is syntactic sugar for a suitable VAL-REC. Indeed, the reason that Typed  $\mu$ Scheme has VAL-REC is that it is easier to typecheck VAL-REC and LAMBDA independently than to typecheck DEFINE directly.

$$\frac{\begin{array}{c} \langle \text{VAL-REC}(f, \tau_1 \times \dots \times \tau_n \rightarrow \tau, \text{LAMBDA}(\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e)), \Delta, \Gamma \rangle \rightarrow \Gamma' \\ \langle \text{DEFINE}(f, \tau, \langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e), \Delta, \Gamma \rangle \rightarrow \Gamma' \end{array}}{\langle \text{DEFINE}(f, \tau, \langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e), \Delta, \Gamma \rangle \rightarrow \Gamma'} \quad (\text{DEFINE})$$

### Type checking

I don't give you a type checker for Typed  $\mu$ Scheme; implementing the type checker is Exercise 19 on page 404. Type checking requires an expression or definition, a type environment, and a kind environment. Calling `typeof(e, Δ, Γ)` should return a  $\tau$  such that  $\Gamma \vdash e : \tau$ , or if no such  $\tau$  exists, it should raise the exception `TypeError`. Calling `typdef(d, Δ, Γ)` should return a pair  $(\Gamma', s)$ , where  $\langle d, \Delta, \Gamma \rangle \rightarrow \Gamma'$  and  $s$  is a string that represents the type of the thing defined.

LeftAsExercise  
S237a

375. `<type checking for Typed μScheme [prototype] 375>` ≡

<code>typeof : exp * kind env * tyex env -&gt; tyex</code> <code>typdef : def * kind env * tyex env -&gt; tyex env * string</code>
---

```

fun typeof _ = raise LeftAsExercise "typeof"
fun typdef _ = raise LeftAsExercise "typdef"
```

<sup>8</sup>In a lazy language like Haskell, a right-hand side is not evaluated until its value is needed, so a definition like `(val-rec [x : int] x)` is legal, but evaluating `x` produces an infinite loop (sometimes called a “black hole.”)

Type system	Concept	Interpreter
$d$	Definition	<code>def</code> (page 370)
$e$	Expression	<code>exp</code> (page 370)
$x$	Name	<code>name</code> (page 310)
$\alpha$	Type variable	<code>name</code> (page 310)
$\tau$	Type	<code>tyex</code> (page 366)
$\kappa$	Kind	<code>kind</code> (page 364)
$\Gamma$	Type environment	<code>tyex env</code> (pages 310 and 366)
$\Delta$	Kind environment	<code>kind env</code> (pages 310 and 364)
$\Delta \vdash \tau :: \kappa$	Kind checking	$\text{kindof}(\tau, \Delta) = \kappa$ , also $\text{kind } \tau = \kappa$ (pages 387 and 388) <code>asType</code> ( $\tau, \Delta$ ) (page 389)
$\tau \text{ where } \Delta \vdash \tau :: *$	Kind checking	
$\Delta, \Gamma \vdash e : \tau$	Typecheck $e$	$\text{typeof}(e, \Delta, \Gamma) = \tau$ , and often $\text{ty } e = \tau$ (left as an exercise, page 375)
$\langle d, \Delta, \Gamma \rangle \rightarrow \Gamma'$	Typecheck $d$	$\text{typedef}(d, \Delta, \Gamma) = (\Gamma', s)$ (left as an exercise, page 375)
$\text{ftv}(\tau)$	Free type variables	<code>freetyvars</code> $\tau$ (page 381)
$\text{ftv}(\Gamma)$	Free type variables	<code>freetyvarsGamma</code> $\Gamma$ (page 381)
$\tau[\alpha \mapsto \tau']$	Capture-avoiding substitution	<code>tysubst</code> ( $\tau, \{\alpha \mapsto \tau'\}$ ) (page 384)
$\forall \alpha . \tau$ becomes	Instantiation	<code>instantiate</code> ( $\forall \alpha . \tau, [\tau'], \Delta$ )
$\tau[\alpha \mapsto \tau']$		(page 385)
$\tau \equiv \tau'$	Type equivalence	<code>eqType</code> ( $\tau, \tau'$ ) (page 379)
$x \in \text{dom } \Gamma$	Definedness	<code>find</code> ( $x, \Gamma$ ) terminates without raising an exception (page 311)
$\Gamma(x)$	Type lookup	<code>find</code> ( $x, \Gamma$ ) (page 311)
$\Gamma\{x \mapsto \tau\}$	Binding	<code>bind</code> ( $x, \tau, \Gamma$ ) (page 312)
$\Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$	Binding	<code>bindList</code> ( $x_1, \dots, x_n, \tau_1, \dots, \tau_n,$ $\Gamma$ ) (page 312)
<code>int, bool, ...</code>	Base types	<code>inttype, booletype, ...</code> (page 390)
$\tau_1 \times \dots \times \tau_n \rightarrow \tau$	Function type	<code>FUNTY([<math>\tau_1, \dots, \tau_n</math>], <math>\tau</math>)</code> (page 366)

Table 6.5: Correspondence between Typed  $\mu$ Scheme’s type system and code

To implement these functions, you need function `eqType`, which tells when two types are equivalent, and function `instantiate`, which instantiates polymorphic types. Equivalence and instantiation are the topics of the next two sections. Also, to enforce the side condition on the TYLAMBDA rule on page 374, use function `freetyvarsGamma`, also defined below, to find out what type variables are free in a type environment.

### 6.6.6 Type equivalence and type-variable renaming

Many typing rules require that two types be the same. For example, the two branches of an `if` expression have to have the same type. And in Typed  $\mu$ Scheme, two types may be considered the same even if they are not identical. For example,

the types  $\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$  and  $\forall \beta. \beta \text{ list} \rightarrow \text{int}$  are considered to be the same—the *names* of bound type variables  $\alpha$  and  $\beta$  are irrelevant. This is because the only thing we can do with a quantified type is substitute for its bound type variables, and once we have substituted, the names are gone. In general, type  $\forall \alpha_1 . \tau_1$  is equivalent to  $\forall \alpha_2 . \tau_2$  if for every possible  $\tau$ ,  $\tau_1[\alpha_1 \mapsto \tau]$  is equivalent to  $\tau_2[\alpha_2 \mapsto \tau]$ . When two types are equivalent, we write  $\tau \equiv \tau'$ .

A type-equivalence relation must be justified as part of proving type soundness. Soundness says that well-typed programs don’t go wrong; in particular, if a well-typed program has a subterm  $e$  of type  $\tau$ , a type-soundness theorem allows us to change the program by substituting any other term  $e'$  of the *same* type  $\tau$ . (While the changed program might produce a different answer, it’s still guaranteed not to go wrong.) To justify a notion of type equivalence, say  $\tau \equiv \tau'$ , we prove that if we substitute a term  $e''$  of an *equivalent* type  $\tau'$ , the program still won’t go wrong.

### §6.6.6

#### Type equivalence and type-variable renaming

377

#### *The names of parameters are irrelevant*

In Typed  $\mu$ Scheme, two types are equivalent if one can be obtained from the other by renaming bound type variables. And a bound type variable originates as a type parameter in a type-lambda. Bound type variables and type-lambdas may seem new and mysterious, but just like the parameters in an ordinary lambda, the parameters in a type-lambda can be renamed without changing the meaning of the code. Let’s look at examples of each.

In  $(\lambda(x)(+x n))$ , I can rename  $x$  to  $y$  without changing the meaning of the code:

```
(lambda (x) (+ x n)) ; two equivalent muScheme functions
(lambda (y) (+ y n))
```

And in a  $(\lambda(a)(\dots))$ , I can similarly rename ‘ $a$ ’ to ‘ $b$ ’ without changing the meaning of the code:

377a. *(transcript 366b)*  $\equiv$

377c

```
-> (val id1 (type-lambda ['a] (lambda ([x : 'a]) x)))
id1 : (forall ['a] ('a -> 'a))
-> (val id2 (type-lambda ['b] (lambda ([x : 'b]) x)))
id2 : (forall ['b] ('b -> 'b))
```

The renaming gives functions  $\text{id1}$  and  $\text{id2}$  types that are syntactically different, but still equivalent: one type is obtained from the other by renaming ‘ $a$ ’ to ‘ $b$ ’. In fact, function  $\text{id1}$  has *every* type that can be obtained from  $(\forall a ('a -> 'a))$  by renaming ‘ $a$ ’. Here is some evidence:

377b. *(type-tests-id.tus 377b)*  $\equiv$

```
(check-type id1 (forall ['a] ('a -> 'a)))
(check-type id1 (forall ['b] ('b -> 'b)))
(check-type id1 (forall ['c] ('c -> 'c)))
```

377c. *(transcript 366b)*  $\equiv$

377a 382a

```
-> (use type-tests-id.tus)
```

All 3 tests passed.

When *can’t* we rename a bound type variable? An example needs at least two bound type variables. Imagine a function that takes a value of any type and returns a value of type ‘ $c$ '; the function has type  $(\forall a ('a -> 'c))$ . I can rename ‘ $a$ ’ to ‘ $b$ ’ without changing the type:

```
(forall ['a] ('a -> 'c)) ; equivalent types
(forall ['b] ('b -> 'c))
```

I can't rename 'a to 'c; that changes the type:

```
(forall ['c] ('c -> 'c)) ; not equivalent to the first two
```

As illustrated above, type `(forall ['c] ('c -> 'c))` is the type of the identity function, and it's not the same as `(forall ['a] ('a -> 'c))`. Functions of these types have to behave differently: a function of type `(forall ['a] ('a -> 'c))` ignores its argument, and a function of type `(forall ['c] ('c -> 'c))` returns its argument.

That last renaming is invalid because it *captures* type variable '`c`: '`c` is free in the original type but bound in the new type, so its meaning has been changed. Whenever we rename a bound type variable, whether it is bound by `forall` or type-lambda, we must not capture any free type variables. (The same restriction applies to the formal parameters of a lambda expression; for example, it is OK to rename `x` to `y` in `(lambda (x) (+ n))`, but it is not OK to rename `x` to `n`; `(lambda (n) (+ n n))` is not the same function!) Also, when we substitute a type  $\tau$  for a free type variable, we must not capture any free type variables of  $\tau$ .

### Soundness of type equivalence in Typed $\mu$ Scheme

Why is it sound to consider types equivalent if one can be obtained from the other by renaming bound type variables? Because if two types differ only in the names of their bound type variables, there is no combination of instantiations and substitutions that can distinguish them. To show what it means to distinguish types by instantiation and substitution, let's compare the three types above. First I instantiate each type at  $\tau_1$ , then I substitute  $\tau_2$  for free occurrences of '`c`:

Original type	After instantiation	After substitution
<code>(forall ['a] ('a -&gt; 'c))</code>	$(\tau_1 \rightarrow 'c)$	$(\tau_1 \rightarrow \tau_2)$
<code>(forall ['b] ('b -&gt; 'c))</code>	$(\tau_1 \rightarrow 'c)$	$(\tau_1 \rightarrow \tau_2)$
<code>(forall ['c] ('c -&gt; 'c))</code>	$(\tau_1 \rightarrow \tau_1)$	$(\tau_1 \rightarrow \tau_1)$

No matter how  $\tau_1$  and  $\tau_2$  are chosen, the first two `forall` types produce identical results. But when  $\tau_1$  and  $\tau_2$  are chosen intelligently—`int` and `bool` will do—the first two `forall` types become `(int -> bool)`, but the third one becomes `(int -> int)`, which is different.

### Rules and code for type equivalence

Typed  $\mu$ Scheme's type equivalence  $\tau \equiv \tau'$  is defined by a proof system. A type variable or type constructor is equivalent to itself, and type equivalence is structural through function types, constructor applications and quantifications.

$\text{EQUALVARIABLES}$ <hr/> $\alpha \equiv \alpha$	$\text{EQUALCONSTRUCTORS}$ <hr/> $\mu \equiv \mu$
$\text{EQUIVFUNS}$ $\frac{\tau_i \equiv \tau'_i, 1 \leq i \leq n \quad \tau \equiv \tau'}{\tau_1 \times \cdots \times \tau_n \rightarrow \tau \equiv \tau'_1 \times \cdots \times \tau'_n \rightarrow \tau'}$	
$\text{EQUIVAPPLICATIONS}$ $\frac{\tau_i \equiv \tau'_i, 1 \leq i \leq n \quad \tau \equiv \tau'}{(\tau_1, \dots, \tau_n) \tau \equiv (\tau'_1, \dots, \tau'_n) \tau'}$	$\text{EQUIVQUANTIFIEDS}$ $\frac{\tau \equiv \tau'}{\forall \alpha_1, \dots, \alpha_n . \tau \equiv \forall \alpha_1, \dots, \alpha_n . \tau'}$

These five rules make syntactically identical types equivalent. We also need to be able to rename bound type variables. As long as I don't capture a free variable of  $\tau$ , I can rename any  $\alpha_i$  to a new variable  $\beta$ :

$$\frac{\beta \notin \text{ftv}(\tau) \quad \beta \notin \{\alpha_1, \dots, \alpha_n\}}{\forall \alpha_1, \dots, \alpha_n . \tau \equiv \forall \alpha, \dots, \alpha_{i-1}, \beta, \alpha_{i+1}, \alpha_1, \dots, \alpha_n . \tau[\alpha_i \mapsto \beta]} \quad (\text{EQUIVRENAMED})$$

The second side condition ensures that even after the renaming, the bound type variables are all distinct.

To permit variables to be renamed on the left side of the  $\equiv$  sign, I make type equivalence symmetric:

$$\frac{\tau \equiv \tau'}{\tau' \equiv \tau} \quad (\text{SYMMETRY})$$

Type equivalence is also reflexive and transitive (Exercise 24 on page 406).

Typed  $\mu$ Scheme's type-equivalence relation is implemented by function `eqType`. Most of the code works by straightforward structural induction over the type: for `TYVAR`, `TYCON`, `CONAPP`, and `FUNTY` there is only one rule that applies and one way to check equivalence.

**379a.** *(type equivalence for Typed  $\mu$ Scheme 379a)*  $\equiv$  (S394b)

<code>eqType : tyex * tyex -&gt; bool</code>
<code>eqTypes : tyex list * tyex list -&gt; bool</code>

*(infinite supply of type variables S393c)*

```

fun eqType (TYVAR a, TYVAR a') = a = a'
| eqType (TYCON c, TYCON c') = c = c'
| eqType (CONAPP (tau, taus), CONAPP (tau', taus')) =
    eqType (tau, tau') andalso eqTypes (taus, taus')
| eqType (FUNTY (taus, tau), FUNTY (taus', tau')) =
    eqType (tau, tau') andalso eqTypes (taus, taus')
| eqType (FORALL (alphas, tau), FORALL (alphas', tau')) =
    <Boolean saying if FORALL (alphas, tau) ≡ FORALL (alphas', tau') 379b>
| eqType _ = false
and eqTypes (taus, taus') = ListPair.allEq eqType (taus, taus')
```

The only case that needs more than structural induction is the case of two quantified types  $\forall \alpha_1, \dots, \alpha_n . \tau$  and  $\forall \alpha'_1, \dots, \alpha'_n . \tau'$ . I compare them by renaming the bound type variables on *both* sides to  $\beta_1, \dots, \beta_n$ . Internal function `ok` ensures that no  $\beta_i$  is free in either  $\tau$  or  $\tau'$ .<sup>9</sup> After renaming, I compare these two types:

$$\begin{aligned} & \forall \beta_1, \dots, \beta_n . \tau[\alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n] \text{ and} \\ & \forall \beta_1, \dots, \beta_n . \tau'[\alpha'_1 \mapsto \beta_1, \dots, \alpha'_n \mapsto \beta_n]. \end{aligned}$$

According to rule `EQUIVQUANTIFIEDS`, the comparison succeeds if the first body type,  $\tau[\alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n]$ , is equivalent to  $\tau'[\alpha'_1 \mapsto \beta_1, \dots, \alpha'_n \mapsto \beta_n]$ . Function `rename` is implemented using substitution; both are defined in the next section.

**379b.** *(Boolean saying if FORALL (alphas, tau) ≡ FORALL (alphas', tau') 379b)*  $\equiv$  (379a)

```

let fun ok a =
    not (member a (freetyvars tau) orelse member a (freetyvars tau'))
    val betas = streamTake (length alphas, streamFilter ok infiniteTyvars)
    in length alphas = length alphas' andalso
       eqType (rename (alphas, betas, tau), rename (alphas', betas, tau'))
    end
```

<sup>9</sup>Because I rename *all* the  $\alpha_i$ 's and  $\alpha'_i$ 's, I don't have to worry about a  $\beta$  colliding with an existing  $\alpha_i$  or  $\alpha'_i$ .

## §6.6.6

### Type equivalence and type-variable renaming

379

CONAPP	366a
FORALL	366a
freetyvars	381a
FUNTY	366a
infiniteTyvars	S393c
member	S240b
rename	385a
streamFilter	S253a
streamTake	S254a
TYCON	366a
TYVAR	366a

Type variables  $\beta_1, \dots, \beta_n$  (betas) are drawn from an infinite stream. Streams and stream operators, as well as the particular stream `infiniteTyvars`, are defined in the Supplement. Because the stream contains infinitely many type variables, of which only finitely many can be free in  $\tau$  or  $\tau'$ , I am guaranteed to find  $n$  good ones.

Function `eqType` can be used in the implementation of any typing rule that requires two types to be the same. To justify the use of equivalence instead of identity, I extend the type system with the following rule, which says that if  $e$  has a type, it also has any equivalent type.

$$\frac{\Delta, \Gamma \vdash e : \tau \quad \tau \equiv \tau'}{\Delta, \Gamma \vdash e : \tau'} \quad (\text{EQUIV})$$

### 6.6.7 Instantiation and renaming by capture-avoiding substitution

Capture-avoiding substitution is tricky—even eminent professors sometimes get it wrong. We’ll start with examples, but first we need to get precise about free and bound type variables.

#### Free and bound type variables

Type variables may occur *free* or *bound*, and we substitute only for *free* occurrences. A free type variable acts like a global variable; a bound type variable acts like a formal parameter. And a *binding occurrence* is an appearance next to a `forall`. Here is an example type that has all three kinds of occurrences:

Example type A	$('c \rightarrow (\text{forall } ['a] ('a \rightarrow 'c)))$
Free occurrence of ' $c$ in A	$('c \rightarrow (\text{forall } ['a] ('a \rightarrow 'c)))$
Binding occurrence of ' $a$ in A	$('c \rightarrow (\text{forall } ('a) ('a \rightarrow 'c)))$
Bound occurrence of ' $a$ in A	$('c \rightarrow (\text{forall } ('a) ('a \rightarrow 'c)))$
Free occurrence of ' $c$ in A	$('c \rightarrow (\text{forall } ('a) ('a \rightarrow 'c)))$

As the wording suggests, “free” and “bound” are not absolute properties; they are relative to a particular type. For “ $\alpha$  is free in  $\tau$ ” I write “ $\alpha \in \text{ftv}(\tau)$ ,” where  $\text{ftv}(\tau)$  represents the free type variables of  $\tau$ . Here’s a proof system, which resembles the proof system for free *term* variables in Section 5.6 on page 323:<sup>10</sup>

$$\begin{array}{c} \frac{}{\alpha \in \text{ftv}(\alpha)} & \frac{\alpha \in \text{ftv}(\tau_i)}{\alpha \in \text{ftv}((\tau_1, \dots, \tau_n) \tau)} & \frac{\alpha \in \text{ftv}(\tau)}{\alpha \in \text{ftv}((\tau_1, \dots, \tau_n) \tau)} \\[10pt] \frac{\alpha \in \text{ftv}(\tau_i)}{\alpha \in \text{ftv}(\tau_1 \times \dots \times \tau_n \rightarrow \tau)} & & \frac{\alpha \in \text{ftv}(\tau)}{\alpha \in \text{ftv}(\tau_1 \times \dots \times \tau_n \rightarrow \tau)} \\[10pt] \frac{\alpha_i \in \text{ftv}(\tau) \quad 1 \leq i \leq n}{\alpha_i \text{ is bound in } \forall \alpha_1, \dots, \alpha_n . \tau} & & \frac{\alpha \in \text{ftv}(\tau) \quad \alpha \neq \alpha_i, \quad 1 \leq i \leq n}{\alpha \in \text{ftv}(\forall \alpha_1, \dots, \alpha_n . \tau)} \end{array}$$

To illustrate the relative nature of “free” and “bound,” type variable ' $a$  occurs bound in type `(forall ['a] ('a -> 'c))`, but it occurs free in type `('a -> 'c)`.

<sup>10</sup>“Binding occurrence” doesn’t need a proof system; binding occurrences are those introduced by  $\forall$ .

The free type variables of a type are computed by function `freetyvars`. Bound type variables are removed using `diff`. Using `foldl`, `union`, `diff`, and `reverse` puts type variables in the set in the order of their first appearance.

**381a.** *(sets of free type variables in Typed μScheme 381a) ≡*

```
fun freetyvars t =
  let fun free (TYVAR v,          ftvs) = insert (v, ftvs)
       | free (TYCON _,           ftvs) = ftvs
       | free (CONAPP (ty, tys), ftvs) = foldl free (free (ty, ftvs)) tys
       | free (FUNTY (tys, ty), ftvs) = foldl free (free (ty, ftvs)) tys
       | free (FORALL (alphas, tau), ftvs) =
          union (diff (free (tau, emptyset), alphas), ftvs)
  in reverse (free (t, emptyset))
  end
```

(S394b) 381b ▷

§6.6.7

Instantiation and  
renaming by  
capture-avoiding  
substitution

381

The free type variables of a type environment, which are needed to enforce the side condition in rule TYLAMBDA page 374, are computed by calling function `freetyvarsGamma`.

**381b.** *(sets of free type variables in Typed μScheme 381a) +≡*

(S394b) ▲381a

```
fun freetyvarsGamma Gamma =
  foldl (fn ((x, tau), ftvs) => union (ftvs, freetyvars tau)) emptyset Gamma
```

*Substitution is for free variables only*

Free occurrences govern the behaviors of both renaming and substitution. When we rename bound type variables in a `forall`, we rename only those occurrences that are free in the body of the `forall`. And when we substitute for a type variable, we substitute only for free occurrences of that variable. These rules may seem arbitrary, but they are motivated by a principle of observational equivalence: If types  $\tau_1$  and  $\tau_2$  are equivalent, and if we substitute  $\tau$  for  $\alpha$  in both types, then the resulting types  $\tau_1[\alpha \mapsto \tau]$  and  $\tau_2[\alpha \mapsto \tau]$  should still be equivalent. For example, here are two equivalent types:

```
('c -> (forall ['a] ('a -> 'c))) ; example A
('c -> (forall ['b] ('b -> 'c))) ; example B
```

Suppose I wish to substitute '`c`' for '`b`'. In example A, there are no occurrences of '`b`', and nothing happens. What about example B? There are no *free* occurrences of '`b`', just one binding occurrence and one bound occurrence. So also, nothing happens! And when nothing happens to two equivalent types, the results are still equivalent.

If I were to substitute for the binding occurrence of '`b`' or the bound occurrence of '`b`', or for both, the resulting type would no longer be equivalent to example type A.

```
('c -> (forall ['c] ('b -> 'c))) ;; WRONG B1
('c -> (forall ['b] ('c -> 'c))) ;; WRONG B2
('c -> (forall ['c] ('c -> 'c))) ;; WRONG B3
```

CONAPP	366a
diff	S240b
emptyset	S240b
FORALL	366a
FUNTY	366a
insert	S240b
reverse	S241c
TYCON	366a
TYVAR	366a
union	S240b

Each of the faulty “substitutions” goes wrong in a different way:

- B1. I substitute for the binding occurrence of '`b`' but not for the bound occurrence in the body. If I now rename the bound type variable '`c`' to '`a`', I get the type  $('c -> (\forall [a] (b -> a)))$ . The outer part now matches example A, but the inner function type  $(b -> a)$  is not equivalent to example A's  $(a -> c)$ .

B2. I substitute for the bound occurrence of ' $b$ ' in the body but not for the binding occurrence in the `forall`. If I now rename the bound type variable ' $b$ ' to ' $a$ ', I get  $('c \rightarrow (\forall [a] ('c \rightarrow 'a)))$ , and again, inner function type  $('c \rightarrow 'c)$  is not equivalent to example A's  $('a \rightarrow 'c)$ .

B3. I substitute for both binding and bound occurrences of ' $b$ '. If I now rename the bound type variable ' $c$ ' to ' $a$ ', I get  $('c \rightarrow (\forall [a] ('a \rightarrow 'a)))$ , and again, inner function type  $('a \rightarrow 'a)$  is not equivalent to example A's  $('a \rightarrow 'c)$ .

When you're substituting for ' $b$ ', and you hit a `forall` that binds ' $b$ ', you must leave it alone. If you're uncomfortable, imagine "leave it alone" as a three-step procedure:

1. First rename the bound ' $b$ ' to ' $z$ ', which preserves equivalence.
2. Now substitute ' $c$ ' for ' $b$ '. But there are no ' $b$ 's—it's just like example A!
3. Finally rename the bound ' $z$ ' back to ' $b$ ', which again preserves equivalence.

Here's one more example of not substituting for a bound type variable. I define a polymorphic value `strange` of type  $\forall \alpha . \forall \alpha . \alpha \rightarrow \alpha$ :

**382a.** *(transcript 366b)*  $\equiv$   $\triangleleft 377c \ 382b \triangleright$   
 $\rightarrow (\text{val } \text{strange} \atop \text{---} \atop (\text{type-lambda} ['a] \atop (\text{type-lambda} ['a] \atop (\text{lambda} ([x : 'a]) x)))) \atop \text{strange} : (\forall [a] (\forall [a] ('a \rightarrow 'a)))$

To instantiate `strange` at `int`, I strip the outer  $\forall \alpha.$ , and I substitute `int` for free occurrences of  $\alpha$  in  $\forall \alpha . \alpha \rightarrow \alpha$ . But there are no free occurrences! Type variable  $\alpha$  is bound in  $\forall \alpha . \alpha \rightarrow \alpha$ , and substituting `int` yields  $\forall \alpha . \alpha \rightarrow \alpha$ :

**382b.** *(transcript 366b)*  $\equiv$   $\triangleleft 382a$   
 $\rightarrow [@ \text{strange int}] \atop <\text{function}> : (\forall [a] ('a \rightarrow 'a))$

It's strange but true.

### Substitution avoids capturing variables

Limiting substitution to free occurrences is not enough. We also have to make sure that substitution does not accidentally change a free occurrence to a bound occurrence. That is, supposing we substitute  $\tau$  for ' $c$ ', every type variable that is free in  $\tau$  must also be free in the result. As an example, let us substitute `(list 'b)` for ' $c$ ' in example types A and B:

```
('c \rightarrow (\forall [a] ('a \rightarrow 'c))) ; example A
('c \rightarrow (\forall [b] ('b \rightarrow 'c))) ; example B
((list 'b) \rightarrow (\forall [a] ('a \rightarrow (list 'b)))) ; A substituted
((list 'b) \rightarrow (\forall [b] ('b \rightarrow (list 'b)))) ; B substituted WRONG
```

In both examples, we substitute for two free occurrences of ' $c$ '. The type we substitute has one free occurrence of ' $b$ '. In example A, the result has, as expected, two free occurrences of ' $b$ '. But in example B, the second free occurrence of ' $b$ ' has become a *bound* occurrence. We say variable ' $b$ ' is *captured*.

Capture is a problem in any computation that involves substitutions—think *macros*—and we already know the solution: the problem with substitution here is

the same problem as the faulty let sugar for `||` in Section 2.13.3 on page 167. That section presents this faulty template for implementing  $(|| e_1 e_2)$ :

$(|| e_1 e_2)$  is almost `(let ([x e_1]) (if x x e_2))`,

The template fails if `x` appears as a free variable in  $e_2$ —as we substitute  $e_2$  into the template for `||`, variable `x` is *captured* and its meaning is changed. To avoid capture, we rename the template’s bound variable `x` to something that is not free in  $e_2$ .

In a polymorphic type system, we avoid capture in the same way: by renaming a bound type variable. And in the type system, renaming is easy to justify: when we rename, instead of substituting into example type `B`, we are substituting into an equivalent type. In the example above, the only type variable bound in `B` is `'b`, and we rename it to `'z`:

```
('c -> (forall ('z) ('z -> 'c))) ; equivalent to B
((list 'b) -> (forall ('z) ('z -> (list 'b)))) ; and now substituted
```

Now the substitution is correct, and the result is equivalent to `A` substituted.

### Specifying and implementing substitution

Let’s generalize from the example to a specification. I specify a simpler case than I implement: instead of simultaneously substituting  $n$  types for  $n$  type variables, I substitute one type  $\tau$  for one type variable  $\alpha$ . I substitute only for free occurrences, and I never allow a variable to be captured. The judgment form is  $\boxed{\tau'[\alpha \mapsto \tau] \equiv \tau''}$ , pronounced “ $\tau'$  with  $\alpha$  going to  $\tau$  is equivalent to  $\tau''$ .”

Substitution for a type variable changes that type variable and no other. And substitution preserves the structure of constructors, constructor applications, and function types.

$$\frac{}{\alpha[\alpha \mapsto \tau] \equiv \tau} \quad \frac{\alpha \neq \alpha'}{\alpha'[\alpha \mapsto \tau] \equiv \alpha'} \quad \frac{}{\mu[\alpha \mapsto \tau] \equiv \mu}$$

$$\frac{\tau'[\alpha \mapsto \tau] \equiv \tau''}{((\tau'_1, \dots, \tau'_n) \tau')[\alpha \mapsto \tau] \equiv (\tau'_1[\alpha \mapsto \tau], \dots, \tau'_n[\alpha \mapsto \tau]) \tau''}$$

$$(\tau'_1 \times \dots \times \tau'_n \rightarrow \tau')[\alpha \mapsto \tau] \equiv \tau'_1[\alpha \mapsto \tau] \times \dots \times \tau'_n[\alpha \mapsto \tau] \rightarrow \tau'[\alpha \mapsto \tau]$$

Substitution into a quantified type is for free variables only, and it may not capture a free type variable of  $\tau$ :

$$\frac{\alpha \notin \{\alpha_1, \dots, \alpha_n\} \quad \text{ftv}(\tau) \cap \{\alpha_1, \dots, \alpha_n\} = \emptyset}{(\forall \alpha_1, \dots, \alpha_n . \tau')[\alpha \mapsto \tau] \equiv \forall \alpha_1, \dots, \alpha_n . (\tau'[\alpha \mapsto \tau])}$$

If you want to substitute, but substitution would lead to variable capture, avoid capture by substituting into an equivalent type:

$$\frac{\tau' \equiv \tau''}{\tau'[\alpha \mapsto \tau] \equiv \tau''[\alpha \mapsto \tau]}$$

In Typed  $\mu$ Scheme, substituting for a single type variable isn’t enough; instantiation substitutes for multiple type variables simultaneously. A substitution is represented by a value of type `tyex env`, which is passed to function `tysubst` as parameter `varenv`. This environment maps each type variable to the type that should

be substituted for it. If the type variable is not mapped, substitution leaves it unchanged.

384a. *(capture-avoiding substitution for Typed  $\mu$ Scheme 384a)*  $\equiv$

(S394b) 385a▷

```

fun tysubst (tau, varenv) =
  let <definition of renameForallAvoiding for Typed  $\mu$ Scheme (left as an exercise)>
    fun subst (TYVAR a) = (find (a, varenv) handle NotFound _ => TYVAR a)
    | subst (TYCON c) = (TYCON c)
    | subst (CONAPP (tau, taus)) = CONAPP (subst tau, map subst taus)
    | subst (FUNTY (taus, tau)) = FUNTY (map subst taus, subst tau)
    | subst (FORALL (alphas, tau)) =
        <use varenv to substitute in tau; don't capture or substitute for any alphas 384c>
      in subst tau
    end
  
```

Substitution into a quantified type requires that we not substitute for a bound variable and that we avoid capturing any variables. Postponing for the moment the issue of capture, we can prevent substitution for a bound type variable by extending varenv so that each bound type variable is mapped to itself:

384b. *(substitute varenv in FORALL (alphas, tau) (works only if there is no capture) 384b)*  $\equiv$

```

let val varenv' = bindList (alphas, map TYVAR alphas, varenv)
in FORALL (alphas, tysubst (tau, varenv'))
end
  
```

To avoid capture, we must identify and rename bindings that might capture a variable. The scenario has three parts:

- A type  $\tau_{new}$  is substituted for a variable that appears free in  $\forall \alpha_1, \dots, \alpha_n . \tau$ .
- Among the free variables of type  $\tau_{new}$  is one of the very type variables  $\alpha_i$  that appears under the  $\forall$ .
- To avoid capturing  $\alpha_i$ , the bound  $\alpha_i$  has to be renamed.

Below,  $\alpha_i$ 's that have to be renamed are put in a set called `actual_captures`. If the set is empty, the code above works, and I use it. Otherwise, the variables in `actual_captures` are renamed by function `renameForallAvoiding`.

384c. *(use varenv to substitute in tau; don't capture or substitute for any alphas 384c)*  $\equiv$  (384a)

```

let val free          = freetyvars (FORALL (alphas, tau))
  val new_taus       = map (subst o TYVAR) free
  val potential_captures = foldl union emptyset (map freetyvars new_taus)
  val actual_captures = inter (potential_captures, alphas)
in if null actual_captures then
    <substitute varenv in FORALL (alphas, tau) (works only if there is no capture) 384b>
  else
    subst (renameForallAvoiding (alphas, tau, potential_captures))
end
  
```

When capture may occur, function `renameForallAvoiding` renames the alphas to avoid potentially captured variables. Its specification requires it to return a type that is equivalent to `FORALL (alphas, tau)` but that does not result in variable capture. In detail, `renameForallAvoiding([ $\alpha_1, \dots, \alpha_n$ ],  $\tau$ , C)` returns a type  $\forall \beta_1, \dots, \beta_n . \tau'$  with these properties:

$$\begin{aligned} \forall \beta_1, \dots, \beta_n . \tau' &\equiv \forall \alpha_1, \dots, \alpha_n . \tau \\ \{\beta_1, \dots, \beta_n\} \cap C &= \emptyset \end{aligned}$$

The implementation of `renameForallAvoiding` is Exercise 28 on page 407.

Renaming is a special case of substitution. We substitute one set of variables for another.

**385a.** *(capture-avoiding substitution for Typed μScheme 384a)*  $\vdash \equiv$  (S394b)  $\triangleleft 384a \ 385b \triangleright$

```
rename : name list * name list * tyex -> tyex
fun rename (alphas, betas, tau) =
  tysubst (tau, bindList (alphas, map TYVAR betas, emptyEnv))
```

Instantiation is also implemented by substitution. Most of the code is error checking. We instantiate only quantified types, we instantiate only at actual types of kind TYPE, and we instantiate only with the right number of types.

**385b.** *(capture-avoiding substitution for Typed μScheme 384a)*  $\vdash \equiv$  (S394b)  $\triangleleft 385a$

```
instantiate : tyex * tyex list * kind env -> tyex
List.find : ('a -> bool) -> 'a list -> 'a option
```

```
fun instantiate (FORALL (formals, tau), actuals, Delta) =
  (case List.find (fn t => not (eqKind (kindof (t, Delta), TYPE)))
    of SOME t => raise TypeError
        ("instantiated at type constructor '' ^ "
         "typeString t ^ '', which is not a type")
     | NONE =>
       (tysubst (tau, bindList (formals, actuals, emptyEnv))
        handle BindListLength =>
         raise TypeError
         "instantiated polymorphic term at wrong number of types"))
| instantiate (tau, _, _) =
  raise TypeError ("tried to instantiate term " ^
                  "of non-quantified type " ^ typeString tau)
```

The Standard ML function `List.find` takes a predicate and searches a list for an element satisfying that predicate.

### 6.6.8 Subverting the type system through variable capture

If you're wondering why we go to all the trouble of avoiding variable capture, this section demonstrates: if capture is permitted, anyone can subvert the type system so that a value of any type can be cast to a value of any other type. The demonstration is short; it uses just two type-lambdas and two type variables.

The demonstration uses a Curried function that first takes an argument, then takes a function to apply to the argument, then returns the application. Without types, code written in untyped μScheme might look like this:

**385c.** *(μScheme transcript 385c)*  $\equiv$

```
-> (val flip-apply (lambda (x) (lambda (f) (f x))))
-> ((flip-apply '(a b c)) reverse)
(c b a)
-> (val apply-to-symbols (flip-apply '(a b c)))
-> (apply-to-symbols reverse)
(c b a)
-> (apply-to-symbols cdr)
(b c)
```

To add types, give `x` type  $\beta$  and `f` type  $\beta \rightarrow \alpha$ , so `flip-apply` has type

$$\text{flip-apply} : \forall \beta . \beta \rightarrow (\forall \alpha . (\beta \rightarrow \alpha) \rightarrow \alpha).$$

§6.6.8  
Subverting the type  
system through  
variable capture

385

bindList	312c
BindListLength	312c
cdr	$\mathcal{P}$ 164a
CONAPP	366a
emptyEnv	311a
emptyset	S240b
eqKind,	in Typed μScheme 364b
find	311b
FORALL	366a
freetyvars	381a
FUNTY	366a
inter	S240b
kindof	387b
NotFound	311b
renameForall-Avoiding	407
TYCON	366a
TYPE,	in Typed μScheme 364a
in Typed μScheme	364a
TypeError	S237b
typeString	S394c
TYVAR	366a
union	S240b

This perfectly reasonable type means if we supply a value of type  $\beta$  and a function of type  $\beta \rightarrow \alpha$ , we can get a value of type  $\alpha$ . Defining `flip-apply` with this type is part of Exercise 33 on page 409.

**386a.** *(variable-capture transcript 386a)* 386b ▷

```
-> (val flip-apply <typed version of flip-apply (left as an exercise)>)
  flip-apply : (forall ['b] ('b -> (forall ['a] (('b -> 'a) -> 'a))))
```

Given `flip-apply`, the idea used to subvert the type system is to substitute '`a`' for '`b`' and then '`b`' for '`a`'. If the first substitution is done incorrectly, '`a`' is captured, and we can produce a polymorphic function with a senseless type:

**386b.** *(variable-capture transcript 386a)*  $+ \equiv$  386a 386c ▷

```
-> (type-lambda ['a] [@ flip-apply 'a]) ; variable 'a is captured!
  <function> : (forall ['a] ('a -> (forall ['a] (('a -> 'a) -> 'a))))
```

The key part here is the result type  $\forall \alpha . (\alpha \rightarrow \alpha) \rightarrow \alpha$ . This type says that for any type  $\alpha$ , given an identity function on type  $\alpha$ , we can manufacture a value of type  $\alpha$ . That's nonsense.

Having captured '`a`', I instantiate the problematic result type at '`b`:

**386c.** *(variable-capture transcript 386a)*  $+ \equiv$  386b 386d ▷

```
-> (val pre-cast
  (type-lambda ['a 'b]
    (lambda ([x : 'a])
      [@ ([@ flip-apply 'a] x) 'b]))
  pre-cast : (forall ['a 'b] ('a -> (('b -> 'b) -> 'b)))
```

To finish the demonstration, modify `pre-cast` by supplying an identity function in the right place (Exercise 33 again). The result is function `cast` of type  $\forall \alpha, \beta. \alpha \rightarrow \beta$ :

**386d.** *(variable-capture transcript 386a)*  $+ \equiv$  386c 386e ▷

```
-> (val cast <definition of cast (left as an exercise)>)
  cast : (forall ['a 'b] ('a -> 'b))
```

Function `cast` can be used to change a value of any type to any other type. For example, we can make a function out of the number 42—but when we apply the function, the evaluator reports a bug in the type checker.

**386e.** *(variable-capture transcript 386a)*  $+ \equiv$  386d

```
-> ([@ cast int (int -> int)] 42)
  42 : (int -> int)
-> ((([@ cast int (int -> int)] 42) 0)
  bug in type checking: applied non-function
```

### 6.6.9 Preventing capture with type-lambda

To make the type system solid, it's not enough to substitute correctly into quantified types; we must also take care when *creating* quantified types. The introduction form for a quantified type is `type-lambda`, and unlike its cousin the ordinary `lambda`, `type-lambda` restricts the name of its formal (type) parameters:

$$\frac{\alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n \quad \Delta\{\alpha_1 :: *, \dots, \alpha_n :: *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \dots, \alpha_n, e) : \forall \alpha_1, \dots, \alpha_n. \tau} \quad (\text{TYLAMBDA})$$

The restriction  $\alpha_i \notin \text{ftv}(\Gamma)$  prevents a form of variable capture.

The essential property of a polymorphic term `TYLAMBDA`( $\alpha, e$ ) is that we can instantiate the term by substituting any type  $\tau$  for  $\alpha$ . But if  $\alpha$  already stands for something else, there's trouble. As a first example, I can begin a term by creating an environment in which `x` has type '`a`:

```
(type-lambda ['a] (lambda ([x : 'a]) ...))
```

Within the `...`, the typing environment binds `x` to '`a`', so '`a`' is a free type variable of  $\Gamma$ . Suppose I break the rules now by allowing an *inner* type-lambda to bind '`a`' a second time:

```
(type-lambda ['a] (lambda ([x : 'a])
    (type-lambda ['a] (lambda ([y : 'a]) ...))))
```

In the position of the `...` in the new example, it looks like `x` and `y` both have the same type, but they can be given different types. Now I have a little hole in my type system. For example, I can compare two values of any types for equality:

**387a.** *(transcript with no restriction on type-lambda 387a)≡*

```
-> (val bad= (type-lambda ['a] (lambda ([x : 'a])
    (type-lambda ['a] (lambda ([y : 'a])
        ([@ = 'a] x y))))))
bad= : (forall ['a] ('a -> (forall ['a] ('a -> bool))))
-> (val worse= (type-lambda ['a 'b]
    (lambda ([x : 'a] [y : 'b])
        ([@ ([@ bad= 'a] x) 'b] y))))
worse= : (forall ['a 'b] ('a 'b -> bool))
```

This hole is actually not so little. In fact, by using a similar trick, I can make a value of any type masquerade as a value of any other type. The details can be found in Exercises 31 and 32 on pages 408 and 409.

### 6.6.10 Other building blocks of a type checker

The main programming exercise for Section 6.6 is to write a type checker for Typed  $\mu$ Scheme. The functions for equivalence, substitution, and instantiation, which are presented above, play key roles. This section presents a few other useful functions, including those used to check kinds and those used to assign types to primitive functions.

#### Checking a kind

When we see a type-level expression, we need to know if it is well formed. In Typed  $\mu$ Scheme, a type-level expression is well formed if it has a *kind*, as described in Section 6.6.2 on page 364. Function `kindof` implements the kinding judgment  $\Delta \vdash \tau :: \kappa$ , which says that given kind environment  $\Delta$ , type-level expression  $\tau$  has kind  $\kappa$ . Given  $\Delta$  and  $\tau$ , `kindof`( $\tau$ ,  $\Delta$ ) returns a  $\kappa$  such that  $\Delta \vdash \tau :: \kappa$ , or if no such kind exists, it raises the exception `TypeError`.

**387b.** *(kind checking for Typed  $\mu$ Scheme 387b)≡*

```
fun kindof (tau, Delta) =
  let <definition of internal function kind 388a>
    in kind tau
    end
```

(S394b) 389b▷

<code>kindof : tyex * kind env -&gt; kind</code>
<code>kind   : tyex           -&gt; kind</code>

kind 388a

The internal function `kind` computes the kind of `tau`; the environment `Delta` is assumed. The implementation of `kind` is derived directly from the kinding rules. The derivation follows the same process as the derivation of a type checker from typing rules or the derivation of an interpreter from operational semantics. As usual, I repeat the rules to show the connection with the code.

A type variable is looked up in the environment.

$$\frac{\alpha \in \text{dom } \Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)} \quad (\text{KINDINTROVAR})$$

The parser in Section Q.3 guarantees that the name of a type variable begins with a quote mark, so it is distinct from any type constructor.

**388a.** *(definition of internal function kind 388a)*  $\equiv$  (387b) 388b▷  
 fun kind (TYVAR a) =  
   (find (a, Delta)  
     handle NotFound \_ => raise TypeError ("unknown type variable " ^ a))

A type constructor is also looked up.

$$\frac{\mu \in \text{dom } \Delta}{\Delta \vdash \text{TYCON}(\mu) :: \Delta(\mu)} \quad (\text{KINDINTROCON})$$

**388b.** *(definition of internal function kind 388a)*  $\equiv$  (387b) ▷388a 388c▷  
 | kind (TYCON c) =  
   (find (c, Delta)  
     handle NotFound \_ => raise TypeError ("unknown type constructor " ^ c))

To form a function type, FUNTY combines any number of argument types with a result type.

$$\frac{\Delta \vdash \tau_i :: *, 1 \leq i \leq n \quad \Delta \vdash \tau :: *}{\Delta \vdash \tau_1 \times \cdots \times \tau_n \rightarrow \tau :: *} \quad (\text{KINDFUNCTION})$$

**388c.** *(definition of internal function kind 388a)*  $\equiv$  (387b) ▷388b 388d▷  
 | kind (FUNTY (args, result)) =  
   let fun badKind tau = not (eqKind (kind tau, TYPE))  
   in if badKind result then  
       raise TypeError "function result is not a type"  
     else if List.exists badKind args then  
       raise TypeError "argument list includes a non-type"  
     else  
       TYPE  
   end

The arguments are inspected using Standard ML function `List.exists`, which corresponds to the  $\mu$ Scheme function `exists?`.

A type constructor may be applied only in ways that are consistent with its kind.

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \quad \Delta \vdash \tau_i :: \kappa_i, 1 \leq i \leq n}{\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \dots, \tau_n]) :: \kappa} \quad (\text{KINDAPP})$$

**388d.** *(definition of internal function kind 388a)*  $\equiv$  (387b) ▷388c 389a▷  
 | kind (CONAPP (tau, actuals)) =  
   (case kind tau  
     of ARROW (formal\_kinds, result\_kind) =>  
       if eqKinds (formal\_kinds, map kind actuals) then  
         result\_kind  
       else  
         raise TypeError ("type constructor " ^ typeString tau ^  
                           " applied to the wrong arguments")  
     | TYPE =>  
       raise TypeError ("tried to apply type " ^ typeString tau ^  
                           " as type constructor"))

A quantified type must always have kind  $*$ .

$$\frac{\Delta \{ \alpha_1 :: *, \dots, \alpha_n :: * \} \vdash \tau :: *}{\Delta \vdash \text{FORALL}(\langle \alpha_1, \dots, \alpha_n \rangle, \tau) :: *} \quad (\text{KINDALL})$$

The quantified variables  $\alpha_1, \dots, \alpha_n$  may be used in  $\tau$ , so we insert them into  $\Delta$  before checking the kind of  $\tau$ .

**389a.** *(definition of internal function kind 388a)*  $\vdash \equiv$  (387b)  $\triangleleft$  388d

```

| kind (FORALL (alphas, tau)) =
  let val Delta' =
    foldl (fn (a, Delta) => bind (a, TYPE, Delta)) Delta alphas
  in case kindof (tau, Delta')
    of TYPE      => TYPE
    | ARROW _   =>
      raise TypeError "quantified a non-nullary type constructor"
  end

```

§6.6.10  
Other building  
blocks of a type  
checker

389

*Variables and parameters must have kind TYPE*

A type-level expression used to describe a variable or parameter must have kind TYPE. Function asType ensures it.

**389b.** *(kind checking for Typed  $\mu$ Scheme 387b)*  $\vdash \equiv$  (S394b)  $\triangleleft$  387b

```

fun asType (ty, Delta) =
  case kindof (ty, Delta)
    of TYPE      => ty
    | ARROW _   => raise TypeError ("used type constructor ``" ^ typeString ty ^ ``'' as a type")

```

asType : tyex \* kind env -> tyex

*Evaluation in the presence of polymorphism*

This chapter is about types, but we also need to know how code is evaluated. I have chosen an operational semantics in which types have no effect at run time; the rules for evaluating expressions are therefore the same as the rules for  $\mu$ Scheme, plus rules for type abstraction and application. These new rules specify that the evaluator behaves as if these type abstraction and application aren't there.

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{TYAPPLY}(e, \tau_1, \dots, \tau_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{TYAPPLY})$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{TYLAMBDA}(\langle \alpha_1, \dots, \alpha_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{TYLAMBDA})$$

This semantics is related to program transformation called *type erasure*: if you start with a program written in Typed  $\mu$ Scheme, and you remove all the TYAPPLYs and the TYLAMBDAs, and you remove the types from the LAMBDAs and the definitions, and you rewrite VALREC to VAL, then what's left is a  $\mu$ Scheme program.

The evaluator for Typed  $\mu$ Scheme resembles the evaluator for  $\mu$ Scheme in Chapter 5. The code for the new forms acts as if TYAPPLY and TYLAMBDA aren't there.

**389c.** *(alternatives for ev for TYAPPLY and TYLAMBDA 389c)*  $\equiv$  (S398a)

```

| ev (TYAPPLY (e, _)) = ev e
| ev (TYLAMBDA (_, e)) = ev e

```

The rest of the evaluator can be found in Appendix Q.

As in Typed Impcore, a new definition must not change the type of an existing name. In Typed Impcore, the problem is avoided by permitting a name to be redefined only when the redefinition preserves the existing type. In Typed  $\mu$ Scheme, as described in Exercise 46 in Chapter 2, a new definition never affects an existing name; instead, VAL always creates a new binding. In a VAL binding, the right-hand side is evaluated in the old environment; in a VAL-REC binding, the right-hand side

ARROW,	in Typed $\mu$ Scheme
	364a
in Typed $\mu$ Scheme	
	S425a
bind	312b
CONAPP	366a
Delta	387b
eqKind,	in Typed $\mu$ Scheme
	364b
in Typed $\mu$ Scheme	
	S425b
eqKinds,	in Typed $\mu$ Scheme
	364b
in Typed $\mu$ Scheme	
	S425b
ev	S398a
find	311b
FORALL	366a
FUNTY	366a
kindof	387b
NotFound	311b
TYAPPLY	370a
TYCON	366a
TYLAMBDA	370a
TYPE,	in Typed $\mu$ Scheme
	S425a
in Typed $\mu$ Scheme	
	364a
TypeError	S237b
typeString	S394c
TYVAR	366a

is evaluated in the new environment. The type system guarantees that the result of evaluation does not depend on the unspecified value with which  $\ell$  is initialized.

$$\frac{\ell \notin \text{dom } \sigma}{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{VAL})$$

$$\frac{\ell \notin \text{dom } \sigma}{\langle \text{VAL-REC}(x, \tau, e), \rho, \sigma \rangle \rightarrow \langle \rho[x \mapsto \ell], \sigma[\ell \mapsto v] \rangle} \quad (\text{VAL-REC})$$

The code that implements these rules is in Appendix Q.

#### Primitive type constructors of Typed $\mu$ Scheme

The code above gives representations of and operations on types, but it doesn't make it easy to write types. For example, inside the interpreter, the type of `cons` has to be written using this enormous phrase:

```
FORALL (["'a"]),
  FUNTY ([TYVAR "'a", CONAPP (TYCON "list", [TYVAR "'a"])),
         CONAPP (TYCON "list", [TYVAR "'a"])))
```

To make it easier to define the primitive operations of Typed  $\mu$ Scheme, I provide these representations of types:

**390a.** *(types for Typed  $\mu$ Scheme 366a) +≡*

```
val inttype = TYCON "int"
val booletype = TYCON "bool"
val symtype = TYCON "sym"
val unittype = TYCON "unit"
val tvA = TYVAR "'a"
fun listtype ty = CONAPP (TYCON "list", [ty])
```

(S394b)  $\triangleleft$  366a

inttype	: tyex
booletype	: tyex
symtype	: tyex
unittype	: tyex
tvA	: tyex
listtype	: tyex -> tyex

These representations are used internally. When Typed  $\mu$ Scheme code refers to a type, it uses the primitive bindings for the type constructors shown in chunk 365.

Types like `int` and `bool` have natural representations. The `unit` type does not truly need a run-time representation, but my interpreter is set up to expect one. Moreover, because a programmer may compare values of type `unit` for equality, and because my implementation of equality compares representations, the representation needs to be the same everywhere. It is defined here:

**390b.** *(utility functions on  $\mu$ Scheme, Typed  $\mu$ Scheme, and nano-ML values 390b) ≡* (S373a S394b)

```
val unitVal = NIL
```

unitVal	: value
---------	---------

#### Selected primitive functions of Typed $\mu$ Scheme

Each primitive function has a name, a value, and a type. Most of them appear in the Supplement, but to show you how primitives are defined, a few appear here.

As in Chapter 5, primitive values are made using functions `unary0p`, `binary0p`, and `arith0p`. But if something goes wrong at run time, the Typed  $\mu$ Scheme versions don't raise the `RuntimeError` exception; they raise `BugInTypeChecking`. And

Typed  $\mu$ Scheme's primitives need types. As the type of the arithmetic primitives, I define `arithtype`.

**391a.** *(utility functions and types for making Typed  $\mu$ Scheme primitives 391a)*  $\equiv$  (S394a)

<code>unaryOp : (value -&gt; value) -&gt; (value list -&gt; value)</code>
<code>binaryOp : (value * value -&gt; value) -&gt; (value list -&gt; value)</code>
<code>arithOp : (int * int -&gt; int) -&gt; (value list -&gt; value)</code>
<code>arithtype : tyex</code>

```
val arithtype =
  FUNTY ([inttype, inttype], inttype)
```

As in Chapter 5, I use the chunk *(primitive functions for Typed  $\mu$ Scheme :: 391b)* to cons up all the primitives into one giant list, and I use that list to build the initial basis. Each primitive has a type as well as a value.

**391b.** *(primitive functions for Typed  $\mu$ Scheme :: 391b)*  $\equiv$  (391e) 391c▷

```
("+", arithOp op +, arithtype) ::  
("-", arithOp op -, arithtype) ::  
("*", arithOp op *, arithtype) ::  
("/", arithOp op div, arithtype) ::
```

The list primitives have polymorphic types.

**391c.** *(primitive functions for Typed  $\mu$ Scheme :: 391b)*  $\langle+$  (391e) < 391b

```
"null?", unaryOp (BOOLV o (fn (NIL) => true | _ => false))  
  , FORALL ([["'a"]], FUNTY ([listtype tvA], booletype)) ::  
("cons", binaryOp (fn (a, b) => PAIR (a, b))  
  , FORALL ([["'a"]], FUNTY ([tvA, listtype tvA], listtype tvA))) ::  
("car", unaryOp (fn (PAIR (car, _)) => car  
  | v => raise RuntimeError  
    ("car applied to non-list " ^ valueString v))  
  , FORALL ([["'a"]], FUNTY ([listtype tvA], tvA))) ::  
("cdr", unaryOp (fn (PAIR (_, cdr)) => cdr  
  | v => raise RuntimeError  
    ("cdr applied to non-list " ^ valueString v))  
  , FORALL ([["'a"]], FUNTY ([listtype tvA], listtype tvA))) ::
```

Other primitives are relegated to the Supplement.

### Typed $\mu$ Scheme's basis

A basis comprises a kind environment, a type environment, and a value environment.

**391d.** *(definition of basis for Typed  $\mu$ Scheme 391d)*  $\equiv$  (S393d)

```
type basis = kind env * tyex env * value ref env
```

The initial basis starts with the kinds of the primitive type constructors, plus the types and values of the primitive functions and values.

**391e.** *(definition of primBasis for Typed  $\mu$ Scheme 391e)*  $\equiv$  (S394a)

```
val primBasis =
  let fun addKind ((name, kind), kinds) =
    bind (name, kind, kinds)
  val kinds = foldl addKind emptyEnv
    ((primitive type constructors for Typed  $\mu$ Scheme :: 365) [])
  fun addPrim ((name, prim, funty), (types, values)) =
    (bind (name, funty, types)
     , bind (name, ref (PRIMITIVE prim), values)
    )
  val (types, values) = foldl addPrim (emptyEnv, emptyEnv)
    ((primitive functions for Typed  $\mu$ Scheme :: 391b) [])
```

<code>kinds : kind</code>	<code>env</code>
<code>types : tyex</code>	<code>env</code>
<code>values : value ref</code>	<code>env</code>
<code>primBasis : basis</code>	

§6.6.10  
Other building  
blocks of a type  
checker  
391

<code>arithOp</code>	S389e
<code>binaryOp</code>	S389d
<code>bind</code>	312b
<code>BOOLV</code>	370b
<code>CONAPP</code>	366a
<code>emptyEnv</code>	311a
<code>type env</code>	310b
<code>FORALL</code>	366a
<code>FUNTY</code>	366a
<code>type kind,</code>	in Typed $\mu$ Scheme
	364a
<code>in Typed <math>\mu</math>Scheme</code>	S425a
<code>NIL,</code>	in nano-ML 415b
<code>in Typed <math>\mu</math>Scheme</code>	370b
<code>PAIR</code>	370b
<code>PRIMITIVE</code>	370b
<code>RuntimeError</code>	S366c
<code>TYCON</code>	366a
<code>type tyex</code>	366a
<code>TYVAR</code>	366a
<code>unaryOp</code>	S389d
<code>type value</code>	370b
<code>valueString</code>	314

```

fun addVal ((name, v, ty), (types, values)) =
  ( bind (name, ty, types)
  , bind (name, ref v, values)
  )
val (types, values) =
  foldl addVal (types, values)
  (<primitives that aren't functions, for Typed  $\mu$ Scheme :: S400d> [])
in (kinds, types, values)
end

```

With the primitives in place, the basis is completed by reading and evaluating the predefined functions. The code is relegated to the Supplement.

## 6.7 TYPE SYSTEMS AS THEY REALLY ARE

Typed Impcore is a good model of a monomorphic language, but real languages are more complicated. As one example, most programming languages, especially monomorphic ones, use product types with named fields. These are often called “record” or “struct” types. The type rules are mostly straightforward; type checking involves matching names, not using positional notation.

Typed  $\mu$ Scheme is not a good model for any widely used language—the annotations put too heavy a burden on the programmer. No serious language designer would ask programmers to use a polymorphic type system that requires explicit types everywhere; it’s especially burdensome to have to instantiate every polymorphic value explicitly. Typed  $\mu$ Scheme is, however, a good model for an intermediate form to which a real polymorphic source language could be translated. It becomes an even better, more powerful model, if we permit type-lambda to quantify over a type variable of *any* kind, not just of kind \*. This sort of parameterization is available in some versions of the functional language Haskell.

In this chapter, comparing types for equality is fairly straightforward: two types are equal if and only if they apply the same constructor to equal arguments. Quantified types are considered equal up to renaming of bound type variables, but that is a minor matter. In real languages, comparing types for equality is a more complicated business, primarily because of *generativity*. A syntactic form is generative if every appearance creates a distinct type, different from any other type. As an example, the product-type constructor in C, called `struct`, is generative. Standard ML’s sum-type definition form, called `datatype`, is also generative. Languages without generativity are sometimes said to compare types using *structural equivalence*; languages with generativity are said to use *name equivalence* or *occurrence equivalence*.

Many languages permit an operator to be used at more than one type, but not at infinitely many types. For example, the `+` operator in ML may be used at two types: `int * int -> int` and `real * real -> real`. Such an operator is not parametrically polymorphic; instead, it is said to be *overloaded*. The `+` operator in C is even more heavily overloaded; because of implicit type conversions, it may be used at many types. In more recent languages, including Ada, C++, and Haskell, programmers can define new overloaded operators or functions. Overloading can complicate a type system; but it can also be done simply, as in Chapter 9 of this book.

Type systems used for research go far beyond what is presented in this book. In systems based on *dependent types*, for example, type checking can be undecidable while still giving useful results in practice!

## 6.8 SUMMARY

Type systems have been called the world’s most successful formal method. Types guide the construction of programs: a strategy used throughout Chapter 2 is that if you are writing a function that consumes a value of type  $\tau$ , the body of your function should be the elimination construct for  $\tau$ . (The dual strategy, which is used less often, is that if you are trying to produce a value of type  $\tau'$ , perhaps the body of your function should include the introduction construct for  $\tau'$ .) Types also provide a relatively painless way of documenting code, and they rule out many silly programming errors.

If types are good, polymorphic types are better. Polymorphism is a key strategy for making code reusable, robustly. The polymorphism in Typed  $\mu$ Scheme is easy to implement, but unpleasant to use—it should be hidden inside a compiler. But similar forms of polymorphism can be easy to use, and sometimes a great pleasure. These are found in Chapters 7 to 9.

The simplest, most easily implemented technique for enforcing a type discipline is a type checker. Most type checkers are easy to implement because most type systems have one rule for each syntactic form. But if you add sophisticated features, parts of a type checker can become more challenging. In Typed  $\mu$ Scheme, these parts include type equivalence, which is an interesting aspect of many experimental type systems, and substitution, which is a ubiquitous, annoying problem.

### 6.8.1 Key words and phrases

**TYPE** A specification for a **TERM**. Or a means of classifying terms. Or a collection of values, called the **INHABITANTS**.

**TERM** The pointy-headed theory word for “expression.” More generally, a syntactic form that is computed with at run time.

**TYPE SYSTEM** A language’s *type system* encompasses both the set of **TYPES** that can be expressed in the language and the rules that say what **TERMS** have what types.

**TYPE CHECKER** A part of a language’s implementation that enforces the rules of the type system, by checking code before it is run.

**MONOMORPHIC TYPE SYSTEM** If every **TERM**, variable, and function in a language has at most one **TYPE**, that language uses a *monomorphic* type system. For example, Pascal’s type system is monomorphic. A type system may also be **POLYMORPHIC**.

**POLYMORPHIC TYPE SYSTEM** If a **TERM**, variable, or function in a language may be used at more than one **TYPE**, that language uses a *polymorphic* type system. For example, ML’s type system is polymorphic. A type system may also be **MONOMORPHIC**.

**POLYMORPHISM** A language is polymorphic if it is possible to write programs that operate on values of more than one type. For example, Scheme and ML are polymorphic languages. A value is polymorphic if it can be used at more than one type; for example, the list `length` function is polymorphic because it can operate on many types of lists.

**PARAMETRIC POLYMORPHISM** The form of polymorphism that uses type parameters and **INSTANTIATION**.

**TYPE ABSTRACTION** In PARAMETRIC POLYMORPHISM, the INTRODUCTION FORM for a polymorphic type. In Typed  $\mu$ Scheme, written type-lambda.

**TYPE APPLICATION** In PARAMETRIC POLYMORPHISM, the ELIMINATION FORM for a polymorphic type. It substitutes actual type parameters for quantified type variables. It is a form of INSTANTIATION.

**TYPE CONSTRUCTOR** The fundamental unit from which TYPES are built. Type constructors come in various KINDS. Nullary type constructors such as int and bool are types all by themselves; they have kind \*. Other type constructors, such as list and array, have to be applied to types to make types.

**MONOTYPE** A type that cannot be instantiated. For example, the function type  $\text{int list} \rightarrow \text{int}$  is a monotype. (As contrasted with a POLYTYPE.)

**POLYTYPE** A type that can be instantiated in more than one way. In Typed  $\mu$ Scheme, a polytype has the form  $\forall \alpha_1, \dots, \alpha_n . \tau$ . (As contrasted with a MONOTYPE.)

**INSTANTIATION** The process of determining at what type a polymorphic value is used. In Typed  $\mu$ Scheme, a polymorphic value is instantiated explicitly using the TYPE-APPLICATION form @. In ML, instantiation is implicit and is handled automatically by the language implementation.

**QUANTIFIED TYPE** A type formed with the universal quantifier  $\forall$ . Also called a POLYTYPE.

**FORMATION RULE** A rule that says how to make a well-formed type. For example,  $(\text{array int})$  is a well-formed type.

**INTRODUCTION FORM** A syntactic form used to create a value of a given type. For example, the introduction form for a function type is lambda. An introduction form “puts information in” to the value. The information can be recovered using an ELIMINATION FORM. The type rule for an introduction form can be called an INTRODUCTION RULE.

**ELIMINATION FORM** A syntactic form used to observe a value of a given type. For example, the elimination form for a function type is function application. An elimination form “takes information out” that was put in using an INTRODUCTION FORM. The type rule for an elimination form can be called an ELIMINATION RULE.

**KIND** A means of classifying type constructors and types. Using kinds makes it possible to handle an unbounded number of TYPE CONSTRUCTORS using finitely many FORMATION RULES.

**GENERATIVITY** If a language construct always creates a new type distinct from any other type, that construct is called *generative*. Examples of generative constructs include C’s struct and ML’s datatype.

### 6.8.2 Further reading

Pierce (2002) has written a wonderful textbook covering many aspects of typed programming languages. Cardelli (1997) presents an alternative view of type systems; his tutorial inspired some of the material in this chapter.

Reynolds (1974) presents the polymorphic, typed lambda calculus now known as System F, which is the basis of Typed  $\mu$ Scheme. I hope you will agree with Reynolds’s characterization of his own work:

<i>Exercises</i>	<i>Sections</i>	<i>Notes</i>
1 to 3	6.1	Type-system fundamentals: type errors vs run-time errors; introduction forms vs elimination forms (§6.3.1); inhabitants of sum and product types (§6.4).
4 to 7	6.3.1, 6.4	Extending a monomorphic language with lists, records, sums, or mutable references.
8 and 9	6.6.3	Coding in a polymorphic language
10 to 15	6.4, 6.6.2, 6.6.3, 6.6.5	Extending a polymorphic language with queues, pairs, sums, polymorphic references, or records.
16 and 17	6.1.5, 6.6.5, 1.7	Writing typing derivations.
18 and 19	6.2, 6.3, 6.6.5	Type checking: add arrays to Typed Impcore's type checker, implement a type checker for Typed $\mu$ Scheme.
20 to 24	6.1, 6.6.5	Metatheory: types are unique, expressions have well-formed types, syntactic sugar preserves typing, $\equiv$ is an equivalence relation (§6.6.6).
25 and 26	6.1.5, 6.5	Metatheory about implementation: type checking terminates and prevents bugs.
27 and 28	6.6.7	Capture-avoiding substitution: rename variables to avoid capture; prove that substitution terminates.
29 to 33	6.1.5, 6.6.5, 6.6.7	Holes in type systems: what goes wrong when restrictions are lifted.

§6.9. Exercises

395

Table 6.6: Synopsis of all the exercises, with most relevant sections

Although this language is hardly an adequate vehicle for programming, it seems to pose the essence of the type structure problem, and it is simple enough to permit a brief but rigorous exposition of its semantics.

Using types to guide the construction of programs is a key part of the “design recipe” method of Felleisen et al. (2018). Because the language used by Felleisen et al. does not have a static type checker, the types are written only in comments—but they are there. Crestani and Sperber (2010) describe an extension in which type “signatures” are used to check types at run time.

Perhaps surprisingly, type systems can be used to guarantee safety even in C programs, although at a significant run-time penalty. Systems such as CCured (Necula, McPeak, and Weimer 2002) run C programs at about half the speed of unsafe compilers, but their error-detection power is comparable to that of tools such as Purify and Valgrind, which may slow down a program by a factor of 5 to 10.

## 6.9 EXERCISES

The exercises are summarized in Table 6.6. This chapter’s exercises are unusually diverse; they include programming, adding new type rules, proving things about type systems, extending interpreters, and subverting type systems. Here are some of my favorites:

- Nothing solidifies your understanding of type systems like writing a type checker. Exercise 19 on page 404 asks you to write a type checker for

Typed  $\mu$ Scheme, using the typing rules as your specification. If you want your type checker to be sound, you will also want to complete the implementation of capture-avoiding substitution (Exercise 28 on page 407). An easier alternative, or a warmup, would be to extend the type checker for Typed Impcore so it supports arrays (Exercise 18 on page 403).

- Another way to develop understanding is to write type rules for familiar language constructs (Exercises 4 to 7, starting on page 400). The easiest and more familiar constructs are for lists (Exercise 4).
- To understand both the power and the agony of programming with explicit polymorphism, implement `exists?` or `all?` in Typed  $\mu$ Scheme (Exercise 9 on page 401).
- To understand how explicit polymorphism benefits implementors and language designers, extend Typed  $\mu$ Scheme with new type constructors (Exercises 10 to 13, starting on page 401). You won't have to change any infrastructure. The easiest, most familiar new type constructor is the `pair` type constructor (Exercise 11).
- Type systems lend themselves well to metatheory. My favorite metatheoretic exercise is to show that in Typed  $\mu$ Scheme, any type that classifies a term is well formed and has kind  $*$  (Exercise 23 on page 406). Exercise 20 on page 404 calls for similar reasoning but has a more familiar conclusion: type checking is deterministic.

#### 6.9.1 Retrieval practice and other short questions

- A. What's an example of a checked run-time error in Impcore that is guaranteed to be prevented by the type system of Typed Impcore?
- B. What's an example of a checked run-time error in Impcore that is *not* guaranteed to be prevented by the type system of Typed Impcore?
- C. How many *values* inhabit the type `bool`?
- D. How many *terms* have type `bool`?
- E. In Typed Impcore, a global variable may have an array type. How many distinct array types are possible?
- F. When you look up a name  $x$  in environment  $\Gamma_\xi$ , what information about  $x$  do you get back?
- G. How do you pronounce the judgment form  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$ ?
- H. How do you pronounce the judgment form  $\Delta, \Gamma \vdash e : \tau$ ?
- I. When the Typed Impcore interpreter checks the type of a `WHILE` loop, is the type checking guaranteed to terminate? Why or why not?
- J. When the Typed Impcore interpreter checks the type of a `WHILE` loop, how does the type of the loop's body affect the type of the loop?
- K. If the body of a `WHILE` loop doesn't have a type, what happens?
- L. In Typed Impcore, what syntactic form is the introduction form for array types?
- M. In Typed  $\mu$ Scheme, what syntactic form is the introduction form for function types?

- N. In Typed  $\mu$ Scheme, what syntactic form is the elimination form for function types?
- O. In Typed  $\mu$ Scheme, what syntactic form is the introduction form for polymorphic types?
- P. In Typed  $\mu$ Scheme, what syntactic form is the elimination form for polymorphic types?
- Q. In Typed  $\mu$ Scheme, what is the type of the primitive function “null”?
- R. In Typed  $\mu$ Scheme, the primitive function `cons` does not have a function type—it has a polymorphic type. How do you get it to act as a function?
- S. In both Typed Impcore and Typed  $\mu$ Scheme, what types have to be checked to make sure they are well formed? Where would an ill-formed type come from?
- T. In Typed  $\mu$ Scheme, why aren’t there type-formation rules for list types? How does the type checker know if a list type is well formed?
- U. In Typed  $\mu$ Scheme, what’s the relationship between instantiation and substitution?
- V. Why does Typed  $\mu$ Scheme need a complicated  $\equiv$  relation? Why doesn’t Typed Impcore need such a thing?

### §6.9. Exercises

397

#### 6.9.2 Type-system fundamentals

1. *Differences between type errors and run-time errors.* Programming-language people get good at thinking about *phases* of computation. This chapter introduces a typechecking phase, which happens before the evaluation phase. Part of thinking about phases is learning how different things can go wrong in different phases; a type error is not the same as a run-time error. To convince yourself that you understand what can go wrong in different phases, create unit tests using both `check-error` and `check-type-error`.
  - (a) Using the `array-at` primitive, create one test for each phase.
  - (b) Using an integer-arithmetic primitive, create one test for each phase.

All four tests should pass.

2. *Introduction forms and elimination forms.* In this exercise, you classify syntactic forms (and their associated rules) as introduction forms or elimination forms. The exercise is modeled on communication techniques found in languages like PML/Pegasus, Concurrent ML, and Haskell.

A expression of type  $\text{PROTO}(\tau)$  is a set of instructions, or *protocol*, for communicating with a remote server. Such an expression can be *run* by a special syntactic form, which communicates with the server. When a communication is run, the local interpreter gets an *outcome* of type  $\tau$ . Here is a grammar, with informal explanations:

$\text{exp} ::= \text{send } \text{exp}$	Send a value to the server
$\text{receive}$	Receive a value from the server
$\text{do } x \leftarrow \text{exp}_1 \text{ in } \text{exp}_2$	Protocol $\text{exp}_1$ , whose outcome is $x$ , followed by $\text{exp}_2$
$\text{locally } \text{exp}$	Produce result $\text{exp}$ locally, without communicating
$\text{run } \text{exp}$	Run a protocol

	UNITTYPE	INTTYPE	BOOLTYPE
$\tau$ is a type	UNIT is a type	INT is a type	BOOL is a type
		ARRAYTYPE	
		$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}}$	

Figure 6.7: Type-formation rules for Typed Impcore

$\boxed{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$	<b>LITERAL</b> $\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LITERAL}(v) : \text{INT}}$
<b>FORMALVAR</b> $\frac{x \in \text{dom } \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\rho(x)}$	<b>GLOBALVAR</b> $\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\xi(x)}$
<b>FORMALASSIGN</b> $\frac{x \in \text{dom } \Gamma_\rho \quad \Gamma_\rho(x) = \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$	<b>GLOBALASSIGN</b> $\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi \quad \Gamma_\xi(x) = \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}$
$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{IF}(e_1, e_2, e_3) : \tau}$	
<b>WHILE</b> $\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}}$	
<b>BEGIN</b> $\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \quad \dots \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n}$	
<b>EMPTYBEGIN</b> $\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}() : \text{UNIT}}$	
<b>APPLY</b> $\frac{\Gamma_\phi(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(f, e_1, \dots, e_n) : \tau}$	
<b>EQ</b> $\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{EQ}(e_1, e_2) : \text{BOOL}}$	<b>PRINTLN</b> $\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{PRINTLN}(e) : \text{UNIT}}$

Figure 6.8: Typing rules for Typed Impcore expressions

$$\boxed{\langle t, \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma'_\xi, \Gamma'_\phi \rangle} \quad \begin{array}{c} \text{NEWVAL} \\ \frac{\Gamma_\xi, \Gamma_\phi, \{ \} \vdash e : \tau \quad x \notin \text{dom } \Gamma_\xi}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi \{ x \mapsto \tau \}, \Gamma_\phi \rangle} \end{array}$$

$$\begin{array}{c} \text{OLDVAL} \\ \frac{\Gamma_\xi, \Gamma_\phi, \{ \} \vdash e : \tau \quad \Gamma_\xi(x) = \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \end{array} \quad \begin{array}{c} \text{EXP} \\ \frac{\Gamma_\xi, \Gamma_\phi, \{ \} \vdash e : \tau}{\langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle} \end{array}$$

DEFINE

$$\frac{\begin{array}{c} \tau_1, \dots, \tau_n \text{ are types} \\ \tau_f = \tau_1 \times \dots \times \tau_n \rightarrow \tau \\ f \notin \text{dom } \Gamma_\phi \\ \Gamma_\xi, \Gamma_\phi \{ f \mapsto \tau_f \}, \{ x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n \} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \{ f \mapsto \tau_f \} \rangle}$$

REDEFINE

$$\frac{\begin{array}{c} \tau_1, \dots, \tau_n \text{ are types} \\ \Gamma_\phi(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau \\ \Gamma_\xi, \Gamma_\phi \{ f \mapsto \tau_1 \times \dots \times \tau_n \rightarrow \tau \}, \{ x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n \} \vdash e : \tau \end{array}}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi, \Gamma_\phi \rangle}$$

§6.9. Exercises

399

Figure 6.9: Typing rules for Typed Impcore definitions

Here are the rules:

$$\begin{array}{ll} \text{PROTO} & \text{SEND} \\ \frac{\tau \text{ is a type}}{\text{PROTO}(\tau) \text{ is a type}} & \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{send } e : \text{PROTO(UNIT)}} \\ \text{RECEIVE} & \text{Do} \\ \frac{\tau \text{ is a type}}{\Gamma \vdash \text{receive} : \text{PROTO}(\tau)} & \frac{\begin{array}{c} \Gamma \vdash e_1 : \text{PROTO}(\tau) \\ \Gamma, x : \tau \vdash e_2 : \text{PROTO}(\tau') \end{array}}{\Gamma \vdash \text{do } x \leftarrow e_1 \text{ in } e_2 : \text{PROTO}(\tau')} \\ \text{LOCALLY} & \text{RUN} \\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{locally } e : \text{PROTO}(\tau)} & \frac{\Gamma \vdash e : \text{PROTO}(\tau)}{\Gamma \vdash \text{run } e : \tau} \end{array}$$

Please classify each rule as a formation rule, an introduction rule, or an elimination rule. Justify your answers.

3. *Counting inhabitants.* Type `bool` is inhabited by the two values `#t` and `#f`. Let's say type `lettergrade` is inhabited by values *A*, *B*, *C*, *D*, and *F*.
- List all the values inhabited by product type `bool`  $\times$  `lettergrade`.
  - List all the values inhabited by sum type `bool`  $+$  `lettergrade`.
  - Are your results consistent with the words “sum” and “product”? Justify your answer.

## 6.9.3 Extending a monomorphic language

4. *Rules for lists in Typed Impcore.* Imagine you want to add lists to Typed Impcore. Use the same technique we use for arrays: devise new abstract syntax to support lists, and write appropriate type-formation, type-introduction, and type-elimination rules. The rules should resemble the rules shown in Section 6.4.

Review the discussion of rules in the sidebar on page 356, and make it obvious which rules are formation rules, which rules are introduction rules, and which rules are elimination rules. Divide the rules into three groups, and label each group. Take care with your classification; when you have a recursive type like lists, you can't always classify rules by looking at types above and below the line.

- Some rules can be classified just by looking to see where list types appear. For example, a rule for `null?` should have a list type in the premise but not in the conclusion, so `null?` has to be an elimination form. Similarly, a `car` rule should have a list type in the premise but not necessarily in the conclusion, so it too has to be an elimination form.
- Other rules have list types in both premises and conclusion. When you have forms like `cons` and `cdr`, which both take and produce lists, you have to fall back on thinking about information. Does a form put new information into a value, which can later be extracted by another form? Then it is an introduction form. Does a form put in no new information, but only extract information that is already present? Then it is an elimination form.

Your abstract syntax should cover all the list primitives defined in Chapter 2: the empty list, test to see if a list is empty, `cons`, `car`, and `cdr`. Your abstract syntax may differ from the abstract syntax used in  $\mu$ Scheme.

Be sure your rules are deterministic: it should be possible to compute the type of an expression given only the syntax of the expression and the current type environment.

5. *Rules for records in Typed Impcore.* Give type rules for records with named fields.
6. *Rules for sums in Typed Impcore.* Give type rules for sums with named variants.
7. *Rules for mutable references in Typed Impcore.* Mutable cells can be represented by a type constructor `ref`. The appropriate operations are `ref`, `!`, and `:=`. The function `ref` is like the function `allocate` in Chapter 2; applying `ref` to a value  $v$  allocates a new mutable cell and initializes it to hold  $v$ . Applying `!` to a mutable cell returns the value contained in that cell. Applying `:=` to a mutable cell and a value replaces the contents of the cell with the value.

Give type rules for a type constructor for mutable cells. (See also Exercise 13.)

## 6.9.4 Coding in a polymorphic language

8. *Folds.* Implement `foldl` and `foldr` in Typed  $\mu$ Scheme.
- Both functions should have the same polymorphic type. Give it.
  - Write an implementation of each function.

9. *Higher-order, polymorphic linear search.* Implement `exists?` and `all?` in Typed  $\mu$ Scheme.

- (a) Both functions should have the same polymorphic type. Give it.
- (b) Write an implementation of `exists?` using recursion.
- (c) Write an implementation of `all?` using your implementation of `exists?` and De Morgan's laws.

### 6.9.5 Extending a polymorphic language

#### §6.9. Exercises

401

10. *Add queues to Typed  $\mu$ Scheme.* A great advantage of a polymorphic type system is that the language can be extended without touching the abstract syntax, the values, the type checker, or the evaluator. Without changing any of these parts of Typed  $\mu$ Scheme, extend Typed  $\mu$ Scheme with a queue type constructor and the polymorphic values `empty-queue`, `empty?`, `put`, `get-first`, and `get-rest`. (A more typical functional queue provides a single `get` operation which returns a pair containing both the first element in the queue and any remaining elements. If instead you write the two functions `get-first` and `get-rest`, you won't have to fool with pair types.)

- (a) What is the kind of the type constructor `queue`? Add it to the initial  $\Delta$ .
  - (b) What are the types of `empty-queue`, `empty?`, `get-first`, `get-rest`, and `put`?
  - (c) Add the new primitive functions to the initial  $\Gamma$  and  $\rho$ . You will need to write implementations in Standard ML.
11. *Add pairs to Typed  $\mu$ Scheme.* Without changing the abstract syntax, values, type checker, or evaluator of Typed  $\mu$ Scheme, extend Typed  $\mu$ Scheme with the `pair` type constructor and the polymorphic functions `pair`, `fst`, and `snd`.
- (a) What is the kind of the type constructor `pair`? Add it to the initial  $\Delta$ .
  - (b) What are the types of `pair`, `fst`, and `snd`?
  - (c) Add the new primitive functions to the initial  $\Gamma$  and  $\rho$ . As you add them to  $\rho$ , you can use the same implementations that we use for `cons`, `car`, and `cdr`.
12. *Add sums to Typed  $\mu$ Scheme.* Without changing the abstract syntax, values, type checker, or evaluator of Typed  $\mu$ Scheme, extend Typed  $\mu$ Scheme with the `sum` type constructor and the polymorphic functions `left`, `right`, and `either`.

- (a) What is the kind of the type constructor `sum`? Add it to the initial  $\Delta$ .
- (b) What are the types of `left`, `right`, and `either`?
- (c) Page 358 gives algebraic laws for pair primitives in a monomorphic language. If the sum primitives were added to a monomorphic language, what would be the laws relating `LEFT`, `RIGHT`, and `EITHER`?
- (d) Since `left`, `right`, and `either` have the polymorphic types in part 12(b), what are the laws relating them?
- (e) Add `left`, `right`, and `either` to the initial  $\Gamma$  and  $\rho$  of Typed  $\mu$ Scheme. Try representing a value of sum type as a PAIR containing a tag and a value.

13. Add references to Typed  $\mu$ Scheme. In Typed  $\mu$ Scheme, it is not necessary to add any special abstract syntax to support mutable cells as in Exercise 7. Give the kind of the `ref` constructor and the types of the operations `ref`, `!`, and `:=`.
14. Add simple polymorphic records to Typed  $\mu$ Scheme. Extend Typed  $\mu$ Scheme by adding polymorphic records with named fields. Types of fields are not specified; instead, the extension creates functions that work like the record functions in Chapter 2, except these functions are polymorphic.

- (a) Add a new form of definition. It should have this concrete syntax:

```
def ::= (record record-name ({field-name}))
```

The abstract syntax can be this:

```
RECORD of name * name list
```

- (b) Modify `typdef` so that it returns a new kind environment  $\Delta'$  as well as a new type environment  $\Gamma'$ .
- (c) A record definition should extend  $\Delta$  by adding a new type constructor `record-name` with kind  $*_1 \times \dots \times *_n \Rightarrow *$ , where  $n$  is the number of `field-names`.
- (d) The record `record` should add a new function `make-record-name`. This function should take one argument for each field and build a record value.
- (e) For each field, the record definition should add a function named by joining the `record-name` and `field-name` with a dash. This function should extract the named field from the structure and return its value.

Here is an example.

402. *(exercise transcript 402)* ≡

```
-> (record assoc key val)
-> (val p ((@ make-assoc sym int) 'class 152))
p : (assoc sym int)
-> ((@ assoc-key sym int) p)
class : sym
-> ((@ assoc-val sym int) p)
152 : int
```

403▷

15. Add more expressive records to Typed  $\mu$ Scheme. Extend Typed  $\mu$ Scheme by adding records with fields that are named and typed. This extension creates record functions that are monomorphic, with types determined by the types in the record specification.

- (a) Add a new kind of definition. It should have this concrete syntax:

```
def ::= (typed-record (record-name {'type-variable-name}) 
({[field-name : type]}))
```

This sort of record is also polymorphic, but under more control: type parameters are listed explicitly, and the type of each field is declared.

The abstract syntax can be this:

```
TYPED_RECORD of name * name list * (name * tyex) list
```

- (b) Modify `typdef` so that it returns a new kind environment  $\Delta'$  as well as a new type environment  $\Gamma'$ .

- (c) The typed-record definition should extend  $\Delta$  by adding a new type constructor *record-name* with kind  $*_1 \times \cdots \times *_n \Rightarrow *$ , where  $n$  is the number of *type-variable-names* in the definition.
- (d) The typed-record definition should add a new function *make-record-name*. This function should take one argument for each field and build a record value.
- (e) For each field, the typed-record definition should add a function named by joining the *record-name* and *field-name* with a dash. This function should extract the named field from the structure and return its value.

## §6.9. Exercises

403

Here is an example.

**403.** (exercise transcript 402) +≡  
 ↗ (typed-record (assoc 'a) ([key : sym] [value : 'a]))  
 ↗ (val p ((@ make-assoc int) 'class 152))  
 p : (assoc int)  
 ↗ ((@ assoc-key int) p)  
 class : sym  
 ↗ ((@ assoc-value int) p)  
 152 : int

↳ 402

### 6.9.6 Typing derivations

16. *The type of a polymorphic function in Typed μScheme.*

- (a) What is the type of the following function?

```
(type-lambda ('a) (type-lambda ('b)
  (lambda ([f : ('a -> 'b)] [x : 'a]) (f x))))
```

- (b) Using the type rules from the chapter, give a derivation tree proving the correctness of your answer to part (a).

17. *The type of a polymorphic function in extended Typed μScheme.* Suppose we get sick and tired of writing @ signs everywhere, so we decide to extend Typed μScheme by making PAIR, FST, and SND abstract syntax instead of functions.

- (a) What is the type of the following function?

```
(type-lambda ('a) (type-lambda ('b)
  (lambda ([p : (pair 'a 'b)]) (pair (snd p) (fst p)))))
```

- (b) Using the type rules from the chapter, give a derivation tree proving the correctness of your answer to part (a).

### 6.9.7 Implementing type checking

18. *Type checking for arrays.* Finish the type checker for Typed Impcore so that it handles arrays. It is sufficient to give code for the four cases in code chunk 357.

$\kappa$ is a kind	KINDFORMATIONTYPE	$\kappa_1, \dots, \kappa_n$ are kinds $\kappa$ is a kind
	$*$ is a kind	$\kappa_1 \times \dots \times \kappa_n \Rightarrow \kappa$ is a kind

Figure 6.10: Kind-formation rules for Typed  $\mu$ Scheme

$\Delta \vdash \tau :: \kappa$	KINDINTROCON $\mu \in \text{dom } \Delta$ $\Delta \vdash \text{TYCON}(\mu) :: \Delta(\mu)$	KINDINTROVAR $\alpha \in \text{dom } \Delta$ $\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)$
KINDAPP $\Delta \vdash \tau :: \kappa_1 \times \dots \times \kappa_n \Rightarrow \kappa$ $\Delta \vdash \tau_i :: \kappa_i, 1 \leq i \leq n$ $\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \dots, \tau_n]) :: \kappa$		
$\Delta \vdash \tau_i :: *, 1 \leq i \leq n$	KINFUNTION $\Delta \vdash \tau :: *$	KINDALL $\Delta\{\alpha_1 :: *, \dots, \alpha_n :: *\} \vdash \tau :: *$ $\Delta \vdash \text{FORALL}((\alpha_1, \dots, \alpha_n), \tau) :: *$

Figure 6.11: Kinding rules for Typed  $\mu$ Scheme types

19. *Type checking for Typed  $\mu$ Scheme.* Write a type checker for Typed  $\mu$ Scheme. That is, implement `typdef` in code chunk 375. Although you could write this checker by cloning and modifying the type checker for Typed Impcore, you will probably get better results if you build a checker from scratch by following the type rules for Typed  $\mu$ Scheme, which are shown in Figure 6.12 on the facing page and in Figure 6.13 on page 406.

- When a type in a program is supposed to be the type of a variable, the rules require that type to have kind  $*$ . You will find such types in the syntax of a `define`, `val-rec`, and `lambda`. The requirement that such types have kind  $*$  can be enforced by the function `asType` on page 389.
- When you typecheck `LETREC`, verify that every right-hand side is a `LAMBDA`.
- When you typecheck literals, use the rules on page 372. Although these rules are incomplete, they should suffice for anything the parser can produce. If a literal `PRIMITIVE` or `CLOSURE` reaches your type checker, the impossible has happened, and your code should raise an appropriate exception.

#### 6.9.8 Metatheory

20. *Types are unique.* Prove that an expression in Typed Impcore has at most one type. That is, prove that given environments  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$  and abstract-syntax tree  $e$ , there is at most one  $\tau$  such that  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$ .
21. *Desugaring preserves types.* The sidebar on page 371 notes that the only definition form we really need is `val`; `define` and `val-rec` can be expressed as syntactic sugar. The desugaring of `define` is given in the text; in this exercise, you desugar `val-rec`.

$\boxed{\Delta, \Gamma \vdash e : \tau}$	$\begin{array}{c} \text{VAR} \\ x \in \text{dom } \Gamma \quad \Gamma(x) = \tau \\ \hline \Delta, \Gamma \vdash \text{VAR}(x) : \tau \end{array}$
$\begin{array}{c} \text{SET} \\ \Delta, \Gamma \vdash e : \tau \quad x \in \text{dom } \Gamma \quad \Gamma(x) = \tau \\ \hline \Delta, \Gamma \vdash \text{SET}(x, e) : \tau \end{array}$	$\begin{array}{c} \text{WHILE} \\ \Delta, \Gamma \vdash e_1 : \text{bool} \quad \Delta, \Gamma \vdash e_2 : \tau \\ \hline \Delta, \Gamma \vdash \text{WHILE}(e_1, e_2) : \text{unit} \end{array}$
$\begin{array}{c} \text{IF} \\ \Delta, \Gamma \vdash e_1 : \text{bool} \quad \Delta, \Gamma \vdash e_2 : \tau \quad \Delta, \Gamma \vdash e_3 : \tau \\ \hline \Delta, \Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau \end{array}$	
$\begin{array}{c} \text{BEGIN} \\ \Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\ \hline \Delta, \Gamma \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n \end{array}$	$\begin{array}{c} \text{EMPTYBEGIN} \\ \hline \Delta, \Gamma \vdash \text{BEGIN}() : \text{unit} \end{array}$
$\begin{array}{c} \text{LET} \\ \Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \quad \Delta, \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \\ \hline \Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau \end{array}$	
$\begin{array}{c} \text{LETSTAR} \\ \Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1 \rangle, \text{LETSTAR}(\langle x_2, e_2, \dots, x_n, e_n \rangle, e)) : \tau \quad n > 0 \\ \hline \Delta, \Gamma \vdash \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau \end{array}$	
$\begin{array}{c} \text{EMPTYLETSTAR} \\ \Delta, \Gamma \vdash e : \tau \\ \hline \Delta, \Gamma \vdash \text{LETSTAR}(\langle \rangle, e) : \tau \end{array}$	
$\begin{array}{c} \text{LETREC} \\ \Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \\ \Delta, \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\ \Delta, \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \\ \text{Every } e_i \text{ has the form LAMBDA}(\dots) \\ \hline \Delta, \Gamma \vdash \text{LETREC}(\langle x_1 : \tau_1, e_1, \dots, x_n : \tau_n, e_n \rangle, e) : \tau \end{array}$	
$\begin{array}{c} \text{LAMBDA} \\ \Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \quad \Delta, \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau \\ \hline \Delta, \Gamma \vdash \text{LAMBDA}(\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau \end{array}$	
$\begin{array}{c} \text{APPLY} \\ \Delta, \Gamma \vdash e : \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\ \hline \Delta, \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \tau \end{array}$	
$\begin{array}{c} \text{TYLAMBDA} \\ \alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n \quad \Delta\{\alpha_1 :: *, \dots, \alpha_n :: *\}, \Gamma \vdash e : \tau \\ \hline \Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \dots, \alpha_n, e) : \forall \alpha_1, \dots, \alpha_n . \tau \end{array}$	
$\begin{array}{c} \text{TYAPPLY} \\ \Delta, \Gamma \vdash e : \forall \alpha_1, \dots, \alpha_n . \tau \quad \Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \\ \hline \Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \dots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n] \end{array}$	

Figure 6.12: Typing rules for Typed  $\mu$ Scheme expressions

$$\begin{array}{c}
 \text{VAL} \\
 \boxed{\langle d, \Delta, \Gamma \rangle \rightarrow \Gamma'} \qquad \frac{\Delta, \Gamma \vdash e : \tau}{\langle \text{VAL}(x, e), \Delta, \Gamma \rangle \rightarrow \Gamma \{ x \mapsto \tau \}}
 \end{array}$$

$$\text{VALREC} \qquad \Delta \vdash \tau :: *$$

$$\frac{\Delta, \Gamma \{ x \mapsto \tau \} \vdash e : \tau \quad e \text{ has the form LAMBDA}(\dots)}{\langle \text{VAL-REC}(x, \tau, e), \Delta, \Gamma \rangle \rightarrow \Gamma \{ x \mapsto \tau \}}$$

$$\text{EXP} \qquad \frac{\langle \text{VAL(it, e)}, \Delta, \Gamma \rangle \rightarrow \Gamma'}{\langle \text{EXP}(e), \Delta, \Gamma \rangle \rightarrow \Gamma'}$$

$$\text{DEFINE} \qquad \frac{\langle \text{VAL-REC}(f, \tau_1 \times \dots \times \tau_n \rightarrow \tau, \text{LAMBDA}(\langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e)), \Delta, \Gamma \rangle \rightarrow \Gamma'}{\langle \text{DEFINE}(f, \tau, \langle x_1 : \tau_1, \dots, x_n : \tau_n \rangle, e), \Delta, \Gamma \rangle \rightarrow \Gamma'}$$

Figure 6.13: Typing rules for Typed  $\mu$ Scheme definitions

- (a) Express VAL-REC as syntactic sugar. That is, specify a translation from an arbitrary VAL-REC form into a combination of VAL and LETREC forms.
  - (b) Prove that your translation preserves typing. That is, prove that a VAL-REC form is well typed if and only if its desugaring is well typed. And prove that when both are well typed, the final type environments on the right-hand side of the  $\rightarrow$  judgment are equal.
22. *The type of a Typed Impcore expression is well formed.* Using a metatheoretic argument about typing derivations in Typed Impcore, prove that if there is a derivation of a typing judgment  $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$ , there is also a derivation of the judgment “ $\tau$  is a type.” Use structural induction on the derivation of the typing judgment. (For a similar exercise using Typed  $\mu$ Scheme, see Exercise 23 on page 406.)
23. *The type of a Typed  $\mu$ Scheme expression is well formed.* In Typed  $\mu$ Scheme, types like list and pair are well formed, with kinds  $* \Rightarrow *$  and  $* \times * \rightarrow *$  respectively, but they are not the types of any term: no expression can have type list or pair. The type of a term must have kind  $*$ . Using a metatheoretic argument about typing derivations in Typed  $\mu$ Scheme, prove that if there is a derivation of a typing judgment  $\Delta, \Gamma \vdash e : \tau$ , there is also a derivation of the judgment  $\Delta \vdash \tau :: *$ . Use structural induction on the derivation of the typing judgment. (For a similar exercise using Typed Impcore, see Exercise 22 on page 406.)
24. “*Type equivalence*” is an equivalence. Prove that  $\equiv$ , as defined in Section 6.6.6 on page 376, is an equivalence relation:
- (a) For any  $\tau$ ,  $\tau \equiv \tau$ .
  - (b) For any  $\tau$  and  $\tau'$ , if  $\tau \equiv \tau'$ , then  $\tau' \equiv \tau$ .
  - (c) For any  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$ , if  $\tau_1 \equiv \tau_2$  and  $\tau_2 \equiv \tau_3$ , then  $\tau_1 \equiv \tau_3$ .

In each case, structure your proof by assuming you have a derivation of the fact or facts assumed, and construct a derivation of the conclusion.

### 6.9.9 Metatheory about implementation

25. *Type-checking terminates.* Using an argument about the rules in the type system, prove that type checking for Typed Impcore always terminates.
26. *Type checking is sound.* Show that if an expression in Typed Impcore has a type, and if the values stored in the value environments  $\xi$ ,  $\phi$ , and  $\rho$  inhabit the types in the corresponding type environments, then the eval function never raises the exception BugInTypeChecking.

§6.9. Exercises

### 6.9.10 Capture-avoiding substitution

407

27. *Proof that substitution terminates.* Function tysubst on page 384 works by defining and calling an inner recursive function subst, which which tysubst is mutually recursive. We need to know that no matter what Typed  $\mu$ Scheme code a programmer writes, tysubst and subst terminate. Of particular concern is the recursive call in chunk 384c: given a FORALL type, the code makes a recursive call on a similar FORALL type. Could this process repeat forever?

Prove that tysubst terminates by showing that at every recursive call something is getting smaller. You might consider assigning each type a pair of numbers and show that the pair shrinks lexicographically. One number worth considering is the number of bound type variables that are in the range of the substitution.

28. *Renaming type variables.* Implement renaming to avoid variable capture: Given a type  $\forall \alpha_1, \dots, \alpha_n . \tau$  and a set of captured type variables  $C$ , rename as many  $\alpha_i$ 's as necessary to avoid conflicts with variables in  $C$  and with free variables of  $\tau$ . Do so in the body of function renameForallAvoiding, which is nested within function tysubst on page 384:

407. *(definition of renameForallAvoiding for Typed  $\mu$ Scheme [prototype] 407)* ≡

```
renameForallAvoiding : name list * tyex * name set -> tyex
fun renameForallAvoiding (alphas, tau, captured) =
  raise LeftAsExercise "renameForallAvoiding"
```

Calling renameForallAvoiding( $[\alpha_1, \dots, \alpha_n], \tau, C$ ) must choose variables  $\beta_i$  not in  $C$  and return a type  $\forall \beta_1, \dots, \beta_n . \tau'$  with these properties:

$$\begin{aligned} \forall \beta_1, \dots, \beta_n . \tau' &\equiv \forall \alpha_1, \dots, \alpha_n . \tau \\ \{\beta_1, \dots, \beta_n\} \cap C &= \emptyset \end{aligned}$$

For each  $\alpha_i$ , there are two cases:

- If  $\alpha_i \notin C$ , then it doesn't need to be renamed, and to maximize readability of the resulting type, let  $\beta_i = \alpha_i$ .
- If  $\alpha_i \in C$ , then  $\beta_i$  must be a new variable that does not appear in  $C$ , is not free in  $\tau$ , and is different from every  $\alpha_i$ .

LeftAsExercise  
S237a

To find a  $\beta_i$ , use function `freshName`, which returns a name based on  $\alpha$  but not in a given set of type variables.

408. (shared utility functions on sets of type variables 408)≡ (S394b)

```
fun freshName (alpha, avoid) = freshName : name * name set -> name
  let val basename = stripNumericSuffix alpha
      val candidates =
        streamMap (fn n => basename ^ "-" ^ intString n) naturals
        fun ok beta = not (member beta avoid)
      in  case streamGet (streamFilter ok candidates)
          of SOME (beta, _) => beta
          | NONE => let exception Can'tHappen in raise Can'tHappen end
      end
```

### 6.9.11 Holes in type systems

29. *Changing a thing's type can break the type system.* If a global variable or function is already defined, Typed Impcore doesn't let you write a new definition at a different type. This exercise shows why such definitions aren't permitted.
  - (a) Remove the restriction that a `val` binding may not change the type of the value bound. (An easy way to do this is to change the condition in chunk 350d so that it says "if true.")
  - (b) With the restriction removed, create a Typed Impcore program whose evaluation raises `BugInTypeChecking`, e.g., by adding 1 to an array.
  - (c) Restore the restriction on `val`, and remove the restriction that a `define` binding may not change the type of a function.
  - (d) With the restriction removed, create a Typed Impcore program whose evaluation raises `BugInTypeChecking`, e.g., by doing an array lookup on an integer.
30. *Reference cells with polymorphic contents are unsound.* In a polymorphic system, the `ref` constructor (Exercise 13) leads to unsoundness: it can be used to subvert the type system. The trick is to use `ref` operations at a *polymorphic* type. If you have a polymorphic mutable cell, you can instantiate the `ref` operation at one type and the `!` operation at a different type, enabling you to write a function that, for example, converts a Boolean to an integer.
  - (a) Using polymorphic references, trigger the `BugInTypeChecking` exception in Typed  $\mu$ Scheme by adding 1 to `#t`.
  - (b) Write an *identity* function of type `bool → int`
  - (c) Write a polymorphic function of type  $\forall \alpha, \beta . \alpha \rightarrow \beta$ .
  - (d) Close this dreadful loophole in the system by making `ref` abstract syntax and permitting it to be instantiated only at a monotype.
31. A *type-lambda may not abstract over a variable in the environment.* In Typed  $\mu$ Scheme, remove the restriction that a type-lambda may not abstract over a type variable that's free in the type environment. Now
  - (a) Write the `const` function of type  $\forall \alpha . \alpha \rightarrow \forall \beta . \beta \rightarrow \alpha$ .
  - (b) Write a similar function `unsafe-const` that uses  $\alpha$  in both places, instead of the two distinct  $\alpha$  and  $\beta$ . (Change the inner  $\beta$  to an  $\alpha$ .)

- (c) Use `unsafe-const` to introduce a variable that has type `int` but value `#t`.
- (d) Trigger `BugInTypeChecking` by adding one to this variable.
32. *Misuse of type-lambda can give any term any type.* The discussion of Typed  $\mu$ Scheme rule `TYLAMBDA` on page 374 observes that if the side conditions are not enforced, then
- $$\{\alpha :: *\}, \{x : \alpha\} \vdash \text{TYLAMBDA}(\alpha, x) : \forall \alpha . \alpha.$$

### \$6.9. Exercises

409

This observation suggests that by wrapping `TYLAMBDA` in code that introduces both  $\alpha$  and  $\beta$  into the kind environment and  $x$  into the type environment, then instantiating `TYLAMBDA` at  $\beta$ , one might well be able to define a function of type  $\forall \alpha, \beta . \alpha \rightarrow \beta$ .

- (a) Define such a function. Call it `cast`, or perhaps `Obj.magic`.
  - (b) To enable `cast` to be accepted by the interpreter, Remove the restriction that a type-lambda may not abstract over a type variable that's free in the type environment.
  - (c) Use `cast` to trigger `BugInTypeChecking`.
33. *Variable capture can break the type system.* In Typed  $\mu$ Scheme, change the code in chunk 384c so that substitution is always done naïvely, in a way that allows the capture of a  $\forall$ -bound variable. (It suffices to write `if true` instead of `if null actual_captures`.)
- (a) Define a typed, polymorphic version of function `flip-apply` from page 385, which should have type  $\forall \beta . \beta \rightarrow (\forall \alpha . (\beta \rightarrow \alpha) \rightarrow \alpha)$ .
  - (b) Using type-lambda and suitable instantiations of `flip-apply`, define a function `cast` that behaves like the identity function but has type  $\forall \alpha, \beta . \alpha \rightarrow \beta$ . This function is *not* acceptable; in a correct version of Typed  $\mu$ Scheme, it doesn't typecheck. But if you allow variable capture, you can write it.
  - (c) Use `cast` to introduce a variable that has type `int` but value `#t`.
  - (d) Trigger `BugInTypeChecking` by adding one to this variable.

<code>intString</code>	S238f
<code>member</code>	S240b
<code>naturals</code>	S252a
<code>streamFilter</code>	S253a
<code>streamGet</code>	S250b
<code>streamMap</code>	S252d
<code>stripNumericSuffix</code>	
	S240a

## CHAPTER CONTENTS

---

7.1	NANO-ML: A NEARLY APPLICATIVE LANGUAGE	412	7.6.1	Functions on types and type schemes	442
7.2	ABSTRACT SYNTAX AND VALUES	414	7.6.2	Type environments	445
7.3	OPERATIONAL SEMANTICS	415	7.6.3	Constraints and constraint solving	446
7.3.1	Rules for expressions	415	7.6.4	Type inference	448
7.3.2	Rules for evaluating definitions	417	7.6.5	Primitives	450
			7.6.6	Predefined functions	451
7.4	TYPE SYSTEM FOR NANO-ML	417	7.7	HINDLEY-MILNER AS IT REALLY IS	451
7.4.1	Types, type schemes, and type environments	418	7.8	SUMMARY	452
7.4.2	Simple type constructors	419	7.8.1	Key words and phrases	452
7.4.3	Substitution, instances, and instantiation	419	7.8.2	Further reading	454
7.4.4	Functions that compare, create, and print types	422	7.9	EXERCISES	454
7.4.5	Type rules for nano-ML	422	7.9.1	Retrieval practice	455
7.4.6	Nondeterministic typing, principal types, and type testing	427	7.9.2	Manipulating type inference	456
			7.9.3	Substitution	456
			7.9.4	Type schemes and principal types	457
7.5	FROM TYPE RULES TO TYPE INFERENCE	427	7.9.5	Typing rules and their properties	457
7.5.1	The method of explicit substitutions	429	7.9.6	Properties of constraints and constraint solving	458
7.5.2	The method of explicit constraints	430	7.9.7	Implementation of constraint solving and type inference	459
7.5.3	Solving constraints	438	7.9.8	Extending nano-ML	460
7.6	THE INTERPRETER	442	7.9.9	Improving the interpreter	461

## ML and type inference

*It soon appeared intolerable to have to declare—for example—a new maplist function for mapping a function over a list, every time a new type of list is to be treated. Even if the maplist function could possess what Strachey called “parametric polymorphism,” it also appeared intolerable to have to supply an appropriate type explicitly as a parameter, for each use of this function.*

Robin Milner, *How ML Evolved*

Chapter 6 presents two statically typed languages: Typed Impcore and Typed  $\mu$ Scheme. Typed Impcore is easy to program in and easy to write a type checker for, but because it is monomorphic, it can accommodate new type constructors and polymorphic operations only if its syntax and type checker are extended. Typed  $\mu$ Scheme is also easy to write a type checker for, and as a polymorphic language, it can accommodate new type constructors and polymorphic functions with no change to its syntax or its type checker, but it is difficult to program in: as Milner observed, supplying a type parameter at every use of every polymorphic value soon becomes intolerable. This chapter presents a language that offers the best of both worlds: nano-ML. Nano-ML is polymorphic and therefore easy to extend, but it is also easy to program in. This ease of use is delivered by a new typing algorithm: instead of type checking, nano-ML uses *type inference*.

A language with type inference doesn’t require explicit type annotations; instead, the types of variables and parameters are discovered by an algorithm. Type inference is used in such languages as Haskell, Miranda, OCaml, Standard ML, and TypeScript. Type inference forces us to give up some expressive power: the type-inference problem for a language as powerful as Typed  $\mu$ Scheme is undecidable. To make type inference decidable, we restrict polymorphism. The most popular restriction is the *Hindley-Milner type system*. In this type system, a quantified  $\forall$  type may appear only at top level; a  $\forall$  type may *not* be passed to a type constructor. In particular, a  $\forall$  type may not appear as an argument in a function type.

I present the Hindley-Milner type system and its type inference in the context of nano-ML, a language that is closely related to Typed  $\mu$ Scheme.

- Like Typed  $\mu$ Scheme, nano-ML has first-class, higher-order functions, and its values are S-expressions.
- Like Typed  $\mu$ Scheme, nano-ML has polymorphic types that are checked at compile time.
- Unlike Typed  $\mu$ Scheme, nano-ML has *implicit* types. In nano-ML, the programmer never writes a type or a type constructor.

- Unlike Typed  $\mu$ Scheme, nano-ML has no mutation. Nano-ML lacks `set`, and its names stand for values, not for mutable locations. Because there is no mutation, nano-ML programs are nearly always written in *applicative* style. Only input/output is done imperatively, with `println`, `print`, and `printu`.
- Unlike Typed  $\mu$ Scheme, nano-ML restricts polymorphism: quantified types appear only at top level. In exchange for the restriction, we get a benefit: nano-ML automatically instantiates polymorphic values, and it also automatically introduces polymorphic types. A nano-ML programmer never has to write @ or `type-lambda`.

Nano-ML, Typed  $\mu$ Scheme, and  $\mu$ Scheme are closely related. If we erase the types from a Typed  $\mu$ Scheme program, we get a valid  $\mu$ Scheme program. If the program does not use `set` or `while`, and if it uses `type-lambda` appropriately, it is also a valid nano-ML program.

Like our interpreter for Typed  $\mu$ Scheme, our interpreter for nano-ML is based on the  $\mu$ Scheme interpreter from Chapter 5. As with the type checker from Chapter 6, I present the rules for type inference but leave substantial parts of the implementation as Exercises.

## 7.1 NANO-ML: A NEARLY APPLICATIVE LANGUAGE

Aside from the type system, nano-ML differs from  $\mu$ Scheme by forbidding mutation.<sup>1</sup> Mutation is the archetypal example of an *imperative feature*. Although nano-ML does not have mutation, it does have other imperative features: printing primitives, `error`, and `begin` (see sidebar).

Nano-ML and  $\mu$ Scheme have subtly different definition forms. In  $\mu$ Scheme, a `val` definition can mutate an existing binding, but in nano-ML, `val` always creates a new binding. To enable the definition of recursive functions, nano-ML uses a `val-rec` definition form like Typed  $\mu$ Scheme's `val-rec`. And like Typed  $\mu$ Scheme, nano-ML uses `define` as syntactic sugar for a combination of `val-rec` and `lambda`.

Nano-ML needs fewer primitives than  $\mu$ Scheme. Because nano-ML has a type system, we know at compile time what is a symbol, a number, and a function, so nano-ML doesn't need type predicates `symbol?`, `number?`, `boolean?`, or `function?`. Nano-ML does need the `null?` predicate, which is used to tell the difference between empty and non-empty lists, but it does not also need `pair?`.

Except for the addition of `val-rec`, the concrete syntax of nano-ML, which is shown in Figure 7.1 on page 414, is mostly a subset of that of  $\mu$ Scheme. But because nano-ML is a typed language like Typed Impcore and Typed  $\mu$ Scheme, it also gets the `check-type` and `check-type-error` unit tests. And because it has type inference, it also gets a new test, `check-principal-type`.

The `check-type` test serves the same role as the corresponding test in Typed  $\mu$ Scheme, but as explained in Section 7.4.6, it is more permissive. To test for type equivalence in nano-ML, use `check-principal-type`.

**412.** *(transcript 412)≡* 419b▷

```
-> (check-principal-type revapp
    (forall ['a] ((list 'a) (list 'a) -> (list 'a)))
    -> (define revapp (xs ys)
        (if (null? xs)
            ys
            (revapp (cdr xs) (cons (car xs) ys))))
```

Function `revapp` is written exactly as we wrote it in  $\mu$ Scheme.

---

<sup>1</sup>To add mutation soundly requires some subtlety in the type system, and in this chapter we are going for simplicity. Mutation is relegated to Exercise 23.

### Sidebar: Imperative features

The early chapters of this book show both procedural and functional languages. The procedural language, Impcore, has only second-class functions, and as in C, most programs will probably use `set`, `while`, and `begin`. The functional languages,  $\mu$ Scheme and Typed  $\mu$ Scheme, share these imperative features with Impcore, but they emphasize first-class functions. Nano-ML shares first-class functions with  $\mu$ Scheme and Typed  $\mu$ Scheme, but it has fewer imperative features.

What are imperative features, and why do we care? A feature is imperative if *when the feature is used, different orders of evaluation can produce different results*.<sup>a</sup>

- The `set` construct, also called assignment or mutation, is an imperative feature: if two expressions assign to the same location, the order of evaluation matters. In particular, after two assignments, the mutable location holds the value written by the second assignment.
- Input/output is an imperative feature: when printing, order matters. If different `print` expressions are evaluated in different orders, the program's output is different. Similarly, if a program reads  $x$  and  $y$  from its input, it may produce different results depending on the order in which the variables are read.
- Exceptions are an imperative feature: if different expressions raise different exceptions, order matters. Which exception is raised depends on which expression is evaluated first. In  $\mu$ Scheme, `error` is a similar imperative feature, because the error message depends on the order of evaluation.

Languages with imperative features usually have other features that are not themselves imperative but are related.

- Sequencing with `begin` is not an imperative feature, but it enables programmers to control order of evaluation. In languages such as C, C++, and OCaml, which have imperative features but do not say in what order a function's arguments are evaluated, sequencing is essential; without it, some programs' behavior would be impossible to predict.
- Loops are useful only in the presence of imperative features. Without imperative features, for example, a `while` expression has only uninteresting outcomes: nontermination, immediate termination, or a run-time error.

Why should we care about imperative features? For two reasons:

- When imperative features are absent, we can reason about programs more easily, especially using algebraic laws. We can more easily build correct programs, and we can more easily transform programs to make them more efficient.
- When imperative features are absent or restricted, a compiler can often produce better code. In ML-like languages, restrictions on mutation have been used to reduce memory requirements, improve instruction scheduling, and reduce the overhead of garbage collection.

car	P
cdr	P
cons	P
null?	P

A language containing no imperative features may be called *pure*; a language containing imperative features may be called *impure*.

<sup>a</sup>For purposes of deciding whether a feature is imperative, we don't consider nontermination to be a "different result." (A reordering that makes a program terminate more often is typically benign.)

```

def      ::= (val variable-name exp)
          | (val-rec variable-name exp)
          | exp
          | (define function-name (formals) exp)
          | (use file-name)
          | unit-test

unit-test ::= (check-expect exp exp)
           | (check-assert exp)
           | (check-error exp)
           | (check-type exp type-exp)
           | (check-principal-type exp type-exp)
           | (check-type-error exp)

exp      ::= literal
          | variable-name
          | (if exp exp exp)
          | (begin {exp})
          | (exp {exp})
          | (let-keyword ({(variable-name exp)}) exp)
          | (lambda (formals) exp)

let-keyword ::= let | let* | letrec

type-exp  ::= type-constructor-name
          | 'type-variable-name
          | (forall [{'type-variable-name}] type-exp)
          | ({type-exp} -> type-exp)
          | (type-exp {type-exp})

formals   ::= {variable-name}

literal   ::= numeral | #t | #f | 'S-exp | (quote S-exp)

S-exp     ::= literal | symbol-name | ({S-exp})

numeral   ::= sequence of digits, possibly prefixed with a minus sign

*-name    ::= sequence of characters not a numeral and not containing
              (,), [,], {, }, ;, or whitespace

```

Figure 7.1: The concrete syntax of nano-ML

## 7.2 ABSTRACT SYNTAX AND VALUES OF NANO-ML

Nano-ML's abstract syntax is the same as  $\mu$ Scheme's, minus WHILEX and SET. As in Chapter 5, I define it by presenting my implementation.

**414.** *(definitions of exp and value for nano-ML 414)*  $\equiv$

(S405c)

```

datatype exp = LITERAL of value
             | VAR      of name
             | IFX      of exp * exp * exp
             | BEGIN    of exp list
             | APPLY    of exp * exp list
             | LETX    of let_kind * (name * exp) list * exp
             | LAMBDA  of name list * exp
and let_kind = LET | LETREC | LETSTAR
and (definition of value for nano-ML 415b)

```

The BEGIN form is intended for use with primitive functions `println` and `print`.

Except for VALREC, definitions are as in  $\mu$ Scheme.

**415a.** *(definition of def for nano-ML 415a)*  $\equiv$

(S405c)

```
datatype def = VAL      of name * exp
             | VALREC of name * exp
             | EXP      of exp
             | DEFINE of name * (name list * exp)
```

In the operational semantics nano-ML and  $\mu$ Scheme have the same values, and their representations are similar enough that I can reuse the projection, embedding, and printing functions from Chapter 5.

**415b.** *(definition of value for nano-ML 415b)*  $\equiv$

(414)

```
value = SYM      of name
          | NUM      of int
          | BOOLV    of bool
          | NIL
          | PAIR     of value * value
          | CLOSURE  of lambda * (unit -> value env)
          | PRIMITIVE of primop
withtype primop = value list -> value (* raises RuntimeError *)
and lambda = name list * exp
```

What's changed is the representations of closures.

If a recursive function names itself, the function's closure must contain an environment that refers to the closure itself. In  $\mu$ Scheme, the environment maps the function's name to a mutable cell, and after the closure is created, the cell is updated to hold the closure. In nano-ML, an environment maps each name to a value, and at the time the closure's environment might be created, the closure itself doesn't yet exist, so it can't be in the environment. Instead, I build each closure's environment *lazily*; that is, instead of storing an environment in the closure, I store a function that produces an environment on demand. The laziness ensures that the environment need not be produced until after the closure is created.

### 7.3 OPERATIONAL SEMANTICS

Because nano-ML doesn't have mutation, and because the effects of the imperative primitives aren't specified formally, nano-ML's operational semantics is simple. The abstract machine has no locations and no store; evaluating an expression produces a value, period. The judgment is  $\langle e, \rho \rangle \Downarrow v$ . The environment  $\rho$  maps a name to a value, not to a mutable location as in  $\mu$ Scheme. And evaluating a definition produces a new environment; the form of that judgment is  $\langle d, \rho \rangle \rightarrow \rho'$ .

#### 7.3.1 Rules for expressions

type env	310b
type name	310a

I hope that most of the rules are self-explanatory.

$$\overline{\langle \text{LITERAL}(v), \rho \rangle \Downarrow v} \quad (\text{LITERAL})$$

$$\frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \rho \rangle \Downarrow \rho(x)} \quad (\text{VAR})$$

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \quad v_1 \neq \text{BOOLV}(\#f) \quad \langle e_2, \rho \rangle \Downarrow v_2}{\langle \text{IF}(e_1, e_2, e_3), \rho \rangle \Downarrow v_2} \quad (\text{IFTRUE})$$

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \quad v_1 = \text{BOOLV}(\#f) \quad \langle e_3, \rho \rangle \Downarrow v_3}{\langle \text{IF}(e_1, e_2, e_3), \rho \rangle \Downarrow v_3} \quad (\text{IFFALSE})$$

The rules for BEGIN are a cheat; the purpose of BEGIN is to force order of evaluation, but our rules are so simplified that they don't enforce an order of evaluation.

$$\frac{}{\langle \text{BEGIN}(), \rho \rangle \Downarrow \text{NIL}} \quad (\text{EMPTYBEGIN})$$

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \quad \langle e_2, \rho \rangle \Downarrow v_2 \quad \dots \quad \langle e_n, \rho \rangle \Downarrow v_n}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \rho \rangle \Downarrow v_n} \quad (\text{BEGIN})$$

Just as in  $\mu$ Scheme, LAMBDA captures an environment in a closure, and APPLY uses the captured environment. Because nano-ML does not store actual parameters in mutable locations, its rules are simpler than  $\mu$ Scheme's rules.

$$\frac{}{\langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho \rangle} \quad (\text{MKCLOSURE})$$

$$\frac{\langle e, \rho \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e_c), \rho_c \rangle \quad \langle e_i, \rho \rangle \Downarrow v_i, \quad 1 \leq i \leq n \quad \langle e_c, \rho_c \{ x_1 \mapsto v_1, \dots, x_n \mapsto v_n \} \rangle \Downarrow v}{\langle \text{APPLY}(e, e_1, \dots, e_n), \rho \rangle \Downarrow v} \quad (\text{APPLYCLOSURE})$$

The semantic rule for applying a nano-ML primitive is to apply the function attached to that primitive. The implementation is equally simple.

$$\frac{\langle e, \rho \rangle \Downarrow \text{PRIMITIVE}(f) \quad \langle e_i, \rho \rangle \Downarrow v_i, \quad 1 \leq i \leq n \quad f(v_1, \dots, v_n) = v}{\langle \text{APPLY}(e, e_1, \dots, e_n), \rho \rangle \Downarrow v} \quad (\text{APPLYPRIMITIVE})$$

Because a LET-bound name stands for a value, not a location, rules for LET forms are also simplified.

$$\frac{\langle e_i, \rho \rangle \Downarrow v_i, \quad 1 \leq i \leq n \quad \langle e, \rho \{ x_1 \mapsto v_1, \dots, x_n \mapsto v_n \} \rangle \Downarrow v}{\langle \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho \rangle \Downarrow v} \quad (\text{LET})$$

As in  $\mu$ Scheme, a LETSTAR expression requires a sequence of environments.

$$\frac{\langle e_1, \rho_0 \rangle \Downarrow v_1 \quad \rho_1 = \rho_0 \{ x_1 \mapsto v_1 \} \quad \vdots \quad \langle e, \rho_{n-1} \rangle \Downarrow v_n \quad \rho_n = \rho_{n-1} \{ x_n \mapsto v_n \} \quad \langle e, \rho_n \rangle \Downarrow v}{\langle \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho_0 \rangle \Downarrow v} \quad (\text{LETSTAR})$$

LETREC is the tricky one. The expressions have to be evaluated in an environment in which their names are already bound to the resulting values. In other words, to evaluate each  $e_i$ , we have to have  $\rho'$ , but to build  $\rho'$ , we have to know  $v_i$ . It seems like it should be impossible to make progress, but because the expressions are all lambda abstractions, we can pull it off.

$$\frac{\rho' = \rho \{ x_1 \mapsto v_1, \dots, x_n \mapsto v_n \} \quad \langle e_1, \rho' \rangle \Downarrow v_1 \quad \dots \quad \langle e_n, \rho' \rangle \Downarrow v_n \quad \langle e, \rho' \rangle \Downarrow v}{\langle \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e), \rho \rangle \Downarrow v} \quad (\text{LETREC})$$

Because each  $e_i$  is a LAMBDA, we know what value it's going to produce: a closure that captures the current environment and the body of the LAMBDA. And in eval, that makes it possible to build  $\rho'$  without calling eval recursively (chunk S407c). The resulting  $\rho'$  satisfies the equations in the premises.

### 7.3.2 Rules for evaluating definitions

In  $\mu$ Scheme or Typed  $\mu$ Scheme, evaluating a definition can change the store. In nano-ML, because there is no mutation, a definition is evaluated primarily for its effect on the environment.<sup>2</sup> We therefore write the judgment in the form  $\langle d, \rho \rangle \rightarrow \rho'$ .

Nano-ML's definitions differ from  $\mu$ Scheme's in several significant ways:

- In nano-ML, as in full ML and in Typed  $\mu$ Scheme, a VAL definition never mutates a previous binding; it always adds a new binding. (See Exercise 46 on page 196 in Chapter 2.) If existing functions or other values depend on the old binding, those functions refer to the old binding, not the new one. If you are using an interactive interpreter, and you change one binding but you don't re-enter the definitions of the functions that depend on the old binding, you may be baffled by the subsequent behavior of the interpreter. This problem trips many programmers new to ML, but after experience, most ML programmers view VAL's behavior as an important feature.

$$\frac{\langle e, \rho \rangle \Downarrow v}{\langle \text{VAL}(x, e), \rho \rangle \rightarrow \rho \{x \mapsto v\}} \quad (\text{VAL})$$

- Like Typed  $\mu$ Scheme but unlike  $\mu$ Scheme, nano-ML has VAL-REC. The semantics requires a  $\rho'$  that binds  $f$  to a closure containing  $\rho'$ .

$$\frac{\rho' = \rho \{f \mapsto (\text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e), \rho')\}}{\langle \text{VAL-REC}(f, \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e)), \rho \rangle \rightarrow \rho'} \quad (\text{VALREC})$$

The implementation supports this recursion using the same tricks used to support recursion in LAMBDA and LETREC.

- In nano-ML, as in Typed  $\mu$ Scheme, the definition DEFINE( $f, a, e$ ) is syntactic sugar for VAL-REC( $f, \text{LAMBDA}(a, e)$ ).

$$\frac{\langle \text{VAL-REC}(f, \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e)), \rho \rangle \rightarrow \rho'}{\langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \rho \rangle \rightarrow \rho'} \quad (\text{DEFINE})$$

The rule for expressions is just like the rule in  $\mu$ Scheme; evaluating a top-level expression  $e$  is syntactic sugar for evaluating a binding to it.

$$\frac{\langle \text{VAL(it}, e), \rho \rangle \rightarrow \rho'}{\langle \text{EXP}(e), \rho \rangle \rightarrow \rho'} \quad (\text{EXP})$$

## 7.4 TYPE SYSTEM FOR NANO-ML

Just as in Typed Impcore and Typed  $\mu$ Scheme, the type system of nano-ML determines which terms have types, and the implementation accepts a definition only if its terms have types. As before, the types of terms are specified by a formal proof system. Before getting into the proof rules, I discuss the elements of the system.

---

<sup>2</sup>Evaluating a definition may also print.

### 7.4.1 Types, type schemes, and type environments

Just as in Typed  $\mu$ Scheme, we build types using four elements:

- Type variables, which we write using  $\alpha$
- Type constructors, which we write generically using  $\mu$  or specifically using a name such as `int` or `list`
- Constructor application, which we write using the ML notation  $(\tau_1, \dots, \tau_n) \tau$
- Quantification, which we write using  $\forall$

In nano-ML, unlike in Typed  $\mu$ Scheme, quantified types are restricted: a type quantified with  $\forall$  may appear only at top level, never as an argument to a type constructor. In nano-ML, this restriction is built in to the *syntax* of types:

- A type built with type variables, type constructors, and constructor application is written using the metavariable  $\tau$ :

$$\tau ::= \alpha \mid \mu \mid (\tau_1, \dots, \tau_n) \tau$$

A  $\tau$  is called a *type*.

- A quantified type is written using the metavariable  $\sigma$ :

$$\sigma ::= \forall \alpha_1, \dots, \alpha_n . \tau$$

A  $\sigma$  is called a *type scheme*.

In the code, a  $\tau$  is represented by a `ty` and a  $\sigma$  by a `type_scheme`:<sup>3</sup>

**418.** *(representation of Hindley-Milner types 418)* (S408c)

```
type tyvar = name
datatype ty = TYVAR of tyvar          (* type variable alpha *)
            | TYCON of tycon          (* type constructor mu *)
            | CONAPP of ty * ty list   (* type-level application *)
```

`datatype type_scheme = FORALL of tyvar list * ty`

The set of type schemes  $\sigma$  in which the  $\alpha_1, \dots, \alpha_n$  is empty is isomorphic to the set of types  $\tau$ . The isomorphism relates each type  $\tau$  to the type scheme `FORALL([], tau)`. A type without  $\forall$  or a type scheme in which the  $\alpha_1, \dots, \alpha_n$  is empty is sometimes called a *monotype* or *ground type*. A type scheme in which the  $\alpha_1, \dots, \alpha_n$  is not empty is sometimes called a *polytype*.

When I write a type application in code, I put the type constructor *before* its arguments, as in

`CONAPP(TYCON "list", [TYCON "int"]).`

But in the text, I put the type constructor *after* its arguments, as in `int list`. This notation is consistent with ML notation.

In nano-ML, all type constructors are predefined; no program can add new ones. The ability to add new type constructors is found in the more advanced bridge languages  $\mu$ ML (Chapter 8) and Molecule (Chapter 9). In nano-ML, predefined constructors `args` and `function` appear in the typing rules for functions. Other constructors, like `bool`, `int`, `sym`, and so on, give types to literals or primitives.

---

<sup>3</sup>Nano-ML's representation has only four of the five forms found in Typed  $\mu$ Scheme. The fifth form, `FUNTY`, is represented in nano-ML as a nested application of type constructors `function` and `args` (chunk 422c). Coding function types in this way simplifies type inference.

To write types made with `args` and `function`, I use the ML abbreviations:

Type	Abbreviation
$(\tau_1, \tau_2)$ function	$\tau_1 \rightarrow \tau_2$
$(\tau_1, \dots, \tau_n)$ args	$\tau_1 \times \tau_2 \cdots \times \tau_n$

In nano-ML, as in Typed  $\mu$ Scheme, a type environment is written using the Greek letter  $\Gamma$ . In nano-ML, a type environment  $\Gamma$  maps a term variable<sup>4</sup> to a type scheme. Type environments are used only during type inference, not at run time.

In nano-ML, unlike in Typed  $\mu$ Scheme, types don't appear in code. And types inferred by the system are guaranteed to be well formed, so we don't have to specify formation rules, and we don't need a system of kinds. (A type may be written in a unit test, but if such a type is ill formed, the test just fails; no special checking is needed.) Kinds are used again in  $\mu$ ML (Chapter 8) and in full ML.

#### 7.4.2 Simple type constructors

Because all type constructors are predefined, we can represent a type constructor simply by its name. Because type names cannot be redefined, a name like `int` always means “integer,” and two type constructors are the same if and only if they have the same name.

**419a.** (definitions of `tycon`, `eqTycon`, and `tyconString` for named type constructors 419a)  $\equiv$  (S408c)

<pre>type tycon = name fun eqTycon (mu, mu') = mu = mu' fun tyconString mu = mu</pre>	<pre>type tycon eqTycon : tycon * tycon -&gt; bool tyconString : tycon -&gt; string</pre>
---	---

#### 7.4.3 Substitution, instances, and instantiation

The great virtue of the nano-ML type system is that *the system automatically instantiates polymorphic values*. As an example, we define `empty-list` with type  $\forall \alpha. \alpha \text{ list}$ , then instantiate it at types `int` and `sym` list:

**419b.** (transcript 412)  $\vdash$  ▷ 412 424 ▷

```
-> (val empty-list '())
() : (forall ['a] (list 'a))
-> (val p (pair (cons 1 empty-list) (cons '(a b c) empty-list)))
(PAIR (1) ((a b c))) : (pair (list int) (list (list sym)))
```

In Typed  $\mu$ Scheme, we would have to instantiate `empty-list` explicitly, writing `(@ empty-list int)` and `(@ empty-list (list sym))`. We would also have to instantiate `pair` and `cons` explicitly.<sup>5</sup> An explicit instantiation or type application substitutes types written by the programmer for the quantified type variables of a polymorphic type. In ML, the implementation figures out what types to substitute. Milner's type inference figures out the types by computing an appropriate *substitution*. Because substitutions play a much greater role in ML than in Typed  $\mu$ Scheme, we represent them explicitly.

cons	$\mathcal{P}$
type name	310a
pair	$\mathcal{P}$
type tycon	497b

A substitution is a finite map from type variables to types; one is written using the Greek letter  $\theta$  (pronounced “THAYT-uh”). A  $\theta$  has many interpretations:

- As a function from type variables to types
- As a function from types to types
- As a function from type schemes to type schemes

<sup>4</sup>Term variables, which appear in terms (expressions) and are bound by `let` or `lambda`, stand for values. Don't confuse them with type variables, which stand for types. The name of a term variable begins with a letter or symbol; the name of a type variable begins with the ASCII quote ('') character.

<sup>5</sup>These instantiations are what Professor Milner found intolerable.

- As a function from type environments to type environments
- As a function from type-equality constraints to type-equality constraints
- As a function from typing judgments to typing judgments
- As a function from typing derivations to typing derivations

These interpretations are all related and mutually consistent. They all appear in the math, and some appear in my code. The interpretation I use most is the function from types to types. Such a function  $\theta$  is a substitution if it preserves type constructors and constructor application:

- For any type constructor  $\mu$ ,  $\theta\mu = \mu$ .
- For any constructor application  $\text{CONAPP}(\tau, \langle \tau_1, \dots, \tau_n \rangle)$ ,

$$\theta(\text{CONAPP}(\tau, \langle \tau_1, \dots, \tau_n \rangle)) = \text{CONAPP}(\theta\tau, \langle \theta\tau_1, \dots, \theta\tau_n \rangle).$$

Or, using informal ML-like notation,

$$\theta((\tau_1, \dots, \tau_n) \tau) = (\theta\tau_1, \dots, \theta\tau_n) (\theta\tau).$$

In the common case where the  $\tau$  being applied is a simple constructor  $\mu$ ,  $\theta\mu = \mu$  and

$$\theta((\tau_1, \dots, \tau_n) \mu) = (\theta\tau_1, \dots, \theta\tau_n) \mu.$$

To be a substitution, a function from types to types must meet one other condition:

- The set  $\{\alpha \mid \theta\alpha \neq \alpha\}$  must be *finite*. This set is the set of variables substituted for. It is called the *domain* of the substitution, and it is written  $\text{dom } \theta$ .

Such a function is defined by `tysubst` in Chapter 6 on page 380. You can verify that the inner function `subst` has all the properties claimed above (Exercise 3).

Substitution determines when  $\tau'$  is an *instance* of  $\tau$ : like Milner (1978), we write  $\tau' \leqslant \tau$  if and only if there exists a substitution  $\theta$  such that  $\tau' = \theta\tau$ . The instance relation  $\tau' \leqslant \tau$  is pronounced in two ways: not only “ $\tau'$  is an *instance* of  $\tau$ ” but also “ $\tau$  is at least as general as  $\tau'$ .”

To extend the instance relation to type schemes, we relate  $\tau' \leqslant \forall \alpha_1, \dots, \alpha_n . \tau$  if and only if there exists a substitution  $\theta$  such that  $\text{dom } \theta \subseteq \{\alpha_1, \dots, \alpha_n\}$  and  $\theta\tau = \tau'$ . The first condition says that the instantiating substitution  $\theta$  may substitute only for type variables that are bound by the  $\forall$ .

To *instantiate* a type scheme  $\sigma = \forall \alpha_1, \dots, \alpha_n . \tau$  is to choose a  $\tau' \leqslant \sigma$ . When we instantiate this type scheme, we may substitute for the type variables  $\alpha_1, \dots, \alpha_n$  and *only* for those variables. It’s like instantiation in Typed  $\mu$ Scheme, except in ML, the system instantiates each  $\sigma$  automatically.

Of all the ways we can interpret a substitution, the one I use to *represent* a substitution is a finite map from type variables to types. In my code, I have a standard representation for such a map: an environment of type `ty env`. To interpret a substitution as a function from type variables to types, we apply `vars subst` to it:

**420.** *(shared utility functions on Hindley-Milner types 420)  $\equiv$*

*(S408c) 421a▷*

<pre>type subst = ty env fun vars subst theta =   (fn a =&gt; find (a, theta) handle NotFound _ =&gt; TYVAR a)</pre>	<pre>type subst vars subst : subst -&gt; (name -&gt; ty)</pre>
--	--

As the code shows, the function defined by a substitution is *total*. If type variable  $a$  is not in the domain of the substitution, the function leaves  $a$  unchanged.

A substitution is most often interpreted as a function from types to types. That interpretation is provided by function `tysubst`. The code is almost the same as the code on page 384 in Chapter 6—and there are no pesky quantified types to deal with.

**421a.** *(shared utility functions on Hindley-Milner types 420)* +≡ (S408c) ▷ 420 421b ▷

```
fun tysubst theta =
  let fun subst (TYVAR a) = vars subst theta a
      | subst (TYCON c) = TYCON c
      | subst (CONAPP (tau, taus)) = CONAPP (subst tau, map subst taus)
    in subst
  end
```

$\begin{array}{l} \text{tysubst : subst} \rightarrow (\text{ty} \rightarrow \text{ty}) \\ \text{subst : ty} \rightarrow \text{ty} \end{array}$

§7.4  
Type system for  
*nano-ML*

---

421

A function produced by `tysubst` has type  $\text{ty} \rightarrow \text{ty}$  and so can be composed with any other function of the same type, including all functions that correspond to substitutions. To be precise, if  $\theta_1$  and  $\theta_2$  are substitutions, then the composition  $\text{tysubst } \theta_2 \circ \text{tysubst } \theta_1$  is a function from types to types (and also corresponds to a substitution). Composition is really useful, but a substitution *data structure*  $\theta$  is strictly more useful than the corresponding *function*  $\text{tysubst } \theta$ . For one thing, we can interrogate  $\theta$  about its domain. To have the best of both worlds, I define a function for composing substitutions, which obeys the algebraic law

$$\text{tysubst}(\text{compose}(\theta_2, \theta_1)) = \text{tysubst } \theta_2 \circ \text{tysubst } \theta_1.$$

Function `dom` produces a substitution's domain.

**421b.** *(shared utility functions on Hindley-Milner types 420)* +≡ (S408c) ▷ 421a 421c ▷

```
fun dom theta =
  map (fn (a, _) => a) theta
fun compose (theta2, theta1) =
  let val domain = union (dom theta2, dom theta1)
    val replace = tysubst theta2 o vars subst theta1
  in map (fn a => (a, replace a)) domain
  end
```

$\begin{array}{l} \text{dom : subst} \rightarrow \text{name set} \\ \text{compose : subst} * \text{subst} \rightarrow \text{subst} \end{array}$

bind	312b
bindList	312c
BindListLength	312c
BugInTypeInference	S237c
CONAPP	418
emptyEnv	311a
type env	310b
find	311b
FORALL	418
NotFound	311b
type ty	418
TYCON	418
TYVAR	418
union	S240b

Instantiation is just as in Chapter 6, except no kind environment is needed. Because the system ensures everything has the right kind, it is an internal error to instantiate with the wrong number of arguments. The internal error is signalled by raising the exception `BugInTypeInference`. This exception is raised only when there is a fault in the interpreter; a faulty nano-ML program should never trigger it.

**421c.** *(shared utility functions on Hindley-Milner types 420)* +≡ (S408c) ▷ 421b 421d ▷

```
fun instantiate (FORALL (formals, tau), actuals) =
  tysubst (bindList (formals, actuals, emptyEnv)) tau
  handle BindListLength =>
    raise BugInTypeInference "number of types in instantiation"
```

$\text{instantiate : type\_scheme} * \text{ty list} \rightarrow \text{ty}$

Before we can apply or compose substitutions, we must create them. The simplest nontrivial substitution is one that substitutes a single type for a single variable. To make it easy to create such a substitution, I define a new infix operator `|-->`. The expression `alpha |--> tau` is the substitution that substitutes  $\tau$  for  $\alpha$ . In math, that substitution is written  $(\alpha \mapsto \tau)$ .

**421d.** *(shared utility functions on Hindley-Milner types 420)* +≡ (S408c) ▷ 421c 422a ▷

```
infix 7 |-->
fun a |--> (TYVAR a') = if a = a' then emptyEnv
                           else bind (a, TYVAR a', emptyEnv)
  | a |--> tau        = bind (a, tau, emptyEnv)
```

$\text{|--> : name} * \text{ty} \rightarrow \text{subst}$

The  $\|-\>$  function accepts any combination of  $\alpha$  and  $\tau$ . But if  $\alpha$  appears free in  $\tau$ , for example if  $\tau = \alpha$  list, then the resulting substitution  $\theta$  is not *idempotent*. If  $\theta$  is not idempotent, then  $\theta \circ \theta \neq \theta$ , and moreover,  $\theta\alpha \neq \theta\tau$ . But type inference is all about using substitutions to guarantee equality of types, and we must be sure that every substitution we create is idempotent, so if  $\theta = (\alpha \mapsto \tau)$ , then  $\theta\alpha = \theta\tau$ . If this equality does not hold, there is a bug in type inference. Detecting this bug is the subject of Exercise 2 on page 456.

A final useful substitution is the identity substitution, which is represented by an empty environment.

**422a.** *(shared utility functions on Hindley-Milner types 420)*  $\|-\>$   $\equiv$

```
val idsubst = emptyEnv
```

(S408c) ▷ 421d 422b ▷

idsubst : subst
-----------------

In math, the identity substitution is written  $\theta_I$ , and it is a left and right identity of composition:  $\theta_I \circ \theta = \theta = \theta \circ \theta_I = \theta$ .

My representation of substitutions is simple but not efficient. Industrial implementations of type inference represent each type variable as a mutable cell, and they apply and compose substitutions by mutating those cells.

#### 7.4.4 Functions that compare, create, and print types

Because a Hindley-Milner type contains no quantifiers, the definition of type equivalence is simpler than in Typed  $\mu$ Scheme (chunk 379a).

**422b.** *(shared utility functions on Hindley-Milner types 420)*  $\|-\>$   $\equiv$

```
fun eqType (TYVAR a, TYVAR a') = a = a'
| eqType (TYCON c, TYCON c') = eqTycon (c, c')
| eqType (CONAPP (tau, taus), CONAPP (tau', taus')) =
  eqType (tau, tau') andalso eqTypes (taus, taus')
| eqType _ = false
and eqTypes (taus, taus') = ListPair.allEq eqType (taus, taus')
```

(S408c) ▷ 422a 444a ▷

eqType : ty * ty -> bool
--------------------------

To make it easy to write the types of primitive operations, I provide convenience functions very much like those from Chapter 6.

**422c.** *(functions that create or compare Hindley-Milner types with named type constructors 422c)*  $\equiv$

```
val inttype = TYCON "int"
val booleatype = TYCON "bool"
val symtype = TYCON "sym"
val alpha = TYVAR "a"
val beta = TYVAR "b"
val unittype = TYCON "unit"
fun listtype ty =
  CONAPP (TYCON "list", [ty])
fun pairtype (x, y) =
  CONAPP (TYCON "pair", [x, y])
fun funtype (args, result) =
  CONAPP (TYCON "function", [CONAPP (TYCON "arguments", args), result])
fun asFuntype (CONAPP (TYCON "function",
  [CONAPP (TYCON "arguments", args), result])) =
  SOME (args, result)
| asFuntype _ = NONE
```

inttype : ty
booleatype : ty
symtype : ty
alpha : ty
beta : ty
unittype : ty
listtype : ty -> ty
pairtype : ty * ty -> ty
funtype : ty list * ty -> ty
asFuntype : ty -> (ty list * ty) option

(S408c)

To make it possible to print types and substitutions, Appendix R defines functions `typeString`, `typeSchemeString`, and `substString`.

#### 7.4.5 Type rules for nano-ML

In Typed  $\mu$ Scheme, you may define and use quantified types anywhere, but you have to write `type-lambda` and `@` everywhere. Chapter 6 shows just how un-

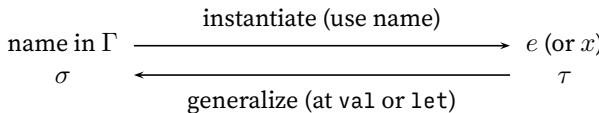


Figure 7.2: Relationship between type schemes and types

§7.4

Type system for  
nano-ML

423

pleasant it is to write these type abstractions and type applications explicitly. ML makes polymorphism lightweight and efficient by eliminating explicit type abstraction and type application; the price is that ML can express fewer types than Typed  $\mu$ Scheme. In ML, the  $\forall$  quantifier appears only in a type scheme, never in a type, and a function cannot expect an argument of quantified type.

To fill in the missing type abstractions and instantiations, nano-ML’s type system does more work than a type checker for Typed  $\mu$ Scheme:

- When a function is defined, the code doesn’t show the types of its arguments; the type system has to figure out a type for each argument.
- When a value is polymorphic, the code doesn’t show how to instantiate it; the type system has to figure out a type at which the value should be instantiated.

The types can be figured out because in the Hindley-Milner type system,  $\forall$  cannot appear just anywhere. In particular, it cannot appear in the type of an expression: *every well-typed expression has a monotype*, although the monotype may have free type variables. *Only a name bound in the environment may have a polytype*. When we take a name from  $\Gamma$  and use it as an expression, we *instantiate* its type scheme to give it a monotype. This instantiation amounts to an implicit @ operation. When we process a `val` or `let` binding, we take the type of the expression and *generalize* it to make a type scheme, which we then put into the environment. This generalization amounts to an implicit type-lambda operation. To visualize what’s going on, see Figure 7.2. (There’s also a sidebar on page 445.)

When we instantiate a type, how do we know what types to substitute for the type variables? When we determine the type of a function, how do we know what types to use for the arguments? Luckily, we don’t have to answer these questions right away. We can first write *nondeterministic* rules such that a nano-ML program is well typed if it can be assigned a type by these rules. Nondeterministic rules make great specifications, but translating them to an algorithm is less obvious than in a type-checked language with deterministic rules. The rest of this section presents the type rules; Sections 7.5.1 and 7.5.2 explain how to get from the nondeterministic rules to a deterministic type-inference algorithm.

#### *Nondeterministic type rules for expressions*

CONAPP	418
emptyEnv	311a
eqTycon,	
in nano-ML	419a
in $\mu$ ML	497c
TYCON	418
TYVAR	418

The typing judgment for an expression has the form  $\Gamma \vdash e : \tau$ , meaning that given type environment  $\Gamma$ , expression  $e$  has type  $\tau$ . An expression may have more than one type; for example, the empty list has many types, and the judgments  $\Gamma \vdash \text{LITERAL(NIL)} : \text{int list}$  and  $\Gamma \vdash \text{LITERAL(NIL)} : \text{bool list}$  are both valid.

The use of a variable is well typed if the variable is bound in the environment. The variable has any type that is an instance of its type scheme.

$$\frac{\Gamma(x) = \sigma \quad \tau \leqslant \sigma}{\Gamma \vdash x : \tau} \quad (\text{VAR})$$

Unlike our rules for operational semantics, this rule does *not* specify a deterministic algorithm. Any  $\tau$  that is an instance of  $\sigma$  is acceptable, and it is not immediately

obvious how to find the “right” one. I say more about this problem in Section 7.5.2; our implementation finds a *most general*  $\tau$  (see also Exercise 5 on page 457).

A conditional expression is well typed if the condition is Boolean and the two branches have the same type  $\tau$ . The type of the conditional expression is also  $\tau$ .

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau} \quad (\text{IF})$$

A sequence is well typed if its subexpressions are well typed.

$$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n} \quad (\text{BEGIN})$$

An empty BEGIN is always well typed with type unit.

$$\frac{}{\Gamma \vdash \text{BEGIN}() : \text{unit}} \quad (\text{EMPTYBEGIN})$$

An application is well typed if the function has arrow type, and if the types and number of actual parameters match the types and number of formal parameters on the left of the arrow.

$$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \quad \Gamma \vdash e : \tau_1 \times \dots \times \tau_n \rightarrow \tau}{\Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \tau} \quad (\text{APPLY})$$

A function is well typed if its body is well typed, in an environment that gives the types of the arguments. The type of the function is formed from the types of its arguments and body.

$$\frac{\Gamma \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau} \quad (\text{LAMBDA})$$

Like the VAR rule, the LAMBDA rule is nondeterministic; types  $\tau_1, \dots, \tau_n$  aren’t in the syntax, and what they should be isn’t obvious.

In the LAMBDA rule, the notation  $\{x_i \mapsto \tau_i\}$  is shorthand for  $\{x_i \mapsto \forall. \tau_i\}$ ; each  $\tau_i$  is converted into a type scheme by wrapping it in an empty  $\forall$ . The type scheme  $\forall. \tau_i$  has only one instance, which is  $\tau_i$  itself. Therefore, when  $x_i$  is used in  $e$ , it always has the same type. How does this property affect a programmer? An ML programmer cannot define a function that *requires* its arguments to be polymorphic. No matter how polymorphic the *actual* parameter may be, inside the function the *formal* parameter has just one type. This property makes type inference decidable.

For example, although we can treat the global variable `empty-list` as polymorphic, consing both an integer and a boolean onto it, we cannot define a function that does the same thing with a formal parameter:

**424.** *(transcript 412)*  $\equiv$  ◀ 419b 425 ▶

```

-> (val empty-list '())
(): (forall ['a] (list 'a))
-> (val p (pair (cons 1 empty-list) (cons #t empty-list)))
(PAIR (1) (#t)) : (pair (list int) (list bool))
-> (val too-polymorphic
      (lambda (empty-list) (pair (cons 1 empty-list) (cons #t empty-list))))
type error: cannot make int equal to bool

```

The example shows the difference between an `empty-list` bound by `lambda` and an `empty-list` bound by `val`. Because `val` is the definition form that corresponds to `let`, this difference is also called the difference between a lambda-bound variable

and a let-bound variable. To illustrate this difference, here is the typing rule for **MLET**, a restricted form of **LET** that binds a single variable.

$$\frac{\Gamma \vdash e' : \tau' \quad \text{ftv}(\tau') - \text{ftv}(\Gamma) = \{\alpha_1, \dots, \alpha_n\} \quad \Gamma \{x \mapsto \forall \alpha_1, \dots, \alpha_n . \tau'\} \vdash e : \tau}{\Gamma \vdash \text{MLET}(x, e', e) : \tau} \quad (\text{MILNER'S LET})$$

Here  $\text{ftv}$  is a function that finds free type variables. Operationally, we first find  $\tau'$ , the type of  $e'$ , but we don't simply extend  $\Gamma$  with  $\{x \mapsto \tau'\}$ . Instead, using  $\forall$ , we *close* over all the free type variables of  $\tau'$  that are not also free type variables of types in  $\Gamma$ . Milner discovered that if a type variable isn't mentioned in the environment, you can instantiate that type variable any way you want, and you can even instantiate it *differently* at different uses of  $x$ . By closing over such type variables, we might give  $x$  a polymorphic type scheme. For example, in the following variation on `too-polyl`, the let-bound variable `empty-list` gets a polymorphic type scheme, and when the polymorphic `empty-list` is used, it is instantiated once with `int` and once with `bool`.

**425.** *(transcript 412)* +≡  
 $\rightarrow (\text{val not-too-polymorphic}$   
 $\quad (\text{let } (\text{empty-list } '())$   
 $\quad \quad (\text{pair } (\text{cons } 1 \text{ empty-list}) (\text{cons } \#t \text{ empty-list})))$   
 $\quad (\text{PAIR } (1) (\#t)) : (\text{pair } (\text{list int}) (\text{list bool}))$

△ 424 427a ▷

If let-bound variables might be polymorphic, why not  $\lambda$ -bound variables? We can't make  $\lambda$ -bound variables polymorphic because we don't know what type of value a  $\lambda$ -bound variable might stand for—it could be any actual parameter. By contrast, we know exactly what type of value a let-bound variable stands for, because it is right there in the program: it is the type of  $e'$ . If  $e'$  could be polymorphic (because it has type variables that don't appear in the environment), that polymorphism can be made explicit in the type scheme associated with  $x$ .

The polymorphic **MLET** is sometimes called "Milner's let," in honor of Robin Milner, who developed ML's type system. The fundamental new operation needed to handle Milner's let is *generalization*. To express generalization, here is function `generalize`, which takes as argument a type  $\tau$  and a set of constrained type variables  $\mathcal{A}$ :

$$\text{generalize}(\tau, \mathcal{A}) = \forall \alpha_1, \dots, \alpha_n . \tau, \text{ where } \{\alpha_1, \dots, \alpha_n\} = \text{ftv}(\tau) - \mathcal{A}.$$

Often  $\mathcal{A}$  is the set of type variables that appear in a type environment, i.e.,  $\mathcal{A} = \text{ftv}(\Gamma)$ . As suggested in Figure 7.2, generalization is a bit like the opposite of instantiation: for any  $\tau$  and  $\Gamma$ , it's always true that  $\tau \leq \text{generalize}(\tau, \text{ftv}(\Gamma))$ . An implementation of `generalize` appears in code chunk 445a.

Using `generalize`, we can rewrite the rule for Milner's let:

$$\frac{\Gamma \vdash e' : \tau' \quad \sigma = \text{generalize}(\tau', \text{ftv}(\Gamma)) \quad \Gamma \{x \mapsto \sigma\} \vdash e : \tau}{\Gamma \vdash \text{MLET}(x, e', e) : \tau} \quad (\text{MILNER'S LET})$$

cons       $\mathcal{P}$   
pair       $\mathcal{P}$

To extend this rule to work on a nano-ML **LET**, which simultaneously binds multiple names, I add a premise for each binding "`let  $x_i = e_i$` ," and I add all the new type schemes  $\sigma_1, \dots, \sigma_n$  to the environment:

$$\frac{\begin{array}{c} \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\ \sigma_i = \text{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \leq i \leq n \\ \Gamma \{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{\Gamma \vdash \text{LET}((x_1, e_1, \dots, x_n, e_n), e) : \tau} \quad (\text{LET})$$

The type rule for LETREC is similar; the difference is that the  $e_i$ 's are checked in type environment  $\Gamma'$ , which itself contains the bindings for the  $x_i$ 's. The essential idea of LETREC is that  $\Gamma'$  is defined using the set  $\{\tau_1, \dots, \tau_n\}$ , and each  $\tau_i$  is in turn defined using  $\Gamma'$ . So  $\Gamma'$  is defined in terms of itself, with types  $\tau_1, \dots, \tau_n$  as intermediaries:

$$\begin{array}{c}
 e_1, \dots, e_n \text{ are all LAMBDA expressions} \\
 \Gamma' = \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \\
 \Gamma' \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\
 \sigma_i = \text{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \leq i \leq n \\
 \frac{\Gamma\{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau}{\Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LETREC})
 \end{array}$$

Within  $\Gamma'$ , the expressions  $e_i$  are *not* polymorphic. Only once all the types of the  $e_i$ 's are fixed is it safe to generalize the types and compute the type of  $e$ . A LETREC is used to define a nest of mutually recursive functions, and because generalize is not applied until the types of all the functions are computed, these functions are not polymorphic when used in each other's definitions—they are polymorphic only when called from  $e$ . Even experienced ML programmers can be surprised by this restriction; in the presence of letrec, functions that look polymorphic may be less polymorphic than you expected.

A rule for LETSTAR would be annoying to write down directly: there is too much bookkeeping. Instead, we treat LETSTAR as syntactic sugar, rewriting it as a nest of LETS.

$$\begin{array}{c}
 \frac{\Gamma \vdash \text{LET}(\langle x_1, e_1 \rangle, \text{LETSTAR}(\langle x_2, e_2, \dots, x_n, e_n \rangle, e)) : \tau \quad n > 0}{\Gamma \vdash \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LETSTAR}) \\
 \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{LETSTAR}(\langle \rangle, e) : \tau} \quad (\text{EMPTYLETSTAR})
 \end{array}$$

### Type rules for definitions

In the operational semantics, a definition produces a new value environment. In the type system, a definition produces a new type environment. The relevant judgment has the form  $\langle d, \Gamma \rangle \rightarrow \Gamma'$ , which says that when definition  $d$  is typed in environment  $\Gamma$ , the new environment is  $\Gamma'$ .

A VAL binding is just like a Milner let binding, and the rules are almost the same.

$$\frac{\Gamma \vdash e : \tau \quad \sigma = \text{generalize}(\tau, \text{ftv}(\Gamma))}{\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad (\text{VAL})$$

VAL-REC requires that a recursive value be defined in the environment used to find its type. As in Typed  $\mu$ Scheme, only LAMBDA forms are accepted.

$$\frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau \quad \sigma = \text{generalize}(\tau, \text{ftv}(\Gamma)) \quad e \text{ has the form LAMBDA}(\dots)}{\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad (\text{VALREC})$$

A top-level expression is syntactic sugar for a binding to it.

$$\frac{\langle \text{VAL(it, e)}, \Gamma \rangle \rightarrow \Gamma'}{\langle \text{EXP}(e), \Gamma \rangle \rightarrow \Gamma'} \quad (\text{EXP})$$

A DEFINE is syntactic sugar for a combination of VAL-REC and LAMBDA.

$$\frac{\langle \text{VAL-REC}(f, \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e)), \Gamma \rangle \rightarrow \Gamma'}{\langle \text{DEFINE}(f, (\langle x_1, \dots, x_n \rangle, e)), \Gamma \rangle \rightarrow \Gamma'} \quad (\text{DEFINE})$$

### 7.4.6 Nondeterministic typing, principal types, and type testing

If the type rules are nondeterministic, how can we use them? We can't simply try all possible types of a term: some terms, like the empty-list literal, have infinitely many types. Fortunately, if an ML term has any type, it has a *principal type*. Type  $\tau_P$  is a principal type of  $e$  if it is a type of  $e$  and it is at least as general as any other type of  $e$ . That is, not only  $\Gamma \vdash e : \tau_P$ , but also if  $\Gamma \vdash e : \tau$ , then  $\tau \leq \tau_P$ . Therefore every type of  $e$  can be obtained by substituting free type variables of  $\tau_P$ .

The idea of principality is easily extended to type schemes. To begin, let us relate two type schemes just as we relate potentially quantified types in Typed μScheme:  $\sigma_g$  is at least as general as  $\sigma_i$ , written  $\sigma_i \leq \sigma_g$ , if we can get  $\sigma_i$  from  $\sigma_g$  by substitution and by *reordering* of type variables. (Because nano-ML has no explicit instantiation, the order of type variables doesn't matter, and two type schemes that differ up to permutations of type variables are equivalent.) Another way to state the relation is that  $\sigma_i \leq \sigma_g$  if and only if  $\sigma_g$  has all of  $\sigma_i$ 's instances. That is, whenever  $\tau \leq \sigma_i$ , it is also true that  $\tau \leq \sigma_g$ .

Given that relation, type scheme  $\sigma_P$  is a principal type scheme of  $e$  in environment  $\Gamma$  if there is a derivation of  $\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma_P\}$ , and furthermore, whenever there is a  $\sigma$  such that  $\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}$ , then  $\sigma \leq \sigma_P$ .

The ideas of generality and principality play a key role in unit-testing types in nano-ML programs. To test a type of an expression, we supply the expression and a type scheme. The given expression must have a type that is at least as general as the given type scheme. For example, the following tests both pass:

**427a.** *(transcript 412) +≡*

△425 433▷

```
-> (define arg1 (x y) x)
-> (check-type arg1 (forall ['a 'b] ('a 'b -> 'a))) ; the principal type
-> (check-type arg1 (forall ['a] ('a 'a -> 'a))) ; a less general type
```

Essentially,  $(\text{check-type } e \sigma)$  checks if a variable defined equal to  $e$  can be bound into the type environment with type scheme  $\sigma$ —that is, if there is a derivation of  $\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}$ .

Sometimes we want to test *exactly* what type a function has—if the function's principal type is more general than we expect, maybe something is wrong. This situation calls for a *check-principal-type* test. Here is an example:

**427b.** *(principal-types.nml 427b) ≡*

427c▷

```
(define arg1 (x y) x)
(check-principal-type arg1 (forall ['a 'b] ('a 'b -> 'a))) ; pass
(check-principal-type arg1 (forall ['a] ('a 'a -> 'a))) ; FAIL
```

As another example, *length* has many types and many type schemes, but only one principal type scheme:

**427c.** *(principal-types.nml 427b) +≡*

△427b

```
(check-type length ((list int) -> int)) ; pass
(check-principal-type length ((list int) -> int)) ; FAIL
(check-type length (forall ['a] ((list (list 'a)) -> int))) ; pass
(check-principal-type length (forall ['a] ((list (list 'a)) -> int))) ; FAIL
(check-type length (forall ['a] ((list 'a) -> int))) ; pass
(check-principal-type length (forall ['a] ((list 'a) -> int))) ; pass
```

The implementations of *check-type* and *check-principal-type*, which are found in Appendix R on page S415, use the  $\leq$  relation on type schemes.

## 7.5 FROM TYPE RULES TO TYPE INFERENCE

The type rules above don't specify an obvious deterministic algorithm for deciding if a nano-ML term has a type, let alone finding a principal type. More precisely,

given  $\Gamma$  and  $e$ , there is no obvious way to find a  $\tau$  such that  $\Gamma \vdash e : \tau$ . Difficulties arise in the LAMBDA rule, where it's not obvious what should be the types of the arguments, and in the VAR rule, where it's not obvious *which* instance of  $\sigma$  we should use as  $\tau$ . To use these rules in an algorithm, we replace each unknown type with a fresh type variable. For example, when we see an argument  $x_i$ , we record its type as  $\alpha_i$ , where  $\alpha_i$  is a new type variable that is not used anywhere else in the program. Each  $\alpha_i$  stands for an unknown type, and when we see more code, we might learn something about it. For example, if  $x_i$  were added to 1, we would learn that  $\alpha_i$  has to be equal to int. We would eventually substitute int for  $\alpha_i$ .

How do we discover an appropriate type to substitute for each fresh type variable? There are two methods worth discussing:

- The first method is the method of *explicit substitutions*. When we want two types to be equal, we call ML function  $\text{unify}(\tau_1, \tau_2)$ . A *unification* algorithm returns a  $\theta$  such that  $\theta(\tau_1) = \theta(\tau_2)$ ; substitution  $\theta$  is called a *unifier* of  $\tau_1$  and  $\tau_2$ . These substitutions are composed to implement *type inference*: given  $\Gamma$  and  $e$ , we can find  $\theta$  and  $\tau$  such that  $\boxed{\theta\Gamma \vdash e : \tau}$ .

The method of explicit substitutions is described by simple mathematics and is easy to prove correct, but it generates an awful lot of substitutions, and they have to be composed in exactly the right order. And it is easy to forget to apply a substitution; in short, the implementation is not easy to get right.

- The second method is the method of *type-equality constraints*. When we want two types  $\tau_1$  and  $\tau_2$  to be equal, we *don't* unify them right away. Instead, we write down the *constraint*  $\tau_1 \sim \tau_2$ , which says that  $\tau_1$  must equal  $\tau_2$ . If we have two or more such constraints, we conjoin them into a single constraint using logical *and*, as in  $C_1 \wedge C_2$ . The constraints are used in type inference in a judgment of the form  $\boxed{C, \Gamma \vdash e : \tau}$ .

The method of explicit constraints is described by more elaborate mathematics and is not as easy to prove correct, but the great thing about it is that the composition operation  $\wedge$  is associative and commutative, so the order in which you compose constraints doesn't matter. And as long as you conjoin all the constraints your code produces, you can't get the conjunction wrong. Using constraints, most of type inference is not so hard to get right.

In the method of explicit constraints, unification happens lazily: when we see a LET binding or a VAL binding, we call on a *constraint solver* to produce a substitution that makes the constraints true. The constraint solver does the same job as *unify*, and solving constraints is only a little bit more complicated than unifying types. And once we have the constraint solver, implementing type inference itself is infinitely easier. For this reason, the method of explicit constraints is the one that I recommend.

Both methods of type inference are justified by the same principle: in a valid derivation, we can substitute for free type variables. That is, if  $\frac{\mathcal{D}}{\Gamma \vdash e : \tau}$  is a valid derivation, then for all substitutions  $\theta$ ,  $\frac{\theta\mathcal{D}}{\theta\Gamma \vdash \theta e : \theta\tau}$  is also a valid derivation.

Which method should you study? It depends what you want to do.

- If you want to read original, primary sources, study explicit substitutions, because that's what Milner (1978) describes.

- If you want to prove that type inference is consistent with the nondeterministic type system in Section 7.4.5 (Exercise 11), study explicit substitutions—the proof is much easier.
- If you want to build something (Exercises 18 and 19), study type-equality constraints—the code is much easier.

### 7.5.1 The method of explicit substitutions

§7.5  
From type rules to  
type inference  
429

When the first algorithm for ML type inference was developed, it used explicit substitutions. If we want to understand explicit substitutions in detail, a good place to start is by developing a syntactic proof system for judgments of the form  $\theta\Gamma \vdash e : \tau$ . Here, for example, is the rule for IF.

$$\frac{\begin{array}{c} \theta_1\Gamma \vdash e_1 : \tau_1 & \theta_2(\theta_1\Gamma) \vdash e_2 : \tau_2 & \theta_3(\theta_2(\theta_1\Gamma)) \vdash e_3 : \tau_3 \\ \theta(\theta_3\tau_2) = \theta(\tau_3) = \tau \\ \theta'(\theta(\theta_3(\theta_2(\tau_1)))) = \theta'(\text{bool}) \end{array}}{(\theta' \circ \theta \circ \theta_3 \circ \theta_2 \circ \theta_1)\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \theta'\tau} \quad (\text{IF})$$

This rule has an operational interpretation:

1. Infer type  $\tau_1$  for  $e_1$ , producing substitution  $\theta_1$ .
2. Apply  $\theta_1$  to the environment  $\Gamma$ , and continue in similar fashion, inferring types and substitutions from  $e_2$  and  $e_3$ .
3. Unify types  $\theta_3\tau_2$  and  $\tau_3$ , producing a new substitution  $\theta$ . The resulting type  $\tau$  becomes (after one more substitution) the type of the entire IF expression.
4. Unify the type of  $e_1$  with `bool`, producing substitution  $\theta'$ . Be careful to apply proper substitutions.
5. Return the composition of the substitutions, together with  $\theta'\tau$ .

This new rule is *sound*, which means that whenever there is a derivation using the new IF rule, there is a corresponding derivation using the original IF rule. A real proof of soundness is beyond the scope of this book,<sup>6</sup> but a hand-waving argument can help you understand how the system works. The idea is to rewrite derivations systematically so that if we are given a derivation in the new system, and we rewrite that derivation, we get a derivation in the old system. In the case of the IF rule, we can rewrite  $(\theta' \circ \theta \circ \theta_3 \circ \theta_2 \circ \theta_1)\Gamma \rightarrow \tilde{\Gamma}$ ,  $(\theta' \circ \theta \circ \theta_3 \circ \theta_2)\tau_1 \rightarrow \tilde{\tau}_1$ , and so on. For example, if  $\theta_1\Gamma \vdash e_1 : \tau_1$ , we apply substitution  $(\theta' \circ \theta \circ \theta_3 \circ \theta_2)$  to both sides, rewrite, and the premise becomes equivalent to  $\tilde{\Gamma} \vdash e_1 : \tilde{\tau}_1$ . A similar rewriting of all the premises enables us to apply the original IF rule to draw the conclusion.

Although the new IF rule is sound, and it has a clear operational interpretation, it requires a *lot* of explicit substitutions; it's a bookkeeping nightmare. But the bookkeeping can be reduced by a trick: extend the typing judgment to *lists* of expressions and types. We write  $\theta\Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n$  as an abbreviation for a set of  $n$  separate judgments:  $\theta\Gamma \vdash e_1 : \tau_1, \dots, \theta\Gamma \vdash e_n : \tau_n$ . When  $n = 1$ , this judgment degenerates to  $\theta\Gamma \vdash e_1 : \tau_1$ . For  $n > 1$ , finding the common substitution  $\theta$  requires combining substitutions from different judgments.

$$\frac{\theta\Gamma \vdash e_1 : \tau_1 \quad \theta'(\theta\Gamma) \vdash e_2, \dots, e_n : \tau_2, \dots, \tau_n}{(\theta' \circ \theta)\Gamma \vdash e_1, \dots, e_n : \theta'\tau_1, \tau_2, \dots, \tau_n} \quad (\text{TYPESOF})$$

---

<sup>6</sup>See the notes on Exercise 11.

By using this new judgment, we can reduce the number of substitutions in any rules that has multiple subexpressions, like IF. For example, here is the application rule:

$$\frac{\theta\Gamma \vdash e, e_1, \dots, e_n : \hat{\tau}, \tau_1, \dots, \tau_n}{\theta'(\hat{\tau}) = \theta'(\tau_1 \times \dots \times \tau_n \rightarrow \alpha), \text{ where } \alpha \text{ is fresh}} \quad (\text{APPLY})$$

The most difficult rule to express using explicit substitutions is probably LETREC. The nondeterministic rule is

$$\frac{\begin{array}{c} e_1, \dots, e_n \text{ are all LAMBDA expressions} \\ \Gamma' = \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \\ \Gamma' \vdash e_i : \tau_i, \quad 1 \leq i \leq n \\ \sigma_i = \text{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \leq i \leq n \\ \Gamma\{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{\Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LETREC})$$

The rule with explicit substitutions is

$$\frac{\begin{array}{c} e_1, \dots, e_n \text{ are all LAMBDA expressions} \\ \Gamma' = \Gamma\{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}, \text{ where all } \alpha_i \text{'s are fresh} \\ \theta\Gamma' \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n \\ \theta'\tau_i = \theta'(\theta\alpha_i), \quad 1 \leq i \leq n \\ \sigma_i = \text{generalize}(\theta'\tau_i, \text{ftv}((\theta' \circ \theta)\Gamma)), \quad 1 \leq i \leq n \\ \theta''(((\theta' \circ \theta)\Gamma)\{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\}) \vdash e : \tau \end{array}}{(\theta'' \circ \theta' \circ \theta)\Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LETREC})$$

The soundness proof requires an argument that the substitution  $\theta''$  does not change any of the type variables that are *bound* in any  $\sigma_i$ .

Using explicit substitutions, rules like LETREC are hard to implement: not only must you compose three or more substitutions, but you must compose them in the right order, and you must apply certain substitutions or even compositions (like  $\theta' \circ \theta$ ) to one or more arguments of other calls. In my experience, people who tackle this implementation task almost always forget a substitution somewhere, leading to an implementation of type inference that almost works, but not quite. There is a better way.

### 7.5.2 The method of explicit constraints

To find that better way, let's return to the nondeterministic type system of Section 7.4.5. In a rule like

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau} \quad (\text{IF})$$

$e_1$  magically has the right type `bool` and  $e_2$  and  $e_3$  magically have the same type  $\tau$ . The idea of explicit constraints is that instead of requiring magic, we allow each each  $e_i$  to have whatever type it wants, which we call  $\tau_i$ . Then we insist that  $\tau_1$  must equal `bool` and  $\tau_2$  must equal  $\tau_3$ . To record our insistence, we use explicit constraints, and we extend the form of typing judgments to add a constraint  $C$  to the typing context. For the IF rule, the constraint we insist on is  $\tau_1 \sim \text{bool} \wedge \tau_2 \sim \tau_3$ . Operators  $\sim$  and  $\wedge$  are explained below.

A judgment of the form  $C, \Gamma \vdash e : \tau$  means “assuming the constraint  $C$  is satisfied, in environment  $\Gamma$  term  $e$  has type  $\tau$ .” Constraints are formed by conjoining simple equality constraints:

- A simple equality constraint has the form  $\tau_1 \sim \tau_2$  (pronounced “ $\tau_1$  must equal  $\tau_2$ ”), and it is satisfied if and only if the types  $\tau_1$  and  $\tau_2$  are equal.
- A conjunction has the form  $C_1 \wedge C_2$  (pronounced “ $C_1$  and  $C_2$ ”), and it is satisfied only if both  $C_1$  and  $C_2$  are satisfied. Just as in ordinary logic, conjunction of constraints is associative and commutative.

The math and the code become easier if we allow a third form of constraint:

- The *trivial constraint* has the form  $\mathbf{T}$ , and it is always considered satisfied. The trivial constraint is a left and right identity of  $\wedge$ . The constraint may be pronounced “trivial” or “true.”

Formally, here is a grammar for constraints:

$$C ::= \tau_1 \sim \tau_2 \mid C_1 \wedge C_2 \mid \mathbf{T}$$

In a typing judgment, the constraint captures what has to be true if a term is going to have a type. A typing judgment has the form  $C, \Gamma \vdash e : \tau$ . Operationally, the expression  $e$  and environment  $\Gamma$  are the inputs to the type checker; the type  $\tau$  and constraint  $C$  are the outputs. The type system is designed so that if it derives  $C, \Gamma \vdash e : \tau$ , and if constraint  $C$  is satisfied, then erasing constraints produces a derivation of  $\Gamma \vdash e : \tau$  that is valid in the original, nondeterministic type system.

Using constraints, we can write the IF rule properly. We must not only create new constraints  $\tau_1 \sim \text{bool}$  and  $\tau_2 \sim \tau_3$ ; we must also remember any old constraints used to give types to the subexpressions  $e_1$ ,  $e_2$ , and  $e_3$ . “Old” constraints propagate from the premises of a rule to the conclusion.

$$\frac{C_1, \Gamma \vdash e_1 : \tau_1 \quad C_2, \Gamma \vdash e_2 : \tau_2 \quad C_3, \Gamma \vdash e_3 : \tau_3}{C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sim \text{bool} \wedge \tau_2 \sim \tau_3, \Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau_2} \quad (\text{IF})$$

The conclusion of this rule conjoins three old constraints with two new simple equality constraints. For the IF expression to have a type, all the constraints needed to give types to  $e_1$ ,  $e_2$ , and  $e_3$  must be satisfied. And it must also be the case that  $\tau_1 \sim \text{bool}$  and  $\tau_2 \sim \tau_3$ . If all the constraints are satisfied, which implies that  $\tau_1 = \text{bool}$  and  $\tau_2 = \tau_3$ , then the rule is equivalent to the original IF rule.

Constraints require a lot less bookkeeping than do the substitutions in Section 7.5.1, but it’s still worth defining a typing judgment that describes lists of expressions and types. We write  $C, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n$  as an abbreviation for a set of  $n$  separate judgments, where  $C$  is the conjunction of the constraints of the individual judgments:

$$\frac{C_1, \Gamma \vdash e_1 : \tau_1 \quad \dots \quad C_n, \Gamma \vdash e_n : \tau_n}{C_1 \wedge \dots \wedge C_n, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n} \quad (\text{TYPESOF})$$

Using this notation, we can simplify some rules. For example, here is the IF rule:

$$\frac{C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3}{C \wedge \tau_1 \sim \text{bool} \wedge \tau_2 \sim \tau_3, \Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau_2} \quad (\text{IF})$$

The application rule uses not only the typing judgment for a list of expressions, but also a fresh type variable. In an application, the function must have an arrow type, so if we apply an expression  $e$  of type  $\hat{\tau}$ ,  $\hat{\tau}$  must be an arrow type. But what arrow type? The argument types are the types of  $e$ ’s actual parameters, but what about the result type? Because we don’t know, we make the result type a fresh type variable  $\alpha$ , and we let its ultimate identity be determined by the constraint  $C$ . Type  $\hat{\tau}$  must therefore be equal to  $\tau_1 \times \dots \times \tau_n \rightarrow \alpha$ :

$$\frac{C, \Gamma \vdash e, e_1, \dots, e_n : \hat{\tau}, \tau_1, \dots, \tau_n \quad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \sim \tau_1 \times \dots \times \tau_n \rightarrow \alpha, \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \alpha} \quad (\text{APPLY})$$

Again, if the constraints are satisfied, the rule is equivalent to the original.

With these rules, we can build an example derivation. Let's assume that the type environment  $\Gamma$  contains these bindings:

$$\Gamma = \{ + : \forall \alpha. \alpha \times \alpha \rightarrow \alpha, \text{cons} : \forall \alpha. \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list} \}.$$

This  $\Gamma$  has no free type variables. Using  $\Gamma$ , if we try to infer a type for  $(+ 1 2)$ , the derivation looks roughly like this, where  $\alpha_{10}$  is a fresh type variable:

$$\frac{\frac{\frac{\dots}{\mathbf{T}, \Gamma \vdash + : \alpha \times \alpha \rightarrow \alpha} \quad \frac{\dots}{\mathbf{T}, \Gamma \vdash 1 : \alpha} \quad \frac{\dots}{\mathbf{T}, \Gamma \vdash 2 : \alpha}}{\mathbf{T} \wedge \mathbf{T} \wedge \mathbf{T} \wedge \alpha \times \alpha \rightarrow \alpha \sim \alpha \times \alpha \rightarrow \alpha_{10}}, \Gamma \vdash (+ 1 2) : \alpha_{10}}$$

If we then substitute  $\alpha$  for  $\alpha_{10}$  in this derivation, we get

$$\frac{\frac{\frac{\dots}{\mathbf{T}, \Gamma \vdash + : \alpha \times \alpha \rightarrow \alpha \sim \alpha} \quad \frac{\dots}{\mathbf{T}, \Gamma \vdash 1 : \alpha} \quad \frac{\dots}{\mathbf{T}, \Gamma \vdash 2 : \alpha}}{\mathbf{T} \wedge \mathbf{T} \wedge \mathbf{T} \wedge \alpha \times \alpha \rightarrow \alpha \sim \alpha \times \alpha \rightarrow \alpha}, \Gamma \vdash (+ 1 2) : \alpha}$$

All the constraints are satisfied, and if we erase them, we are left with a derivation in the original, nondeterministic system.

### Converting nondeterministic rules to use constraints

Almost every nondeterministic rule can be converted to a deterministic, constraint-based rule. We need just two ideas:

- If in the original rule the same type  $\tau$  appears in more than one place, give each appearance its own name (like  $\tau_2$  and  $\tau_3$ ), and introduce constraints forcing the names to be equal.
- If a type  $\tau$  appears in the original rule and we don't know what  $\tau$  is supposed to be, use a fresh type variable.

The first idea is illustrated by the IF rule. To illustrate the second idea, let's convert the nondeterministic VAR rule to use explicit constraints. The nondeterministic rule says

$$\frac{\Gamma(x) = \sigma \quad \tau \leqslant \sigma}{\Gamma \vdash x : \tau} \quad (\text{VAR})$$

We know that  $\tau' \leqslant \sigma$  when  $\sigma = \forall \alpha_1, \dots, \alpha_n. \tau$  and some (unknown) types are substituted for the  $\alpha_1, \dots, \alpha_n$ . For the unknown types, we pick fresh type variables  $\alpha'_1, \dots, \alpha'_n$ .

$$\frac{\Gamma(x) = \forall \alpha_1, \dots, \alpha_n. \tau \quad \alpha'_1, \dots, \alpha'_n \text{ are fresh and distinct}}{\Gamma, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \dots \circ (\alpha_n \mapsto \alpha'_n)) \tau} \quad (\text{VAR})$$

In  $\tau$ , each  $\alpha_i$  is replaced by the corresponding fresh  $\alpha'_i$ , and  $\alpha'_1, \dots, \alpha'_n$  are eventually constrained by the way  $x$  is used. For example, if we infer a type for the expression  $(\text{cons } 1 \text{ '()})$ , although the instance of  $\text{cons}$  uses a fresh type variable, eventually that type variable should be constrained to be equal to  $\text{int}$ . The derivation looks roughly like this:

$$\frac{\frac{\frac{\frac{\Gamma(\text{cons}) = \forall \alpha. \alpha \times \alpha \rightarrow \alpha}{\Gamma, \Gamma \vdash \text{cons} : \alpha_{11} \times \alpha_{11} \rightarrow \alpha_{11}}} \quad \frac{\dots}{\Gamma, \Gamma \vdash 1 : \alpha_{12}} \quad \frac{\dots}{\Gamma, \Gamma \vdash \text{'() : } \alpha_{12}}}{\Gamma, \Gamma \vdash (\text{cons } 1 \text{ '()}) : \alpha_{13}}} {\Gamma, \Gamma \vdash \alpha_{11} \times \alpha_{11} \rightarrow \alpha_{11} \sim \alpha_{12} \times \alpha_{12} \rightarrow \alpha_{13}}, \Gamma \vdash (\text{cons } 1 \text{ '()}) : \alpha_{13}}$$

That big constraint can be rewritten as a simpler, equivalent constraint:

$$\alpha_{11} \sim \text{int} \wedge \alpha_{11} \text{ list} \sim \alpha_{12} \text{ list} \wedge \alpha_{11} \text{ list} \sim \alpha_{13}.$$

Both constraints are solved by the substitution

$$(\alpha_{11} \mapsto \text{int}) \circ (\alpha_{12} \mapsto \text{int}) \circ (\alpha_{13} \mapsto \text{int list}),$$

§7.5

From type rules to  
type inference

433

and the type of `(cons 1 '())` is `int list`.

*Inferring polytypes for bound names*

In the examples above, each type variable is determined to be a completely known type, like `int` or `int list`. But when we define polymorphic values, some types will be unknown. These types will be denoted by free type variables, and eventually we want to create polymorphic type schemes that use  $\forall$  with those free type variables. For example, a function that makes a singleton list should have a polymorphic type:

433. *transcript 412* +≡

◁ 427a 435 ▷

```
-> (val singleton (lambda (x) (cons x '())))
singleton : (forall ['a] ('a -> (list 'a)))
```

Converting from a monotype  $\tau$  with free type variables to a type scheme  $\sigma$  with an explicit `forall` is the most challenging part of type inference; this conversion is called *generalization*.

The easiest example of generalization is in the rule for a `VAL` binding:

$$\frac{\begin{array}{c} C, \Gamma \vdash e : \tau \\ \theta C \text{ is satisfied} \quad \theta \Gamma = \Gamma \\ \sigma = \text{generalize}(\theta \tau, \text{ftv}(\Gamma)) \end{array}}{\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma \{x \mapsto \sigma\}} \quad (\text{VAL})$$

Here's an operational interpretation:

- Typecheck  $e$  in environment  $\Gamma$ , getting back type  $\tau$  and constraint  $C$
- Choose a substitution  $\theta$  such that  $\theta C$  is satisfied, making sure that  $\theta$  does not affect any free type variables of  $\Gamma$  (this step is called *solving C*; how to do it is the subject of Section 7.5.3 below)
- Generalize the type  $\theta \tau$  to form the general type scheme  $\sigma$ , which becomes the type of  $x$  in a new environment  $\Gamma \{x \mapsto \sigma\}$

Assuming we have a valid derivation of the judgment  $C, \Gamma \vdash e : \tau$ , here's why the rule works: for any  $\theta$ ,

- $\theta C, \theta \Gamma \vdash e : \theta \tau$  is a derivable judgment.
- Because  $\theta \Gamma = \Gamma$ , the judgment  $\theta C, \Gamma \vdash e : \theta \tau$  is also derivable.
- Because  $\theta C$  is satisfied,  $\Gamma \vdash e : \theta \tau$  is a derivable judgment in the original, nondeterministic system.
- If the original system can derive type  $\theta \tau$  for  $e$ , it is safe to generalize that type.

cons

P

The `VAL` rule illustrates the key ideas underlying constraint-based inference of polymorphic type schemes:

- We need a substitution  $\theta$  that solves a constraint  $C$ , but  $\theta$  mustn't substitute for any free type variables of the environment  $\Gamma$ .
- Using substitution  $\theta$  on a type  $\tau$  gives us a type we can generalize.

Here's an example. If we type the definition

```
(val singleton (lambda (x) (cons x '()))))
```

then the body of the lambda is checked in type environment  $\Gamma' = \Gamma\{x : \forall.\alpha_{14}\}$ ,<sup>7</sup> and the derivation of the type of the lambda looks something like this:

$$\frac{\begin{array}{c} \Gamma'(\text{cons}) = \forall.\alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list} \\ \mathbf{T}, \Gamma \vdash \text{cons} : \alpha_{15} \times \alpha_{15} \text{ list} \rightarrow \alpha_{15} \text{ list} \end{array}}{\alpha_{15} \times \alpha_{15} \text{ list} \rightarrow \alpha_{15} \text{ list} \sim \alpha_{14} \times \alpha_{16} \text{ list} \rightarrow \alpha_{17}, \Gamma' \vdash (\text{cons } x \ '()) : \alpha_{17}}$$

$$\frac{\Gamma'(x) = \forall.\alpha_{14} \quad \mathbf{T}, \Gamma' \vdash x : \alpha_{14}}{\mathbf{T}, \Gamma \vdash '() : \alpha_{16} \text{ list}}$$

$$\frac{\alpha_{15} \times \alpha_{15} \text{ list} \rightarrow \alpha_{15} \text{ list} \sim \alpha_{14} \times \alpha_{16} \text{ list} \rightarrow \alpha_{17}, \Gamma \vdash (\text{lambda } (x) (\text{cons } x \ '())) : \alpha_{14} \rightarrow \alpha_{17}}$$

The constraint can be simplified to

$$\alpha_{15} \sim \alpha_{14} \wedge \alpha_{15} \text{ list} \sim \alpha_{16} \text{ list} \wedge \alpha_{15} \text{ list} \sim \alpha_{17}$$

and then further simplified to

$$C = \alpha_{15} \sim \alpha_{14} \wedge \alpha_{15} \sim \alpha_{16} \wedge \alpha_{15} \text{ list} \sim \alpha_{17}.$$

This constraint is solved by several substitutions; for example, it is solved by the substitution  $\theta = (\alpha_{14} \mapsto \alpha_{15}) \circ (\alpha_{16} \mapsto \alpha_{15}) \circ (\alpha_{17} \mapsto \alpha_{15} \text{ list})$ . The most interesting premises of the VAL are then

$$\begin{aligned} C, \Gamma \vdash (\text{lambda } (x) (\text{cons } x \ '())) &: \tau \\ \tau = \alpha_{14} \rightarrow \alpha_{17} \quad \theta\tau = \alpha_{15} \rightarrow \alpha_{15} \text{ list} \quad \sigma = \forall\alpha_{15}.\alpha_{15} \rightarrow \alpha_{15} \text{ list} \end{aligned}$$

Name `singleton` is therefore added to  $\Gamma$  with type scheme  $\forall\alpha_{15}.\alpha_{15} \rightarrow \alpha_{15} \text{ list}$ , which we prefer to write in canonical form as  $\forall\alpha.\alpha \rightarrow \alpha \text{ list}$ .

*A term that has no type produces an unsolvable constraint*

In the method of explicit substitutions, when a term has no type, the problem manifests as a call to `unify` with two types that can't be unified. In the method of explicit constraints, when a term has no type, the problem manifests as a constraint that can't be solved. As an example, let's consider the function

```
(lambda (x) (cons x x))
```

Let's assume that `x` is introduced to the environment with the monotype  $\forall.\alpha_{18}$ , that `cons` is instantiated with type  $\alpha_{19} \times \alpha_{19} \text{ list} \rightarrow \alpha_{19} \text{ list}$ , and that the return type of the lambda is type variable  $\alpha_{20}$ . Then the system derives a judgment that looks roughly like this:

$$\alpha_{19} \times \alpha_{19} \text{ list} \rightarrow \alpha_{19} \text{ list} \sim \alpha_{18} \times \alpha_{18} \rightarrow \alpha_{20}, \Gamma \vdash (\text{lambda } (x) (\text{cons } x x)) : \alpha_{18} \rightarrow \alpha_{20}.$$

The constraint can be simplified to

$$\alpha_{19} \sim \alpha_{18} \wedge \alpha_{19} \text{ list} \sim \alpha_{18} \wedge \alpha_{19} \text{ list} \sim \alpha_{20}.$$

<sup>7</sup>Just as in the nondeterministic system, the rule for `lambda` type-checks the body in an extended environment that binds the formal parameter `x` to a monotype. If you wonder why I don't show you a rule for `lambda`, it's because I want you to develop the rule yourself; see Exercise 8 on page 457.

The third simple type equality can be satisfied by substituting  $\alpha_{19}$  list for  $\alpha_{20}$ . The first can be satisfied by substituting  $\alpha_{19}$  for  $\alpha_{18}$  or vice versa. So the full constraint is solvable if and only if the simpler constraint

$$\alpha_{19} \text{ list} \sim \alpha_{19}$$

is solvable (or equivalently, if  $\alpha_{18}$  list  $\sim \alpha_{18}$  is solvable). But there is no possible substitution for  $\alpha_{19}$  that makes  $\alpha_{19}$  list equal to  $\alpha_{19}$ .<sup>8</sup> And after putting the unsolvable constraint into canonical form, that's what the interpreter reports:

435. ⟨transcript 412⟩ +≡ ↳ 433 436 ▾  
 -> (val broken (lambda (x) (cons x x)))  
 type error: cannot make 'a equal to (list 'a)

### A prequel to LETX forms: Adding constraints is OK

What's left are the rules for LETX forms. These forms generalize the types of multiple bound names, which gets complicated. To avoid that complexity, most presentations of this theory work with a simplified “core calculus.” But if you're building an interpreter that infers types, you don't want some simplified core calculus; you want to infer types for code you actually write. To do it soundly, I present a rule that allows us to add a constraint:

$$\frac{C_i, \Gamma \vdash e_i : \tau_i \quad C \text{ has a solution}}{C \wedge C_i, \Gamma \vdash e_i : \tau_i} \quad (\text{OVERCONSTRAINED})$$

The idea is that although the constraint  $C_i$  is *sufficient* to give  $e_i$  a type, we may add another constraint  $C$ , provided that  $C$  has a solution. To prove this rule sound, we appeal to the substitution  $\theta$  that solves  $C$ : if we apply  $\theta$  to the valid derivation associated with the judgment  $C_i, \Gamma \vdash e_i : \tau_i$ , we get another valid derivation.

### Generalization in Milner's let binding

The most difficult part of constraint-based type inference is understanding how to generalize at a LET binding. In  $(\text{let } ([x_1 e_1]) \dots)$ , the key issues are the type of  $e_1$  and the type scheme bound to  $x_1$ . Here's an operational description of what happens to  $e_1$  and  $x_1$  in this let expression, where the type environment is  $\Gamma$ :

- Using a recursive call, we type check  $e_1$  in environment  $\Gamma$ , getting back a type  $\tau_1$  and a constraint  $C$ .
- We solve the constraint  $C$ , getting substitution  $\theta$ .
- Constraint  $C$  affects type variables that are free in  $\tau$  or in  $\Gamma$ . The type variables that are free in  $\tau$  can be eliminated by applying  $\theta$  to  $\tau$ . But we can't substitute for free type variables of  $\Gamma$ . Instead, we use  $\theta$  to build a new constraint  $C'$ . We build  $C'$  by applying  $\theta$  to the free type variables of  $\Gamma$ , which captures every way in which those type variables should be constrained.

$$C' = \bigwedge \{\alpha \sim \theta\alpha \mid \alpha \in \text{dom } \theta \cap \text{ftv}(\Gamma)\}.$$

The notation  $\bigwedge \{\dots\}$  says to form a single constraint by conjoining the constraints in the set. If the set is empty,  $\bigwedge \emptyset = \mathbf{T}$ .

<sup>8</sup>For a proof, consider the number of times that list appears on each side. No matter what you substitute for  $\alpha_{19}$ , there will always be one more list constructor on the left than on the right, so the two sides can never be equal.

- We form  $\sigma_1$  in two steps: First, apply  $\theta$  to  $\tau_1$ , getting the type of  $e_1$  as refined by constraint  $C$ . Second, generalize  $\theta\tau_1$  by quantifying over all of the type variables that are mentioned neither in  $\Gamma$  nor in  $C'$ .

These operations can be formalized as an incomplete rule, which puts the type scheme of  $x_1$  into  $\Gamma'$ :

$$\frac{\begin{array}{c} C, \Gamma \vdash e_1 : \tau_1 \\ \theta C \text{ is satisfied} \quad \theta \text{ is idempotent} \\ C' = \bigwedge \{\alpha \sim \theta\alpha \mid \alpha \in \text{dom } \theta \cap \text{ftv}(\Gamma)\} \\ \sigma_1 = \text{generalize}(\theta\tau_1, \text{ftv}(\Gamma) \cup \text{ftv}(C')) \\ \hline \Gamma' = \Gamma \{x_1 \mapsto \sigma_1\} \end{array}}{\dots} \quad (\text{INCOMPLETE SIMPLE LET})$$

Because  $\theta$  is used to form both  $C'$  and  $\sigma_1$ , it can eventually be used twice, so for soundness, we need it to be idempotent (sidebar on the facing page).

The final operation is to use new environment  $\Gamma'$  to infer the type of the body of the let. Here's what the complete rule looks like, with an arbitrary number of bound variables  $x_1, \dots, x_n$ :

$$\frac{\begin{array}{c} C, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n \\ \theta C \text{ is satisfied} \quad \theta \text{ is idempotent} \\ C' = \bigwedge \{\alpha \sim \theta\alpha \mid \alpha \in \text{dom } \theta \cap \text{ftv}(\Gamma)\} \\ \sigma_i = \text{generalize}(\theta\tau_i, \text{ftv}(\Gamma) \cup \text{ftv}(C')), \quad 1 \leq i \leq n \\ \hline C_b, \Gamma \{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LET})$$

In LETREC, the  $e_i$ 's are checked in an extended environment where each  $x_i$  is bound to a fresh type variable  $\alpha_i$ , and *before* generalization, each type  $\tau_i$  is constrained to be equal to the corresponding  $\alpha_i$ . The rest is the same.

$$\frac{\begin{array}{c} e_1, \dots, e_n \text{ are all LAMBDA expressions} \\ \Gamma' = \Gamma \{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}, \text{ where all } \alpha_i \text{'s are distinct and fresh} \\ C_r, \Gamma' \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n \\ C = C_r \wedge \tau_1 \sim \alpha_1 \wedge \dots \wedge \tau_n \sim \alpha_n \\ \theta C \text{ is satisfied} \quad \theta \text{ is idempotent} \\ C' = \bigwedge \{\alpha \sim \theta\alpha \mid \alpha \in \text{dom } \theta \cap \text{ftv}(\Gamma)\} \\ \sigma_i = \text{generalize}(\theta\tau_i, \text{ftv}(\Gamma) \cup \text{ftv}(C')), \quad 1 \leq i \leq n \\ \hline C_b, \Gamma \{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau} \quad (\text{LETREC})$$

Here's an example of generalization in LET:

436. *<transcript 412>+≡* △435

```

-> (val ss (lambda (y)
  (let ([single (lambda (x) (cons x '()))])
    (single (single y))))
  ss : (forall ['a] ('a -> (list (list 'a)))))

```

The whole derivation is too big to show, so let's look at pieces. Start by assuming that the *body* of the outer lambda is type-checked in an environment

$$\Gamma = \{\text{cons} : \forall \alpha. \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list}, y : \alpha_{20}\}.$$

In a similar environment, the singleton example on page 434 shows a derivation for the lambda, so if  $C = \alpha_{22} \times \alpha_{22} \text{ list} \rightarrow \alpha_{22} \text{ list} \sim \alpha_{21} \times \alpha_{23} \text{ list} \rightarrow \alpha_{24}$ , we can take it for granted that

$$C, \Gamma \vdash (\lambda (x) (\text{cons} x '())) : \alpha_{21} \rightarrow \alpha_{24}$$

## Soundness of generalization with constraints

Here is an informal argument about how generalization works with constraints.

- To compute the type being generalized, we use judgment  $C, \Gamma \vdash e_1 : \tau_1$ . For example, we might have

$$\alpha \sim \text{int}, \Gamma \vdash (+ \times 1) : \alpha,$$

where  $\Gamma = \{x : \alpha\}$ . Even though the type of  $(+ \times 1)$  has a type variable, we can't possibly generalize that variable:  $\alpha$  has to be `int`.

- If we solve the constraint  $C$ , we get the substitution  $\theta = (\alpha \mapsto \text{int})$ . The type  $\tau_1$  is an output, so we can substitute `int` for  $\alpha$  there, getting type  $\theta\tau_1 = \text{int}$ . But  $\Gamma$  is an input, so there's no way to substitute there: the constraint has to be retained.
- In the general case, we take the constraint  $C$  and its solution  $\theta$ , and we split  $\theta$  into two parts, so that  $\theta = \theta_g \circ \theta_l$ , where
  - $\theta_l = \theta|_{\text{dom } \theta \setminus \text{ftv}(\Gamma)}$  is the *local* part; it substitutes for type variables that are mentioned in  $C$  (and possibly in  $\tau$ ) but never in  $\Gamma$ . Substitution  $\theta_l$  can be applied to  $\tau$  and then thrown away.
  - $\theta_g = \theta|_{\text{ftv}(\Gamma)}$  is the *global* part; it substitutes for type variables that are mentioned in  $\Gamma$ . It can be applied to  $\tau$  but not to  $\Gamma$ . We can't throw  $\theta_g$  away, so we convert it to constraint  $C'$ .
- Because  $\sigma_1$  is formed by generalizing  $\theta\tau_1$ , none of the variables in  $\text{dom } \theta$  is generalized, and  $\theta$  is idempotent,  $\theta\sigma_1 = \sigma_1$ .

If we have a valid derivation of  $\theta C', \theta\Gamma \vdash e_1 : \theta\sigma_1$ , we can use the equalities above to recover a valid derivation in the original, nondeterministic system.

is derivable. Now the LET rule comes into play:

$$\begin{aligned} C &= \alpha_{22} \times \alpha_{22} \text{ list} \rightarrow \alpha_{22} \text{ list} \sim \alpha_{21} \times \alpha_{23} \text{ list} \rightarrow \alpha_{24} \\ \tau_1 &= \alpha_{21} \rightarrow \alpha_{24} \\ \theta &= (\alpha_{21} \mapsto \alpha_{22}) \circ (\alpha_{23} \mapsto \alpha_{22}) \circ (\alpha_{24} \mapsto \alpha_{22} \text{ list}) \\ C' &= \bigwedge \{ \} = \mathbf{T} \\ \text{ftv}(\Gamma) &= \{\alpha_{20}\} \\ \sigma_1 &= \text{generalize}(\alpha_{22} \rightarrow \alpha_{22} \text{ list}, \text{ftv}(\Gamma)) = \forall \alpha_{22}. \alpha_{22} \rightarrow \alpha_{22} \text{ list} \end{aligned}$$

The body of the LET, `(single (single x))`, is checked in the extended environment  $\Gamma_e = \Gamma \{ \text{single} : \forall \alpha_{22}. \alpha_{22} \rightarrow \alpha_{22} \text{ list} \}$ . Each instance of `single` gets its own type, and the typing derivation gets crowded. Abbreviating constraint  $C_{inner} = \alpha_{25} \rightarrow \alpha_{25} \text{ list} \sim \alpha_{20} \rightarrow \alpha_{26}$ , we have

$$\frac{\Gamma_e(\text{single}) = \forall \alpha_{22}. \alpha_{22} \rightarrow \alpha_{22} \text{ list}}{\mathbf{T}, \Gamma_e \vdash \text{single} : \alpha_{27} \rightarrow \alpha_{27} \text{ list}} \quad \frac{\Gamma_e(\text{single}) = \forall \alpha_{22}. \alpha_{22} \rightarrow \alpha_{22} \text{ list} \quad \Gamma_e(y) = \forall . \alpha_{20}}{\mathbf{T}, \Gamma_e \vdash y : \alpha_{20}} \quad \frac{\Gamma_e(\text{single}) = \forall \alpha_{22}. \alpha_{22} \rightarrow \alpha_{22} \text{ list} \quad \Gamma_e(y) = \forall . \alpha_{20}}{\mathbf{T}, \Gamma_e \vdash (\text{single } y) : \alpha_{26}}$$

$$\frac{\Gamma_e(\text{single}) = \forall \alpha_{22}. \alpha_{22} \rightarrow \alpha_{22} \text{ list} \quad \Gamma_e(\text{single } y) = \forall . \alpha_{26}}{\alpha_{27} \rightarrow \alpha_{27} \text{ list} \sim \alpha_{26} \rightarrow \alpha_{28} \wedge C_{inner}, \Gamma_e \vdash (\text{single } (\text{single } y)) : \alpha_{28}}$$

The type of the outer lambda is therefore  $\alpha_{20} \rightarrow \alpha_{28}$ , with constraint  $C' \wedge C_b$ , which is

$$\mathbf{T} \wedge \alpha_{27} \rightarrow \alpha_{27} \text{ list} \sim \alpha_{26} \rightarrow \alpha_{28} \wedge \alpha_{25} \rightarrow \alpha_{25} \text{ list} \sim \alpha_{20} \rightarrow \alpha_{26}.$$

This constraint is equivalent to

$$\alpha_{27} \sim \alpha_{26} \wedge \alpha_{27} \text{ list} \sim \alpha_{28} \wedge \alpha_{25} \sim \alpha_{20} \wedge \alpha_{25} \text{ list} \sim \alpha_{26},$$

which is solved by the substitution

$$(\alpha_{27} \mapsto \alpha_{20} \text{ list}) \circ (\alpha_{28} \mapsto \alpha_{27} \text{ list}) \circ (\alpha_{25} \mapsto \alpha_{20}) \circ (\alpha_{26} \mapsto \alpha_{25} \text{ list}),$$

which is equivalent to the substitution

$$(\alpha_{27} \mapsto \alpha_{20} \text{ list}) \circ (\alpha_{28} \mapsto \alpha_{20} \text{ list list}) \circ (\alpha_{25} \mapsto \alpha_{20}) \circ (\alpha_{26} \mapsto \alpha_{20} \text{ list}),$$

The type of the outer lambda is therefore  $\alpha_{20} \rightarrow \alpha_{20} \text{ list list}$ , and at the `VAL` binding, this type is generalized to the type scheme  $\forall \alpha_{20}. \alpha_{20} \rightarrow \alpha_{20} \text{ list list}$ .

### 7.5.3 Solving constraints

As shown by the examples above, the method of explicit constraints reduces type inference to a constraint-solving problem. A fresh type variable that is introduced to represent an unknown type may get constrained, and by solving the constraints, we learn what (if anything) to substitute for each type variable. Substitutions play a crucial role, but they are used only to infer types at `LET` and `VAL`, where potentially polymorphic names are bound. In this section we explore all the details. I hope you use the ideas to implement your own constraint solver.

We begin by defining when a constraint is “satisfied”:

$$\frac{\tau_1 = \tau_2}{\tau_1 \sim \tau_2 \text{ is satisfied}} \quad \frac{\begin{array}{c} C_1 \text{ is satisfied} \quad C_2 \text{ is satisfied} \\ \hline C_1 \wedge C_2 \text{ is satisfied} \end{array}}{\text{T is satisfied}}$$

Constraints, like types, can be converted to other constraints by substitution: when we apply a substitution to a constraint, we may replace type variables in that constraint. A substitution is applied to a constraint using the following algebraic laws:

$$\theta(\tau_1 \sim \tau_2) = \theta\tau_1 \sim \theta\tau_2 \quad \theta(C_1 \wedge C_2) = \theta C_1 \wedge \theta C_2 \quad \theta\mathbf{T} = \mathbf{T}$$

The substitution function is implemented in chunk 447b, and the constraints’ ML representation is shown in chunk 446e.

We *solve* a constraint  $C$  by finding a substitution  $\theta$  such that  $\theta C$  is satisfied; we say that  $\theta$  *solves*  $C$ . Not all constraints can be solved; for example, there is no  $\theta$  that solves `int ~ bool`, and there is no  $\theta$  that solves  $\alpha ~ \alpha \text{ list}$ .

As Exercise 4 on page 456 asks you to show, satisfaction is preserved by substitution. Formally, if  $\theta C$  is satisfied, and if  $\theta'$  is another substitution, then  $\theta'(\theta C)$  is also satisfied. We might also write that  $(\theta' \circ \theta)C$  is satisfied.

At bottom, constraint solving is the same problem as unification (page 428). If you have a constraint solver, then any substitution that solves the constraint  $\tau_1 \sim \tau_2$  is a unifier of  $\tau_1$  and  $\tau_2$ . And if you have a unifier, you can use it to solve constraints—the exact trick is the subject of Exercise 20 on page 460.

A constraint solver is given a constraint  $C$  and returns a substitution  $\theta$  such that  $\theta C$  is satisfied. The solver need consider only three cases:

- $C$  is the trivial constraint  $\mathbf{T}$ , in which case  $\theta_I$  solves  $C$
- $C$  is the conjunction  $C_1 \wedge C_2$ , in which case both sub-constraints  $C_1$  and  $C_2$  must be solved
- $C$  is a simple equality constraint of the form  $\tau_1 \sim \tau_2$

Both conjunctions and simple equality constraints require attention to detail.

Wouldn't it be great if you could solve a conjunction  $C_1 \wedge C_2$  simply by solving  $C_1$ , then solving  $C_2$ , then composing the solutions? It's a pity you can't. Here's a useless rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \quad \tilde{\theta}_2 C_2 \text{ is satisfied}}{(\tilde{\theta}_2 \circ \theta_1)(C_1 \wedge C_2) \text{ may or may not be satisfied}} \quad (\text{UNSOLVEDCONJUNCTION})$$

Even when  $\theta_1$  solves  $C_1$  and  $\tilde{\theta}_2$  solves  $C_2$ , neither  $\tilde{\theta}_2 \circ \theta_1$  nor  $\theta_1 \circ \tilde{\theta}_2$  is guaranteed to solve  $C_1 \wedge C_2$ . I've put a tilde on the substitution  $\tilde{\theta}_2$  because I'm going to treat it as the bad guy; looking at  $\tilde{\theta}_2 \circ \theta_1$ , here's what goes wrong:

- We want  $(\tilde{\theta}_2 \circ \theta_1)(C_1 \wedge C_2)$  to be satisfied. According the rule for satisfying conjunctions, this means both  $(\tilde{\theta}_2 \circ \theta_1)C_1$  and  $(\tilde{\theta}_2 \circ \theta_1)C_2$  must be satisfied.
- By the assumption that  $\theta_1$  solves  $C_1$ ,  $\theta_1 C_1$  is satisfied. And because satisfaction is preserved by substitution,  $(\tilde{\theta}_2 \circ \theta_1)C_1$  is satisfied. So far, so good.
- Constraint  $(\tilde{\theta}_2 \circ \theta_1)C_2 = \tilde{\theta}_2(\theta_1 C_2)$  must also be satisfied. But unfortunately, all we know about  $\tilde{\theta}_2$  is that it solves  $C_2$ , and this information isn't enough to guarantee that it also solves  $\theta_1 C_2$ .

This line of thinking can be carried further by completing Exercises 12 and 16 on page 458, which ask you for a proof and some examples.

What goes wrong here is that in the last step the constraint that must be solved is not  $C_2$ ; it is  $\theta_1 C_2$ . This observation leads to a technique that works, which I express here as a useful inference rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \quad \theta_2(\theta_1 C_2) \text{ is satisfied}}{(\theta_2 \circ \theta_1)(C_1 \wedge C_2) \text{ is satisfied}} \quad (\text{SOLVEDCONJUNCTION})$$

Substitution  $\theta_2$ , which may be different from  $\tilde{\theta}_2$ , does what we need. The rule has an operational interpretation involving a recursive call to the solver, the application of substitution  $\theta_1$ , another recursive call to the solver, and finally the composition of two substitutions. This rule is sound (Exercise 13 on page 458). And so its operational interpretation works as a solver.

I think about the solver like this: if  $\theta_1$  solves  $C_1$ , then  $\theta_1$  represents the *assumptions* that I have to make for  $C_1$  to have a solution. If those assumptions are to hold everywhere, then they somehow must be accounted for when I look at  $C_2$ . And the way to account for assumptions is to *apply* the corresponding substitution.

## Solving simple equality constraints

The last remaining case for a constraint solver is a simple equality constraint of the form  $\tau_1 \sim \tau_2$ . I call this case “simple,” but the code is far from simple: because there are three ways to form a type, there are *nine* possible ways to form a simple equality constraint. Nine cases is a lot to code, but if you organize your code carefully, you can use ML pattern matching to cut them down. And no matter what case you are considering, the goal is always the same: find a  $\theta$  such that  $\theta\tau_1 = \theta\tau_2$ . Let’s tackle the most tricky case first, the easy cases next, and the most involved case last.

The tricky case is one in which the left-hand side is a type variable, giving the constraint the form  $\alpha \sim \tau_2$ . You might hope that this constraint is always solved by the substitution  $(\alpha \mapsto \tau_2)$ . But it works only in some cases:

- If  $\tau_2$  does not mention  $\alpha$ , then  $(\alpha \mapsto \tau_2)\tau_2 = \tau_2$ , and also  $(\alpha \mapsto \tau_2)\alpha = \tau_2$ . Solved!
- If  $\tau_2$  is *equal* to  $\alpha$ , then  $(\alpha \mapsto \alpha)$  is the identity substitution  $\theta_I$ . Also solved.
- If  $\tau_2$  *mentions*  $\alpha$  but is *not equal* to  $\alpha$ —for example, suppose  $\tau_2$  is  $\alpha$  list—then the constraint  $\alpha \sim \tau_2$  has no solution. (The proof is the subject of Exercise 15.)

Type  $\tau_2$  mentions  $\alpha$  if and only if  $\alpha$  occurs free in  $\tau_2$ . The part of your code that checks to see if this happens is called the *occurs check*.

Suppose the left-hand side of the constraint is not a type variable, but the right-hand side is. That is, we have a constraint of the form  $\tau_1 \sim \alpha$ . Well, this constraint has exactly the same solutions as  $\alpha \sim \tau_1$ , and you can just swap the two sides and make a recursive call to your solver.

If neither side is a type variable, then what we are left with is combinations of type constructors and type applications (TYCON and CONAPP). Because every substitution maps constructors to constructors and applications to applications, we needn’t think hard about constraints that equate a TYCON with a CONAPP: they can’t be solved. We also don’t have to think hard if each side of the constraint holds a type constructor: every substitution leaves every constructor unchanged, so a constraint of the form  $\mu \sim \mu$  is solved by the identity substitution, and a constraint of the form  $\mu \sim \mu'$ , where  $\mu \neq \mu'$ , has no solution.

The most complicated case is a constraint in which both sides are constructor applications. Because every substitution must preserve the structure of a constructor application (see the substitution laws on page 420), we can break down such a constraint into a conjunction of smaller constraints:

$$\frac{\theta(\tau \sim \tau' \wedge \tau_1 \sim \tau'_1 \wedge \cdots \wedge \tau_n \sim \tau'_n) \text{ is satisfied}}{\theta(\text{CONAPP}(\tau, \langle \tau_1, \dots, \tau_n \rangle) \sim \text{CONAPP}(\tau', \langle \tau'_1, \dots, \tau'_n \rangle)) \text{ is satisfied}} \quad (\text{SOLVECONAPPCONAPP})$$

This rule is also sound (Exercise 14 on page 459). Operationally, the solver is given the constraint on the bottom. It then builds the constraint on the top, on which it calls itself recursively.

## An example of constraint solving

Using the ideas above, let’s solve the constraint from (cons 1 '()),

$$\mathbf{T} \wedge (\mathbf{T} \wedge \alpha_{11} \times \alpha_{11} \text{ list} \rightarrow \alpha_{11} \text{ list} \sim \text{int} \times \alpha_{12} \text{ list} \rightarrow \alpha_{13}). \quad (7.1)$$

This constraint is big enough to be interesting, but for a complete, formal derivation, it’s a little too big. So let’s solve it informally.

Nano-ML’s type inference is designed for clarity, not performance. If you write a definition that includes hundreds of function applications, you get hundreds of constraints. Solving them may result in hundreds of *thousands* of calls to `tysubst`, which in turn may mean tens of *millions* of string comparisons in `find`—the total cost of environment lookup is quadratic in the number of type variables, and type inference can take hours. Not good.

A production compiler doesn’t have such issues. Production code uses efficient representations of names and environments, and substitution is implemented in constant time by updating the contents of a mutable reference cell, much as in a union-find algorithm.

To avoid nano-ML’s performance issues, avoid deeply nested calls. For example, if you want to build a list of a thousand elements, try something like this:

```
(let*
  ([ns '()]
   [ns (cons 1000 ns)]
   [ns (cons 999 ns)]
   ...
   [ns (cons 1 ns)])
  ns)
```

At every binding the constraint solver is called with a small constraint involving just a few type variables, so type inference is quick.

Alternatively, type inference of large lists can be made efficient by better language design: a *list constructor* builds a list of any length using just a single type variable, which represents the element type of the list (Exercise 22 on page 460).

The constraint is a conjunction, so we first solve the left conjunct, which is  $\mathbf{T}$ . This conjunct is solved by the identity substitution  $\theta_I$ , which we then apply to the right conjunct

$$\mathbf{T} \wedge \alpha_{11} \times \alpha_{11} \text{ list} \rightarrow \alpha_{11} \text{ list} \sim \text{int} \times \alpha_{12} \text{ list} \rightarrow \alpha_{13}. \quad (7.2)$$

The identity substitution leaves this conjunct unchanged, and we continue solving recursively. The exact same set of steps lead us to solve

$$\alpha_{11} \times \alpha_{11} \text{ list} \rightarrow \alpha_{11} \text{ list} \sim \text{int} \times \alpha_{12} \text{ list} \rightarrow \alpha_{13}. \quad (7.3)$$

At this point the final case for a simple equality constraint comes into play: on both sides, we have an application of the  $\rightarrow$  constructor. We use the `SOLVECONAPP` rule to convert this constraint to

$$(\rightarrow \sim \rightarrow) \wedge (\alpha_{11} \times \alpha_{11} \text{ list} \sim \text{int} \times \alpha_{12} \text{ list} \wedge \alpha_{11} \text{ list} \sim \alpha_{13}). \quad (7.4)$$

The left conjunct,  $(\rightarrow \sim \rightarrow)$ , has two equal type constructors and so is solved by  $\theta_I$  which, when applied to the right conjunct, leaves it unchanged. So we solve

$$\alpha_{11} \times \alpha_{11} \text{ list} \sim \text{int} \times \alpha_{12} \text{ list} \wedge \alpha_{11} \text{ list} \sim \alpha_{13}. \quad (7.5)$$

The next step is to solve the left conjunct

$$\alpha_{11} \times \alpha_{11} \text{ list} \sim \text{int} \times \alpha_{12} \text{ list}, \quad (7.6)$$

which requires another application of the SOLVECONAPP rule, asking us to solve

$$(\times \sim \times) \wedge \alpha_{11} \sim \text{int} \wedge \alpha_{11} \text{ list} \sim \alpha_{12} \text{ list}. \quad (7.7)$$

We must next solve

$$\alpha_{11} \sim \text{int} \wedge \alpha_{11} \text{ list} \sim \alpha_{12} \text{ list}, \quad (7.8)$$

and we begin with its left conjunct

$$\alpha_{11} \sim \text{int}. \quad (7.9)$$

Finally we have a case with a type variable on the left, and constraint 7.9 is solved by  $\theta_1 = \alpha_{11} \mapsto \text{int}$ . We then apply  $\theta_1$  to the constraint  $\alpha_{11} \text{ list} \sim \alpha_{12} \text{ list}$ , yielding

$$\text{int list} \sim \alpha_{12} \text{ list}. \quad (7.10)$$

Let's not go through all the steps; constraint 7.10 is solved by  $\theta_2 = \alpha_{12} \mapsto \text{int}$ . Constraints 7.6, 7.7, and 7.8 are therefore solved by the composition of  $\theta_1$  and  $\theta_2$ , which is  $\theta_2 \circ \theta_1 = (\alpha_{12} \mapsto \text{int} \circ \alpha_{11} \mapsto \text{int})$ .

Now we can return to constraint 7.5. We apply  $\theta_2 \circ \theta_1$  to the right conjunct  $\alpha_{11} \text{ list} \sim \alpha_{13}$ , yielding

$$\text{int list} \sim \alpha_{13}, \quad (7.11)$$

which is solved by substitution  $\theta_3 = \alpha_{13} \mapsto \text{int list}$ . The solution to constraint 7.5 is therefore  $\theta_3 \circ \theta_2 \circ \theta_1$ , which is

$$\theta = \theta_3 \circ \theta_2 \circ \theta_1 = \alpha_{13} \mapsto \text{int list} \circ \alpha_{12} \mapsto \text{int} \circ \alpha_{11} \mapsto \text{int}.$$

Substitution  $\theta$  also solves constraints 7.1 to 7.4.

## 7.6 THE INTERPRETER

Most of our interpreter for nano-ML is just like Chapter 5's interpreter for  $\mu$ Scheme. We just add type inference. Significant parts of the implementation of type inference don't appear here, however, because they are left as Exercises.

### 7.6.1 Functions on types and type schemes

This section defines functions that are used throughout type inference.

Function `freetyvars` computes the free type variables of a type. For readability, I ensure that type variables appear in the set in the order of their first appearance in the type, when reading from left to right.

**442.**  $\langle \text{sets of free type variables in Hindley-Milner types} \rangle \equiv$  (S408c)

`freetyvars : ty -> name set`

```
fun freetyvars t =
  let fun f (TYVAR v,           ftvs) = insert (v, ftvs)
       | f (TYCON _,            ftvs) = ftvs
       | f (CONAPP (ty, tys), ftvs) = foldl f (f (ty, ftvs)) tys
  in  reverse (f (t, emptyset))
  end
```

Type system	Concept	Interpreter	
$d$	Definition	<code>def</code> (page 415)	
$e$	Expression	<code>exp</code> (page 414)	
$x$	Variable	<code>name</code> (page 310)	
$\alpha$	Type variable	<code>tyvar</code> (page 418)	§7.6
$\tau$	Type	<code>ty</code> (page 418)	<i>The interpreter</i>
$\sigma, \forall \alpha.\tau$	Type scheme	<code>type_scheme</code> (page 418)	443
$\tau \equiv \tau'$	Type equivalence	<code>eqType</code> ( $\tau, \tau'$ ) (page 422)	
$\Gamma$	Type environment	<code>type_env</code> (page 446)	
$\Gamma(x) = \sigma$	Type lookup	<code>findtyscheme</code> ( $x, \Gamma$ ) = $\sigma$ (page 446)	
$\Gamma\{x \mapsto \sigma\}$	Type binding	<code>bindtyscheme</code> ( $x, \sigma, \Gamma$ ) (page 446)	
$C$	Constraint	<code>con</code> (page 446)	
$\tau_1 \sim \tau_2$	Equality constraint	$\tau_1 \sim \tau_2$ (page 446)	
$C_1 \wedge C_2$	Conjunction	$C_1 \wedge C_2$ (page 446)	
$\mathbf{T}$	Trivial constraint	<code>TRIVIAL</code> (page 446)	
$\bigwedge_i C_i$	Conjunction	<code>conjoinConstraints</code> [ $C_1, \dots, C_n$ ] (page 447)	
$\text{ftv}(\tau)$	Free type variables	<code>freetyvars</code> $\tau$ (page 442)	
$\text{ftv}(\Gamma)$	Free type variables	<code>freetyvarsGamma</code> $\Gamma$ (page 446)	
$\text{ftv}(C)$	Free type variables	<code>freetyvarsConstraint</code> $C$ (page 447)	
$C, \Gamma \vdash e : \tau$	Type inference	<code>typeof</code> ( $e, \Gamma$ ) = ( $\tau, C$ ), also <code>ty</code> $e = (\tau, C)$ (page 448; some parts left as an exercise)	
$\langle d, \Gamma \rangle \rightarrow \Gamma'$	Type inference	<code>typedef</code> ( $d, \Gamma$ ) = ( $\Gamma', s$ ) (page 449)	
$\theta$	A substitution	<code>subst</code> (page 420)	
$\theta_I$	Identity substitution	<code>idsubst</code> (page 422)	
$[\alpha \mapsto \tau]$	Substitution for $\alpha$	$\alpha \mid\!\!-\!\!\!> \tau$ (page 421)	
$\theta\tau$	Substitution	<code>tysubst</code> $\theta \tau$ (page 421)	
$\theta\alpha$	Substitution	<code>vars subst</code> $\theta \alpha$ (page 421)	
$\theta C$	Substitution	<code>cons subst</code> $\theta C$ (page 447)	
$\theta_2 \circ \theta_1$	Composition	<code>compose</code> (page 421)	
$\text{dom } \theta$	Domain	<code>dom</code> (page 421)	
$\theta C \equiv \mathbf{T}$	Constraint solving	$\theta = \text{solve } C$ (left as an exercise, page 448)	
$C \equiv \mathbf{T}$	Solved constraint	<code>isSolved</code> $C$ (page 448)	
$\forall \alpha.\tau$ becomes $\tau[\alpha \mapsto \tau']$	Instantiation	<code>instantiate</code> ( $\forall \alpha.\tau, [\tau']$ ) (page 421)	
$\text{int}, \text{bool}, \dots$	Base types	<code>inttype</code> , <code>booleatype</code> , ... (page 422)	
$\tau_1 \times \dots \times \tau_n \rightarrow \tau$	Function type	<code>funtype</code> ( $[\tau_1, \dots, \tau_n], \tau$ ) (page 422)	
			CONAPP 418 emptyset S240b insert S240b reverse S241c TYCON 418 TYVAR 418

Table 7.3: Correspondence between nano-ML’s type system and code

*Canonical type schemes*

Type variables like '`t136`' are not suitable for use in error messages. A type scheme like `(forall ['t136] ((list 't136) -> int))` is unpleasant to look at, and it is equivalent to `(forall ['a] ((list 'a) -> int))` (when a type variable is  $\forall$ -bound, its name is irrelevant). For readability, we are better off using names like '`a`', '`b`', and so on. Function `canonicalize` renames bound type variables in a type scheme. It replaces each existing bound type variable with a canonically named type variable, taking care not to use the name of any free type variable.

**444a.** *(shared utility functions on Hindley-Milner types 420)*  $\doteqdot$

(S408c) □ 422b 444b ▷

```
canonicalize : type_scheme -> type_scheme
newBoundVars : int * name list -> name list

fun canonicalize (FORALL (bound, ty)) =
  let fun canonicalTyvarName n =
    if n < 26 then "!" ^ str (chr (ord "#" ^ "a" + n))
    else "v" ^ intString (n - 25)
  val free = diff (freetyvars ty, bound)
  fun unusedIndex n =
    if member (canonicalTyvarName n) free then unusedIndex (n+1) else n
  fun newBoundVars (index, []) = []
  | newBoundVars (index, oldvar :: oldvars) =
    let val n = unusedIndex index
      in canonicalTyvarName n :: newBoundVars (n+1, oldvars)
    end
  val newBound = newBoundVars (0, bound)
  in FORALL (newBound,
             tysubst (bindList (bound, map TYVAR newBound, emptyEnv)) ty)
  end
```

Internal function `unusedIndex` finds a name for a bound type variable; it ensures that the name is not the name of any free type variable.

*Fresh type variables*

A type variable that does not appear in any type environment or substitution is called *fresh*. When inferring the type of a function, we need fresh type variables to stand for the (unknown) types of the arguments. When instantiating a polytype, we need fresh type variables to stand for the (unknown) types at which the polytype is instantiated. And when inferring the type of a function application, we need a fresh type variable to stand for the return type.

In my code, fresh type variables come from the `freshtyvar` function. I use a private mutable counter to supply an arbitrary number of type variables of the form `tn`. Because a nano-ML expression or definition does not contain any explicit type variables, the names won't collide with other names.

**444b.** *(shared utility functions on Hindley-Milner types 420)*  $\doteqdot$

(S408c) □ 444a 445a ▷

```
local
  val n = ref 1
in
  fun freshtyvar _ = TYVAR ("'" ^ "t" ^ intString (!n) before n := !n + 1)
end
```

*Generalization and instantiation*

Calling `generalize( $\tau$ ,  $\mathcal{A}$ )` generalizes type  $\tau$  to a type scheme by closing over type variables not in  $\mathcal{A}$ . It also puts the type scheme into canonical form.

## Why generalize and instantiate?

We use quantified types (i.e., type schemes) so we can instantiate them when we look them up in an environment. Instantiation gives us the full effect of polymorphism. Without instantiation, we wouldn't be able to type such ML terms as `(1::nil, true::nil)`. Suppose we had an environment  $\Gamma$  with only types, not type schemes:

$$\Gamma = \{1 : \text{int}, \text{true} : \text{bool}, \text{nil} : \alpha \text{ list}, :: : \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list}\}$$

When type-checking `1::nil`, we would get the constraint  $\alpha \sim \text{int}$ . And when type-checking `true::nil`, we would get the constraint  $\alpha \sim \text{bool}$ . But the conjunction  $\alpha \sim \text{int} \wedge \alpha \sim \text{bool}$  has no solution, and type checking would fail.

Instead, we use `freshInstance` to make sure that every use of a polymorphic value (here `::` and `nil`) has a type different from any other instance. In order to make that work, the environment has to contain polytypes:

$$\Gamma = \{1 : \forall.\text{int}, \text{true} : \forall.\text{bool}, \text{nil} : \forall\alpha.\alpha \text{ list}, :: : \forall\alpha.\alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list}\}$$

Now, we can imagine our sample term like this, writing `::` as a prefix operator so as to show the types:

```
(op :: : 't121 * 't121 list -> 't121 list (1    : int, nil : 't122 list),
 op :: : 't123 * 't123 list -> 't123 list (true : bool, nil : 't124 list))
```

The constraint ' $t121 \sim \text{int} \wedge \text{int} \sim t122 \wedge t123 \sim \text{bool} \wedge \text{bool} \sim t124$ ' does have a solution, and the whole term has the type `int list * bool list`, as desired.

**445a.** *(shared utility functions on Hindley-Milner types 420)*  $\equiv$  (S408c) ▷ 444b 445b ▷

```
fun generalize (tau, tyvars) = [generalize : ty * name set -> type_scheme]
  canonicalize (FORALL (diff (freetyvars tau, tyvars), tau))
```

The dual function, `instantiate`, is defined in chunk 421c. It requires a list of types with which to instantiate, but the common case is to instantiate with fresh type variables. Function `freshInstance` implements this case.

**445b.** *(shared utility functions on Hindley-Milner types 420)*  $\equiv$  (S408c) ▷ 445a ▷

```
fun freshInstance (FORALL (bound, tau)) =
  instantiate (FORALL (bound, tau), map freshtyvar bound)
```

### 7.6.2 Type environments

When we call `generalize`, we use the free type variables of some type environment. And a type environment contains the type of every name ever defined in a nano-ML program, so it can get big. I therefore pay some attention to efficiency. In particular, instead of searching an entire type environment every time I want free type variables, I pay a little extra when I *extend* a type environment—and I find free type variables in constant time.

Here are the operations we perform on type environments:

- Function `bindtyscheme` adds a binding  $x : \sigma$  to the environment  $\Gamma$ . We use `bindtyscheme` to implement the LAMBDA rule and the various LET rules.

bindList	312c
diff	S240b
emptyEnv	311a
FORALL	418
freetyvars	442
instantiate	421c
intString	S238f
member	S240b
tysubst	421a
TYVAR	418

- Function `findtyscheme` looks up a variable  $x$  to find  $\sigma$  such that  $\Gamma(x) = \sigma$ . We use `findtyscheme` to implement the VAR rule.
- Function `freetyvarsGamma` finds the type variables free in  $\Gamma$ , i.e., the type variables free in any  $\sigma$  in  $\Gamma$ . We need `freetyvarsGamma` to get a set of free type variables to use in `generalize`; when we assign a type scheme to a let-bound variable, only those type variables not free in  $\Gamma$  may be  $\forall$ -bound.

If `freetyvarsGamma` were implemented using the obvious representation, of type `type_scheme env`, it would visit every type scheme in every binding in the environment. Because most bindings contribute no free type variables, most visits would be unnecessary. I therefore implement an optimization: with every type environment, I keep a cache of the free type variables. A type environment is represented as follows:

**446a.** *(specialized environments for type schemes 446a)*  $\equiv$

(S408c) 446b ▷

```
type type_env = type_scheme env * name set
```

An empty type environment binds no variables and has no free type variables. Looking up a type scheme ignores the free variables.

**446b.** *(specialized environments for type schemes 446a)*  $+ \equiv$

(S408c) ◁ 446a 446c ▷

val emptyTypeEnv =	emptyTypeEnv : type_env
(emptyEnv, emptyset)	findtyscheme : name * type_env -> type_scheme
fun findtyscheme (x, (Gamma, free)) =	find (x, Gamma)

When adding a new binding, I update the set of free type variables in  $\Gamma$ . I take the union of the existing free type variables with the free type variables of the new type scheme  $\sigma$ .

**446c.** *(specialized environments for type schemes 446a)*  $+ \equiv$

(S408c) ◁ 446b 446d ▷

bindtyscheme : name * type_scheme * type_env -> type_env
--

```
fun bindtyscheme (x, sigma as FORALL (bound, tau), (Gamma, free)) =
    (bind (x, sigma, Gamma), union (diff (freetyvars tau, bound), free))
```

Finally, when we want the free type variables, we just take them from the pair.

**446d.** *(specialized environments for type schemes 446a)*  $+ \equiv$

(S408c) ◁ 446c ▷

freetyvarsGamma : type_env -> name set
--

```
fun freetyvarsGamma (_, free) = free
```

### 7.6.3 Constraints and constraint solving

To highlight the relationship between the code and the math, I use a representation that's close to the math: the  $\sim$  operator is  $\sim$ ; the  $\wedge$  operator is  $/\wedge$ ; and the T constraint is TRIVIAL.

**446e.** *(representation of type constraints 446e)*  $\equiv$

(S405e)

```
datatype con = ~ of ty * ty
             | /\ $\wedge$  of con * con
             | TRIVIAL
infix 4 ~
infix 3 /\ $\wedge$ 
```

(You may know  $\sim$  as ML's negation operator. My code redefines  $\sim$ , but negation can still be referred to by its qualified name `Int. $\sim$` .)

Many of the utility functions defined on types have counterparts on constraints. For example, we can find free type variables in a constraint, and we can substitute for free type variables.

**447a.** *(utility functions on type constraints 447a)*  $\equiv$

(S405e) 447b▷

```
fun freetyvarsConstraint (t ~ t') = union (freetyvars t, freetyvars t')
| freetyvarsConstraint (c /\ c') = union (freetyvarsConstraint c,
                                         freetyvarsConstraint c')
| freetyvarsConstraint TRIVIAL = emptyset
```

§7.6

The interpreter

447

A substitution is applied to a constraint using the following rules:

$$\theta(\tau_1 \sim \tau_2) = \theta\tau_1 \sim \theta\tau_2 \quad \theta(C_1 \wedge C_2) = \theta C_1 \wedge \theta C_2 \quad \theta\mathbf{T} = \mathbf{T}$$

The code resembles the code for `tysubst` in chunk 421a.

**447b.** *(utility functions on type constraints 447a)*  $\equiv$

(S405e) ▷447a 447c▷

```
fun consubst theta =
  let fun subst (tau1 ~ tau2) = tysubst theta tau1 ~ tysubst theta tau2
  | subst (c1 /\ c2) = subst c1 /\ subst c2
  | subst TRIVIAL = TRIVIAL
  in subst
  end
```

I implement the  $\wedge\{\dots\}$  operator using the ML function `conjoinConstraints`. Because I want to preserve the number and order of sub-constraints, I avoid using `foldl` or `foldr`.

**447c.** *(utility functions on type constraints 447a)*  $\equiv$

(S405e) ▷447b 448b▷

```
fun conjoinConstraints [] = TRIVIAL
| conjoinConstraints [c] = c
| conjoinConstraints (c::cs) = c /\ conjoinConstraints cs
```

Two more functions are defined in Appendix R. Function `constraintString` can be used to print constraints, and `untriviate`, whose type is `con -> con`, removes trivial conjuncts from a constraint.

### Constraint solving

If type inference is given an ill-typed program, it produces an *unsolvable* constraint. Examples of unsolvable constraints include `int ~ bool` and  $\alpha$  `list ~ \alpha. Given such a constraint, we want to issue a readable error message, not one full of machine-generated type variables. To do so, function unsatisfiableEquality takes the pair of types that can't be made equal, puts the pair into canonical form, and raises the TypeError exception.`

**447d.** *(constraint solving 447d)*  $\equiv$

(S405e)

```
fun unsatisfiableEquality (t1, t2) =
let val t1_arrow_t2 = funtype ([t1], t2)
  val FORALL (_, canonical) =
    canonicalize (FORALL (freetyvars t1_arrow_t2, t1_arrow_t2))
in case asFuntype canonical
  of SOME ([t1'], t2') =>
      raise TypeError ("cannot make " ^ typeString t1' ^
                      " equal to " ^ typeString t2')
  | _ => let exception Can'tHappen in raise Can'tHappen end
end
```

asFuntype,	
in nano-ML	422c
in $\mu$ ML	S423d
bind	312b
canonicalize	444a
diff	S240b
emptyEnv	311a
emptyset	S240b
type env	310b
find	311b
FORALL	418
freetyvars	442
funtype,	
in nano-ML	422c
in $\mu$ ML	S423d
type name	310a
type ty	418
type type_scheme	
418	
TypeError	S237c
typeString	S411d
tysubst	421a
union	S240b

The implementation of `unsatisfiableEquality` is a little weird. To make a single type out of  $\tau_1$  and  $\tau_2$ , so their variables can be canonicalized together, I make the type  $\tau_1 \rightarrow \tau_2$ . What's weird is that there's no function here—it's just a device to make one type out of two. When I get the canonical version, I immediately take it apart, getting back canonical types  $\tau_1'$  and  $\tau_2'$ .

I don't provide a solver; I'm hoping you will implement one.

**448a.** *(constraint solving [prototype] 448a)≡*

```
fun solve c = raise LeftAsExercise "solve"
```

solve : con -> subst
----------------------

For debugging, it can be useful to see if a substitution solves a constraint.

**448b.** *(utility functions on type constraints 447a) +≡*

(S405e) ▷ 447c

```
fun isSolved TRIVIAL = true
  | isSolved (tau ~ tau') = eqType (tau, tau')
  | isSolved (c /\ c') = isSolved c andalso isSolved c'
fun solves (theta, c) = isSolved (cons subst theta c)
```

isSolved : con -> bool
solves : subst * con -> bool

#### 7.6.4 Type inference

Type inference builds on constraint solving. With constraints in place, all we need is `typeof`, which implements the typing rules for expressions, and `typdef`, which implements the rules for definitions.

##### Type inference for expressions

Given an expression  $e$  and type environment  $\Gamma$ , function `typeof(e, Γ)` returns a pair  $(\tau, C)$  such that  $C, \Gamma \vdash e : \tau$ . It uses internal functions `typesof`, `literal`, and `ty`.

**448c.** *(definitions of typeof and typdef for nano-ML and μML 448c)≡*

(S405e) ▷ 449f▷

typeof : exp * type_env -> ty * con
typesof : exp list * type_env -> ty list * con
literal : value -> ty * con
ty : exp -> ty * con

```
fun typeof (e, Gamma) =
let
  shared definition of typesof, to infer the types of a list of expressions 448d
  function literal, to infer the type of a literal constant (left as an exercise)
  function ty, to infer the type of a nano-ML expression, given Gamma 449a
in
  ty e
end
```

Calling `typesof(( $e_1, \dots, e_n$ ),  $\Gamma$ )` returns  $((\tau_1, \dots, \tau_n), C)$  such that for every  $i$  from 1 to  $n$ ,  $C, \Gamma \vdash e_i : \tau_i$ . The base case is trivial; the induction step uses this rule from Section 7.5.2:

$$\frac{C_1, \Gamma \vdash e_1 : \tau_1 \quad \dots \quad C_n, \Gamma \vdash e_n : \tau_n}{C_1 \wedge \dots \wedge C_n, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n} \quad (\text{TYPESOF})$$

**448d.** *(shared definition of typesof, to infer the types of a list of expressions 448d)≡*

(448c)

```
fun typesof ([] , Gamma) = ([] , TRIVIAL)
  | typesof (e::es, Gamma) =
    let val (tau, c) = typeof (e, Gamma)
        val (taus, c') = typesof (es, Gamma)
    in (tau :: taus, c /\ c')
    end
```

To infer the type of a literal value, we call `literal`, which is left as Exercise 19.

**448e.** *(function literal, to infer the type of a literal constant [prototype] 448e)≡*

```
fun literal _ = raise LeftAsExercise "literal"
```

449a. *(function ty, to infer the type of a nano-ML expression, given Gamma 449a)*  $\equiv$  (448c)

```
fun ty (LITERAL n) = literal n
  ⟨more alternatives for ty 449b⟩
```

To infer the type of a variable, we use fresh type variables to create a most general instance of the variable's type scheme in  $\Gamma$ . No constraint is needed.

449b. *(more alternatives for ty 449b)*  $\equiv$  (449a) 449c  $\triangleright$

```
| ty (VAR x) = (freshInstance (findtyscheme (x, Gamma)), TRIVIAL)
```

Section 7.5.2 shows how to rewrite a nondeterministic type rule to introduce constraints; here is the rule for application:

$$\frac{C, \Gamma \vdash e, e_1, \dots, e_n : \hat{\tau}, \tau_1, \dots, \tau_n \quad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \sim \tau_1 \times \dots \times \tau_n \rightarrow \alpha, \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \alpha} \quad (\text{APPLY})$$

To implement this rule, we let `funty` stand for  $\hat{\tau}$ , `actualtypes` stand for  $\tau_1, \dots, \tau_n$ , and `rettype` stand for  $\alpha$ . The first premise is implemented by a call to `typesof` and the second by a call to `freshtyvar`. The constraint is just as in the rule.

449c. *(more alternatives for ty 449b)*  $\doteqdot$  (449a)  $\triangleleft$  449b 449d  $\triangleright$

```
| ty (APPLY (f, actuals)) =
  (case typesof (f :: actuals, Gamma)
    of ([], _) => let exception Can'tHappen in raise Can'tHappen end
     | (funty :: actualtypes, c) =>
       let val rettype = freshtyvar ()
       in (rettype, c) /\ (funty ~ funtype (actualtypes, rettype)))
      end)
```

The LETSTAR form is most easily implemented as syntactic sugar, and the remaining cases for `ty` are left as exercises.

449d. *(more alternatives for ty 449b)*  $\doteqdot$  (449a)  $\triangleleft$  449c

```
| ty (LETX (LETSTAR, [], body)) = ty body
| ty (LETX (LETSTAR, (b :: bs), body)) =
  ty (LETX (LET, [b], LETX (LETSTAR, bs, body)))
```

449e. *(more alternatives for ty [prototype] 449e)*  $\equiv$

```
| ty (IFX (e1, e2, e3)) = raise LeftAsExercise "type for IFX"
| ty (BEGIN es) = raise LeftAsExercise "type for BEGIN"
| ty (LAMBDA (formals, body)) = raise LeftAsExercise "type for LAMBDA"
| ty (LETX (LET, bs, body)) = raise LeftAsExercise "type for LET"
| ty (LETX (LETREC, bs, body)) = raise LeftAsExercise "type for LETREC"
```

### Typing and type inference for definitions

Given a definition, we extend the top-level type environment. We infer the type of the thing defined, generalize the type to a type scheme, and add a binding to the environment. This step *types* the definition. To report to a user, we also return a string suitable for printing.

449f. *(definitions of typeof and typdef for nano-ML and μML 448c)*  $\doteqdot$  (S405e)  $\triangleleft$  448c

```
fun typdef (d, Gamma) = typdef : def * type_env -> type_env * string
  case d
  of VAL (x, e) => ⟨infer and bind type for VAL (x, e) for nano-ML 450a⟩
  | VALREC (x, e) => ⟨infer and bind type for VALREC (x, e) for nano-ML 450b⟩
  | EXP e => typdef (VAL ("it", e), Gamma)
  | DEFINE (x, lambda) => typdef (VALREC (x, LAMBDA lambda), Gamma)
  (extra case for typdef used only in μML S419e)
```

Forms EXP and DEFINE are syntactic sugar.

§7.6  
The interpreter

449

APPLY,	
in nano-ML	414
in $\mu$ ML	S421c
BEGIN,	
in nano-ML	414
in $\mu$ ML	S421c
cons subst	447b
DEFINE,	
in nano-ML	415a
in $\mu$ ML	S421d
eqType	422b
EXP,	
in nano-ML	415a
in $\mu$ ML	S421d
findtyscheme	446b
freshInstance	445b
freshtyvar	444b
funtype,	
in nano-ML	422c
in $\mu$ ML	S423d
IFX,	
in nano-ML	414
in $\mu$ ML	S421c
LAMBDA,	
in nano-ML	414
in $\mu$ ML	S421c
LeftAsExercise	
	S237a
LET,	
in nano-ML	414
in $\mu$ ML	S421c
LETREC,	
in nano-ML	414
in $\mu$ ML	S421c
LETSTAR,	
in nano-ML	414
in $\mu$ ML	S421c
LETX,	
in nano-ML	414
in $\mu$ ML	S421c
LITERAL,	
in nano-ML	414
in $\mu$ ML	S421c
TRIVIAL	
	446e
VAL,	
in nano-ML	415a
in $\mu$ ML	S421d
VALREC,	
in nano-ML	415a
in $\mu$ ML	S421d
VAR,	
in nano-ML	414
in $\mu$ ML	S421c

The cases for VAL and VALREC resemble each other. We begin with VAL, which computes a type and generalizes it.

$$\frac{\begin{array}{c} C, \Gamma \vdash e : \tau \\ \theta C \text{ is satisfied} \quad \theta\Gamma = \Gamma \\ \sigma = \text{generalize}(\theta\tau, \text{ftv}(\Gamma)) \end{array}}{\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad (\text{VAL})$$

**450a.** *(infer and bind type for VAL (x, e) for nano-ML 450a)*  $\equiv$

```
let val (tau, c) = typeof (e, Gamma)
  val theta    = solve c
  val sigma    = generalize (tysubst theta tau, freetyvarsGamma Gamma)
  in (bindtyscheme (x, sigma, Gamma), typeSchemeString sigma)
end
```

This code takes a big shortcut: we just assume that  $\theta\Gamma = \Gamma$ . How can we get away with this assumption? Because we can prove that a *top-level*  $\Gamma$  *never contains a free type variable*. This property guarantees that  $\theta\Gamma = \Gamma$  for any  $\theta$ . To prove it, try Exercise 10 on page 458.

The code for VALREC is a bit more complicated. We need an environment that binds  $x$  to  $\tau$ , but we don't yet know  $\tau$ . The original rule looks like this:

$$\frac{\begin{array}{c} \Gamma\{x \mapsto \tau\} \vdash e : \tau \quad \sigma = \text{generalize}(\tau, \text{ftv}(\Gamma)) \\ e \text{ has the form LAMBDA}(\dots) \end{array}}{\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad (\text{VALREC})$$

Here's a version with constraints:

$$\frac{\begin{array}{c} C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \quad \alpha \text{ is fresh} \\ \theta(C \wedge \alpha \sim \tau) \text{ is satisfied} \quad \theta\Gamma = \Gamma \\ \sigma = \text{generalize}(\theta\alpha, \text{ftv}(\Gamma)) \\ e \text{ has the form LAMBDA}(\dots) \end{array}}{\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad (\text{VALREC with constraints})$$

As usual, we introduce a fresh type variable to stand for  $\tau$ , then constrain it to be equal to the type of  $e$ .

**450b.** *(infer and bind type for VALREC (x, e) for nano-ML 450b)*  $\equiv$

```
let val alpha    = freshTyvar ()
  val Gamma'   = bindtyscheme (x, FORALL ([]), alpha), Gamma)
  val (tau, c) = typeof (e, Gamma')
  val theta    = solve (c /\
    alpha ~ tau)
  val sigma    = generalize (tysubst theta alpha, freetyvarsGamma Gamma)
  val ()       = (if e is not a LAMBDA, raise TypeError S405a)
  in (bindtyscheme (x, sigma, Gamma), typeSchemeString sigma)
end
```

### 7.6.5 Primitives

As in Typed  $\mu$ Scheme, each primitive has a value and a type. Most of nano-ML's primitives are defined exactly as they are in Typed  $\mu$ Scheme; only a few that are different are shown below. As in Chapters 5 and 6, the values are defined using higher-order functions unaryOp, binaryOp, and arithOp, which are defined in the Supplement. They are identical to the versions in Chapters 5 and 6, except that when there is an error, they raise the exception BugInTypeInference.

Each of the list primitives winds up with a polymorphic type scheme, but I don't write type schemes in the code. Instead, I give each primitive a type, and when

I install the primitive in the initial type environment, I generalize its type. Types are shorter and easier to read than type schemes, and calling `generalize` ensures that the type variables are properly quantified.

**451a.** *(primitives for nano-ML :: 451a)≡*

(S411b)

```

("null?", unaryOp (fn NIL => BOOLV true | _ => BOOLV false),
  funtype ([listtype alpha], booletype)) ::

("cons", binaryOp (fn (a, b) => PAIR (a, b)),
  funtype ([alpha, listtype alpha], listtype alpha)) ::

("car", unaryOp
  (fn (PAIR (car, _)) => car
   | NIL => raise RuntimeError "car applied to empty list"
   | _   => raise BugInTypeInference "car applied to non-list"),
  funtype ([listtype alpha], alpha)) ::

("cdr", unaryOp
  (fn (PAIR (_, cdr)) => cdr
   | NIL => raise RuntimeError "cdr applied to empty list"
   | _   => raise BugInTypeInference "cdr applied to non-list"),
  funtype ([listtype alpha], listtype alpha)) ::
```

§7.7

Hindley-Milner as  
it really is

451

The other primitive worth showing here is `error`. The type of `error`, which is  $\forall \alpha, \beta . \alpha \rightarrow \beta$ , tells us something interesting about its behavior. The type suggests that `error` can produce an arbitrary  $\beta$  without ever consuming one. Such a miracle is impossible; what the type tells us is that the `error` function never returns normally. In nano-ML this type means it either halts the interpreter or fails to terminate; in full ML, a function of this type could also raise an exception.

**451b.** *(primitives for nano-ML and μML :: 451b)≡*

(S411b)

```

("error", unaryOp (fn v => raise RuntimeError (valueString v)),
  funtype ([alpha], beta)) ::
```

The remaining primitives are relegated to the Supplement.

### 7.6.6 Predefined functions

Nano-ML's predefined functions are nearly identical to  $\mu$ Scheme's predefined functions, except for association lists. To represent an association list in nano-ML, we use a list of pairs, not a list of two-element lists. This representation is needed so that keys and values can have different types. And if a key in an association list is not bound, `find` can't return the empty list, because the empty list might not have the right type—instead, `find` causes a checked run-time error. To avert such errors, we provide a `bound?` function. Functions `bind`, `find`, and `bound?` are defined in the Supplement, but their types are shown here:

**451c.** *(types of predefined nano-ML functions 451c)≡*

```

(check-principal-type bind
  (forall ['a 'b] ('a 'b (list (pair 'a 'b)) -> (list (pair 'a 'b)))))
(check-principal-type find
  (forall ['a 'b] ('a (list (pair 'a 'b)) -> 'b)))
(check-principal-type bound?
  (forall ['a 'b] ('a (list (pair 'a 'b)) -> bool)))
```

## 7.7 HINDLEY-MILNER AS IT REALLY IS

The Hindley-Milner type system has been used in many languages, but the first is the one Milner himself worked on: Standard ML. In Standard ML, as in most other languages based on Hindley-Milner, a programmer can mix inferred types with explicit types. For example, in Standard ML one can write explicit type variables after a `val` or `fun` keyword. (This syntax is essentially a way of writing type-lambda.)

alpha,	
in nano-ML	422c
in $\mu$ ML	S432a
beta,	
in nano-ML	422c
in $\mu$ ML	S432a
binaryOp	S408d
bindtyscheme	446c
booletype	422c
BOOLV	415b
BugInTypeInference	S237c
FORALL	418
freetyvarsGamma	
446d	
freshtyvar	444b
funtype,	
in nano-ML	422c
in $\mu$ ML	S423d
Gamma	449f
generalize	445a
listtype	422c
NIL	415b
PAIR	415b
RuntimeError	S366c
solve	448a
typeof	448c
typeSchemeString	S412b
tysubst	421a
unaryOp	S408d
valueString,	
in nano-ML	314
in $\mu$ ML	S448b

Standard ML does not have explicit instantiation; instead it has *type constraints*. An explicit instantiation gives the type at which a polymorphic value is instantiated, whereas a type constraint gives the type of the resulting instance. As an example of explicit instantiation, here's length in Typed  $\mu$ Scheme:

**452a.** *(length function for Typed  $\mu$ Scheme 452a)*  $\equiv$

```
(val-rec
  (forall ['a] (function ((list 'a)) int))
  length
  (type-lambda ('a)
    (lambda (((list 'a) xs))
      (if ((@ null? 'a) xs)
        0
        (+ 1 ((@ length 'a) ((@ cdr 'a) xs)))))))
```

To write the same example in Standard ML, we put the type parameter after val or fun, and we use type constraints on lambda-bound and polymorphic variables:

**452b.** *(length function for Standard ML 452b)*  $\equiv$

```
val 'a rec length =
fn (xs : 'a list) =>
  if (null : 'a list -> bool) xs then 0
  else 1 + (length : 'a list -> int) ((tl : 'a list -> 'a list) xs)
```

Different implementations use different notations for the type parameter: Moscow ML puts 'a before rec, but Standard ML of New Jersey puts 'a after rec.

The Hindley-Milner type system is just a starting point. A good next step is functional language Haskell, whose type system combines Hindley-Milner type inference with operator overloading. Most implementations of Haskell also support more general polymorphism; for example, the Glasgow Haskell Compiler provides an explicit forall that supports lambda-bound variables with polymorphic types.

## 7.8 SUMMARY

Type inference changed the landscape of functional languages. A scant 4 years elapsed from the time that Reynolds (1974) described the polymorphic calculus underlying Typed  $\mu$ Scheme to the time that Milner (1978) presented an algorithm for type inference. Over 40 years later, although it has been extended and elaborated in many innovative ways, Milner's type inference remains a sweet spot in the design of typed languages.

Milner's original formulation manipulates only substitutions generated by unification. From unifications to constraints is just a small step, but the constraint-based, “generate-and-solve” model of type inference has proven resilient and extensible. For type inference today, it is the model of choice.

### 7.8.1 Key words and phrases

**TYPE INFERENCE** An algorithm that reconstructs and checks types in a program that does not necessarily include explicit type declarations for let-bound or  $\lambda$ -bound variables. Also called **TYPE RECONSTRUCTION**.

**TYPE** In ML, a type formed using type constructors, type variables, and function arrows. Does not include any quantification.

**TYPE SCHEME** A quantified type with exactly one universal quantifier, which is the outermost part of the type. The set of quantified variables may be empty, in which case the type scheme is equivalent to a TYPE.

**MONOTYPE** A monomorphic type. May be a **TYPE** or may be a **TYPE SCHEME** that quantifies over an empty list of type variables. In ML, every  $\lambda$ -bound variable has a monotype.

**POLYTYPE** A **TYPE SCHEME** that may be instantiated in more than one way. That is, one that quantifies over a nonempty list of type variables. In ML, only a let-bound variable may have a polytype.

**$\lambda$ -BOUND VARIABLE** A formal parameter to a function.

**LET-BOUND VARIABLE** A variable introduced by `let`, `let*`, `letrec`, `val`, or `val-rec`.

**INSTANTIATION** The process of creating a **TYPE** from a **TYPE SCHEME** by substituting for the quantified type variables, if any. In ML, when a variable is used, its type is automatically instantiated.

**GENERALIZATION** The process of creating a **TYPE SCHEME** from a **TYPE** by quantifying over those free type variables that are not also free in the environment, if any. In ML, when an expression is bound to a **LET-BOUND VARIABLE**, its type is automatically generalized.

**SUBSTITUTION** A finite map from type variables to **TYPES**. Also defines maps from types to types, **CONSTRAINTS** to constraints, and others.

**INSTANCE** A **TYPE** obtained from another type by **SUBSTITUTING** for type variables. Also, a **TYPE SCHEME** obtained from another type scheme by first substituting for bound type variables, then **GENERALIZING** over variables in the range of the substitution.

**AT LEAST AS GENERAL** The relation of a type or a type scheme to its instances. For example, a check-type test checks if an expression's **PRINCIPAL TYPE** is at least as general as the type scheme written in the test.

**MOST GENERAL** Given a set of substitutions, a substitution  $\theta$  is a *most general* substitution if any other member of the set can be obtained from  $\theta$  by composing a substitution. Or given a set of types, a type  $\tau$  is a most general type if any other member of the set can be obtained from  $\tau$  by substitution.

**PRINCIPAL TYPE** A type that can be ascribed to an expression such that any other type ascribable to the expression is an **INSTANCE** of the principal type. In other words, a **MOST GENERAL** type of an expression. Also used as shorthand for **PRINCIPAL TYPE SCHEME**, which is similar. In ML, principal type schemes are unique up to renaming of bound type variables.

**EQUALITY CONSTRAINT** A constraint or requirement on types that must be true if a program is to be well typed. Formed by writing conjunctions of **SIMPLE EQUALITY CONSTRAINTS**, each of which takes the form  $\tau_1 \sim \tau_2$ .

**CONSTRAINT SOLVER** An algorithm that finds a substitution that solves a constraint. Constraint  $C$  is solved by  $\theta$  if all the **SIMPLE EQUALITY CONSTRAINTS** in  $\theta C$  relate identical **TYPES**. To work in type inference, a constraint solver must find a **MOST GENERAL** substitution.

**UNIFICATION** An algorithm that finds a **MOST GENERAL SUBSTITUTION** that makes two types equal. Can be used to implement a **CONSTRAINT SOLVER**, or vice versa.

### 7.8.2 Further reading

The original work on the Hindley-Milner type system appears in two papers. Milner (1978) emphasizes the use of polymorphism in programming, and Hindley (1969) emphasizes the existence of principal types. Milner describes Algorithm W, which is the “method of explicit substitutions” in this chapter. Damas and Milner (1982) show that Algorithm W finds the most general type of every term.

Odersky, Sulzmann, and Wehr (1999) present HM( $X$ ), a general system for implementing Hindley-Milner type inference with abstract constraints. This system is considerably more ambitious than nano-ML; it allows a very broad class of constraints, and it completely decouples constraint solving from type inference. Pottier and Rémy (2005) use the power of HM( $X$ ) to explore a number of extensions to ML. Their tutorial includes code written in the related language OCaml.

Vytiniotis, Peyton Jones, and Schrijvers (2010) argue that as type systems grow more sophisticated, Milner’s LET rule makes it harder, not easier to work with the associated constraints. They recommend that by default, the types of LET-bound names should *not* be generalized.

In the presence of mutable reference cells, Milner’s LET rule is unsound. While the unsoundness can be patched by various annotations on type variables, a better approach is to generalize the type of a LET-bound variable only if the expression to which the variable is bound is a *syntactic value*, such as a variable, a literal, or a lambda expression (Wright 1995).

Cardelli (1997) provides a general tutorial on type systems. Cardelli (1987) has also written a tutorial specifically on type inference; it includes an implementation in Modula-2. The implementation represents type variables using mutable cells and does not use explicit substitutions.

Peyton Jones et al. (2007) show how by adding type annotations, one can implement type inference for types in which a  $\forall$  quantifier may appear to the left an arrow—that is, types in which functions may require callers to pass polymorphic arguments. Such types are an example of *higher-rank types*. The authors present both nondeterministic and deterministic rules. The paper is accompanied by code, and it repays careful study.

Material on Haskell can be found at [www.haskell.org](http://www.haskell.org). A nice implementation of Haskell’s type system, in Haskell, is presented by Jones (1999).

## 7.9 EXERCISES

The exercises are summarized in Table 7.4. There are many that I like, but type inference takes center stage.

- In Exercises 18 and 19 on pages 459 and 460, you implement a constraint solver and finish the implementation of type inference. Before you tackle the constraint solver, I recommend that you do Exercises 12 and 16, which will help you solve conjunction constraints in the right way.
- Exercises 1 and 5 offer nice insights into properties of the type system.
- In Exercises 21 to 23, you extend nano-ML. Of these exercises, Exercise 21 (pair primitives) is probably the easiest, and it’s useful. But Exercise 22 offers the most satisfying benefit to the nano-ML programmer.

<i>Exercises</i>	<i>Section</i>	<i>Notes</i>	
1	Ch. 5	Using type inference to get a term of an unusual type.	
2 to 4	7.4.3	Substitutions: understand idempotence; confirm properties of the implementation; substitution preserves constraint satisfaction (§7.5.3).	
5 to 7	7.4	Principal types: equivalence, uniqueness up to equivalence. Most general instances of type schemes.	
8 to 11	7.4, 7.5	Writing constraint-based rules; consequences of the rules.	
12 to 15	7.5.3	Constraints: solvability of conjunctions, soundness of rules for the solver.	
16 to 20	7.5, 7.6	Implementation of constraint solving and type inference.	
21 to 23	7.6	Extending nano-ML: pairs, a list constructor, mutable reference cells.	
24 and 25	7.6	Improving error messages; elaborating untyped $\mu$ ML terms into Typed $\mu$ Scheme terms.	

§7.9. Exercises

455

Table 7.4: Synopsis of all the exercises, with most relevant sections

### 7.9.1 Retrieval practice and other short questions

- A. The type of the successor function on Church numerals is  $(\forall \alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow (\forall \alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ . This type is valid in Typed  $\mu$ Scheme, with kind  $*$ , but it is not a valid Hindley-Milner type. Why not?
- B. In Typed  $\mu$ Scheme, a programmer may instantiate any expression using the instantiation form, written with the @ operator. In nano-ML, instantiation is performed automatically. What syntactic forms are automatically instantiated?
- C. To ensure that types are well formed, Typed  $\mu$ Scheme uses a system of kinds. Why doesn't nano-ML need such a system?
- D. What kind of a thing is  $\theta$ ? What can a  $\theta$  be applied to?
- E. If  $\theta$  is applied to a type constructor  $\mu$ , what are the possible results?
- F. What's an example of a substitution that's *not* idempotent?
- G. Function type `int × int → int` is printed as `(int int -> int)`. How is it represented in the nano-ML interpreter?
- H. The nondeterministic type rule VAR includes judgment  $\tau \leqslant \sigma$ . What does the judgment mean? Given a particular  $\sigma$ , if we want a  $\tau$  satisfying  $\tau \leqslant \sigma$ , how must such a  $\tau$  be formed?
- I. Suppose that  $\Gamma$ 's only binding is  $\{f \mapsto \forall \alpha. \alpha \rightarrow \beta\}$ . What are  $\Gamma$ 's free type variables?
- J. If  $\sigma = \text{generalize}(\tau, \mathcal{A})$  and  $\mathcal{A}$  is empty, what do we know about the free type variables of  $\sigma$ ?
- K. In the nondeterministic type system, does term `(lambda (x y) x)` have type `(int int -> int)`?
- L. What is the principal type (or principal type scheme) of `(lambda (x y) x)`?
- M. In the initial basis, what principal type scheme should be assigned to the primitive function “`null?`”?
- N. What's the difference, if any, between substitution  $\theta_1 \circ \theta_2$  and substitution  $\theta_2 \circ \theta_1$ ?

- O. What's the difference, if any, between constraint  $C_1 \wedge C_2$  and constraint  $C_2 \wedge C_1$ ?
- P. What's the difference, if any, between constraint  $\tau_1 \sim \tau_2$  and constraint  $\tau_2 \sim \tau_1$ ?
- Q. For type inference, why do I recommend against implementing the method of explicit substitutions? What's an example of a typing rule that illustrates the difficulties of this method?
- R. Can the constraint  $\alpha \sim \text{bool}$  be satisfied? If so, by what substitution? What about constraint  $\alpha \times \text{int} \sim \text{bool} \times \alpha$ ? What about constraint  $\alpha \times \text{int} \sim \text{bool} \times \beta$ ?
- S. If constraint  $\tau \sim \tau_1 \rightarrow \tau_2$  is to be satisfied, what form must  $\tau$  have?
- T. If  $\theta_1$  solves  $C_1$  and  $\theta_2$  solves  $C_2$ , does  $\theta_2 \circ \theta_1$  solve  $C_1 \wedge C_2$ ?
- U. What's the algorithm for solving a constraint of the form  $C_1 \wedge C_2$ ?
- V. In the interpreter, why does  $\Gamma$  have a different representation than it did in Chapter 6?
- W. Given a list of constraints  $C_1, \dots, C_n$ , what interpreter function do you call to combine them into a single constraint  $C_1 \wedge \dots \wedge C_n$ ?
- X. When a constraint can't be solved, what interpreter function should you call?
- Y. In chunk 449c, the combination of `f` and `actuals` into a single list is a little awkward. How does the code work? What's the alternative? Why do you think I coded it this way?

### 7.9.2 Manipulating type inference

1. *Functions with seemingly impossible types.*
  - (a) Without using any primitives, and without using `letrec`, write a function in nano-ML that has type  $\forall \alpha, \beta . \alpha \rightarrow \beta$ .
  - (b) Based on your experience, if you see a function whose result type is a quantified type variable, what should you conclude about that function?

### 7.9.3 Properties of substitutions

2. *Idempotence.* A substitution  $\theta$  is *idempotent* if  $\theta \circ \theta = \theta$ .
  - (a) Give an example of a substitution  $\theta_x = (\alpha \mapsto \tau)$  that is *not* idempotent.
  - (b) Prove that  $\theta_x \circ \theta_x \neq \theta_x$ .
  - (c) Prove that if  $\alpha \notin \text{ftv}(\tau)$ , then  $(\alpha \mapsto \tau)$  is idempotent.
  - (d) Instrument the `|-->` function in the interpreter so that if  $\alpha \in \text{ftv}(\tau)$ , calling  $\alpha |--> \tau$  raises the exception `BugInTypeInference`.
3. *Code that produces substitutions.* Go back to function `tysubst` from Chapter 6 on page 380, and look at the inner function `subst`, which is a function from types to types. Given that `varenv` is finite, show that `subst` satisfies each of the properties claimed for a substitution  $\theta$  on page 420.
4. *Substitution preserves satisfaction.* Look at the definition of satisfaction in Section 7.5.3, and prove that if constraint  $C$  is satisfied and  $\theta$  is a substitution, then  $\theta C$  is also satisfied.

### 7.9.4 Properties of type schemes and principal types

5. *Most general instances.* Prove that for any type scheme  $\sigma$ , there is a *most general* instance  $\tau \leqslant \sigma$ . An instance  $\tau$  is a most general instance if and only if  $\forall \tau' . \tau' \leqslant \sigma \implies \tau' \leqslant \tau$ .
6. *Uniqueness of principal types.* Principal types are unique up to renaming of variables.
  - (a) Give an example of an environment and a term such that the term has more than one principal type. Show two different principal types.
  - (b) Prove that if  $\Gamma \vdash e : \tau_P$  and  $\Gamma \vdash e : \tau'_P$  and both  $\tau_P$  and  $\tau'_P$  are principal types for  $e$  in  $\Gamma$ , then  $\tau'_P$  can be obtained from  $\tau_P$  by renaming variables. Use the definition of principal type on page 427.

7. *Equivalence of type schemes.* The implementation of check-principal-type takes as inputs an expression  $e$  and a type scheme  $\sigma$ . It infers a principal type scheme  $\sigma_P$  for  $e$ , then checks to see that  $\sigma_P$  is equivalent to  $\sigma$ . The two type schemes are equivalent if  $\sigma_P \leqslant \sigma$  and  $\sigma \leqslant \sigma_P$ , which is to say that they have the same instances. But in the *Definition of Standard ML*, equivalence of type schemes is defined syntactically:

Two type schemes  $\sigma$  and  $\sigma'$  are considered equal if they can be obtained from each other by renaming and reordering of bound type variables, and deleting type variables from the prefix which do not occur in the body. [The *prefix* is the list of variables between the  $\forall$  and the dot. —NR]

Prove that these definitions are equivalent.

- (a) Prove that renaming a bound type variable in  $\sigma$  does not change its set of instances.
- (b) Prove that reordering bound type variables in  $\sigma$  does not change its set of instances. It suffices to prove that adjacent type variables can be swapped without changing the set of instances.
- (c) Prove that if a type variable appears in the prefix of  $\sigma$  but does not appear in the body, then removing that variable from the prefix does not change the set of instances.
- (d) Conclude that if type schemes  $\sigma$  and  $\sigma'$  are considered equal according to the *Definition of Standard ML*, then they have the same instances, and so they are also considered equal by check-principal-type.

### 7.9.5 Typing rules and their properties

8. *Adding constraints to typing rules.* Rewrite these rules to use explicit constraints:

- (a) The rule for BEGIN:

$$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n} \quad (\text{BEGIN})$$

- (b) The rule for LAMBDA:

$$\frac{\Gamma \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau} \quad (\text{LAMBDA})$$

- 7
9. *Recursive definitions.* If not for the restriction that the right-hand side of a val-rec must be a lambda, the rules of nano-ML would permit the definition (val-rec  $x$   $x$ ). If the definition were permitted, what type scheme  $\sigma$  would be inferred for  $x$ ? Is that  $\sigma$  inhabited by any values? In other words, is there a value that could be stored in  $\rho$  that is consistent with that  $\sigma$ ?
  10. *Absence of free type variables in top-level type environments.* Prove that in nano-ML, a top-level type environment never contains a free type variable. Your proof should be by induction on the sequence of steps used to create  $\Gamma$ :
    - Prove that an empty type environment contains no free type variables.
    - Using the code in chunk S411b, show that if  $\Gamma$  contains no free type variables, then  $\text{addPrim}((x, p, \tau), (\Gamma, \rho))$  returns a pair in which the new  $\Gamma'$  also contains no free type variables.
    - Show that if  $\Gamma$  contains no free type variables, and if  $\Gamma'$  is specified by  $\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma'$ , then  $\Gamma'$  contains no free type variables.
    - Show that if  $\Gamma$  contains no free type variables, and if  $\Gamma'$  is specified by  $\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma'$ , then  $\Gamma'$  contains no free type variables.

11. *Consistency of type inference with nondeterministic rules.* Prove that whenever there is a derivation of a judgment  $\theta\Gamma \vdash e : \tau$  using the rules for explicit substitutions in Section 7.5.1, then if  $\Gamma' = \theta\Gamma$ , there is also a derivation of  $\Gamma' \vdash e : \tau$  using the nondeterministic rules in Section 7.4.5.

To prove this property for nano-ML would be tedious: there are too many cases. Instead, use a subset, which I'll call "pico-ML," which has just these forms: lambda, function application, variable, and let. Both let and lambda bind exactly one name, and a function application has exactly one argument.

What about the other direction? If there is a derivation using the nondeterministic rules, is there a corresponding derivation using explicit substitutions? Yes, but the corresponding derivation might not derive the same type. The most we can say about a type derivable using the nondeterministic rules is that it must be an *instance* of the type derived using type inference with explicit substitutions. The type derived using type inference is special, because all other derivable types are instances of it; it is the term's principal type. Proving that a principal type exists and that the type-inference algorithm finds one are problems that are beyond the scope of this book.

### 7.9.6 Properties of constraints and constraint solving

12. *Solvability and conjunction.*
  - (a) Using the proof system in Section 7.5.3, prove that if the constraint  $C_1 \wedge C_2$  is solvable, then constraints  $C_1$  and  $C_2$  are also solvable.
  - (b) Find a particular pair of constraints  $C_1$  and  $C_2$  such that  $C_1$  is solvable,  $C_2$  is solvable, but  $C_1 \wedge C_2$  is *not* solvable. *Prove* that  $C_1 \wedge C_2$  is not solvable.
13. *Soundness of the SOLVEDCONJUNCTION rule.* Using the definition of what it means for a constraint to be satisfied, prove that the SOLVEDCONJUNCTION rule on page 439 is sound. That is, prove that whenever both premises hold, so does the conclusion. The rule is reproduced here:

$$\frac{\theta_1 C_1 \text{ is satisfied} \quad \theta_2(\theta_1 C_2) \text{ is satisfied}}{(\theta_2 \circ \theta_1)(C_1 \wedge C_2) \text{ is satisfied}} \quad (\text{SOLVEDCONJUNCTION})$$

14. *Soundness of the SOLVECONAPP rule.* Using the laws for applying substitutions to constraints and to types, prove that the SOLVECONAPP rule on page 440 is sound. That is, prove that a substitution  $\theta$  solves the constraint

$$\text{CONAPP}(\tau, \langle \tau_1, \dots, \tau_n \rangle) \sim \text{CONAPP}(\tau', \langle \tau'_1, \dots, \tau'_n \rangle)$$

if and only if it also solves the constraint

$$\tau \sim \tau' \wedge \tau_1 \sim \tau'_1 \wedge \dots \wedge \tau_n \sim \tau'_n.$$

15. *Need for an occurs check.* Prove that if  $\tau_2$  mentions  $\alpha$  but is *not equal* to  $\alpha$ , then there is no substitution  $\theta$  such that  $\theta\alpha = \theta\tau_2$ . (Hint: count type constructors.)

### 7.9.7 Implementation of constraint solving and type inference

16. *Practice with conjunction constraints.* The most common mistake made in constraint solving is to get conjunctions wrong (Section 7.5.3). Before you tackle a solver, this exercise asks you to develop some examples and to verify that the naïve approach works sometimes, but not always.

- (a) Find two constraints  $C_1$  and  $C_2$  and substitutions  $\theta_1$  and  $\theta_2$  such that

- $C_1$  has a free type variable,
- $C_2$  has a free type variable,
- $\theta_1$  solves  $C_1$ ,
- $\theta_2$  solves  $C_2$ , and
- $\theta_2 \circ \theta_1$  solves  $C_1 \wedge C_2$ .

- (b) Find two constraints  $C_1$  and  $C_2$  and substitutions  $\theta_1$  and  $\theta_2$  such that

- $C_1$  has a free type variable,
- $C_2$  has a free type variable,
- $\theta_1$  solves  $C_1$ ,
- $\theta_2$  solves  $C_2$ , and
- $\theta_2 \circ \theta_1$  does *not* solve  $C_1 \wedge C_2$ .

17. *Understanding a recursive solver.* Pages 440–442 discuss the solution of constraint 7.1, which involves synthesizing and solving constraints 7.2 to 7.11. List the numbered constraints *in the order that they would be solved* by a recursive solver.

18. *Implementing a constraint solver.* Using the ideas in Section 7.5.3, implement a function `solve` which takes as argument a constraint of type `con` and returns an idempotent substitution of type `subst`. (If a substitution is created using only the value `idsubst` and the functions `|-->` and `compose`, then it is guaranteed to be idempotent.) The resulting substitution should solve the constraint, obeying the law

$$\text{solves} (\text{solve } C, C).$$

If the constraint has no solution, call function `unsatisfiableEquality` from chunk 447d, which raises the `TypeError` exception.

19. *Implementing type inference.* Complete the definitions of functions `ty` and `literal` so they never raise `LeftAsExercise`. If your code discovers a type error, it should raise the exception `TypeError`.

The function `literal` should give a suitable type to integer literals, Boolean literals, symbol literals (which have type `sym`), and quoted lists in which all elements have the same type. (This set includes the empty list.) For example, the value `'(1 2 3)` should have type `int list`. Values created using `CLOSURE` or `PRIMITIVE` cannot possibly appear in a `LITERAL` node, so if your `literal` function sees such a value, it would be justified in raising `BugInTypeInference`.

You can adapt the `typeof` function from Chapter 6, but you may find it easier to start from scratch. Whatever you do, don't overlook the `typesof` function.

You will probably find it helpful to refer to the typing rules for nano-ML, which are summarized in Figures 7.5 to 7.8 on pages 463 and 464.

20. *Constraint solving from unification.* The original formulation of type inference relies on unification of explicit substitutions. To show that unification is as powerful as constraint solving, suppose that you have a function `unify` such that given any two types  $\tau_1$  and  $\tau_2$ , `unify`( $\tau_1, \tau_2$ ) returns a substitution  $\theta$  such that  $\theta\tau_1 = \theta\tau_2$ , or if no such  $\theta$  exists, raises an exception. Use `unify` to implement a constraint solver.

*Hint:* To help convert a constraint-solving problem into a unification problem, try using the `SOLVECONAPP` rule on page 440 *in reverse*.

#### 7.9.8 Extending nano-ML

21. *Pair primitives.* Extend nano-ML with primitives `pair`, `fst`, and `snd`. Give the primitives appropriate types. Function `pair` should be used to create pairs of any type, and functions `fst` and `snd` should retrieve the elements of any pair.

22. *A list constructor.* In nano-ML, the most convenient way to build a large list is by using a large expression that contains a great many applications of `cons`. But as described in the sidebar on page 441, type inference using my data structures requires time and space that is quadratic in the number of applications of `cons`. The problem can be addressed through more efficient representations, but there is a surprisingly simple fix through language design: extend nano-ML with a `LIST` constructor that works the same way the square-bracket-and-comma syntax works in Standard ML. Here is a rule of operational semantics:

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \quad \dots \quad \langle e_n, \rho \rangle \Downarrow v_n}{v = \text{PAIR}(v_1, \text{PAIR}(v_2, \dots, \text{PAIR}(v_n, \text{NIL})))} \quad (\text{LIST})$$

And here is a nondeterministic typing rule:

$$\frac{\Gamma \vdash e_1 : \tau \quad \dots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \text{LIST}(e_1, \dots, e_n) : \tau \text{ list}} \quad (\text{LIST})$$

Implement a list constructor for nano-ML, in the following four steps:

- (a) Extend the abstract syntax for `exp` with a case `LIST` of `exp list`.

- (b) Extend the parser to accept  $(\text{list } e_1 \dots e_n)$  to create a LIST node.  
As a model, use the parser for begin.
- (c) Extend the evaluator to handle the LIST case.
- (d) Extend type inference to handle the LIST case.
23. *Mutable reference cells and the value restriction.*
- (a) Add new primitives ref, !, and := with the same meanings as in full ML.  
You will have to use ML “ref cells” to add a new form of value, and you will have to extend the primitiveEquality function to compare ref cells for equality. But you should not have to touch environments or the evaluator.
- (b) Write a program in extended nano-ML that subverts the type system, i.e., that makes the evaluator raise the exception BugInTypeInference. For example, try writing a program that applies + to a non-integer argument. (Hint: (ref (lambda (x) x)).)
- (c) Restore type safety by implementing the “value restriction” on polymorphism: the type of val-bound name is generalized only if the name is bound to syntactic value (a variable, a literal, or a  $\lambda$ -abstraction).

### 7.9.9 Improving the interpreter

24. *Improved error messages.*
- (a) If a type error arises from a function application, show the function and show the arguments. Show what types of arguments the function expects and what the types the arguments actually have.
- (b) If a type error arises from an IF expression, show either that the type of the condition is inconsistent with bool or that the two branches do not have consistent types.
- (c) Highlight the differences between inconsistent types. For example, if your message says
- ```
function cons expected int * int list, got int * int
```
- this is better than saying “cannot unify int and int list,” but it is not as good as showing which argument caused the problem, e.g.,
- ```
function cons expected int * >>int list<<, got int * >>int<<
```
- To complete this part, it would be helpful to define a more sophisticated version of unsatisfiableEquality, which returns strings in which types that don’t match are highlighted.

To associate a type error with a function application, look at the APPLY rule on page 431 and see whether the constraint in the premise is solvable, and if so, whether the constraint in the conclusion is solvable.

25. *Elaboration into explicitly typed terms.* Change the implementation of type inference so that instead of inferring and checking types in one step, the interpreter takes an untyped term and infers an explicitly typed term in Typed  $\mu$ Scheme. Adding information to the original code is an example of *elaboration*. I recommend copying the syntax of Typed  $\mu$ Scheme into a submodule of your interpreter, as in

```
structure Typed = struct
  datatype exp = ...
  :
end
```

You can then translate LAMBDA to Typed.LAMBDA, and so on.

$$\boxed{\Gamma \vdash e : \tau} \quad \frac{\text{VAR}}{\Gamma(x) = \sigma \quad \tau \leqslant \sigma} \quad \frac{}{\Gamma \vdash x : \tau}$$

$$\frac{\text{IF}}{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau} \quad \frac{}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau}$$

$$\frac{\text{APPLY}}{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \quad \Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \rightarrow \tau} \quad \frac{}{\Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \tau}$$

*§7.9. Exercises*  
463

$$\frac{\text{LAMBDA}}{\Gamma \vdash \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e) : \tau_1 \times \cdots \times \tau_n \rightarrow \tau}$$

$$\frac{\text{LET}}{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n} \quad \frac{\sigma_i = \text{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \leq i \leq n}{\Gamma \vdash e : \tau} \quad \frac{}{\Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}$$

$$\frac{\text{LETREC}}{e_1, \dots, e_n \text{ are all LAMBDA expressions}} \quad \frac{\Gamma' = \Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}}{\Gamma' \vdash e_i : \tau_i, \quad 1 \leq i \leq n} \quad \frac{\sigma_i = \text{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \leq i \leq n}{\Gamma \vdash e : \tau} \quad \frac{}{\Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}$$

$$\frac{\text{LETSTAR}}{\Gamma \vdash \text{LET}(\langle x_1, e_1 \rangle, \text{LETSTAR}(\langle x_2, e_2, \dots, x_n, e_n \rangle, e)) : \tau \quad n > 0} \quad \frac{}{\Gamma \vdash \text{LETSTAR}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}$$

$$\frac{\text{EMPTYLETSTAR}}{\Gamma \vdash \text{LETSTAR}(\langle \rangle, e) : \tau} \quad \frac{\text{BEGIN}}{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n} \quad \frac{}{\Gamma \vdash \text{BEGIN}(e_1, \dots, e_n) : \tau_n} \quad \frac{\text{EMPTYBEGIN}}{\Gamma \vdash \text{BEGIN}() : \text{unit}}$$

Figure 7.5: Nondeterministic typing rules for expressions

$$\boxed{\langle d, \Gamma \rangle \rightarrow \Gamma'} \quad \frac{\text{VAL}}{\Gamma \vdash e : \tau \quad \sigma = \text{generalize}(\tau, \text{ftv}(\Gamma))} \quad \frac{}{\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}}$$

$$\frac{\text{VALREC}}{\Gamma\{x \mapsto \tau\} \vdash e : \tau \quad \sigma = \text{generalize}(\tau, \text{ftv}(\Gamma)) \quad e \text{ has the form LAMBDA}(\dots)} \quad \frac{}{\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad \frac{\text{EXP}}{\langle \text{VAL(it, e)}, \Gamma \rangle \rightarrow \Gamma'} \quad \frac{}{\langle \text{EXP}(e), \Gamma \rangle \rightarrow \Gamma'}$$

$$\frac{\text{DEFINE}}{\langle \text{VAL-REC}(f, \text{LAMBDA}(\langle x_1, \dots, x_n \rangle, e)), \Gamma \rangle \rightarrow \Gamma'} \quad \frac{}{\langle \text{DEFINE}(f, (\langle x_1, \dots, x_n \rangle, e)), \Gamma \rangle \rightarrow \Gamma'}$$

Figure 7.6: Nondeterministic typing rules for definitions

$\boxed{C, \Gamma \vdash e : \tau}$	<b>VAR</b> $\Gamma(x) = \forall \alpha_1, \dots, \alpha_n. \tau$ $\alpha'_1, \dots, \alpha'_n \text{ are fresh and distinct}$ $\boxed{\mathbf{T}, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \dots \circ (\alpha_n \mapsto \alpha'_n)) \tau}$
$\boxed{C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3}$	<b>IF</b> $C \wedge \tau_1 \sim \text{bool} \wedge \tau_2 \sim \tau_3, \Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau_2$
$\boxed{C, \Gamma \vdash e, e_1, \dots, e_n : \hat{\tau}, \tau_1, \dots, \tau_n}$	<b>APPLY</b> $C \wedge \hat{\tau} \sim \tau_1 \times \dots \times \tau_n \rightarrow \alpha, \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \alpha$
$\boxed{C, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n}$	<b>LET</b> $\theta C \text{ is satisfied } \theta \text{ is idempotent}$ $C' = \bigwedge \{\alpha \sim \theta \alpha \mid \alpha \in \text{dom } \theta \cap \text{ftv}(\Gamma)\}$ $\sigma_i = \text{generalize}(\theta \tau_i, \text{ftv}(\Gamma) \cup \text{ftv}(C')), \quad 1 \leq i \leq n$ $C_b, \Gamma \{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau$ $\boxed{C' \wedge C_b, \Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}$
$\boxed{C, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n}$	<b>LETREC</b> $e_1, \dots, e_n \text{ are all LAMBDA expressions}$ $\Gamma' = \Gamma \{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}, \text{ where all } \alpha_i \text{'s are distinct and fresh}$ $C_r, \Gamma' \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n$ $C = C_r \wedge \tau_1 \sim \alpha_1 \wedge \dots \wedge \tau_n \sim \alpha_n$ $\theta C \text{ is satisfied } \theta \text{ is idempotent}$ $C' = \bigwedge \{\alpha \sim \theta \alpha \mid \alpha \in \text{dom } \theta \cap \text{ftv}(\Gamma)\}$ $\sigma_i = \text{generalize}(\theta \tau_i, \text{ftv}(\Gamma) \cup \text{ftv}(C')), \quad 1 \leq i \leq n$ $C_b, \Gamma \{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau$ $\boxed{C' \wedge C_b, \Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}$

Figure 7.7: Constraint-based typing rules for expressions

$\boxed{\langle d, \Gamma \rangle \rightarrow \Gamma'}$	<b>VAL</b> $C, \Gamma \vdash e : \tau$ $\theta C \text{ is satisfied } \theta \Gamma = \Gamma$ $\sigma = \text{generalize}(\theta \tau, \text{ftv}(\Gamma))$ $\boxed{\langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma \{x \mapsto \sigma\}}$
$\boxed{C, \Gamma \{x \mapsto \alpha\} \vdash e : \tau \quad \alpha \text{ is fresh}}$	<b>VALREC with constraints</b> $\theta(C \wedge \alpha \sim \tau) \text{ is satisfied } \theta \Gamma = \Gamma$ $\sigma = \text{generalize}(\theta \alpha, \text{ftv}(\Gamma))$ $e \text{ has the form LAMBDA}(\dots)$ $\boxed{\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma \{x \mapsto \sigma\}}$

Figure 7.8: Constraint-based typing rules for definitions

## **II. PROGRAMMING AT SCALE**



Programming at scale is supported by many techniques, tools, and language features, but the key enabling idea is to hide information. The role of a programming language is to hide information about the representation of data, which is called *data abstraction*. There are two techniques: *abstract data types*, which uses a type system to limit access to representation, and *objects*, which uses environments to limit access to representation. These ideas are illustrated by the languages Molecule (Chapter 9) and  $\mu$ Smalltalk (Chapter 10).

To write programs at scale, particularly if you want to describe representations using a type system, you need more data types than can be found in languages like Typed Impcore, Typed  $\mu$ Scheme, and nano-ML. To provide suitable types, this section options with  $\mu$ ML, which extends nano-ML with user-defined types.  $\mu$ ML’s *algebraic data types* give programmers the ability to group parts together, to express choices among forms of a single representation, and to define recursive representations. They also support pattern matching.

Molecule starts with  $\mu$ ML’s data representation and with Typed  $\mu$ Scheme’s type system, adding a modules system that can describe interfaces and implementations. Like Typed  $\mu$ Scheme, Molecule requires no type inference, only type checking.

$\mu$ Smalltalk introduces a different programming paradigm: objects. Code doesn’t call functions; instead, it sends *messages* to objects, and these messages are dispatched dynamically to *methods*. In  $\mu$ Smalltalk, methods are defined by *classes*, and classes can *inherit* methods from other classes. Dynamic dispatch and inheritance are simple mechanisms, but they have far-reaching consequences. Using them well enables new forms of reuse but requires new ways of thinking.

Molecule and  $\mu$ Smalltalk provide data abstraction in different ways. Both hide representations except from code that is inside the abstraction, but what’s considered “inside” is different. Within a Molecule module, an operation can see the representation of any value of any abstract type defined inside the module. Within a  $\mu$ Smalltalk class, an operation can see the representation of just one object: the object on which the operation is defined. The representations of other objects are hidden. These different forms of information hiding lead to different kinds of flexibility. Using Molecule’s abstract data types, it is easy to define operations that inspect multiple representations, like an efficient merge of leftist heaps. But it is impossible to define operations that work with new abstractions defined outside the operation’s cluster: each operation works only with arguments that have the static types given in its signature. Using  $\mu$ Smalltalk’s objects, it is possible to define operations that inspect multiple representations, but to do so requires exposing the representation, violating abstraction. But if abstraction is not violated, then it is easy to use existing operations with new representations: each operation works with any object that responds correctly to the messages in the specification’s protocol. These differences in visibility and interoperability are essential differences that apply when comparing any language that uses abstract data types with any language that uses objects.

Other differences between Molecule and Smalltalk apply to a more limited set of languages. For example, Molecule provides code reuse through parametric polymorphism, and  $\mu$ Smalltalk provides code reuse through inheritance. But a language can have abstract data types without having parametric polymorphism, and a language can also have objects without having classes and inheritance. There are also incidental differences; for example, a Molecule module controls which operations are exported to client code, but a Smalltalk class exposes all operations to every client. The context of its language’s semantics, each decision makes good design sense.

## CHAPTER CONTENTS

---

10.1	OBJECT-ORIENTED PROGRAMMING BY EXAMPLE	620	10.7.5	Choice of representation: Natural numbers	678
10.1.1	Objects and messages	620	10.8	TECHNIQUE IV: OBJECT-ORIENTED PROGRAMMING WITH INVARIANTS	681
10.1.2	Protocols, methods, classes, and inheritance	622	10.9	OPERATIONAL SEMANTICS	685
10.1.3	Pictures: programming with lists and blocks	626	10.9.1	Expressions without return	688
10.1.4	A canvas, on which pictures can be drawn	628	10.9.2	Expressions that return	691
10.1.5	Shapes: taking inheritance seriously	628	10.9.3	Definitions	692
10.1.6	Object-orientation means dynamic dispatch	633	10.10	THE INTERPRETER	693
10.1.7	Review and analysis	633	10.10.1	Abstract syntax	695
10.2	DATA ABSTRACTION ALL OVER AGAIN	635	10.10.2	Evaluating expressions; dynamic dispatch	696
10.3	THE LANGUAGE	637	10.10.3	Evaluating definitions	701
10.3.1	Concrete syntax and its evaluation	637	10.10.4	Class objects and metaclasses	702
10.3.2	Values	639	10.10.5	The primitive classes	703
10.3.3	Names	640	10.10.6	Bootstrapping for literal and Boolean values	705
10.3.4	Message sends, inheritance, dynamic dispatch	641	10.10.7	Primitives	707
10.3.5	Equality: identity and equivalence	642	10.11	SMALLTALK AS IT REALLY IS	707
10.4	THE INITIAL BASIS	645	10.11.1	Programming language as operating system	708
10.4.1	Protocol for all objects	646	10.11.2	Smalltalk-80's language	709
10.4.2	Protocol for classes	647	10.11.3	Smalltalk-80's class hierarchy	712
10.4.3	Blocks and Booleans	647	10.12	OBJECTS AND CLASSES AS THEY REALLY ARE	714
10.4.4	Unicode characters	651	10.13	SUMMARY	715
10.4.5	Collections	651	10.13.1	Key words and phrases	716
10.4.6	Magnitudes and numbers	658	10.13.2	Further reading	719
10.4.7	Natural numbers	661	10.14	EXERCISES	721
10.5	TECHNIQUE I: DISPATCH REPLACES CONDITIONALS	662	10.14.1	Retrieval practice	722
10.6	TECHNIQUE II: ABSTRACT CLASSES	664	10.14.2	Using shape classes	724
10.6.1	Wide protocols: Magnitudes and collections	664	10.14.3	New shapes and canvases	724
10.6.2	Widening to keyed and sequenceable collections	667	10.14.4	Creating new classes from scratch	725
10.6.3	Compromising on protocol: class Array	670	10.14.5	Messages to super	725
10.7	TECHNIQUE III: MULTIPLE REPRESENTATIONS	670	10.14.6	Equality	726
10.7.1	Context: Abstract classes Number and Integer	671	10.14.7	Method redefinition	726
10.7.2	Implementing complex operations: Fraction	673	10.14.8	New methods on existing classes	727
10.7.3	Interoperation with more than one representation: Double dispatch	675	10.14.9	Conditional tests via dynamic dispatch	727
10.7.4	Coercion in Fraction and Integer	677	10.14.10	Collections	727
			10.14.11	New magnitudes	730
			10.14.12	Numbers	731
			10.14.13	Method dispatch	733
			10.14.14	Object-oriented profiling	734

# 10

## *Smalltalk and object-orientation*

*Smalltalk's design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.*

Alan Kay, *The Early History of Smalltalk*

*Data and operations belong together, and most useful program constructs contain both.*

Ole-Johan Dahl and Kristen Nygaard,  
*The Development of the SIMULA Languages*

An abstract data type is not the only way to hide the representation of data: a representation can also be hidden in an *object*. Think of an object as a bundle of operations—often called *methods*—that may share hidden state. An object interacts with other objects by sending *messages*; each message activates a method on the receiving object. An object’s methods can see the representation of the object’s own state, but not the representations of the states of other objects.

Object-oriented languages—or more precisely, procedural languages infused with object-oriented ideas—have been popular since the 1990s. The popular languages draw from ideas originally developed in Simula 67, Smalltalk-80, and Self, which collectively represent three decades of language design. Simula 67 (Nygaard and Dahl 1981) introduced objects and classes, which it layered on top of Algol 60, a procedural language. Smalltalk-80 simplified Nygaard and Dahl’s design by eliminating the underlying procedural language; Smalltalk is purely object-oriented. Self (Ungar and Smith 1987) simplified Smalltalk’s design still further, eliminating classes. For learning, the best of these languages is Smalltalk: Because it is not layered on top of a procedural language, you won’t be distracted by superfluous features. And because it does have classes, you’ll have no trouble relating it to such successor languages as Objective C, C++, Modula-3, Ruby, Java, Ada 95, and Swift—even though most of these successors are actually procedural languages with object-oriented features.

Smalltalk is designed around one idea, simplified as much as possible, and pushed to a logical extreme: everything is an object. In Smalltalk-80, every value is an object: for example, the number 12 is an object; every class is an object; the program is an object; the compiler is an object; every activation on the call stack is an object; the programming environment is an object; every window is an object;

and even most parts of the hardware (display, keyboard, mouse) are modeled by objects. Furthermore, conditionals and loops are implemented not by evaluating special syntactic forms like `if` and `while` but by sending continuations to objects. While  $\mu$ Smalltalk does not go as far as enabling you to treat the interpreter and the hardware as objects, it does ensure that every value and class is an object, and it does implement conditionals and loops by sending continuations.

## 10.1 OBJECT-ORIENTED PROGRAMMING BY EXAMPLE

The evaluation of a Smalltalk expression proceeds as a sequence of actions, each of which typically mutates a variable or sends a message to an object. (An action may also allocate a closure or terminate the execution<sup>1</sup> of a method.) When a message is sent to an object, which is called the *receiver*, that message is *dispatched* to a *method*. Like a function in full Scheme, a Smalltalk method binds *arguments* to formal parameters, then evaluates a sequence of expressions. The method to which a message is dispatched is determined by the *class* of the object receiving the message. A class defines methods, but it also *inherits* methods from a designated *superclass*. And a class defines the representation of each of its objects, which are called *instances* of the class.

While method dispatch and inheritance are important, the heart of object-oriented programming is exchanging messages: “You don’t send messages because you have objects; you have objects because you send messages” (Metz 2013, page 67). In the examples below, messages are exchanged among four sorts of objects:

- Coordinate pairs, which represent locations and which can be added and subtracted as vectors
- Shapes, which can be asked about their locations, can be moved to given locations, and can be drawn on canvases
- Pictures, which collect shapes
- Canvases, on which shapes and pictures can be drawn

After drawing a picture by sending messages, we examine the definitions behind the messages, using Smalltalk in gradually more sophisticated ways.

### 10.1.1 Objects and messages

The object that receives a message is called the message’s *receiver*. On receiving a message, an object activates the execution of a *method*; the method does some internal computation and eventually replies with an *answer*. A method is a bit like a function, and an object is a lot like a bundle of functions that share state—typically mutable state. But unlike a procedural or functional programming language, Smalltalk does not enable the caller to choose what function is called—a caller can choose only what message to send. When the message is received, the method that is executed is determined by the *class* of the receiver.

The syntax of a message send has three parts: an expression that evaluates to the receiver, the name of the message, and zero or more *arguments*:

$$\text{exp} ::= (\text{exp } \text{message-name} \{ \text{exp} \})$$

<sup>1</sup>For Smalltalk, I say “execute” instead of “evaluate.” They mean the same thing, but “execute” carries connotations of imperative features and mutable state, which are typical of Smalltalk.

In this syntax, as in Impcore’s function-call syntax “*(function-name { exp })*,” the operation that is intended is *named*. But the meaning of the name—and therefore its placement—is different:

- In a procedural language like Impcore, the caller determines the code that is executed and identifies it by naming the procedure, which is written first.
- In an object-oriented language like Smalltalk, all the caller gets to do is pick the *name* of the message sent, which has the same name as the method that is eventually executed—but it’s the *receiver* that determines what code is executed. That’s why the receiver is written first.
- In a functional language like  $\mu$ Scheme, the caller doesn’t have to *name* the code that is executed, but the semantics and syntax are like procedural semantics and syntax: the caller determines what function is executed, and the function is written first.

§10.1  
Object-oriented  
programming by  
example

621

Receiver-first syntax is found not only in Smalltalk but in C++, Java, JavaScript, Ruby, Modula-3, Lua, Swift, and essentially all languages that emphasize object-oriented features.

If you have interesting objects, you can write interesting computations just by sending them messages and placing their answers in variables. To illustrate, I use  $\mu$ Smalltalk to draw the picture “ $\bigcirc \square \triangle$ .” To draw this picture, I send messages to objects that represent classes, shapes, the picture, and a “canvas” on which the picture is drawn. To start, I create a circle object by sending the new message to the Circle class.

**621a.** *(transcript 621a)*≡

```
-> (use shapes.smt)      ; load shape classes defined in this section
-> (val c (Circle new))
<Circle>
```

621b▷

The Circle class, like every Smalltalk class, is also an object, and by default, we can create a new instance of a class by sending message new to it. The word new does not signal a special syntactic form the way it does in Java or C++; it’s a message name like any other.

Next I send new to the Square class, creating a new square:

**621b.** *(transcript 621a)*+≡

```
-> (val s (Square new))
<Square>
```

△621a 621c▷

By default, a new shape has radius 1 and is located at the coordinate origin. So both s and c are initially at the origin—and s needs to be moved to the right of c. I send two messages:

- Ask circle c for the location of its East “control point.”
- Tell square s to adjust its position to put its West control point at that same location.

A “control point” is just a point on a square in which a shape is inscribed (page 629). I label control points with symbols like ‘North’, ‘South’, and so on. Using control points, I tell the square, “adjust yourself to place your West control point at the same location as the circle’s East control point”:

**621c.** *(transcript 621a)*+≡

```
-> (s adjustPoint:to: 'West (c location: 'East))
<Square>
```

△621b 622a▷

Both `adjustPoint:to:` and `location:` are *message names*, also called *message selectors*; `s` and `c` are *receivers*; `'West`, `'East`, and `(c location: 'East)` are *arguments*. The combination of message name and arguments forms a *message*. Each message name has a number of colons equal to the number of arguments in the message.

I create a new triangle and adjust it immediately, without first placing it in a variable. The `adjustPoint:to:` message answers the triangle (its receiver).<sup>2</sup>

**622a.** *(transcript 621a) +≡* △621c 622b ▷  
→ (val t ((Triangle new) adjustPoint:to: 'Southwest (s location: 'East)))  
<Triangle>

The expression almost reads like pidgin English: “hey new triangle, adjust your southwest control point to the location of `s`’s east point.”

I now put all three shapes into a picture.

**622b.** *(transcript 621a) +≡* △622a 622c ▷  
→ (val pic (Picture empty))  
<Picture>  
→ (pic add: c)  
<Picture>  
→ (pic add: s)  
<Picture>  
→ (pic add: t)  
<Picture>

Message `empty` is sent to the `Picture` class, which answers a new object that represents an empty picture. Message `add:` is then sent to that object, `pic`, with different arguments. Each `add:` message answers the (modified) picture. To render `pic`, I use a *canvas*, which I create by sending the `new` message to class `TikzCanvas`.

**622c.** *(transcript 621a) +≡* △622b 632b ▷  
→ (val canvas (TikzCanvas new))  
<TikzCanvas>  
→ (pic renderUsing: canvas)  
`\begin{tikzpicture}[x=4pt,y=4pt]`  
`\draw (0,0)ellipse(1 and 1);`  
`\draw (3,1)--(1,1)--(1,-1)--(3,-1)--cycle;`  
`\draw (4,2)--(3,0)--(5,0)--cycle;`  
`\end{tikzpicture}`  
<Picture>

The arcane output is a program written for the `TikZ` package of the `LATEX` typesetting system, which produces “○□△”.

To draw this picture, I send `new` to several class objects, I send `location:` and `adjustPoint:to:` to objects that represent shapes, I send `empty` to the `Picture` class, and I send `add:` and `renderUsing:` to the resulting instance of `Picture`. How do I know what messages to send? And how do the receiving objects know what to do with them? Messages and their associated behaviors are determined by *protocols* and *class definitions*, which are the subject of the next section.

### 10.1.2 Protocols, methods, classes, and inheritance

In Impcore, you can call any function named in the function environment  $\phi$ . In Scheme, you can call any function you can compute, even if it has no name.

<sup>2</sup>Smalltalk uses the verb “to answer” in a nonstandard way. In standard English, when you answer something, the thing answered is a request: you might “answer” an email or “answer” your phone. But in Smalltalk jargon, “answer” means “to send as the answer.” A method might “answer a new object” or “answer a number.” This usage should grate on your ear, but like any other jargon, it soon seems normal.

Class protocol for CoordPair:

withX:y:	aNumber aNumber	Answer a new instance of class CoordPair with the given arguments as its $(x, y)$ coordinates.
----------	-----------------	--

Instance protocol for CoordPair:

x	Answer the $x$ coordinate of the receiver.
y	Answer the $y$ coordinate of the receiver.
* aNumber	Answer a new instance of class CoordPair whose $(x, y)$ coordinates are the coordinates of the receiver, multiplied by the given number.
+ aCoordPair	Answer a new instance of class CoordPair whose $(x, y)$ coordinates are the vector sum of the receiver and the argument.
- aCoordPair	Answer a new instance of class CoordPair whose $(x, y)$ coordinates are the vector difference of the receiver and the argument.
print	Print the receiver in the form $(x, y)$ .

Figure 10.1: Protocols for CoordPair

In Smalltalk, you can send any message that the receiver understands. The message name is not looked up in an environment, as in Impcore, and it does not evaluate to a value, as in Scheme. Instead, it is used by the receiver to determine what code should be executed in response to the message, using the *dynamic dispatch* algorithm described in Section 10.3.4 below.

The messages that an object understands form the object’s *protocol*. Like a Molecule interface, a Smalltalk protocol describes an abstraction by saying what we can do with it: what messages an object understands, what arguments are sent with that message, and how the object responds.

We begin our study of protocols with the protocols related to class CoordPair, which defines an abstraction of  $(x, y)$  coordinate pairs, or equivalently, two-dimensional vectors. In the example above, these pairs represent locations. Their protocols are shown in Figure 10.1.

- The first protocol is the *class protocol* for CoordPair. This protocol specifies the messages that can be sent to the CoordPair class, which is represented by an object named CoordPair. Sending `withX:y:` to the class results in a new object that is an *instance* of the class.
- The second protocol is the *instance protocol* for CoordPair. This protocol specifies the messages that can be sent to any instance of the class. Sending `*` multiplies the receiving vector by a scalar, while sending `+` or `-` does vector arithmetic. Sending `print` prints a textual representation of the instance.

The methods in the instance protocol are divided into three groups: access to coordinates, arithmetic, and printing. Smalltalk-80 called such groups “message categories,” but today we just call them “protocols”—so a class’s instance protocol is a collection of smaller protocols.

The CoordPair abstraction is immutable, as we can tell from its protocol: no message mutates its receiver. In general, we can classify messages using the ideas and terminology introduced in Section 2.5.2 on page 113, which we also use to classify

operations in a Molecule interface. In the class protocol, message `withX:y:` is a *creator*. In the instance protocol, messages `*`, `-`, and `+` are *producers*. The abstraction is immutable, so there are no *mutators*. Messages `x`, `y`, and `print` are *observers*.

The protocols for a class and its instances are determined by a *class definition*. A class definition associates each message name (the interface) with a method (the implementation). In  $\mu$ Smalltalk, a class definition is written with this syntax:

*Smalltalk and  
object-orientation*  

---

10      624

```
def ::= (class class-name
           [subclass-of superclass-name]
           [[ivars {instance-variable-name}]]
           {method-definition})
method-definition ⇒
  (class-method method-name (formals) [[locals locals]] {exp})
  | (method       method-name (formals) [[locals locals]] {exp})
```

From the top down,

- A class definition names the new class and identifies its *superclass*.
- Except for a few kinds of primitive objects, every Smalltalk object is represented by a set of named variables, each of which stands for a mutable location.<sup>3</sup> The location may contain any object. Because every instance of the class gets its own set of named locations, the variables are called *instance variables*. The names of a class's instance variables, if any, are listed just after its superclass. In addition to the names listed, an instance also *inherits* instance-variable names from the superclass.
- A *method definition* determines how an *instance* of the class responds to the corresponding message; the message becomes part of the instance protocol.
- A *class-method definition* determines how the *class itself* responds to the corresponding message; the message becomes part of the class protocol. Both instance methods and class methods may declare local variables.

Like instance variables, the instance methods and class methods of a superclass are inherited by subclasses.

Our first class definition defines `CoordPair`, which implements the `CoordPair` protocols. It is shown in Figure 10.2 on the next page.

- Every new class has to have a superclass, and unless you have a reason to choose something else, you choose class `Object`. Class `Object` defines and implements a protocol useful for all objects, including such operations as printing, checking for equality, and several others. Class `CoordPair` inherits this protocol and the corresponding methods, with two exceptions: the inherited `=` and `print` methods are redefined, or, in object-oriented terminology, *overridden*.
- An instance of class `CoordPair` is represented by the *instance variables* `x` and `y`. As noted in the comment, both `x` and `y` are numbers—this is the representation invariant of the abstraction.

---

<sup>3</sup>In this book, “variable” means the name you look up in an environment, to find a mutable location. But in the literature on Smalltalk-80, “variable” means the mutable location. Alas, terminology is hard to agree on.

```

(class CoordPair
  [subclass-of Object]
  [ivars x y] ; two numbers      ;;;;;; Representation,
                           ;;;;;; as instance variables

  (class-method withX:y: (anX aY)    ;;;;;; Initialization
    ((self new) initializeX:andY: anX aY))
  (method initializeX:andY: (anX aY) ; private
    (set x anX)
    (set y aY)
    self)

  (method x ()                  ;;;;;; Observation of coordinates
  (method y () y)

  (method = (coordPair)          ;;;;;; Equivalence
    ((x = (coordPair x)) & (y = (coordPair y)))

  (method print ()              ;;;;;; Printing
    (left-round print) (x print) (', print) (y print) (right-round print))

  (method * (k)                 ;;;;;; Scaling and translation
    (CoordPair withX:y: (x * k) (y * k)))

  (method + (coordPair)
    (CoordPair withX:y: (x + (coordPair x)) (y + (coordPair y))))
  (method - (coordPair)
    (CoordPair withX:y: (x - (coordPair x)) (y - (coordPair y))))
)

```

§10.1  
Object-oriented  
programming by  
example

625

Figure 10.2: Definition of class CoordPair

To understand the method definitions, we need to understand what names are visible. Unlike a function in a Molecule module, which has access to the representation of every argument whose type is defined in that module, a Smalltalk method has access only to the representation of the receiver; arguments are *encapsulated*. Access to the representation of the receiver is provided by putting the receiver’s instance variables into the environment in which the body of the method is evaluated. For example, in Figure 10.2, method `initializeX:andY:` mentions names `x` and `y`, which are instance variables of the receiver. Access to the representations of the arguments is denied; that is what is meant by “encapsulated.” All we can do with arguments is send messages to them.

Every method also has access to its parameters, as in Scheme, plus any local variables that might be declared with `locals`. And every method can use the special name `self`, which refers to the object receiving the message.<sup>4</sup> The name `self` cannot be assigned to; a method’s receiver is mutated by mutating its instance variables.

With these visibility rules in mind, we can understand the method definitions of class `CoordPair`, which are organized into three groups.

---

<sup>4</sup>Because the receiver does not appear on the list of a method’s formal parameters, referring to it requires a special name. Some object-oriented languages use `this` instead of `self`; others provide syntax by which the programmer can choose a name.

- Instances of class `CoordPair` are created by class method `withX:y:`, then initialized by instance method `initializeX:andY:`. In the class method, the `self` in `(self new)` refers to the *class* object, not to any instance. The class, like all classes, inherits the `new` method from class `Class`; the `new` method creates and answers a new, uninitialized object that is an instance of class `CoordPair`. After sending `new`, class method `withX:y:` finishes by sending the message `initializeX:andY:` to the new object. Method `initializeX:andY:` initializes instance variables `x` and `y`, then answers `self`—when message `withX:y:` is sent to the `CoordPair` class object, the final answer is the newly allocated, correctly initialized instance. This idiom is common: a Smalltalk object is created by sending some message to its class, and because the corresponding class method has no access to the instance variables of the new object, it initializes the object using an instance method. (The class object has no access to the instance variables of any other object, not even one of its own instances.)

Method `initializeX:andY:` is labeled *private*, which means that it is intended to be used only by other methods of the class, or of its subclasses. Private methods are an essential part of object-oriented programming, and different object-oriented languages interpret “private” in slightly different ways. In Smalltalk, private methods are purely a programming convention, which is not enforced by the language. Methods that initialize instance variables are often private.

- An instance of class `CoordPair` is printed by the `print` method, in the form “ $(x, y)$ .” The method’s body is a sequence of message sends. The `print` message may be sent to *any* object; it is part of the protocol for all objects (Section 10.4.1 on page 646). Names `x` and `y` refer to instance variables; `left-round` and `right-round` refer to predefined objects stored in global variables, and `',` is a literal symbol.
- Every instance of class `CoordPair` provides two methods, `x` and `y`, which answer the values of instance variables `x` and `y`.
- Instances of class `CoordPair` are operated on as vectors by methods `*`, `+`, and `-`. The `*` method takes a scalar argument `k` and multiplies the receiver by `k`. The `CoordPair` abstraction is immutable, so the method does not mutate any instance variables; instead it multiplies each instance variable by `k`, then uses the resulting products to create a new object of class `CoordPair`.

The `+` and `-` methods implement vector sum and difference. To compute the sum of the receiver vector `self` and the argument vector `coordPair`, the methods need access not only to the instance variables `x` and `y` of the receiver, but also to the `x` and `y` coordinates of the argument `coordPair`. They get that access by sending messages `x` and `y` to `coordPair`.

The definition of class `CoordPair` illustrates basic Smalltalk. For more advanced techniques, we examine the definitions of classes `Picture` and `TikzCanvas`, which illustrate looping over lists, then the shape classes, which illustrate inheritance.

### 10.1.3 Pictures: programming with lists and blocks

Class `Picture` provides an abstraction of a list of shapes. We can add shapes to the list, and we can draw all the shapes on a *canvas*.

The protocols for pictures are shown in Figure 10.3 on the next page. The class protocol includes just one message: sending `empty` to the class creates an empty

Class protocol for pictures:

empty	The class answers a new instance of itself, which contains no shapes.
-------	---

Instance protocol for pictures:

add: aShape	The shape is added to those remembered by the receiver.
renderUsing: aCanvas	The receiver sends messages to aCanvas, drawing all of its remembered shapes, then answers itself.

§10.1  
Object-oriented  
programming by  
example

627

627. *<shape classes 625> +≡* ▷ 625 629 ▷

```
(class Picture
  [subclass-of Object]
  [ivars shapes] ; the representation: a list of shapes

  (class-method empty ()      ;;;;;;;;;;; Initialization
    ((self new) init))
  (method init () ; private
    (set shapes (List new))
    self)

  (method add: (aShape)      ;;;;;;;;;;; Add a shape,
    (shapes add: aShape)      ;;;;;;;;;;; reply with the picture
    self)

  (method renderUsing: (aCanvas)
    (aCanvas startDrawing)
    (shapes do:                ; draw each shape
      [block (shape) (shape drawOn: aCanvas)])
    (aCanvas stopDrawing)
    self)
)
```

Figure 10.4: Definition of class Picture

picture. The instance protocol includes message `add:`, which adds a shape to a picture, and `renderUsing:`, which asks a picture to draw itself on a canvas. These protocols are implemented by the class definition in Figure 10.4. A picture is represented by the single instance variable `shapes`, which is a list of shapes. An empty picture is initialized using the same idiom as in class `CoordPair`: class method `empty` calls private initialization method `init`, which initializes `shapes` with an empty list. (Class `List` is predefined.)

To add a shape to a picture, we add the shape to the `shapes` list. To render a picture on a canvas, we prepare the canvas, draw each shape, and notify the canvas that drawing is complete.

The operation of drawing each shape is implemented using only message passing; unlike Impcore or  $\mu$ Scheme, Smalltalk does not have any built-in looping syntax. The loop is implemented by sending the `do:` message to the list of shapes. The argument to the `do:` message is a *block*, which is created by evaluating the special syntactic form

$$\exp ::= [\text{block } (\{\text{argument-name}\}) \{\exp\}]$$

Class protocol for a canvas:

<code>new</code>	Answer a new instance of itself, which is not yet prepared for drawing.
------------------	---

Instance protocol for a canvas:

<code>startDrawing</code>	Prepare the receiver for drawing.
<code>stopDrawing</code>	Tell the receiver that drawing is complete.
<code>drawEllipseAt:width:height: aCoordPair aNumber aNumber</code>	The receiver issues drawing commands for an ellipse centered at the point with the given coordinates and with the given width and height.
<code>drawPolygon: aList</code>	The receiver issues drawing commands for a polygon whose vertices are specified by the given list, which is a list of <code>CoordPairs</code> .

Figure 10.5: Protocols for canvases, including `TikzCanvas`

Just like a `lambda` expression in Scheme, a `block` expression evaluates to a closure. But a block is not a function; it is an object, and it is activated when it receives the right sort of message.

In the `renderUsing:` method of class `Picture`, the block is activated once for each shape in the list `shapes`. That's accomplished by sending the `do:` message to the list, with the block as an argument. Sending `do:` is a lot like applying  $\mu$ Scheme's `app` function; the effect is to send the `drawOn:` message to each shape in `shapes`, with the canvas as the argument. This is higher-order imperative programming at its finest: without any special-purpose syntax, we achieve the effect of a `for` loop.<sup>5</sup>

#### 10.1.4 A canvas, on which pictures can be drawn

A canvas object hides a secret: how to draw ellipses and polygons, and how to surround them with the bureaucracy that the rendering engine requires. Class `TikzCanvas`, shown in Figure 10.6 on the facing page, encapsulates what I know about drawing simple pictures with L<sup>A</sup>T<sub>E</sub>X's TikZ package. To switch to a different drawing language, like PostScript, I could define a new class using the same protocol (Exercise 5 on page 724).

The methods of class `TikzCanvas` are ugly—because  $\mu$ Smalltalk does not have strings, the TikZ syntax can be emitted only by a sequence of `print` messages, which are sent both to literal symbols and to predefined Unicode characters like `left-curly`.

Like the `renderUsing:` method of class `Picture`, the `drawPolygon:` method iterates over a list, but here it is a list containing the vertices of the polygon. The block—which is effectively the body of the loop—prints the coordinates of a vertex, followed by the `--` symbol.

The `drawEllipseAt:width:height:` method requires no iteration. Aside from printing, its computational task is to convert width and height to radii, which are what TikZ wants.

#### 10.1.5 Shapes: taking inheritance seriously

The `CoordPair` class illustrates arithmetic and printing, which are found in all languages. The `Picture` and `TikzCanvas` classes illustrate two of Smalltalk's built-in

---

<sup>5</sup>The `drawOn:` message is part of the protocol for shapes; it is defined below.

629. ⟨shape classes 625⟩+≡

▷ 627 630 ▷

```
(class TikzCanvas
    ;;;;;;; Encapsulates TikZ syntax
    [subclass-of Object]
    (method startDrawing ()
        ('begin print)
        (left-curly print) ('tikzpicture print) (right-curly print)
        (left-square print) ('x=4pt,y=4pt print) (right-square println))

    (method stopDrawing ()
        ('end print)
        (left-curly print) ('tikzpicture print) (right-curly println))

    (method drawPolygon: (coord-list)
        ('\draw print) (space print)
        (coord-list do: [block (pt) (pt print) ('-- print)])
        ('cycle print)
        (semicolon println))

    (method drawEllipseAt:width:height: (center w h)
        ('\draw print) (space print) (center print) ('ellipse print)
        (left-round print)
        ((w div: 2) print) (space print) ('and print) (space print)
        ((h div: 2) print)
        (right-round print)
        (semicolon println))
    )
```

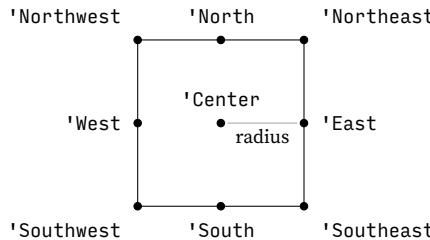
§10.1  
Object-oriented  
programming by  
example

629

Figure 10.6: Definition of class TikzCanvas

abstractions: lists and blocks, which are similar to abstractions found in  $\mu$ Scheme (lists and closures). Our final example—shapes—illustrates inheritance, which is found only in object-oriented languages. Inheritance is used to create related abstractions that share functionality.

Notionally, every shape is inscribed in a square. The square is specified by its location and its *radius*, which is half its side. A square also has nine “control points,” each of which is named with a symbol:



We draw three different shapes, and for each each shape,

- The circumscribing square and the control points are the same.
- The drawing is different.

Because the circumscribing square and the control points are the same for all shapes, all shapes can share the implementation of the first part of the shape protocol shown in Figure 10.7 on the next page (messages `location:`, `locations:`,

Class protocol for shapes:

<b>new</b>	The class answers a new instance of itself, whose radius is 1 and whose center control point is at the coordinate origin.
------------	---

Instance protocol for shapes:

<b>location: aSymbol</b>	Answers the location of the receiver's control point that is named by the given symbol ('North, 'South, ...). If the given symbol does not name a control point, the result is a checked run-time error.
<b>locations: symbols</b>	Given a list or array <b>symbols</b> , each of which names a control point, return the list of the corresponding locations. If any symbol in <b>symbols</b> does not name a control point, the result is a checked run-time error.
<b>adjustPoint:to: aSymbol aCoordPair</b>	Changes the location of the receiver so that the control point named by the given symbol is located at the given coordinate pair. Answers the receiver. If the given symbol does not name a control point, the result is a checked run-time error.
<b>scale: aNumber</b>	Multiply the size (radius) of the receiver by the given number.
<b>drawOn: aCanvas</b>	The receiver draws itself on the given canvas.

Figure 10.7: Protocols for shapes

**adjustPoint:to:, and scale:).** But because each shape is drawn differently, each shape needs its own implementation of **drawOn:.** This kind of “sharing with a difference” is exactly what inheritance is designed to provide.

- Shared methods **location:, locations:, adjustPoint:to:, and scale:.** are defined in a common *superclass*, which I call **Shape**. Class **Shape** (Figure 10.8 on the facing page) also defines instance variables **center** and **radius**, which specify the square in which the shape is inscribed.
- Each **drawOn:.** method is defined in a unique *subclass*, which inherits the representation and the other methods of **Shape**.

My design separates concerns: the superclass knows only how to use control points to move and resize shapes, and a subclass knows only how to draw a shape.

Because each subclass defines only the **drawOn:.** method, the subclass definitions are tiny, requiring little programming effort. For example, I draw a circle by drawing an ellipse whose width and height are both twice the radius of the circumscribing square:

```
630. <shape classes 625>+≡                                     ▷629 631b▷
  (class Circle
    [subclass-of Shape]
    ;; no additional representation
    (method drawOn: (canvas)
      (canvas drawEllipseAt:width:height: center (2 * radius) (2 * radius)))
  )
```

What's new here is in the **subclass-of** form: class **Circle** inherits from **Shape**, not from **Object**. So it gets the representation and methods of **Shape**.

**631a.** *(definition of class Shape 631a)≡*

```

(class Shape
  [subclass-of Object]
  [ivars center radius] ;; CoordPair and number

  (class-method new ()
    ((super new) center:radius: (CoordPair withX:y: 0 0) 1))
  (method center:radius: (c r) ;; private
    (set center c)
    (set radius r)
    self)

  (method location: (point-name)
    (center + ((point-vectors at: point-name) * radius)))

  (method locations: (point-names) [locals locs]
    (set locs (List new))
    (point-names do: [block (pname) (locs add: (self location: pname))])
    locs)

  (method adjustPoint:to: (point-name location)
    (set center (center + (location - (self location: point-name))))
    self)

  (method scale: (k)
    (set radius (radius * k))
    self)

  (method drawOn: (canvas)
    (self subclassResponsibility))
)

```

§10.1  
Object-oriented  
programming by  
example

---

631

Figure 10.8: Definition of class Shape

As another example, class Square also inherits from Shape. Its `drawOn:` method draws the polygon whose vertices are the four corners of the square:

**631b.** *(shape classes 625)+≡* △630 632a△

```

(class Square
  [subclass-of Shape]
  ;; no additional representation
  (method drawOn: (canvas)
    (canvas drawPolygon:
      (self locations: '(Northeast Northwest Southwest Southeast))))
)

```

The `locations:` method converts the list of symbols into a list of coordinate pairs, which is then sent with `drawPolygon::`. You can define other shapes, including Triangle, in the Exercises.

Both Circle and Square inherit from Shape, whose definition is shown in Figure 10.8.

- A shape is created by sending message `new` to its class. Class method `new` uses the now-familiar pattern of creating a new object, then using a private method to initialize it. But there is something different here: to allocate the new object, I send the `new` message to `super`, not to `self`. Sending a message to `super` is a language feature that is described in detail in Section 10.3.4

on page 642; for now, accept that sending new to super sends the new message to class Shape, but in a way that dispatches to the new method inherited from class Class. Sending new to self would dispatch to the redefined new method, which would cause infinite recursion.

- The location associated with a control point is computed by the location: method, which finds a vector associated with the control point, multiplies that vector by the radius, and adds the resulting vector to the center. To associate a vector with each control point, I define a global variable point-vectors of class Dictionary (page 655).

**632a.** *(shape classes 625) +≡*  
(val point-vectors (Dictionary new))  
(point-vectors at:put: 'Center' (CoordPair withX:y: 0 0))  
(point-vectors at:put: 'East' (CoordPair withX:y: 1 0))  
(point-vectors at:put: 'Northeast' (CoordPair withX:y: 1 1))  
; ... six more definitions follow ...

&lt;631b

Sending at:put: to a dictionary stores a key-value pair. Sending at: to the dictionary, as in method location: of class Shape, answers the value stored with the given key.

- A list of locations is computed by the locations: method, which uses the same do: as the renderUsing: method of the Picture class. In locations:, the do: loop adds each location to a list locs, which is stored in a local variable of the method.
- A shape is moved by adjusting a given control point to a given location. The adjustPoint:to: method updates the center of the shape, computing a new center by sending + and - messages to objects of class CoordPair. This arithmetic is hidden from clients that send messages to shapes; such clients know only about the names of control points, not about the arithmetic.
- A shape is changed in size by changing its instance variable radius, which is done in the scale: method.
- A shape is drawn by sending it the drawOn: message, but if drawOn: is sent to an instance of class Shape, the message send fails with a special run-time error:

**632b.** *(transcript 621a) +≡*  
-> ((Shape new) drawOn: canvas)  
Run-time error: subclass failed to implement a method it was responsible for  
Method-stack traceback:  
Sent 'subclassResponsibility' in shapes.smt, line 83  
Sent 'drawOn:' in standard input, line 41

&lt;622c 641a&gt;

Sending drawOn: to an instance of class Shape causes an error because we're not supposed to create instances of class Shape—we're supposed to instantiate only *subclasses* like Circle, Square, and Triangle. A class like Shape, which is intended to collect reusable methods but not to be instantiated, is called an *abstract class*. The purpose of an abstract class is to make it easy to define subclasses, which we can then instantiate. That purpose is signaled by defining method drawOn: to send the subclassResponsibility message.

In Smalltalk, the “abstract class,” like the “private method,” is a programming convention, not a language feature. Client code should never create an instance of an abstract class; it should instead create instances of concrete subclasses.

### 10.1.6 Object-orientation means dynamic dispatch

With this extended example to learn from, we can start to identify the essence of Smalltalk. In some ways, Smalltalk resembles Impcore:

- Smalltalk has no static types.
- Idiomatic code is imperative: it uses set and it evaluates expressions in sequence.
- A message send, like an Impcore function call, always uses a name.

§10.1  
Object-oriented  
programming by  
example

633

In Impcore, calling a function requires you to name the function, and in Smalltalk, sending a message requires you to name the message. But in Impcore, a function name always refers to the same function, no matter what arguments are passed; the code that is executed is determined by the caller. Whereas in Smalltalk, a message name may be implemented by any number of methods, and the code that is executed—the method that a message is *dispatched* to—is determined by the receiver. For example, in class `CoordPair`, the `+` and `-` being defined are dispatched to when the `+` or `-` message is sent to an object of class `CoordPair`. But in the body of each method, the `+` and `-` messages are sent to *numbers*, and those messages dispatch to different methods.

Even a single message send from a single place in the source code could be dispatched to different methods on different evaluations. For example, the `renderUsing:` method of class `Picture` sends the `drawOn:` message to each of a list of shapes. But depending on whether a shape is a `Circle`, `Square`, or `Triangle`, that message dispatches to a different method. At compile time, we don't know what method will be executed—the method is chosen at run time, by examining the receiver. The algorithm that associates a message name with a method is called *dynamic dispatch*, and it is the essence of Smalltalk. The algorithm is explained in Section 10.3.4 on page 641.

### 10.1.7 Review and analysis

The shapes example illustrates message sends, method dispatch, protocols, representation, object creation, and inheritance.

- A computation is implemented by a small army of objects that exchange *messages*. Each message is dispatched to a *method*, which is chosen at run time based on the class of the *receiver*. As often as not, a method is executed for its side effects, like updating instance variables, or printing.
- A method may be designated *private*, which means that its use should be limited to certain senders. For example, a message designated as private might be sent only by an object to itself, or by a class to its instances. In Smalltalk, such a designation is purely advisory: it records a programmer's intent, but it is not enforced. In practice, “*private*” may also designate a message that is intended to be sent by instances of a class to other objects that are believed to be of the same class.
- The set of messages an object understands, together with the behavior that is expected in response to those message, is the object's *protocol*. Smalltalk does not have a static type system, and protocols are specified informally. An object's protocol is determined by the class of which the object is an *instance*. The same protocol can be implemented by many different classes, which are often related by *inheritance*. But they don't have to be related: any

canvas      622c

programmer can define an entirely new class, like `PSCanvas` (Exercise 5 on page 724), that implements an existing protocol.

Although Smalltalk does not have a static type system, it does have other ways of ensuring that a message is accompanied by the correct number of arguments: for example, the number of colons in a message's name equals the number of arguments the message expects. This little requirement costs almost nothing, but like a type, it serves as documentation that is checked by the compiler. And it catches mistakes at compile time. In proper Smalltalk-80 syntax, the colons break the name into parts, which are *interleaved* among the arguments. This concrete syntax, while unusual, leads to code that I enjoy reading (Section 10.11).

- An object is represented by a fixed set of named *instance variables*. An object's instance variables are visible only to the object's own methods; other objects, even other instances of the same class, cannot see the instance variables. Chapters 8 and 9 discuss the role of a *sum of products* in representing linked data structures. In Smalltalk, an object's instance variables form the product in a sum of products.
- A new object is created by sending a message to its class, often the `new` message. The corresponding class method can initialize the object's instance variables using an instance method, which often exists only for this purpose. Occasionally a new object is created by evaluating some other syntactic form, like a block expression, a literal integer, a literal symbol, or a class definition.
- Every class definition specifies a *superclass*, from which the new class inherits both methods and instance variables. A class can be defined whose sole purpose is to be inherited from, like the `Shape` class. Such a class, which is not intended for instantiation, is called an *abstract class*. A class whose methods send the `subClassResponsibility` message is always abstract.

It's also worth thinking about how Smalltalk compares with procedural and functional languages.

- A method is like a function or a procedure, and an object is like a bundle of functions that share mutable state. And since objects themselves are sent with messages and are returned as answers, object-oriented programming is like programming with higher-order functions, only more so: any time you send an object with a message, you may be setting off a higher-order computation. This is why we study functions before objects.
- In Impcore, Scheme, and ML, conditional expressions are a normal way to do business. We would ask, “are you a circle or a triangle or a square,” and we would draw the shape accordingly. Not in Smalltalk! Instead, we do our conditional decision-making via dynamic dispatch: just send the `drawOn:` message, and based on the class of the receiver, method dispatch makes the choice for us.

Multiple classes that all respond to `drawOn:` form a sum in the *sum of products* discussed in Chapters 8 and 9. And because `drawOn:` is dispatched dynamically, we can easily add new elements to the sum—like new shapes—without touching any existing code. This open extensibility, which can't be replicated using algebraic data types, is characteristic of object-oriented programming.

- Making decisions via dynamic dispatch influences the structure of programs: compared with the number of function definitions and function calls in a functional program, a similar object-oriented program likely has more method definitions and more message sends. For example, to draw a picture, we send messages to shapes, which send messages back to the picture object. For a programmer who is used to procedural or functional programming, distributing a computation over multiple methods defined on multiple classes may make it hard to identify the algorithm: the “procedure,” meaning the sequence of steps being performed, is not written in one place.

Moving from procedural programming to object-oriented programming requires a change in mindset. If you are used to seeing the steps of an algorithm laid out in sequence in a single block of code, you are going to be frustrated by object-oriented programming: what the procedural programmer thinks of as a simple sequence of commands is usually distributed over many methods defined on many classes, and it may be hard to identify. The object-oriented programmer focuses on the protocols. If the protocols are well designed and each object does one job, the individual methods become much easier to understand than a typical procedure full of conditionals and `while` loops. Moreover, although the conceptual procedure may be hard to identify, the individual commands become easy to reuse: for example, we can use inheritance to create new shapes that differ only in their drawing methods. Learning to design with protocols and message passing—though difficult—is the key to becoming a productive object-oriented programmer.

- If you want to learn about open, extensible systems, you need something to extend. Looking just at the Smalltalk language gives too narrow a view—Smalltalk-80 ships with a huge hierarchy of predefined classes. The basic reference on Smalltalk-80, by [Goldberg and Robson \(1983\)](#), devotes only 90 pages to the language, but 300 pages to the predefined classes. To program effectively in Smalltalk, you must learn about the predefined classes, like the lists and dictionaries used in the pictures example.

## 10.2 DATA ABSTRACTION ALL OVER AGAIN

Like an abstract data type in a Molecule module, a Smalltalk class defines an abstraction. Everything said about abstractions in Section 9.6 applies to Smalltalk as well. You need to think about and specify abstractions. You need to design representations. And to get your code right, you must keep in mind your abstraction functions and representation invariants.

Much of what is said about program design in Section 9.6 also applies to program design with objects. But objects are not the same as abstract types; they have different capabilities, and they are customarily used in different ways.

- Abstractions and their interfaces still have to be designed. Abstractions are still either mutable or immutable. But Smalltalk, which is built on set expressions, favors mutable abstractions, while Molecule, which is built on case expressions, favors immutable abstractions. And in Smalltalk, as in other object-oriented languages, the customary, built-in abstractions also favor mutability.
- As in Molecule, operations can be classified as creators, producers, observers, or mutators. But in Molecule, all these operations are specified in

*Data abstraction for “open” and “closed” systems*

Systems based on abstract data types are “closed,” and can thereby be made reliable. Systems based on objects and protocols are “open,” and can thereby be made extensible.

Using abstract data types, as in Molecule, a representation is hidden in a module, and access is mediated by a module type that exports an abstract type  $T$ . The line between “visible” and “hidden” is fixed:

- Only operations inside the module can see the representation of values of type  $T$ .
- An operation inside the module can see the representation of *any* value of type  $T$ , no matter where it comes from.

Once the module is sealed, the representation of  $T$  can’t be seen by new modules. And new abstractions can’t be made compatible with  $T$ : any new module’s abstract types, even if defined identically to  $T$ , cannot be used in a context that expects  $T$ . The only way to add new, compatible code is to break open an existing module and edit it. Sometimes you even have to change existing interfaces.

Using objects, as in Smalltalk, a representation is hidden inside an object, and access is mediated by a protocol. The line between “visible” and “hidden” is drawn differently:

- Only methods defined on an object can see that object’s representation.
- A method defined on an object can see *only* that object’s representation, not the representation of any other object. This principle, which Cook (2009) calls the *autognostic principle*, is a defining characteristic of object-oriented programming.

The representation of the object can be seen by new methods, provided those methods are defined in a subclass. And new abstractions can be made compatible with old ones: if a new object implements an old protocol, than it can be used in any context that expects the old protocol.

one place: a module type. In Smalltalk, while mutators, producers, and observers are all part of an instance protocol, creators are not: you can’t create an object by sending a message to an instance that doesn’t yet exist. You usually create an object by sending a message to the object’s class, and an object’s creators are therefore part of a different protocol.

Objects and abstract data types support data abstraction in different ways, and these differences lead to differences in client code.

- In Molecule, an abstraction is specified by a named abstract data type, and operations of the abstraction work only with values of that type. In Smalltalk, an abstraction is specified by a protocol, and operations of the abstraction work with objects of any class that implements the protocol. If an object walks like a fraction, talks like a fraction, and exchanges messages in the protocol for fractions, then for programming purposes, it’s a fraction—even

if its definition doesn't inherit from class `Fraction`. This property is called *behavioral subtyping*, or by Ruby programmers, "duck typing."

- In a Smalltalk method, `self` is the only object whose representation is accessible. Objects sent as arguments must implement the expected protocols, but the method cannot see their representations. Methods that wish they could see arguments' representations, like `+` and `-` on class `CoordPair`, have to send messages to those arguments instead—usually messages like `x` and `y`. In Molecule, the analogous operations get to see all relevant representations directly.
- A Smalltalk class can specify not a single abstraction, but a family of related abstractions. Members of the family are implemented by subclasses. Examples shown in this chapter include shapes, numbers, and "collections." Additional examples found only in full Smalltalk include input streams and user-interface widgets.

Compared with Molecule modules, Smalltalk classes are more flexible and easier to reuse, but operations that wish to see multiple representations are harder to implement. And unlike Molecule, Smalltalk does not have a type checker, so it cannot guarantee that every operation is given abstractions that it knows how to work with. To understand how these tradeoffs work, study the design and implementation of the `Collection` hierarchy, shown in Sections 10.4.5, 10.6.1, 10.6.2, and 10.8.

## 10.3 THE $\mu$ SIMPLY TALK LANGUAGE

### 10.3.1 Concrete syntax and its evaluation

$\mu$ Smalltalk shares some expression forms with Impcore and  $\mu$ Scheme; for example, all three languages have variables, `set`, and `begin`. But instead of function application,  $\mu$ Smalltalk has message send. In a send, the message, like an Impcore function, is identified by name.<sup>6</sup> Every message name has an *arity*, which is the number of arguments that the message expects. If the *message-name* begins with a nonletter, its arity is 1. If the *message-name* begins with a letter, its arity is the number of colons in the name.

In addition to message send and to forms that are shared with Impcore and  $\mu$ Scheme, a  $\mu$ Smalltalk expression can be a *block*, which is like a `lambda` abstraction in Scheme. A block specifies zero or more formal parameters and a body. The body contains a sequence of expressions, which are executed in, well, sequence. Because parameterless blocks are used frequently to implement control flow,  $\mu$ Smalltalk provides a special syntax for a parameterless block: the  $\mu$ Smalltalk expression `{e1 ... en}` is syntactic sugar for `[block () e1 ... en]`. This syntax is borrowed from Ruby, a descendant of Smalltalk-80.<sup>7</sup>

$\mu$ Smalltalk also has three forms not related to anything in Impcore or  $\mu$ Scheme: `return`, `primitive`, and `compiled-method`. The complete set of expression forms is defined by this grammar:

<sup>6</sup>Impcore's functions are not values, but  $\mu$ Scheme's functions *are* values. Are  $\mu$ Smalltalk's functions values? It depends what you mean by "function." If you mean "block," then yes, blocks are values. If you mean "message name," then no, message names are not values. (In Smalltalk-80, the message name is called a *message selector*, and it can be given as a value with the `perform:` message.)

<sup>7</sup>Smalltalk-80 writes blocks in square brackets, but in this book, square brackets work just like round brackets—you mix square and round brackets in whatever way you find easiest to read.

```


$$\begin{aligned} \text{exp} ::= & \text{literal} \\ | & \text{variable-name} \\ | & (\text{set variable-name } \text{exp}) \\ | & (\text{begin } \{\text{exp}\}) \\ | & (\text{exp message-name } \{\text{exp}\}) \\ | & [\text{block } (\{\text{argument-name}\}) \{\text{exp}\}] \\ | & \{\{\text{exp}\}\} \\ | & (\text{return exp}) \\ | & (\text{primitive primitive-name } \{\text{exp}\}) \\ | & (\text{compiled-method } (\{\text{argument-name}\}) [[\text{locals } \text{locals}]] \{\text{exp}\}) \end{aligned}$$


```

Literal expressions include integers and symbols as in  $\mu$ Scheme. And in place of list literals,  $\mu$ Smalltalk has array literals.

$\begin{aligned} \text{literal} ::= & \text{integer-literal} \\   & 'name \\   & '(\{\text{array-element}\}) \end{aligned}$	$\begin{aligned} \text{array-element} ::= & \text{integer-literal} \\   & name \\   & (\{\text{array-element}\}) \end{aligned}$
---	---

Expression forms for literals, variables, set, and begin are evaluated as in Impcore or  $\mu$ Scheme. But message send, although it may look like a function call, is its own beast; it is discussed in Section 10.3.4 below. A block, whether it is written using the block keyword or the curly braces, is evaluated very much like  $\mu$ Scheme's lambda: it captures the environment and forms a closure. The return form is a control operator that is subtly different from the control operators in Chapter 3: evaluating a return terminates the method in which the return appears. A primitive expression is evaluated almost like a call to a primitive function in Impcore: the primitive is named, it takes arguments, and it returns one object as a result. And a compiled-method expression evaluates to a method object, which can be added to an existing class.

The definition forms of  $\mu$ Smalltalk include the usual val, exp, define, use, and unit tests that we see in Impcore and  $\mu$ Scheme, as well as a class definition. But in  $\mu$ Smalltalk, define defines a block, not a function.

```


$$\begin{aligned} \text{def} ::= & (\text{val variable-name } \text{exp}) \\ | & \text{exp} \\ | & (\text{define block-name } (\text{formals}) \text{ exp}) \\ | & (\text{class subclass-name} \\ & \quad [\text{subclass-of superclass-name}] \\ & \quad [[\text{ivars } \{\text{instance-variable-name}\}]] \\ & \quad \{\text{method-definition}\}) \\ | & (\text{use file-name}) \\ | & \text{unit-test} \\ \text{unit-test} ::= & (\text{check-expect } \text{exp } \text{exp}) \\ | & (\text{check-assert } \text{exp}) \\ | & (\text{check-error } \text{exp}) \\ | & (\text{check-print } \text{exp } \text{name}) \\ \\ \text{method-definition} ::= & (\text{method } \text{method-name } (\text{formals}) [[\text{locals } \text{locals}]] \{\text{exp}\}) \\ | & (\text{class-method } \text{method-name } (\text{formals}) [[\text{locals } \text{locals}]] \{\text{exp}\}) \\ \text{formals} ::= & \{\text{formal-parameter-name}\} \\ \text{locals} ::= & \{\text{local-variable-name}\} \end{aligned}$$


```

A method is specified by giving its formal parameters and body.

The true-definition forms `val`, top-level `exp`, and `define` are evaluated exactly as in  $\mu$ Scheme. Class definitions are unique to  $\mu$ Smalltalk; a class definition adds a new *class object* to the global environment. The creation of class objects is discussed in detail in Section 10.10.4 on page 702.

To understand how message sends and class objects work, we should first study  $\mu$ Smalltalk's values.

### 10.3.2 Values

§10.3  
*The language*

639

In  $\mu$ Smalltalk, every value is an object. As in Smalltalk-80, you have to learn about the predefined objects and their protocols. I describe the protocols in Section 10.4, but there are a lot of them, and a preview will help.

#### *Properties shared by all values*

Every value is an object, and every object is an instance of some class. Class `Object` is the root of the class hierarchy, so every other class inherits from it. This means that every object responds to messages defined on instances of class `Object`. These messages include `=`, `==`, `println`, `class`, and a number of others. They are a bit like predefined functions in other languages, with an important difference: you can redefine them.

#### *The undefined object, nil*

The value `nil` is the sole instance of class `UndefinedObject`. It conventionally represents a bad, missing, undefined, or uninitialized value. This usage is quite different from the use of `nil` in some dialects of Lisp, Scheme, Prolog, and ML, where `nil` represents the empty list.

#### *Other predefined objects*

A literal symbol, written '`name`', is an object of class `Symbol`. Like all objects, symbols can usefully be printed and compared for equality. A symbol also understands a hash message, to which it responds with an integer.

The Boolean objects `true` and `false` are predefined. A Boolean object responds to the message `ifTrue:ifFalse:`, whose arguments are two *blocks*. Depending on the value of the receiver, one block or the other is evaluated. Boolean objects also respond to many other messages, including `and:`, `or:`, and `not`.

A block is an object that understands messages `value`, `value:`, `value:value:`, and so on. A block behaves just like a `lambda` abstraction, except that it is activated by sending it one of these messages, not by applying it—Smalltalk does not have function application. A block also responds to the message `whileTrue:`, whose argument is a block. As long as sending `value` to the receiver answers `true`, the argument block is executed.

#### *Classes are objects*

Every class definition creates a class object, which itself has a class that inherits (through intermediaries) from class `Class`. A class object can respond to new and possibly to other class-dependent initialization messages. In  $\mu$ Smalltalk, a class object also responds to messages that provide information about the class, or even alter it: `printProtocol` prints all the messages that a class and its instances know how to respond to; `superclass` answers the class object of the receiver's superclass,

if any; and `addSelector:withMethod:` can add a new method or change an existing one.

An object that represents a class is also an instance of a class. The object representing class *C* is an instance of *C*'s *metaclass*, and in fact it is the *only* instance of *C*'s metaclass. If an object *c* is an instance of class *C*, the methods defined on *C* determine what messages *c* can respond to. Similarly, the methods defined on *C*'s metaclass determine what messages class *C* can respond to. In other words, the metaclass provides a place to put *C*'s class methods. Because every metaclass inherits from class *Class*, every class object responds to the new message, which is defined on class *Class*.

In full Smalltalk-80, class objects and metaclasses provide many *reflective* facilities, which enable the language to manipulate its own objects. You can learn the names of methods and instance variables, execute methods by name, create new subclasses, and much more, all by sending messages. Such reflective facilities are powerful—making it possible to implement Smalltalk-80 class browsers and debuggers in Smalltalk itself.

### 10.3.3 Names

Just as Impcore has distinct name spaces for functions and values,  $\mu$ Smalltalk has distinct name spaces for message names and variables. The name of a variable is resolved *statically*; that is, we can tell by looking at a class definition what each variable name in each method must stand for. The name of a message is resolved *dynamically*: until a message is sent, we can't tell what method will be executed to respond to it.

The name of a variable stands for one of the following possibilities:

1. A formal parameter of a block
2. A local variable of a method
3. A formal parameter of a method
4. An instance variable of an object
5. A global variable

A name stands for the first possibility that is consistent with the source code where the name appears. For example, if a name appears in a block, is not a formal parameter of that block, but is a local variable of the method in which the block appears, then it's a local variable (possibility 2). Not all possibilities apply in all contexts; for example, a name in a top-level expression doesn't appear inside a method or a class definition, so it must stand either for a parameter of a block or for a global variable.

In possibility 4, a name *x* appearing in a method of class *C* can stand for an instance variable if *x* is declared as an instance variable in class *C* or in any of *C*'s *ancestors*. (To avoid confusion, if *x* is declared as an instance variable of an ancestor, it may not also be declared as an instance variable of class *C*.)

The rules for the meanings of names enforce a form of encapsulation: *only a method executed on an object has access to the instance variables of that object*. In other words, an object's instance variables are hidden from all other objects; not even another object of the same class can get access to them. To an object, all *other* objects are abstract. (See also Exercise 36 on page 731, part a.) This encapsulation mechanism differs from Molecule's mechanism: in Molecule, if an operation is located inside a module that defines an abstract type, it automatically has access to the representation of *any* value of that type. A Smalltalk method, by contrast, has access to the representation of only its receiver.

The names `self` and `super` are not, properly speaking, variables; for one thing, `self` and `super` cannot be mutated with `set`. Within a method, `self` stands for the object that received the message which caused the method to be executed. The name `super` stands for the same object, but with different rules for method dispatch, which are described in the next section.

Unlike the name of a variable, the name of a message—called a *message selector*—does not, by itself, tell us what method will be executed when the message is sent. Instead, the method is determined by a *combination* of the message selector and the object to which the message is sent. This object, the receiver, may be different on different executions, even on different iterations of the same loop—and so may be the method. The process by which a method is identified happens at run time, and it is called *dynamic dispatch*.

#### 10.3.4 Message sends, inheritance, and dynamic dispatch

Except when a message is sent to `super`, an object of class  $C$  responds to any message for which a method is defined in  $C$ —or in any of  $C$ 's ancestors. When object of class  $C$  receives a message with selector  $m$ , it looks in  $C$ 's definition for a method named  $m$ . If that fails, it looks in the definition of  $C$ 's superclass, and so on. When it finds a definition of a method for  $m$ , it executes that method. If it reaches the top of the hierarchy without finding a definition of  $m$ , an error has occurred: “message not understood.”

For example, when `location:` is sent to an object of class `Circle`, class `Circle` does not define a `location:` method, but `Circle`'s superclass, class `Shape`, *does* define a `location:` method, so the message is dispatched to the `location:` method defined on class `Shape`. On the other hand, if `drawOn:` is sent to an object of class `Circle`, it is immediately dispatched to the `drawOn:` method defined on class `Circle`—class `Shape` is not consulted.

Unlike an ancestor's instance variable, an ancestor's method can be redefined. When a method is defined in  $C$  and also in an ancestor, and the relevant message is sent to an object of class  $C$ , it is  $C$ 's definition of the method that is executed. This method-dispatch rule produces outcomes that might surprise you, so let's look at a contrived example.

Suppose classes `B` (the superclass or ancestor) and `C` (the subclass) are defined as follows:

641a. *(transcript 621a)* +≡

```
-> (class B [subclass-of Object]
     (method m1 () (self m2))
     (method m2 () 'B))
-> (class C [subclass-of B]
     (method m2 () 'C))
```

△632b 641b ▷

Method `m2`, which is defined in superclass `B`, is also defined in subclass `C`. We say that `m2` is *overridden*.

Object B

When message `m1` or `m2` is sent to an instance of class `B` or `C`, what happens? When `m1` or `m2` is sent to an instance of class `B`, the answer is the symbol `B`. And when `m2` is sent to an instance of class `C`, method dispatch finds the definition of `m2` within `C`, and the result is symbol `C`. These cases are easy. The potentially surprising case occurs when `m1` is sent to an instance of class `C`:

641b. *(transcript 621a)* +≡

```
-> (val x (C new))
-> (x m1)
C
```

△641a 645a ▷

Here is what happens:

1. When `m1` is sent to `x`, `x` is an instance of class `C`, but class `C` does not define method `m1`. But `m1` is defined by `C`'s superclass, class `B`. The message is dispatched to `B`'s `m1` method, which executes.
2. `B`'s `m1` method sends message `m2` to `self`—which is to say, to `x`. The search for method `m2` begins in `x`'s class, namely `C`.
3. A definition of `m2` is found in `C`, and the message send dispatches to it. Executing class `C`'s `m2` method answers the symbol `C`.

This example illustrates a crucial fact about Smalltalk: *the search for a method begins in the class of the receiver*. Many more examples of method dispatch appear below.

Does method dispatch *always* begin in the class of the receiver? Almost always. A message sent to `super` is dispatched in a special way. The message is sent to `self`, but the method search starts in the superclass of the class in which the message to `super` appears. That is, unlike the starting place for a normal message send, the starting place for a message to `super` is *statically determined*, independent of the class of the receiver `self`. This behavior is useful when one wants to guarantee to execute a particular method from the superclass.

We typically send to `super` when we wish to *add* to the behavior of an inherited method, not simply replace it. The most common examples are class methods that initialize objects, like method `new` in class `Shape`. Every class has a `new` method. A properly designed `new` method not only allocates a new object but also establishes the private invariants of its class. Simply sending `new` to `self` executes only the `new` method defined on the class of the object being created. But if there are invariants associated with the superclass, those invariants need to be established too. The idiomatic way to establish *all* the invariants is for each subclass to send `new` to `super`, establishing the superclass invariants, then execute additional code to establish any subclass invariants. By “chaining together” `new` methods in this way, we establish the invariants of every superclass. If a subclass has no invariants of its own, it can take a shortcut and simply inherit `new`.

### 10.3.5 Equality: identity and equivalence

In Section 10.4 below, we dive into Smalltalk’s initial basis, starting with the messages that every object understands. But first, it’s time to confront a problem we’ve danced around since Chapter 2: when should two objects be considered equal? This problem is so central, and yet so seldom addressed well, that when you encounter a new programming language, equality is one of the first things you should look at. The designer of every language has to say whether equality is built in, what it means, and where it can be used.

Almost all languages support constant-time equality tests on simple, scalar values like integers and Booleans. Beyond those values, there are many design choices, and when considering equality on more structured data like records, sums, and arrays, there is no single right answer (Noble et al. 2016). Moreover, when you have abstract types, it is all too easy to provide a wrong answer. Before we examine what’s been done wrong and what we need to do right, let’s review some designs you’ve already seen.

- C has only one form of equality, and it applies only to scalar data (integers, Booleans, floating-point numbers, and similar) and to pointers. Two pointers are equal if and only if they point to the same memory; viewed at a higher

level of abstraction, C's `==` operator tests for *object identity*. (A well-known beginner's mistake is to use `==` to compare strings for equality; that comparison demands `strcmp`.) Structured data like `structs` and `unions` cannot be compared for equality, and C famously does not have arrays—only pointers and memory.<sup>8</sup>

- $\mu$ Scheme has two forms of equality, written `=` and `equal?`. The `=` operator works only on scalar data; given two pairs, it always returns `#f`—according to  $\mu$ Scheme's `=`, a cons cell is not even equal to itself. The `equal?` predicate, on the other hand, is *structural similarity*; it returns `#t` whenever two values have the same structure, even if the structure is arbitrarily large.

Full Scheme, in which cons cells are mutable, has a more principled design, but it also makes some compromises regarding efficiency. Function `equal?` acts as in  $\mu$ Scheme, providing structural similarity. Function `eqv?` provides object identity, and it compares not only scalar data but also cons cells, vectors, and procedures. And function `eq?` provides a predicate that gives object identity on structured data, but given numeric data and character data, it is more efficient but less predictable than `eqv?`.

- Standard ML has “polymorphic equality” comparison on integers, Booleans, and mutable reference cells, but not on functions or floating-point numbers. Types that can be compared using polymorphic equality are said to “admit equality”; there is even a special family of type variables that can be instantiated only with types that admit equality. A type of constructed data (tuple type, record type, or algebraic data type) admits equality if and only if all its components admit equality. Polymorphic equality hacks together object identity (for arrays and references) with structural similarity (for constructed data) in a hybrid that gets common cases right.

OCaml, like Scheme, offers two forms of equality; the `=` sign means structural similarity, and the `==` means object identity (and is fully defined for mutable types only).

All these designs get abstract data wrong. For evidence, review the association lists in Section 2.9 on page 134. Two association lists should be considered equal if and only if each contains all the key-value pairs found in the other. This equivalence relation, which is the correct one, is much coarser than equivalence of representation. If two association lists have the same representation, they definitely represent the same associations, and therefore if two association lists represent different associations, they definitely have different representations. But two association lists may have different representations and yet represent the same associations. If you use Scheme's `equal?` or ML's polymorphic equality, you get wrong answers. Smalltalk's *built-in* `=` and `==` messages, which correspond roughly to  $\mu$ Scheme's `equal?` and `=`, pose the same risk. To be sure their answers are right, classes may have to override the built-in `=` method. Knowing how to override the built-in methods requires deep understanding of what it could mean to consider two objects equivalent.

### *Notions of equivalence*

To get equality right, Smalltalk uses the same methodology as Molecule. The methodology is based on a central principle of programming-language theory, called

---

<sup>8</sup>“Does not have arrays” stretches the truth. In initialized data, array notation and pointer notation mean different things. On this topic, I recommend consulting Van der Linden (1994, Chapter 4).

*observational equivalence*. The pointy-headed theory idea is, speaking informally, that “two things are observationally equivalent if there is no computational context that can distinguish them.” If we specialize this idea for equality and programming, we get something that makes more sense to a programmer who doesn’t normally talk about computational contexts:

*Two values should be considered equal if no program can tell them apart.*

This principle is the guiding principle behind ML’s polymorphic equality. The principle has two immediate consequences:

- When you’re comparing mutable abstractions, like dictionaries, an object may compare equal only with itself. That’s because if you have two different mutable objects, you can tell them apart by mutating one and seeing that the other doesn’t change. Therefore, on mutable data, equality must be implemented as *object identity*.
- When you’re comparing immutable, non-atomic representations like large integers, structurally similar representations should be considered equal.

In Smalltalk, object identity is implemented by the `==` method. Here is how object identity is defined on class `Object`:

**644a.** *(methods of class Object 644a)≡*

```
(method == (anObject) (primitive sameObject self anObject))  
(method != (anObject) ((self == anObject) not))
```

644b ▷

The problem with observation by “any program” is the same as the problem with object identity and with ML’s polymorphic identity: sometimes it says two things are different when they really should be considered the same. When you’re trying to tell if objects are equivalent, there are some observations you might want to rule out:

- You might want to rule out *mutation* as a way of observing differences. For example, perhaps right now lists `ns` and `ms` have the same elements, so you’d like to consider them equivalent, even if adding number 80 to list `ns` (but not `ms`) would enable you to tell them apart.
- When you’re comparing *abstractions*, they should be considered equivalent if no *client code* can tell them apart. For example, if finite maps are represented as association lists, and if no combination of `find` and `bind` operations can tell two maps apart, then they should be considered equivalent, even if they are represented differently (page 134).
- Finally, in a language like Smalltalk, we want to rule out reflection as a means of distinguishing objects. For example, sending the `class` message to try to distinguish two objects should be considered the same kind of observation as poking at the representation.

To decide equivalence of two objects,  $\mu$ Smalltalk rules out *all* of these observations:

*Two objects are considered equivalent if, without mutating either object, client code cannot tell them apart.*

In Smalltalk, this is the equivalence that is used by `check-expect`, and it is implemented by the `=` method. Each class has to think for itself about how `=` should be defined, but as a default, a conservative approximation is defined on all objects:

**644b.** *(methods of class Object 644a)+≡*

```
(method = (anObject) (self == anObject))  
(method != (anObject) ((self = anObject) not))
```

△644a 663b ▷

### Why no mutation?

If object-orientation is all about data abstraction, then of course objects should be considered differently only if *client* code can tell them apart. But why shouldn't the client code be allowed to mutate? If we're considering the definition of equivalence in isolation, this restriction makes little sense. But in the context of a whole language design, the restriction makes pragmatic sense: if you already have object identity, and if you permit client code to mutate objects, then client-code observation gives you nothing beyond what you already get with object identity. A second form of observational equivalence is useful only if it's different from what you already have.

This default is conservative in the sense that if it says two objects are equivalent, they really are. If two objects are equivalent but not identical, however, the default = will report, incorrectly, that they are different.

### Identity versus equivalence: an example

Here's an example that uses lists to illustrate the distinction between identity and equivalence. The two lists ns and ms are built differently:

645a. *(transcript 621a)* +≡  
-> (val ns (List new))  
List( )  
-> (ns addFirst: 3)  
-> (ns addFirst: 2)  
-> (ns addFirst: 1)  
-> ns  
List( 1 2 3 )  
-> (val ms (List new))  
List( )  
-> (ms addLast: 1)  
-> (ms addLast: 2)  
-> (ms addLast: 3)  
-> ms  
List( 1 2 3 )

◁641b 645b▷

Sure enough, they aren't the same object, but because they represent the same list, they are equivalent:

645b. *(transcript 621a)* +≡  
-> (ns == ms)  
<False>  
-> (ns = ms)  
<True>

◁645a 647a▷

## 10.4 THE INITIAL BASIS OF $\mu$ SMALLTALK

Much of Smalltalk's power comes from its impressive primitive and predefined objects and classes. To use Smalltalk effectively, you must learn to use the initial classes and to define new classes that inherit from them. By the standards of Smalltalk-80,  $\mu$ Smalltalk's hierarchy of predefined classes is small, but by the standards of this book, it is large—sufficient to help you learn what Smalltalk programming is like, but too large to digest all at once. To use the predefined classes, all you

Object instance protocol	
<code>== anObject</code>	Answer whether the argument is the same object as the receiver.
<code>!= anObject</code>	Answer whether the argument is not the same object as the receiver.
<code>= anObject</code>	Answer whether the argument should be considered equal to the the receiver, even if they are not identical.
<code>!= anObject</code>	Answer whether the argument should be considered different from the receiver.
<code>isNil</code>	Answer whether the receiver is nil.
<code>notNil</code>	Answer whether the receiver is not nil.
<code>print</code>	Print the receiver on standard output.
<code>println</code>	Print the receiver, then a newline, on standard output.
<code>error: aSymbol</code>	Issue a run-time error message which includes aSymbol.
<code>subclassResponsibility</code>	Report to the user that a method specified in the superclass of the receiver should have been implemented in the receiver's class.
<code>leftAsExercise</code>	Report to the user that a method should have been implemented as an exercise.
<code>class</code>	Answer the class of the receiver.
<code>isKindOf: aClass</code>	Answer whether the argument, aClass, is a superclass or class of the receiver. Use only at the read-eval-print loop.
<code>isMemberOf: aClass</code>	Answer whether the argument, aClass, is exactly the receiver's class. Use only at the read-eval-print loop.

Figure 10.9: Protocol for all objects

need to know is what abstractions they represent and what protocols they understand, as presented in this section (10.4). But to develop good  $\mu$ Smalltalk skills, you will want to study techniques used in the *implementations* of the predefined classes, as presented in Sections 10.5 to 10.8.

#### 10.4.1 Protocol for all objects

Every  $\mu$ Smalltalk object responds to the messages in Figure 10.9, which are defined on the primitive class `Object`. Messages `=` and `!=` test for equivalence, while `==` and `!=!` test for identity. Messages `isNil` and `notNil` test for “definedness.”

The `print` message is used by the  $\mu$ Smalltalk interpreter itself to print the values of objects; `println` follows with a newline.

The `error:` message is used to report errors. The `subclassResponsibility` message, which also reports an error when sent, is used to mark a class as abstract (page 632). A method that sends `subclassResponsibility` is sometimes called a “marker method.” Finally, some methods of some classes are meant for you to implement, as exercises; these methods send message `leftAsExercise` to `self`.

The three methods `class`, `isKindOf:`, and `isMemberOf:` are meant for introspection into objects. Message `class` gives the class of a receiver, and messages `isKindOf:` and `isMemberOf:` test properties of that class. For example, this session tells us that 3 is from some proper subclass of `Number`, and that the symbol '3 is not a number at all.

**647a.** *(transcript 621a)* +≡  
-> (3 isKindOf: Number)  
<True>  
-> (3 isMemberOf: Number)  
<False>  
-> ('3 isKindOf: Number)  
<False>

△645b 647b ▷

§10.4  
*The initial basis*

647

These methods can sometimes distinguish objects that should be considered equivalent. They are suitable for building system infrastructure or for exploring how Smalltalk works, interactively; except possibly for `class`, they shouldn't find their way into your code. If you're tempted use `isKindOf:` or `isMemberOf:` in definitions of other methods, don't. Decisions should never be made by using `isKindOf:` or `isMemberOf:`; they should be made by dispatching to a method that already knows what class it's defined on. To see the right way to do things, consult Section 10.5 on pages 663 and 664: Look at the way `isNil` and `notNil` are implemented in classes `Object` and `UndefinedObject`, or look at the implementations of methods on class `True`.

#### 10.4.2 Protocol for classes

Because a class is also an object, a class can answer messages. Every class object inherits from `Class`, and it responds to the protocol in Figure 10.10 (as well as to every message in the `Object` protocol). Parts of the protocol approximate capabilities that a full Smalltalk system provides in a graphical user interface: messages `printProtocol` and `printLocalProtocol` are meant for interactive exploration, and message `addSelector:withMethod` enables you to modify or extend built-in classes.

#### 10.4.3 Blocks and Booleans

A block is a Smalltalk object that corresponds to a Scheme closure. Quite commonly, a block expects no arguments, in which case it is conventionally written in curly brackets (Smalltalk-80 uses square brackets). A bracketed expression of the form `{exp}` evaluates to a block object; the expression inside the brackets is not evaluated until the block is sent the `value` message.

**647b.** *(transcript 621a)* +≡  
-> (val index 0)  
-> {{set index (index + 1)}}  
<Block>  
-> index  
0  
-> {{set index (index + 1)}} value)  
1  
-> index  
1

△647a 648 ▷

<code>new</code>	The receiver is a class; answer a new instance of that class. A class may override <code>new</code> , e.g., if it needs arguments to initialize a newly created instance.
<code>name</code>	The receiver is a class; answer its name.
<code>superclass</code>	The receiver is a class; answer its superclass, or if it has no superclass, answer <code>nil</code> .
<code>printProtocol</code>	The receiver is a class; print the names of all the messages it knows how to respond to, and the names of the messages its instances know how to respond to.
<code>printLocalProtocol</code>	The receiver is a class; print the names of the class and instance methods that are defined by this class, not those that are inherited from its superclass.
<code>methodNames</code>	The receiver is a class; answer an array containing the name of every method defined on the class.
<code>compiledMethodAt: aSymbol</code>	The receiver is a class; answer the compiled method with the given name defined on that class. (If no method with that name exists, causes a checked run-time error.)
<code>removeSelector: aSymbol</code>	The receiver is a class; if the receiver defines a method with the given name, remove it.
<code>addSelector:withMethod: aSymbol aCompiledMethod</code>	The receiver is a class; updates the receiver's internal state to associate the given compiled method with the given name. The method may replace an existing method or may be entirely new. The name must be consistent with the arity of the method.

Figure 10.10: Protocol for classes

Just like any object, a block can be assigned to a variable and used later.

648. *(transcript 621a) +≡* <647b 649>

```

-> (val incrementBlock {(set index (index + 1))})
<Block>
-> (val sumPlusIndexSquaredBlock {(sum + (index * index))})
<Block>
-> (val sum 0)
0
-> (set sum (sumPlusIndexSquaredBlock value))
1
-> (incrementBlock value)
2
-> (set sum (sumPlusIndexSquaredBlock value))
5

```

What is most surprising about such blocks is not their semantics but the way they are used: as continuations. Smalltalk uses blocks to implement conditional

Instance protocol for Booleans:

<code>ifTrue:ifFalse: trueBlock falseBlock</code>	If the receiver is true, evaluate <code>trueBlock</code> , otherwise evaluate <code>falseBlock</code> .
<code>ifTrue: trueBlock</code>	If the receiver is true, evaluate <code>trueBlock</code> , otherwise answer <code>nil</code> .
<code>ifFalse: falseBlock</code>	If the receiver is false, evaluate <code>falseBlock</code> , otherwise answer <code>nil</code> .
<code>and: alternativeBlock</code>	If the receiver is true, answer the value of the argument; otherwise, answer <code>false</code> (short-circuit conjunction).
<code>or: alternativeBlock</code>	If the receiver is false, answer the value of the argument; otherwise, answer <code>true</code> (short-circuit disjunction).
<code>&amp; aBoolean</code>	Answer the conjunction of the receiver and the argument.
<code>  aBoolean</code>	Answer the disjunction of the receiver and the argument.
<code>not</code>	Answer the complement of the receiver.
<code>eqv: aBoolean</code>	Answer <code>true</code> if the receiver is equivalent to the argument.
<code>xor: aBoolean</code>	Answer <code>true</code> if the receiver is different from the argument (exclusive or).

§10.4  
The initial basis  
649

Figure 10.11: Protocols for Booleans

expressions, much as the pure  $\lambda$ -calculus uses functions to implement conditional expressions. In other languages, including Scheme, an expression of the form  $(\text{if } e_1 e_2 e_3)$  is treated specially by the interpreter, which first evaluates  $e_1$ , then evaluates either  $e_2$  or  $e_3$ , as needed. In  $\mu$ Smalltalk, as in full Smalltalk-80, `if` is not built in; the *only* control structure is message passing.<sup>9</sup> In  $\mu$ Smalltalk, writing  $(e_1 \text{ ifTrue:ifFalse: } e_2 e_3)$ , asks the interpreter to evaluate *all three* expressions  $e_1$ ,  $e_2$ , and  $e_3$  before dispatching to method `ifTrue:ifFalse:` on the object that  $e_1$  evaluates to. This behavior is not what we mean by a conditional. To write a conditional in Smalltalk, we *delay* the evaluation of  $e_2$  and  $e_3$  by putting them in blocks: a conditional takes the form  $(\text{ifTrue:ifFalse: } e_1 \{e_2\} \{e_3\})$ .<sup>10</sup> This transcript illustrates the difference:

649. ⟨transcript 621a⟩+≡

```

-> ((sum < 0) ifTrue:ifFalse: {'negative'} {'nonnegative'})
nonnegative
-> ((sum < 0) ifTrue:ifFalse: 'negative' 'nonnegative')
Run-time error: Symbol does not understand message value
Method-stack traceback:
  Sent 'value' in predefined classes, line 28
  Sent 'ifTrue:ifFalse:' in standard input, line 152

```

◀648 650▶

index 647b

<sup>9</sup>In fact, implementations of Smalltalk-80 have traditionally “open coded” conditionals, preventing user-defined classes from usefully defining method `ifTrue:ifFalse:` (ANSI 1998). The Self language manages efficiency without such hacks (Chambers and Ungar 1989; Chambers 1992).

<sup>10</sup>This code is another example of *continuation-passing style* (Section 2.10). In Smalltalk, control flow is implemented by passing two blocks—the continuations—to a Boolean. The `true` object continues by executing the first block; `false` continues by executing the second.

## Instance protocol for blocks:

<code>value</code>	Evaluate the receiver and answer its value.
<code>value: anArgument</code>	Allocate a fresh location to hold the argument, bind that location to the receiver's formal parameter, evaluate the body of the receiver, and answer the result.
<code>value:value: arg1 arg2</code>	
<code>value:value:value: arg1 arg2 arg3</code>	
<code>value:value:value:value: arg1 arg2 arg3 arg4</code>	Like <code>value:</code> , but with two, three, or four arguments.
<code>whileTrue: bodyBlock</code>	Send <code>value</code> to the receiver, and if the response is true, send <code>value</code> to <code>bodyBlock</code> and repeat. When the receiver responds <code>false</code> , answer <code>nil</code> .
<code>whileFalse: bodyBlock</code>	Send <code>value</code> to the receiver, and if the response is false, send <code>value</code> to <code>bodyBlock</code> and repeat. When the receiver responds <code>true</code> , answer <code>nil</code> .

Figure 10.12: Protocols for blocks

On the first line, `(sum < 0)` produces the Boolean object `false`. Sending message `ifTrue:ifFalse:` to `false` causes `false` to send `value` to the block `{'nonnegative'}`, which answers the symbol `'nonnegative'`, which is the result of the entire expression. On the second line, `false` sends `value` to the symbol `'nonnegative'`, which results in an error message and a stack trace.

The continuation-passing messages of the Booleans are shown in the top half of Figure 10.11 on the previous page. In addition to classic conditionals, these methods also include short-circuit `and:` and `or::`. The bottom half of the figure shows the messages that implement simple Boolean operations.

Like conditionals, loops are implemented in continuation-passing style. But a loop is implemented not by sending a message to a Boolean, but by sending a message to a block, which holds the condition. The condition must be a block because it must be re-evaluated on each iteration of the loop.

```
650. ⟨transcript 621a⟩ +≡                                     ◁ 649 654a ▷
-> ({(sum < 10000)} whileTrue: { (set sum (5 * sum)) (sum println) })
25
125
625
3125
15625
nil
```

When the loop terminates, the `whileTrue:` method answers `nil`, which the interpreter prints.

Loops and conditionals use parameterless blocks, but a block may also take parameters. Such a block must be written using the `block` keyword, as in this block from the `drawPolygon:` method of class `TikzCanvas`:

```
(coord-list do: [block (pt) (pt print) ('-- print)])
```

A block that expects one argument understands the `value:` message. Smalltalk also provides `value:value:`, and so on up to four arguments, as shown in Figure 10.12.

#### 10.4.4 Unicode characters

If you want to print something that you can't write as a literal symbol, like a newline or a left parenthesis, you have to treat it as a Unicode character, which is typically specified by an integer "code point" of 16 or more bits. But nobody wants to read code points, so  $\mu$ Smalltalk defines the following objects of class `Char`, which you can print by sending the `print` message:

newline	semicolon	left-round	left-square	left-curly
space	quotemark	right-round	right-square	right-curly

§10.4

The initial basis

You can define more characters using `new:`, as in `(Char new: 955)`.

651

#### 10.4.5 Collections

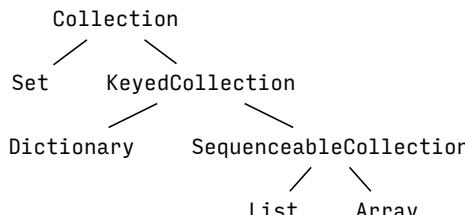
All programmers need ways of grouping individual data elements into larger collections. Most languages provide one or two ways of collecting data; for example, Scheme has its lists, and Molecule has both lists and arrays. Other collections have to be defined by the programmer. But  $\mu$ Smalltalk ships with four useful collections: sets, lists, arrays, and dictionaries. Each of these collections inherits, sometimes indirectly, from class `Collection`, which is a subclass of `Object`.

Smalltalk-80 also has a collection hierarchy, which includes similar classes but is structured somewhat differently (Section 10.11.3 on page 713). My version is inspired by Budd's (1987) Collection hierarchy.

What is a "collection"? It's an abstraction, like the abstractions written in Molecule in Chapter 9. But Smalltalk provides data abstractions using objects, not abstract data types, and object-oriented abstractions work differently. In Smalltalk, a collection abstraction has nothing to do with any type, and whether it has anything to do with the `Collection` class is a matter of convenience. A collection abstraction is an object that responds to the *collection protocol*. That protocol is specified in Figure 10.13 on page 653, and for convenience, we associate it with class `Collection`. The most fundamental parts of the protocol are as follows:

- A collection must respond to the `do:` message. When a message of the form `(collection do: [block (x) body])` is sent, the receiver must respond by evaluating `body` once for each item `x` that it contains. A `do:` method acts like the  $\mu$ Scheme `app` function, but it works on any collection, not just on lists.
- Collections are mutable; a typical collection object should respond to message `add:` by adding the argument to itself, and it should respond to `remove:` by removing the argument from itself. Some collections, like `Arrays`, have a fixed number of elements; when receiving `add:` or `remove:` messages, such collections report errors.

Class `Collection` specifies the protocol, but like class `Shape` from Section 10.1, it is an abstract class: it defines many useful methods, but it is intended to be inherited from, not instantiated. The implementation of the protocol is spread out over three abstract classes and four concrete classes:



sum

648

Here is an informal description of what each class does or what it is:

**Collection:** A collection contains objects. Class `Collection` is an abstract class that implements all but four of the methods in the collection protocol. The remaining four methods are implemented by subclasses.

**Set:** A value of class `Set` represents a set. A set contains objects in no particular order, with no duplicates (as decided by `=`). Class `Set` implements the four crucial methods missing from the `Collection` class, and it can be used to instantiate objects—it is a concrete subclass.

**KeyedCollection:** A keyed collection contains objects that are associated with keys, or equivalently, it contains key-value pairs. A keyed collection shares a lot of protocol with an ordinary collection, and its protocol includes additional messages that can add, retrieve, and mutate objects by using a key. `KeyedCollection` is an abstract class.

**Dictionary:** A value of class `Dictionary` represents a finite map. Class `Dictionary` is a concrete subclass of `KeyedCollection`; it is used to instantiate objects. It assumes as little as possible about keys: in a dictionary, keys need only to be comparable for equality.

**SequenceableCollection:** A sequenceable collection is a sequence, which the designers of Smalltalk view as a keyed collection whose keys are consecutive integers. Class `SequenceableCollection` is another abstract class that adds yet more protocol to `KeyedCollection`.

**List:** A list represents a sequence that can grow or shrink. Objects can easily be added to or removed from the beginning or the end of a list. Unlike a list in Scheme or ML, a Smalltalk list is *mutable*. Class `List` is a concrete subclass that can be used to instantiate lists.

**Array:** An array represents a sequence whose elements can be accessed in constant time—but unlike a list, an array cannot grow or shrink. Although class `Array` inherits from `SequenceableCollection` and therefore indirectly from `Collection`, it does not implement the full collection protocol: it implements only those methods that can be implemented without growing or shrinking. Class `Array` is a concrete subclass that can be used to instantiate arrays.

The collection protocol is described in detail in Figure 10.13 on the facing page.

- Each collection *class* responds to a `with:` message, which creates a collection containing a single element, and to a `withAll:` message, which creates a collection that contains all the elements from another collection.
- The instance methods in the first group are *mutators*; they define ways of adding and removing elements. In  $\mu$ Smalltalk, a mutator typically answers the receiver, which makes it easy to send several mutation messages to the same object in sequence.
- The instance methods in the second group are *observers*; they define ways of finding out about the elements in a collection. The `includes:` and `occurrencesOf:` observers ask about elements directly; the `detect:` and `detect:ifNone:` observers ask for *any* element satisfying a given predicate, which is typically represented as a block. The `=` observer compares two collections to see if they contain equal things (according to `=`).

Class protocol for Collection:

<code>with: anObject</code>	Create and answer a singleton collection holding the given object.
<code>withAll: aCollection</code>	Create and answer a collection that holds all the elements of the argument.

Public instance protocol for Collection:

<code>add: newObject</code>	Include the argument, <code>newObject</code> , as one of the elements of the receiver. Answer the receiver.
<code>addAll: aCollection</code>	Add every element of the argument to the receiver; answer the receiver.
<code>remove: oldObject</code>	Remove the argument, <code>oldObject</code> , from the receiver. If no element is equal to <code>oldObject</code> , report an error; otherwise, answer the receiver.
<code>remove:ifAbsent: oldObject exnBlock</code>	Remove the argument, <code>oldObject</code> , from the receiver. If no element is equal to <code>oldObject</code> , answer the result of evaluating <code>exnBlock</code> ; otherwise, answer the receiver.
<code>removeAll: aCollection</code>	Remove every element of the argument from the receiver; answer the receiver or report an error.
<code>isEmpty</code>	Answer whether the receiver has any elements.
<code>size</code>	Answer how many elements the receiver has.
<code>includes: anObject</code>	Answer whether the receiver has <code>anObject</code> .
<code>occurrencesOf: anObject</code>	Answer how many of the receiver's elements are equal to <code>anObject</code> .
<code>detect: aBlock</code>	Answer the first element <code>x</code> in the receiver for which <code>(aBlock value: x)</code> is true, or report an error if none.
<code>detect:ifNone: aBlock exnBlock</code>	Answer the first element <code>x</code> in the receiver for which <code>(aBlock value: x)</code> is true, or answer <code>(exnBlock value)</code> if none.
<code>= aCollection</code>	Answer whether the contents of the receiver are similar to the contents of the argument.
<code>do: aBlock</code>	For each element <code>x</code> of the collection, evaluate <code>(aBlock value: x)</code> .
<code>inject:into: aValue binaryBlock</code>	Evaluate <code>binaryBlock</code> once for each element in the receiver. The first argument of the block is an element from the receiver; the second argument is the result of the previous evaluation of the block, starting with <code>aValue</code> . Answer the final value of the block.
<code>select: aBlock</code>	Answer a new collection like the receiver, containing every element <code>x</code> of the receiver for which <code>(aBlock value: x)</code> is true.
<code>reject: aBlock</code>	Answer a new collection like the receiver, containing every element <code>x</code> of the receiver for which <code>(aBlock value: x)</code> is false.
<code>collect: aBlock</code>	Answer a new collection like the receiver, containing <code>(aBlock value: x)</code> for every element <code>x</code> of the receiver.

- The instance methods `do:` and `inject:into:` are *iterators*. An iterator is a special kind of observer that repeats a computation once for every element in a collection. The `do:` method performs a computation only for side effect, while `inject:into:` accumulates and answers a value. These two methods correspond to the  $\mu$ Scheme functions `app` and `foldl`, but they are defined on *all* collections, not only on lists.
- The instance methods in the final group are *producers*; given a collection, such a method makes a new collection, without mutating the original collection. Methods `select:` and `collect:` correspond to the  $\mu$ Scheme functions `filter` and `map`, which are also defined in ML.

`Collection` is an abstract class, so a client should not send a `new`, `with:`, or `withAll:` message directly to `Collection`—only to one of its concrete subclasses.

The simplest subclass of `Collection` is `Set`; to get a feel for sets and for the `Collection` protocol, look at this transcript:

**654a.** *(transcript 621a)* +≡ △650 654b▷

```

-> (val s (Set new))
Set( )
-> (s size)
0
-> (s add: 2)
Set( 2 )
-> (s add: 'abc')
Set( 2 abc )
-> (s includes: 2)
<True>
-> (s add: 2)
Set( 2 abc )

```

Remember that when a message is sent to a receiver, the method has access not only to the values of the arguments but also to the receiver. For example the definition of the `add:` method has only one argument, that being the item to be added, but it also has access to the collection to which the item is added.

Our next examples mix two species of collections: sets and arrays. To start, I build a set `s` from an array, and then I add to `s` all the elements of another array.

**654b.** *(transcript 621a)* +≡ △654a 654c▷

```

-> (set s (Set withAll: '(1 2 3 1 2 3)))
Set( 1 2 3 )
-> (s addAll: '(1 2 3 a b c d e f))
Set( 1 2 3 a b c d e f )
-> (s includes: 'b)
<True>

```

This kind of mixed computation, where I pass an array as an argument to a set operation, is not possible using abstract data types. But using objects, it's easy: the `addAll:` method treats the argument object abstractly. As long as the argument responds to the `Collection` protocol, `addAll:` doesn't care what class it's an instance of. The `addAll:` method interacts with the argument only by sending it the `do:` message.

Other messages that work with any collection include `removeAll:` and `reject::`

**654c.** *(transcript 621a)* +≡ △654b 657▷

```

-> (s removeAll: '(e f))
Set( 1 2 3 a b c d )
-> (val s2 (s reject: [block (x) (x isKindOf: Number)]))
Set( a b c d )

```

Object-orientation also provides an effortless form of polymorphism. In the example, set `s` is initialized from the array `'(1 2 3 1 2 3)`, which contains duplicate elements. The `withAll:` class method, which is defined on all collections, eliminates the duplicates. To identify duplicates, the methods of class `Set` send the `=` message to individual elements. Because the `=` message is dispatched dynamically, it automatically has the right semantics for comparing elements, no matter what classes the elements are instances of. In Scheme, by contrast, getting equality right requires either passing an equality function as a parameter, storing it in a data structure, or capturing it in a closure (Section 2.9 on page 133). But in Smalltalk, as in every object-oriented language, an object acts like a closure—it is data bundled with code—and the bundle includes the right version of `=`, with no additional coding required.

### *Keyed collections and class Dictionary*

`KeyedCollection` is another abstract class; the abstraction is a set of key-value pairs in which no key occurs more than once. This abstraction is one of the most frequently used abstractions in computing, and it has many names: depending on context, it may be called an “associative array,” a “dictionary,” a “finite map,” or a “table.” Smalltalk uses the word “keyed collection” to encompass not only general-purpose key-value data structures but also special-purpose structures, including lists and arrays.

An object of class `KeyedCollection` responds to the `Collection` protocol as if it were a simple collection of values, with no keys. But `KeyedCollection` adds additional messages to the protocol (Figure 10.14 on the next page), and using these new messages, you can present a key and use it to look up, change, or remove the corresponding value.

- The `at:put:` message is a new mutator; `(kc at:put: key value)` modifies collection `kc` by associating `value` with `key`.
- The new observers `at:` and `keyAtValue:` answer the value associated with a key or the key associated with a value. The observer `at:ifAbsent:` can be used to learn whether a given key is in the collection. To learn whether a given `value` is in a collection, we use the observer `includes:` from the `Collection` protocol. Each of these observers works with a key that is *equivalent* to the key originally used with `at:put:`—it is not necessary to present an identical object.
- The observer `associationAt:` and the iterator `associationsDo:` provide access to the key-value pairs directly, in the form of `Association` objects.
- Mutators `removeKey:` and `removeKey:ifAbsent:` can remove a key-value pair by presenting an equivalent key.

Among these new messages, the most frequently used are `at:` and `at:put:`. Among the old messages that `KeyedCollection` inherits from `Collection`, the most frequently used may be `size`, `isEmpty`, and `includes:`.

`KeyedCollection` is an abstract class; a client should create instances of subclasses only. The simplest subclass is called `Dictionary`; it represents a collection as a list of associations, and it assumes only that its keys respond correctly to the `=` message. I encourage you to implement a more efficient version that uses a hash table or a search tree (Exercises 28 and 29).

## New instance protocol for KeyedCollection:

<code>at:put: key value</code>	Modify the receiver by associating key with value. May add a new value or replace an existing value. Answer the receiver.
<code>removeKey: key</code>	Modify the receiver by removing key and its associated value. If key is not in the receiver, report an error; otherwise, answer the value associated with key.
<code>removeKey:ifAbsent: key aBlock</code>	Modify the receiver by removing key and its associated value. If key is not in the receiver, answer the result of sending value to aBlock; otherwise, answer the value associated with key.
<code>at: key</code>	Answer the value associated with key, or report an error if there is no such value.
<code>at:ifAbsent: key aBlock</code>	Answer the value associated with key, or the result of evaluating aBlock if there is no such value.
<code>includesKey: key</code>	Answer true if there is some value associated with key, or false otherwise.
<code>keyAtValue: value</code>	Answer the key associated with value, or if there is no such value, report an error.
<code>keyAtValue:ifAbsent: value aBlock</code>	Answer the key associated with value, or if there is no such value, answer the result of evaluating aBlock.
<code>associationAt: key</code>	Answer the Association in the collection with key key, or if there is no such Association, report an error.
<code>associationAt:ifAbsent: key exnBlock</code>	Answer the Association in the collection with key key, or if there is no such Association, answer the result of evaluating exnBlock.
<code>associationsDo: aBlock</code>	Iterate over all Associations in the collection, evaluating aBlock with each one.

## Class protocol for Association:

<code>withKey:value: key value</code>	Create a new association with the given key and value.
---------------------------------------	--

## Instance protocol for Association:

<code>key</code>	Answer the receiver's key.
<code>value</code>	Answer the receiver's value.
<code>setKey: key</code>	Set the receiver's key to key.
<code>setValue: value</code>	Set the receiver's value to value.

Figure 10.14: Protocols for keyed collections

New instance protocol for SequenceableCollection:

first	Answer the first element of the receiver.
firstKey	Answer the integer key that is associated with the first value of the receiver.
last	Answer the last element of the receiver.
lastKey	Answer the integer key that is associated with the last value of the receiver.

§10.4

The initial basis

657

Figure 10.15: New protocol for sequenceable collections

### Sequenceable collections

The abstract class SequenceableCollection represents a keyed collection whose keys are consecutive integers—in other words, a sequence. This class defines additional protocol, which is shown in Figure 10.15. Methods `firstKey` and `lastKey` answer the first and last keys, and `first` and `last` answer the first and last values.

*Lists* A List is a sequence that can change size: we can add or remove an element at either end. The add and remove operations appear in the protocol shown in Figure 10.16. Unlike a Scheme list, a Smalltalk list is *mutable*: adding or removing an element mutates the original list, rather than creating a new one. The `add:` message is a synonym for `addLast:`.

Here is how List is used:

657. ⟨transcript 621a⟩+≡  
    -> (val xs (List new))  
    List( )  
    -> (xs addLast: 'a)  
    List( a )  
    -> (xs add: 'b)  
    List( a b )  
    -> (xs addFirst: 'z)  
    List( z a b )  
    -> (xs first)  
    z  
    -> (xs addFirst: 'y)  
    List( y z a b )  
    -> (xs at: 2)  
    a  
    -> (xs removeFirst)  
    y  
    -> xs  
    List( z a b )

*Arrays* An array is a sequence that cannot change size, but that implements `at:` and `at:put:` in constant time. Arrays are found in many programming languages; in  $\mu$ Smalltalk, every array is one-dimensional, and the first element's key is 0. (In Smalltalk-80, the first element's key is 1.)

An instance of Array responds to the messages of the SequenceableCollection protocol. (Because an array cannot change size, it responds to `add:` and `remove:` with errors.) The Array class responds to the messages of the SequenceableCollection class protocol, and also to message `new::`

new: anInteger Create and answer an array of size `anInteger` in which each element is nil.

<code>addLast: anObject</code>	Add anObject to the end of the receiver and answer the receiver.
<code>addFirst: anObject</code>	Add anObject to the beginning of the receiver and answer the receiver.
<code>removeFirst</code>	Remove the first object from the receiver and answer that object. Causes an error if the receiver is empty.
<code>removeLast</code>	Remove the last object from the receiver and answer that object. Causes an error if the receiver is empty.

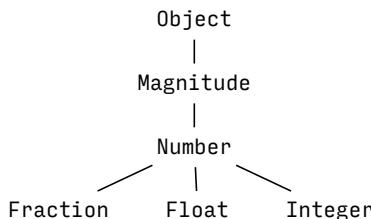
Figure 10.16: New protocol for lists

*Inheriting from Collection*

To define a new subclass of `Collection` might seem like an enormous job; after all, Figure 10.13 shows a great many messages, which any `Collection` object must understand. But the `Collection` class is carefully structured so that a subclass need implement only *four* of the methods in Figure 10.13: `do: :to`, to iterate over elements, `add: :to`, to add an element, `remove:ifAbsent: :to`, to remove an element, and `=`, to compare contents. The details are shown in Section 10.6.1.

10.4.6 *Magnitudes and numbers*

Collections make up a large part of  $\mu$ Smalltalk's initial basis. Much of the rest of the basis deals with numbers. The numeric classes illustrate the same classic use of inheritance as the collection classes: each numeric representation defines a minimal set of representation-specific methods like `+`, `*`, and negated, and they all share common, inherited implementations of generic methods, like `-`, `isNegative`, and `abs`. The main part of the numeric sub-hierarchy is



On page 660, this hierarchy is extended with large integers.

Class `Magnitude` supports comparisons, as well as `min: :max:` operations. The `Magnitude` protocol can be implemented by any totally ordered abstraction, like a number, a date, or a time. Magnitudes can be used in the implementations of search trees, sorted collections, and so on. `Number` supports not only comparisons but also arithmetic, plus other operations that can be performed only on numbers: you can ask a `Number` about its sign; you can compute a power, square, or square root; and you can *coerce* (convert) a `Number` to be an integer, a fraction, or a floating-point number. Protocols for both `Magnitude` and `Number` are shown in Figure 10.17 on the facing page.

In Smalltalk-80, predefined magnitudes include dates, times, and characters; in  $\mu$ Smalltalk, the predefined magnitudes include only natural numbers and the `Number` classes. Other magnitudes can be defined (Exercise 33 on page 730).  $\mu$ Smalltalk's predefined numbers are more interesting: objects of class `Integer`

Instance protocol for Magnitude:

= aMagnitude	Answer whether the receiver equals the argument.
< aMagnitude	Answer whether the receiver is less than the argument.
> aMagnitude	Answer whether the receiver is greater than the argument.
<= aMagnitude	Answer whether the receiver is no greater than the argument.
>= aMagnitude	Answer whether the receiver is no less than the argument.
min: aMagnitude	Answer the lesser of the receiver and aMagnitude.
max: aMagnitude	Answer the greater of the receiver and aMagnitude.

§10.4  
The initial basis  
659

Instance protocol for Number:

negated	Answer the negation of the receiver.
reciprocal	Answer the reciprocal of the receiver.
abs	Answer the absolute value of the receiver.
+ aNumber	Answer the sum of the receiver and the argument.
- aNumber	Answer the difference of the receiver and the argument.
* aNumber	Answer the product of the receiver and the argument.
/ aNumber	Answer the quotient of the receiver and the argument. The quotient is <i>not</i> rounded to the nearest integer, so the quotient of two integers may be a different kind of number.
isNegative	Answer whether the receiver is negative.
isNonnegative	Answer whether the receiver is nonnegative.
isStrictlyPositive	Answer whether the receiver is positive ( $> 0$ ). <sup>11</sup>
raisedToInteger: anInteger	Answer the receiver, raised to the integer power anInteger.
squared	Answer the receiver squared.
sqrtWithin: eps	Answer a number that is within eps of the square root of the receiver.
sqrt	Answer the square root of the receiver, to within some eps defined by the implementation.
coerce: aNumber	Answer a Number that is of the same <i>kind</i> as the receiver, but represents the same <i>value</i> as the argument.
asInteger	Answer the integer nearest to the value of the receiver.
asFraction	Answer the fraction nearest to the value of the receiver.
asFloat	Answer the floating-point number nearest to the value of the receiver.

Figure 10.17: Protocols for magnitudes and numbers

<sup>11</sup>The word “strictly” is from Smalltalk-80, which inexplicably uses “positive” to mean  $\geq 0$ .

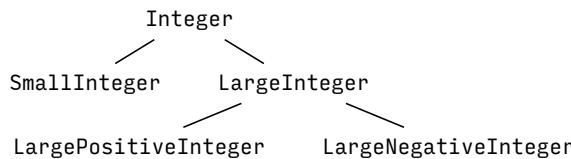
<code>div: anInteger</code>	Answer the quotient of the receiver and the argument, with division rounding towards $-\infty$ .
<code>mod: anInteger</code>	Answer the modulus of the receiver and the argument, with division rounding towards $-\infty$ .
<code>gcd: anInteger</code>	Answer the greatest common denominator of the receiver and the argument.
<code>lcm: anInteger</code>	Answer the least common multiple of the receiver and the argument.
<code>timesRepeat: aBlock</code>	If the receiver is the integer $n$ , send value to <code>aBlock</code> $n$ times.

Figure 10.18: New protocol for integers

represent integers, and instances of `Fraction` and `Float` represent rational numbers. A `Fraction` represents a rational number as the ratio of numerator  $n$  to denominator  $d$ . A `Float` represents a rational number of the form  $m \cdot 10^e$ , where  $m$  is an integer mantissa and  $e$  is an integer exponent.

These predefined numeric classes support all of the arithmetic and comparison operations in the `Magnitude` and `Number` protocols, but they support only *homogeneous* operations: for example, you compare integers only with integers, fractions only with fractions, and floating-point numbers only with floating-point numbers. The same goes for addition, subtraction, multiplication, and so on. You can't ask if the fraction  $3/4$  is less than the integer  $1$ —but if you could, it would be not only convenient but also more in the spirit of object-oriented programming. I encourage you to implement this *mixed* arithmetic yourself (Exercise 36 on page 731).

**Integers** Integers provide not just the standard arithmetic operations expected of all numbers, but also `div:`, `mod:`, `gcd:`, and `lcm:`, which are defined only on integers. It is also possible to evaluate a block an integral number of times.  $\mu$ Smalltalk is designed to support two forms of integer: small integers, which I implement, and large integers, I invite you to implement (Exercise 38). The integer sub-hierarchy looks like this:



These classes are discussed in Section 10.7.3 on page 675, and they all answer the protocol shown in Figure 10.18.

**Fractions** A `Fraction` represents a rational number as a fraction. For example, in the session below, sending `(two sqrtWithin: epsilon)` answers  $\frac{17}{12}$ , which approximates  $\sqrt{2}$  to within `epsilon` ( $\frac{1}{10}$ ). The approximation,  $1.41\bar{6}$ , is quite close to the actual value of  $1.4142+$ .

660. <transcript 621a>+≡  
 -> (val two (Fraction num:den: 2 1))  
 -> (val epsilon (Fraction num:den: 1 10))  
 1/10  
 -> (val root2 (two sqrtWithin: epsilon))  
 17/12

&lt;657 661a&gt;

One can get better precision by decreasing epsilon. And it is more convenient and idiomatic to get fractions by dividing integers and by using asFraction than by sending messages to class Fraction.

661a. ⟨transcript 621a⟩+≡  
-> (val epsilon (1 / 100))  
1/100  
-> (val root2 ((2 asFraction) sqrtWithin: epsilon))  
577/408

△660 661b ▷

The implementation computes the square root of  $n$  by using the Newton-Raphson technique for finding a zero of the function  $x^2 - n$ . Newton-Raphson produces accurate results quickly; the approximation  $\frac{577}{408}$  is accurate to five decimal places. To make the Fraction class useful for more ambitious computations, you need to finish the implementation of large-integer arithmetic (Exercises 37 and 38) so that class Fraction can compute large numerators and denominators without risking overflow.

§10.4  
The initial basis  
661

*Floating-point numbers* A number in floating-point form is represented by a mantissa  $m$  and exponent  $e$ , which represent the number  $m \cdot 10^e$ . Both  $m$  and  $e$  can be negative. A Float satisfies the representation invariant that  $|m| \leq 32767$ ; this restriction, which provides about 15 bits of precision, ensures we can multiply two mantissas without having the results overflow.

Float can be used as follows:

661b. ⟨transcript 621a⟩+≡  
-> (val epsilon ((1 / 100) asFloat))  
1x10^-2  
-> ((2 asFloat) sqrtWithin: epsilon)  
14142x10^-4

△661a

The result is good to five decimal places.

#### 10.4.7 Natural numbers

A natural number is more than a magnitude but less than a full Number:  $\mu$ Smalltalk's class Natural supports limited arithmetic, plus a couple of new observers. A natural number is a form of magnitude, and class Natural is a subclass of Magnitude. In addition to the protocol for Magnitude, including comparisons, class Natural and its instances respond to the protocol in Figure 10.19 on the following page.

Natural numbers may be added and multiplied without fear of overflow—the point of the abstraction is that the size of a natural number is limited only by the amount of memory available. Natural numbers may also be subtracted, provided that the argument is no greater than the receiver. And a natural number may be divided by a small, positive integer; the s in sdiv: and smod: stands for “short.” (As noted on page S21, long division is beyond the scope of this book.)

The protocol for instances of Natural includes an observer decimal:, which converts the receiver to a list of decimal digits. (For efficiency, the receiver is expected to use an internal representation with a base much larger than 10.)

Finally, the Natural protocol includes three methods that are intended to promote efficiency:

- Method sdivmod:with: computes both quotient and remainder in a single operation. It is necessary in order to implement division in linear time. If a division operation were to send both sdiv: and smod:, division could take time exponential in the number of digits, which is not acceptable.
- Method subtract:withDifference:ifNegative: combines comparison and subtraction into a single operation. Implemented independently, each of

Class protocol for Natural:

<code>fromSmall: anInteger</code>	Answer a natural-number object whose value is equal to the value of the argument, which must be a <i>nonnegative</i> integer.
-----------------------------------	---

Protocol for natural-number objects:

<code>+ aNatural</code>	Answer the sum of the receiver and the argument.
<code>* aNatural</code>	Answer the product of the receiver and the argument.
<code>- aNatural</code>	Answer the difference of the receiver and the argument, or if this difference is not a natural number, fail with a run-time error.
<code>sdiv: aSmallInteger</code>	Answer the largest natural number whose value is at most the quotient of the receiver and the argument.
<code>smod: aSmallInteger</code>	Answer a small integer which is the remainder when the receiver is divided by the argument.
<code>decimal</code>	Answer a List containing the <i>decimal</i> digits of the receiver, most significant digit first.
<code>sdivmod:with: aSmallInteger aBlock</code>	An object <i>n</i> receiving ( <i>n sdivmod:with: d b</i> ) answers the result of sending ( <i>b value:value: Q r</i> ), where <i>Q</i> is <i>n div d</i> and <i>r</i> is <i>n mod d</i> .
<code>subtract:withDifference:ifNegative: aNatural diffBlock negativeBlock</code>	Subtract aNatural from the receiver to obtain difference <i>d</i> . If the difference is a natural number, answer ( <i>diffBlock value: d</i> ). If the difference is negative, answer ( <i>negativeBlock value</i> ).
<code>isZero</code>	Answer true if the receiver is zero, and false otherwise.

Figure 10.19: Protocols for natural numbers

these operations could take linear time, and comparison comes “for free” with a subtraction.

- Method `isZero` can sometimes be implemented more efficiently than `=`.

The implementation of class `Natural` is left to you, as Exercise 37. Ideas are presented on page 678.

## 10.5 TECHNIQUE I: METHOD DISPATCH REPLACES CONDITIONALS

To get started programming in  $\mu$ Smalltalk, you can focus on the examples from Section 10.1 and on the protocols and informal descriptions of the predefined classes in Section 10.4. But to internalize object-oriented ways of thinking and programming, you need to study how things are done. To show you, the next four sections use excerpts from the predefined classes to present four sets of techniques:

- This section (10.5) shows how in Smalltalk, your code should make decisions by dispatching messages to the right methods, not by evaluating conditionals. The technique is illustrated with example methods defined on classes `Object`, `UndefinedObject`, and `Boolean`.

- Section 10.6 shows how you can reuse code by building abstract classes that provide many useful methods on top of just a few subclass responsibilities. The technique is illustrated with example methods defined on collection classes.
- Section 10.7 shows how you can define methods that want to look at representations of more than one object, even though a method defined on an object has access only to its own instance variables. The technique is illustrated with example methods defined on numeric classes.
- Section 10.8 shows how you can integrate ideas from Sections 10.5 and 10.6 with program-design ideas from Chapter 9, including an abstraction function and representation invariant. The techniques are illustrated with a complete implementation of class `List`.

We start with decisions: The most fundamental difference between an object-oriented program and a functional or procedural program is how decisions are made. An object-oriented program doesn't ask an object, "how were you formed?" Instead, it sends a message that asks a different question or makes a request. The form of the receiver becomes known not by evaluating an expression but by dispatching to a method. For example, we can ask any object, "are you nil?" by sending it message `isNil` or `notNil`. These methods are defined in ordinary μSmalltalk, on classes `Object` and `UndefinedObject`. Before we study the code, here's how *not* to do it—two ways to test for `nil` by evaluating an expression:

```
(method isNil () (self == nil)) ; embarrassing
(method isNil () (self isMemberOf: UndefinedObject)) ; more embarrassing
```

A real Smalltalk programmer sneers at this code. The proper way to implement case analysis is to use method dispatch. For `isNil` there are only two possible cases: the method should answer `true` or `false`. We arrange that on the `nil` object, the `isNil` method answers `true`, and that on all other objects, it answers `false`. We need only two method definitions: one on class `UndefinedObject`, which is used only to answer messages sent to `nil`, and one on class `Object`, which all other classes inherit. We implement `notNil` the same way. The definitions on class `UndefinedObject` are

**663a.** *(methods of class UndefinedObject 663a) ≡*

```
(method isNil () true)
(method notNil () false)
```

On `Object`, they are

**663b.** *(methods of class Object 644a) +≡*

△ 644b

```
(method isNil () false)
(method notNil () true)
```

The code contains no conditionals; decisions are made entirely by method dispatch. For example, when `isNil` is sent to `nil`, `nil` is an instance of class `UndefinedObject`, so the message dispatches to `UndefinedObject`'s `isNil` method, which answers `true`. But when `isNil` is sent to any other object, method search starts in the class of that object and eventually reaches class `Object`, where it dispatches to `Object`'s `isNil` method, which answers `false`. A `notNil` message works the same way.

If you take object-oriented programming seriously, you will *never use an explicit conditional if you can achieve the same effect using method dispatch*. Method dispatch is preferred because it is *extensible*: to add new cases, you just add new classes—think, for example, about adding new kinds of shapes to the pictures in Section 10.1.

To add new cases to a conditional, you would have to edit the code—at every location where a similar conditional decision is made. Method dispatch makes it easier to evolve the code. And in many implementations, it is also more efficient.

Some conditionals can't be avoided. For example, if we need to know if a number  $n$  is at least 10, we must use a conditional like `ifTrue:` on the Boolean object answered by `(n >= 10)`. But `ifTrue:` is itself implemented using method dispatch. Smalltalk doesn't have conditional and looping control structures like  $\mu$ Scheme's `if` and `while`; it has only message passing and return. Conditionals are implemented by sending continuations to Boolean objects, and loops are implemented by sending continuations to block objects.<sup>12</sup>

The Boolean protocol is defined in class `Boolean`, but class `Boolean` does nothing for itself—it defines only subclass responsibilities (Appendix U). These responsibilities are implemented in the two subclasses `True` and `False`, which have a single instance each: an object `true` of class `True` and an object `false` of class `False`. Conditional control is implemented by defining each method differently in each subclass. Here, the unique object of class `True` knows it represents truth, and it implements its operations accordingly:

**664.** *<predefined  $\mu$ Smalltalk classes and values 664>* ≡

```
(class True
  [subclass-of Boolean]
  (method ifTrue:ifFalse: (trueBlock falseBlock) (trueBlock value))
  (method ifFalse:ifTrue: (falseBlock trueBlock) (trueBlock value))
  (method ifTrue: (trueBlock) (trueBlock value))
  (method ifFalse: (falseBlock) nil)

  (method not () false)
  (method eqv: (aBoolean) aBoolean)
  (method xor: (aBoolean) (aBoolean not))
  (method & (aBoolean) aBoolean)
  (method | (aBoolean) self)

  (method and: (alternativeBlock) (alternativeBlock value))
  (method or: (alternativeBlock) self)
)
```

There is no explicit conditional logic here: by the time a message dispatches to one of these methods, all decisions have been made. The implementation of class `False`, which is similar, is left as Exercise 16.

## 10.6 TECHNIQUE II: ABSTRACT CLASSES

Object-oriented code can achieve a form of polymorphism not readily available in languages like Scheme and ML: By relying on subclass responsibilities, methods like `min:` and `max:` (for magnitudes) or `detect:` and `select:` (for collections) can be implemented once and reused for many different subclasses. This section starts with two examples—magnitudes and collections—and then goes deeper into the technique by showing how different subclasses can refine and extend the Collection protocol. This is a great technique to emulate in your own designs.

### 10.6.1 Implementing wide protocols: Magnitudes and collections

The `Magnitude` protocol suits any abstraction that is totally ordered, even those that do not support arithmetic: numbers, dates, times, and so on. A subclass has only

---

<sup>12</sup>This implementation of conditionals is the same one used in the *Church encoding* of Booleans in the *lambda calculus*—a tool used in the theoretical study of programming languages.

two responsibilities: comparisons = and <.<sup>13</sup> The other comparisons, as well as min: and max:, are implemented using <.

**665a. (numeric classes 665a)≡** (S560c) 672c▷

```
(class Magnitude
  [subclass-of Object] ; abstract class
  (method = (x) (self subclassResponsibility)) ; may not inherit
  (method < (x) (self subclassResponsibility))
  (method > (y) (y < self))
  (method <= (x) ((self > x) not))
  (method >= (x) ((self < x) not))
  (method min: (aMagnitude)
    ((self < aMagnitude) ifTrue:ifFalse: {self} {aMagnitude}))
  (method max: (aMagnitude)
    ((self > aMagnitude) ifTrue:ifFalse: {self} {aMagnitude}))
)
```

§10.6  
Technique II:  
Abstract classes

665

Every subclass can use the methods defined here.

Another a big protocol that relies on just a few subclass responsibilities is the `Collection` protocol. This protocol is a joy to work with; it includes not only object-oriented analogs to functions like `exists?`, `map`, `filter`, and `foldl`, but also many other methods, which do things like add, remove, find, and count elements (Figure 10.13 on page 653). And unlike their Scheme analogs, these operations support not only lists but also several other forms of collection. All this functionality is provided using just four subclass responsibilities: a collection class must define methods `do:`, `add:`, `remove:ifAbsent:`, and `=`.

**665b. (collection classes 665b)≡** (S560c) 668a▷

```
(class Collection
  [subclass-of Object] ; abstract
  (method do: (aBlock) (self subclassResponsibility))
  (method add: (newObject) (self subclassResponsibility))
  (method remove:ifAbsent: (oldObject exnBlock)
    (self subclassResponsibility))
  (method = (aCollection) (self subclassResponsibility))
  (other methods of class Collection 665c)
)
```

If you want to study an implementation of these subclass responsibilities, look at class `Set` (Section U.1.8). To see how they are used, look below.

To create a singleton collection, we send `add:` to a new instance; to create a collection holding all of an argument's elements, we send `addAll:` to a new instance.

**665c. (other methods of class Collection 665c)≡** (665b) 665d▷

```
(class-method with: (anObject)
  ((self new) add: anObject))
(class-method withAll: (aCollection)
  ((self new) addAll: aCollection))
```

When `addAll:` is sent to an object of a subclass, the message dispatches to the method shown here, which is defined on class `Collection`. It is implementing using `do:` and `add::`.

**665d. (other methods of class Collection 665c)+≡** (665b) ▷ 665c 666a▷

```
(method addAll: (aCollection)
  (aCollection do: [block (x) (self add: x)])
  self)
```

When method `addAll:` sends `do:` and `add::`, they dispatch to the methods defined on the subclass.

<sup>13</sup>According to the rules of Smalltalk, a magnitude may not inherit the default implementation of = from class `Object`, which is object identity. That's why method = is redefined as a subclass responsibility.

Removal works in the same way, building on `do:` and `remove:ifAbsent::`.

```
666a. <other methods of class Collection 665c>+≡ (665b) ▷665d 666b▷
  (method remove: (oldObject)
    (self remove:ifAbsent: oldObject {self error: 'remove-was-absent'}}))
  (method removeAll: (aCollection)
    (aCollection do: [block (x) (self remove: x)])
    self)
```

In addition to these mutators, the Collection protocol defines a host of observers, including `isEmpty` and `size`, among others (page 653). The default implementations given here iterate through the elements of the collection using `do::`.

```
666b. <other methods of class Collection 665c>+≡ (665b) ▷666a 666c▷
  (method size () [locals n]
    (set n 0)
    (self do: [block (_) (set n (n + 1))])
    n)
  (method occurrencesOf: (anObject) [locals n]
    (set n 0)
    (self do: [block (x) ((x = anObject) ifTrue: {set n (n + 1)})])
    n)
```

Using a linear search to compute `size`, for example, may seem inefficient, but if a subclass knows a more efficient way to compute the number of elements, it redefines the `size` method. And for some collections, like `List`, there really is no better way to compute `size` or count occurrences.

Other observers also iterate using `do::`, but they can cut an iteration short by evaluating a `return` expression. Like `size` and `occurrencesOf::`, these default methods are as efficient as is possible with a linked list.

```
666c. <other methods of class Collection 665c>+≡ (665b) ▷666b 666d▷
  (method isEmpty ()
    (self do: [block (_) (return false)])
    true)
  (method includes: (anObject)
    (self do: [block (x) ((x = anObject) ifTrue: {return true})])
    false)
```

```
666d. <other methods of class Collection 665c>+≡ (665b) ▷666c 666e▷
  (method detect:ifNone: (aBlock exnBlock)
    (self do: [block (x) ((aBlock value: x) ifTrue: {return x})])
    (exnBlock value))
  (method detect: (aBlock)
    (self detect:ifNone: aBlock {self error: 'no-object-detected'}))
```

In addition to mutators and observers, the Collection protocol also provides iterators. These iterators are akin to  $\mu$ Scheme's higher-order functions on lists. For example, `do::` resembles  $\mu$ Scheme's `app`, and `inject:into::` resembles  $\mu$ Scheme's `foldl`. And here is another case where data abstraction using objects offers an advantage over abstract data types: unlike  $\mu$ Scheme's `foldl`, `inject:into::` works on *any* collection, not just on lists.

```
666e. <other methods of class Collection 665c>+≡ (665b) ▷666d 667a▷
  (method inject:into: (aValue binaryBlock)
    (self do: [block (x) (set aValue (binaryBlock value:value: x aValue))])
    aValue)
```

The methods `select::`, `reject::` and `collect::` resemble  $\mu$ Scheme's `filter` and `map` functions. Like `inject:into::`, they work on all collections, not just on lists.

<code>species</code>	Answer a class that should be used to create new instances of collections like the receiver, to help with the implementation of <code>select:</code> , <code>collect:</code> , and similar methods.
<code>printName</code>	Print the name of the object's class, to help with the implementation of <code>print</code> . (Almost all <code>Collection</code> objects print as the name of the class, followed by the list of elements in parentheses. <code>Array</code> objects omit the name of the class.)

Figure 10.20: Private methods internal to `Collection` classes.

The implementations use `species`, which is a private message used to create “a new collection like the receiver” (Figure 10.20).

**667a.** *(other methods of class Collection 665c)* +≡ (665b) ▷ 666e 667b  
 (method `select:` (aBlock) [locals temp]  
   (set temp ((self `species`) new))  
   (self do: [block (x) ((aBlock value: x) ifTrue: { (temp add: x) })])  
   temp)  
 (method `reject:` (aBlock)  
   (self select: [block (x) ((aBlock value: x) not)]))  
 (method `collect:` (aBlock) [locals temp]  
   (set temp ((self `species`) new))  
   (self do: [block (x) (temp add: (aBlock value: x))])  
   temp)

A `species` defaults to the class of the receiver.

**667b.** *(other methods of class Collection 665c)* +≡ (665b) ▷ 667a 667c  
 (method `species` () (self `class`))

Finally, `Collection` defines its own `print` method. By default, a collection prints as the name of its class, followed by a parenthesized list of its elements.

**667c.** *(other methods of class Collection 665c)* +≡ (665b) ▷ 667b  
 (method `print` ()  
   (self `printName`)  
   (left-round print)  
   (self do: [block (x) (space print) (x print)])  
   (space print)  
   (right-round print)  
   self)  
 (method `printName` () (((self `class`) name) print))

Both methods may be overridden by subclasses.

### 10.6.2 Widening a protocol: Keyed and sequenceable collections

`Collection` isn't just an abstract class with multiple implementations. It's an abstraction that is refined into more abstractions:

- Keyed collections, which collect key-value pairs and can be indexed by key
- Sequenceable collections, which are keyed by consecutive integers

Each of these collections refines the protocol defined by its superclass. To study this refinement, we ask these questions:

- Which of the subclass responsibilities inherited from the superclass does it *implement*?

- What subclass responsibilities inherited from the superclass does it *pass on* to its own subclasses?
  - What new subclass responsibilities does it *add*?
  - What methods inherited from the superclass does it *override*? On what grounds?

Here are the answers:

	<i>Implements</i>	<i>Passes on</i>	<i>Adds</i>	<i>Overrides</i>
<i>Keyed</i>	do:, =	add, remove:ifAbsent	at:put, associationsDo:, removeKey:ifAbsent:	—
	associationsDo:	add, at:put, remove:ifAbsent, removeKey:ifAbsent, species	firstKey, lastKey	at:IfAbsent:

## *Implementation of KeyedCollection*

A keyed collection provides access to key-value pairs. Any subclass must define method `associationsDo:`, which replaces `do:`, method `removeKey:ifAbsent:`, which replaces `remove:ifAbsent:`, and method `at:put:`, which sometimes replaces `add:`. The key-value pairs answer the Association protocol.

**668a.** ⟨collection classes 665b⟩+≡

(S560c) <665b 669d>

```
(class KeyedCollection
  [subclass-of Collection] ; abstract class
  (method associationsDo: (aBlock) (self subclassResponsibility))
  (method removeKey:ifAbsent: (key exnBlock) (self subclassResponsibility))
  (method at:put: (key value) (self subclassResponsibility))
  <other methods of class KeyedCollection 668b>
)
```

Given `associationsDo:`, we implement the required `do:` method by iterating over associations.

**668b.** *⟨other methods of class KeyedCollection 668b⟩* ≡

(668a) 668c▷

```
(method do: (aBlock)
  (self associationsDo: [block (anAssoc) (aBlock value: (anAssoc value))]))
```

Every method in the “at” family, as well as `includesKey:`, is ultimately implemented on top of `associationAt:ifAbsent:`, which uses `associationsDo::`

**668c.** ⟨other methods of class KeyedCollection 668b⟩+≡

(668a) ◁ 668b 669a ▷

```
(method at: (key)
    (self at:ifAbsent: key {(self error: 'key-not-found)})})
(method at:ifAbsent: (key exnBlock)
    ((self associationAt:ifAbsent: key {(return (exnBlock value))}) value)))
(method includesKey: (key)
    ((self associationAt:ifAbsent: key {}) notNil))
(method associationAt: (key)
    (self associationAt:ifAbsent: key {(self error: 'key-not-found)})})
(method associationAt:ifAbsent: (key exnBlock)
    (self associationsDo: [block (x) (((x key) = key) ifTrue: {(return x)})])
    (exnBlock value)))
```

When a key is found, method `associationAt:ifAbsent:` terminates the search immediately by evaluating a return expression. All the other methods benefit from this efficiency. I use the same technique for finding keys.

```
669a. ⟨other methods of class KeyedCollection 668b⟩+≡ (668a) ▷ 668c 669b ▷
(method keyAtValue: (value)
  (self keyAtValue:ifAbsent: value { (self error: 'value-not-found') }))
(method keyAtValue:ifAbsent: (value exnBlock)
  (self associationsDo: [block (x)
    (((x value) = value) ifTrue: { (return (x key)) })])
  (exnBlock value))
```

To remove a key, use `removeAt:ifAbsent:`.

```
669b. ⟨other methods of class KeyedCollection 668b⟩+≡ (668a) ▷ 669a 669c ▷
(method removeKey: (key)
  (self removeKey:ifAbsent: key { (self error: 'key-not-found') }))
```

Rather than have to implement = separately for dictionaries, lists, and arrays, I can implement it once, here, for all keyed collections. Two keyed collections are equivalent if they have equivalent associations. For efficiency, I look first for an association that's in the receiver but not in the argument. If I find one, the collections are not equivalent, and I return `false` immediately. Otherwise, I just have to confirm that both receiver and argument have the same number of associations—if so, they are equivalent.

```
669c. ⟨other methods of class KeyedCollection 668b⟩+≡ (668a) ▷ 669b
(method = (collection)
  (self associationsDo: ; look for an 'assn' not in 'collection'
    [block (assn)
      (((assn value) != (collection at:ifAbsent: (assn key) { (return false) })))
       ifTrue:
         { (return false) }])
  ((self size) = (collection size)))
```

The classic keyed collection is `Dictionary`. My implementation, which appears in Section U.1.4, is a simple list of key-value pairs, just like the `env` type in Chapter 5. Implementations using hash tables or search trees can be explored in Exercises 28 and 29.

### *Implementation of SequenceableCollection*

The abstract class `SequenceableCollection` defines methods used by keyed collections whose keys are consecutive integers. Sequenceable collections are implemented by predefined classes `List` and `Array`.

```
669d. ⟨collection classes 665b⟩+≡ (S560c) ▷ 668a 682a ▷
(class SequenceableCollection
  [subclass-of KeyedCollection] ; abstract class
  (method firstKey () (self subclassResponsibility))
  (method lastKey () (self subclassResponsibility))
  (method last () (self at: (self lastKey)))
  (method first () (self at: (self firstKey)))
  (method at:ifAbsent: (index exnBlock) [locals current]
    (set current (self firstKey))
    (self do: [block (v)
      ((current = index) ifTrue: { (return v) })
      (set current (current + 1))])
    (exnBlock value))
  ⟨other methods of class SequenceableCollection 670a⟩
)
```

Because keys are consecutive integers, method `at:ifAbsent:` can track the value of the key inside a `do:` loop, without ever allocating an Association. This implementation is more efficient than the generic implementation inherited from class `KeyedCollection`.

Method `associationsDo:` also loops over consecutive keys.

**670a.** *(other methods of class SequenceableCollection 670a)≡* (669d)

```
(method associationsDo: (bodyBlock) [locals i last]
  (set i (self firstKey))
  (set last (self lastKey))
  ({{(i <= last)} whileTrue:
    {(bodyBlock value: (Association withKey:value: i (self at: i)))}
    (set i (i + 1))}})
```

### 10.6.3 Compromising on protocol: class Array

Classes `KeyedCollection` and `SequenceableCollection` refine the `Collection` protocol, adding new operations. Sometimes, however, a class may want to *remove* operations from a protocol; the goal is to reuse methods defined in a superclass while implementing only some of its subclass responsibilities. A classic example is an array: it is a sequenceable collection, and `at:` and `at:put:` take only constant time, but after it is allocated, an array cannot grow or shrink. As a result, the subclass responsibilities `add:`, `remove:ifAbsent`, and `removeKey:ifAbsent` are not implemented; sending any of those messages results in a checked run-time error.

**670b.** *(other methods of class Array 670b)≡* (S541c) 670c▷

```
(method add: (x) (self fixedSizeError))
(method remove:ifAbsent: (x b) (self fixedSizeError))
(method removeKey:ifAbsent: (x b) (self fixedSizeError))
(method fixedSizeError () (self error: 'arrays-have-fixed-size'))
```

Because class `Array` exists to promote efficiency, it overrides many inherited methods; in particular, it uses primitives to implement methods `size`, `at:`, and `at:put:`. These methods are then used to implement `firstKey`, `lastKey`, and `do:`.

**670c.** *(other methods of class Array 670b)+≡* (S541c) ▷670b

```
(method firstKey () 0)
(method lastKey () ((self size) - 1))
(method do: (aBlock) [locals index]
  (set index (self firstKey))
  (self size) timesRepeat:
    {(aBlock value: (self at: index))
    (set index (index + 1))})
```

Instance methods `select:` and `collect:` and class method `withAll:` are left as Exercises 20 and 21, and the rest of `Array` is relegated to Appendix U.

## 10.7 TECHNIQUE III: MULTIPLE REPRESENTATIONS THE OBJECT-ORIENTED WAY

Collection abstractions are relatively easy to implement, in part because most operations involve only one collection: the receiver. And even operations that do involve a second collection, as an argument, don't have to look at the argument's representation; methods `addAll:` and `removeAll:` simply use `do:` to iterate over

the argument's elements. But in some abstractions, like the leftist heap from Chapter 9 (page 565), an operation like heap merge does have to look at the representation of an argument. Such an operation is called *complex* (Cook 2009), and in a pure object-oriented language like Smalltalk, complex operations are not so easy to implement.

Complex operations are found throughout the protocols for magnitudes and numbers (page 659). Here are two examples:

- Operation `<` on fractions needs to look at the numerators and denominators of both fractions.
- Operation `*` on floating-point numbers needs to look at the mantissas and exponents of both numbers.

In a language that uses abstract data types, like Molecule, complex operations like these are easy to implement: if an operation can see the definition of a type, it can see the representation of every argument of that type. But in a pure object-oriented language, like Smalltalk, a complex method isn't so easy to implement: it can see only the representation of the receiver, and all it can do with an argument is send messages to it. To figure out what messages to send, we have several options:

- Our first option is to extend the argument's protocol with new messages that provide the necessary access to its representation. This technique is illustrated using classes `Number`, `Integer`, and especially `Fraction` (Sections 10.7.1 and 10.7.2).
- If we don't know the argument's representation, we might not know what new messages it can respond to. In such a case, the *receiver* can send a message to the argument telling the *argument* what new messages the receiver can respond to. This technique, called *double dispatch*, is illustrated with the integer classes (Section 10.7.3).
- Another option, if we're not sure about representations, is to *coerce* one object to have the same representation as another. For example, if we want to add `aNumber` to a fraction, we might coerce `aNumber` to a fraction. Coercion is illustrated on the `Fraction` and `Integer` classes (Section 10.7.4).

All these options can work with any representation. To help you integrate these techniques into your own programming, I recommend a case study for which there is more than one reasonable representation: arithmetic on natural numbers. A natural number could reasonably be represented as an array of digits or as a list of digits, and each representation suggests its own set of new messages that are analogous to the new messages defined on class `Fraction` (Section 10.7.5).

### 10.7.1 A context for complex operations: Abstract classes `Number` and `Integer`

Before studying complex operations on fractions, we look at the context in which fractions are defined. A fraction is a number, and abstract class `Number` (Figure 10.17 on page 659) defines two groups of subclass responsibilities: arith-

metic methods (+, \*, negated, and reciprocal) and coercion methods (asInteger, asFraction, asFloat, and coerce:).

**672a.** *(definition of class Number 672a)* ≡

```
(class Number
  [subclass-of Magnitude] ; abstract class
 ;;;;;; arithmetic
  (method + (aNumber)      (self subclassResponsibility))
  (method * (aNumber)      (self subclassResponsibility))
  (method negated ()       (self subclassResponsibility))
  (method reciprocal ()   (self subclassResponsibility))

 ;;;;;; coercion
  (method asInteger ()     (self subclassResponsibility))
  (method asFraction ()    (self subclassResponsibility))
  (method asFloat ()       (self subclassResponsibility))
  (method coerce: (aNumber) (self subclassResponsibility))
  <other methods of class Number 672b>
)
```

Subclasses of Number must also implement subclass responsibilities for magnitudes: methods = and <. Methods =, <, +, \*, and coerce: take another Number as argument, and all except coerce: turn out to be complex.

To get a feel for the class, let's see how subclass responsibilities are used to implement other parts of the Number protocol (page 659). Arithmetic methods are implemented on top of subclass arithmetic methods, and sign tests are implemented on top of comparison and coercion:

**672b.** *(other methods of class Number 672b)* ≡

(672a)

```
(method - (y) (self + (y negated)))
(method abs () ((self isNegative) ifTrue:ifFalse: { (self negated) } { self }))
(method / (y) (self * (y reciprocal)))

(method isNegative () (self < (self coerce: 0)))
(method isNonnegative () (self >= (self coerce: 0)))
(method isStrictlyPositive () (self > (self coerce: 0)))
```

The Number protocol also requires methods squared, sqrt, sqrtWithin:, and raisedToInteger:, which are relegated to the Supplement.

Before we get to class Fraction, we should also sketch class Integer, which provides the gcd: operation needed to put a fraction in lowest terms. The gcd: method is one of the four division-related methods gcd:, lcm:, div:, and mod:. Only div: has to be implemented in subclasses.

**672c.** *(numeric classes 665a) +≡*

(S560c) &lt;665a 673b&gt;

```
(class Integer
  [subclass-of Number] ; abstract class
  (method div: (n) (self subclassResponsibility))
  (method mod: (n) (self - (n * (self div: n))))
  (method gcd: (n) ((n = (self coerce: 0))
                     ifTrue:ifFalse: { self } { (n gcd: (self mod: n)) }))
  (method lcm: (n) (self * (n div: (self gcd: n))))
  <other methods of class Integer 673a>
)
```

Class Integer typically has three concrete subclasses: SmallInteger, for integers that fit in a machine word (Appendix U); LargePositiveInteger, for arbitrarily large positive integers; and LargeNegativeInteger, for arbitrarily large negative integers (Section 10.7.3). In  $\mu$ Smalltalk, only SmallInteger is defined; the other two are meant to be added by you (Exercise 38).

Fractions use integer operations, but integers also use fraction operations: When integers are divided using the standard method `/`, the result is a fraction.

**673a.** *(other methods of class Integer 673a)*  $\equiv$

(672c) 678b▷

```
(method reciprocal () (Fraction num:den: 1 self))
(method / (aNumber) ((self asFraction) / aNumber))
```

### 10.7.2 Implementing complex operations: class Fraction

Now can look at the complex methods of class `Fraction`. What makes a method complex is its need to inspect the representation of an argument. For a fraction, that representation comprises a numerator `num` and denominator `den`, both of which are integer instance variables.

**673b.** *(numeric classes 665a)*  $\doteq\equiv$

(S560c) ▲672c 678a▷

```
(class Fraction
  [subclass-of Number]
  [ivars num den]
  (method print () (num print) ('/ print) (den print) self)
  ⟨other methods of class Fraction 673c⟩
)
```

The abstraction function for `Fraction` maps this representation to the ratio  $\frac{\text{num}}{\text{den}}$ . Such ratios can be compared, added, and multiplied only if each method has access to the numerator and denominator of its argument, not just its receiver. Access is provided by private methods `num` and `den`.

**673c.** *(other methods of class Fraction 673c)*  $\equiv$

(673b) 673d▷

```
(method num () num) ; private
(method den () den) ; private
```

Methods `num` and `den` are used to implement the four complex operations `=`, `<`, `*`, and `+`. Each implementation relies on and guarantees these representation invariants:

1. The denominator is always positive.
2. If the numerator is zero, the denominator is 1.
3. The numerator and denominator are reduced to lowest terms, that is, their only common divisor is 1.

These invariants imply that two `Fraction` objects represent the same fraction if and only if they have the same numerator and denominator, and they enable the following implementations of the comparison methods from class `Magnitude`:

**673d.** *(other methods of class Fraction 673c)*  $\doteq\equiv$

(673b) ▲673c 673e▷

```
(method = (f) ((num = (f num)) and: { (den = (f den)) }))
(method < (f) ((num * (f den)) < ((f num) * den)))
```

The `<` method uses the law that  $\frac{n}{d} < \frac{n'}{d'}$  if and only if  $n \cdot d' < n' \cdot d$ , which holds only because  $d$  and  $d'$  are positive. And it's worth emphasizing that argument `f` doesn't have to be an instance of class `Fraction`; it's enough if `f` is a number and if it responds sensibly to messages `num` and `den`.

Methods `=` and `<` rely on the representation invariants. To establish the invariants for any given fraction, class `Fraction` uses two private methods: method `signReduce` establishes invariant 1, and method `divReduce` establishes invariants 2 and 3:

**673e.** *(other methods of class Fraction 673c)*  $\doteq\equiv$

(673b) ▲673d 674a▷

```
(method signReduce () ; private
  ((den isNegative) ifTrue:
    { (set num (num negated)) (set den (den negated)) })
  self)
```

§10.7  
Technique III:  
Multiple  
representations

673

**674a.** *(other methods of class Fraction 673c)* +≡  
 (method divReduce () [locals temp] ; private  
   ((num = 0) ifTrue:ifFalse:  
     {(set den 1)}  
     {(set temp ((num abs) gcd: den))}  
     {(set num (num div: temp))}  
     {(set den (den div: temp))})  
   self)

When a new Fraction is created by public class method num:den:, all three invariants are established by private method initNum:den:.

**674b.** *(other methods of class Fraction 673c)* +≡  
 (class-method num:den: (a b) ((self new) initNum:den: a b))  
 (method setNum:den: (a b) (set num a) (set den b) self) ; private  
 (method initNum:den: (a b) ; private  
   (self setNum:den: a b)  
   (self signReduce)  
   (self divReduce))

Private method setNum:den: sets the numerator and denominator of a fraction, but it does not establish the invariants. It is used in multiplication and addition.

Multiplication is specified by the law  $\frac{n}{d} \cdot \frac{n'}{d'} = \frac{n \cdot n'}{d \cdot d'}$ , but the right-hand side could violate invariant 2 or 3. (Numerator  $n \cdot n'$  and denominator  $d \cdot d'$  can have common factors, but  $d \cdot d'$  cannot be negative.) The multiplication method therefore sends divReduce to the naive result.

**674c.** *(other methods of class Fraction 673c)* +≡  
 (method \* (f)  
   (((Fraction new) setNum:den: (num \* (f num)) (den \* (f den))) divReduce))

Addition is specified by the law  $\frac{n}{d} + \frac{n'}{d'} = \frac{n \cdot d' + n' \cdot d}{d \cdot d'}$ . The result may violate invariants 2 and 3, but divReduce can eliminate common factors, as in the example  $\frac{1}{2} + \frac{1}{2} = \frac{4}{4} = \frac{1}{1}$ . In addition, the computation of  $d \cdot d'$  might overflow. To make overflow less likely, I define temp = lcm(d, d'), putting temp in the denominator, and using  $\frac{\text{temp}}{d}$  and  $\frac{\text{temp}}{d'}$  as needed. Without this tweak, the square-root computations in Section 10.4.6 would overflow.

**674d.** *(other methods of class Fraction 673c)* +≡  
 (method + (f) [locals temp]  
   (set temp (den lcm: (f den)))  
   (((Fraction new) setNum:den:  
     (((num \* (temp div: den)) +  
       ((f num) \* (temp div: (f den))))  
       temp)  
     divReduce))

Method + is the last of the complex methods. But it's not the last of the methods that rely on or guarantee invariants. For example, reciprocal must not leave a negative fraction with a negative denominator. Sending signReduce to the inverted fraction sets things right regardless of the original sign. (The reciprocal of zero cannot be put into reduced form. Nothing can be done about it.)

**674e.** *(other methods of class Fraction 673c)* +≡  
 (method reciprocal()  
   (((Fraction new) setNum:den: den num) signReduce))

Negation simply negates the numerator; the invariants are guaranteed to be maintained.

**674f.** *(other methods of class Fraction 673c)* +≡  
 (method negated () ((Fraction new) setNum:den: (num negated) den))

The invariants enable the sign tests to inspect only the receiver's numerator. These tests are much more efficient than the versions inherited from `Number`.

**675a.** *(other methods of class Fraction 673c)*  $\equiv$  (673b)  $\triangleleft$  674f 677b  $\triangleright$   
(method `isNegative` () (`num isNegative`))  
(method `isNonnegative` () (`num isNonnegative`))  
(method `isStrictlyPositive` () (`num isStrictlyPositive`))

The final responsibilities of subclass `Fraction` involve coercion, which is the topic of Section 10.7.4 (page 677).

### 10.7.3 Interoperation with more than one representation: Double dispatch

Given methods `num` and `den`, we can compare fractions with fractions, add fractions to fractions, and so on. None of these operations can see the representation of its argument, but they don't have to: it's enough to know that the argument responds to messages `num` and `den`. But when we want to implement the same operations on integers, things are not so simple.

The difficulty arises because Smalltalk supports two representations of integers: small and large. A small integer must fit in a machine word; a large integer may be arbitrarily large. Both are integers, so both should respond to a `+` message, but unlike the `+` method on fractions, the `+` method on integers doesn't always implement the same algorithm. Instead, the algorithm depends on what its being added:

- If two small integers are being added, the algorithm is to use a hardware addition instruction.
- If two large integers are being added, the algorithm is to add the digits pairwise, with carry digits, as described in Appendix B.
- If a large integer and a small integer are being added, the algorithm is to coerce the small integer to a large integer, then add the resulting large integers.

How is the right algorithm chosen? The `+` method doesn't interrogate objects to ask about their classes. (The object-oriented motto is, "Don't ask; tell.") Instead, when a small integer receives a `+` message, it sends another message to its *argument*, saying, "I'm a small integer; add me to yourself."

**675b.** *(SmallInteger methods revised to use double dispatch 675b)*  $\equiv$   
(method `+` (`anInteger`) (`anInteger addSmallIntegerTo: self`))

The right algorithm is chosen by method `addSmallIntegerTo::`.

- If `addSmallIntegerTo:` is sent to a small integer, the method uses a machine primitive to add `self` to the argument.
- If `addSmallIntegerTo:` is sent to a large integer, the method coerces the argument to a large integer, then sends the receiver a `+` message.

This technique, by which a complex binary operation is implemented in two message sends instead of one, is called *double dispatch*.

The story so far glosses over an important fact about large-integer operations; a large integer has both a sign and a magnitude, and the algorithm for adding large integers depends on the sign: If two integers have the same sign, their magnitudes are added, but if they have different signs, their magnitudes are subtracted. As always, we don't write a conditional interrogating an integer about its sign; instead, we put positive and negative integers in different classes, and we decide on the algorithm by dispatching to a method on the right class. That means that a large-integer

Class protocol for LargeInteger:

<code>fromSmall: aSmallInteger</code>	Answer a large integer whose value is equal to the value of the argument.
<code>withMagnitude: aNatural</code>	Answer an instance of the receiver whose magnitude is the given magnitude.

Protocol for large-integer objects:

<code>addSmallIntegerTo: aSmallInteger</code>	Answer the sum of the argument and the receiver.
<code>addLargePositiveIntegerTo: aLargePositiveInteger</code>	Answer the sum of the argument and the receiver.
<code>addLargeNegativeIntegerTo: aLargeNegativeInteger</code>	Answer the sum of the argument and the receiver.
<code>multiplyBySmallInteger: aSmallInteger</code>	Answer the product of the argument and the receiver.
<code>multiplyByLargePositiveInteger: aLargePositiveInteger</code>	Answer the product of the argument and the receiver.
<code>multiplyByLargeNegativeInteger: aLargeNegativeInteger</code>	Answer the product of the argument and the receiver.
<code>magnitude</code>	Answer an object of class Natural that represents the absolute value of the receiver.
<code>sdiv: aSmallInteger</code>	Answer the largest natural number whose value is at most the quotient of the receiver and the argument.
<code>smod: aSmallInteger</code>	Answer a small integer which is the remainder when the receiver is divided by the argument.

Figure 10.21: Private protocols for large integers

class must support *three* double-dispatch methods for addition: it can be told to add a *small* integer to itself, add a large *positive* integer to itself, or add a large *negative* integer to itself. These methods are shown, along with other private methods for large integers, in Figure 10.21.

Given these methods, the implementations of `+` and `*` invoke the method appropriate to the operation and the class of the receiver. For example, method `+` on class `LargePositiveInteger` is implemented as follows:

```
(method + (anInteger) (anInteger addLargePositiveIntegerTo: self))
```

The rest of Figure 10.21 describes the other methods needed to implement arithmetic. Method `magnitude` plays the same role for large integers that methods `num` and `den` play for fractions. And methods `sdiv:` and `smod:` provide a protocol for dividing a large integer by a small integer. (Division of a large integer by another large integer requires long division, which is beyond the scope of this book.)

A starter kit for class `LargeInteger` is shown in Figure 10.22 on the next page. The `LargeInteger` class is meant to be abstract; do not instantiate it. Instead, define subclasses `LargePositiveInteger` and `LargeNegativeInteger`, which you can then instantiate. You can find the details in Exercise 38.

**677a.** *(large integers 677a)≡*

```
(class LargeInteger
  [subclass-of Integer]
  [ivars magnitude]

  (class-method withMagnitude: (aNatural)
    ((self new) magnitude: aNatural))
  (method magnitude: (aNatural) ; private, for initialization
    (set magnitude aNatural)
    self)

  (method magnitude () magnitude)

  (class-method fromSmall: (anInteger)
    ((anInteger isNegative) ifTrue:ifFalse:
      {{{(self fromSmall: 1) + (self fromSmall: ((anInteger + 1) negated)))}
       negated}}
     {{{(LargePositiveInteger new) magnitude: (Natural new: anInteger))}})
  (method asLargeInteger () self)
  (method isZero () (magnitude isZero))
  (method = (anInteger) ((self - anInteger) isZero))
  (method < (anInteger) ((self - anInteger) isNegative))

  (method div: (_) (self error: 'long-division-not-supported'))
  (method mod: (_) (self error: 'long-division-not-supported'))

  (method sdiv: (aSmallInteger) (self leftAsExercise))
  (method smod: (aSmallInteger) (self leftAsExercise))
)
```

§10.7  
Technique III:  
Multiple  
representations

677

Figure 10.22: Abstract class LargeInteger

#### 10.7.4 Coercion between abstractions in Fraction and Integer

No matter what class it's defined on, when you use a binary method like `<` or `+`, you have to make sure the receiver and the argument are compatible. In most situations you'll know that you have compatible numbers, but when you don't, you can make them compatible using *coercion*. In Smalltalk, every number is required to be able to coerce itself to be an integer, a floating-point number, or a fraction. Coercion methods typically use the public protocol of the classes they are coercing themselves to, like these methods defined on class Fraction:

**677b.** *(other methods of class Fraction 673c) +≡*

(673b) ▷ 675a 677c ▷

```
(method asInteger () (num div: den))
(method asFloat () ((num asFloat) / (den asFloat)))
(method asFraction () self)
```

To coerce itself to another class of number, a fraction divides num by den. Coercion to an integer uses integer division `div:`; coercion to a floating-point number first coerces num and den to floating point, then divides. (To coerce itself to a fraction, a fraction needn't divide at all.)

To make two numbers compatible, you may tell one to coerce its argument to be like itself. A fraction coerces its argument to a fraction.

**677c.** *(other methods of class Fraction 673c) +≡*

(673b) ▷ 677b

```
(method coerce: (aNumber) (aNumber asFraction))
```

The coercion methods on classes Integer and Float follow the same structure. Class Float is relegated to Appendix U, but the Integer methods are shown here.

**678a.** *⟨numeric classes 665a⟩* +≡ (S560c) ▷ 673b

```

(class Natural
  [subclass-of Magnitude]
  ; instance variables left as an exercise

  (class-method fromSmall: (anInteger) (self leftAsExercise))

  (method = (aNatural) (self leftAsExercise))
  (method < (aNatural) (self leftAsExercise))

  (method + (aNatural) (self leftAsExercise))
  (method * (aNatural) (self leftAsExercise))
  (method - (aNatural)
    (self subtract:withDifference:ifNegative:
      aNatural
      [block (x) x]
      {(self error: 'Natural-subtraction-went-negative)})})
  (method subtract:withDifference:ifNegative: (aNatural diffBlock exnBlock)
    (self leftAsExercise))

  (method sdiv: (n) (self sdivmod:with: n [block (q r) q]))
  (method smod: (n) (self sdivmod:with: n [block (q r) r]))
  (method sdivmod:with: (n aBlock) (self leftAsExercise))

  (method decimal () (self leftAsExercise))
  (method isZero () (self leftAsExercise))

  (method print () ((self decimal) do: [block (x) (x print)])))
)
```

Figure 10.23: Template for a definition of class Natural

Just as a fraction must know what integer or floating-point operations to use to divide its numerator by its denominator, an integer must know what fractional or floating-point operations to use to represent an integer. In this case, it's self divided by 1 and self times a base to the power 0, respectively.

**678b.** *⟨other methods of class Integer 673a⟩* +≡ (672c) ▷ 673a 678c ▷

```

(method asFraction () (Fraction num:den: self 1))
(method asFloat () (Float mant:exp: self 0))
```

Just as in class Fraction, the other two methods simply exploit the knowledge that the receiver is an integer:

**678c.** *⟨other methods of class Integer 673a⟩* +≡ (672c) ▷ 678b

```

(method asInteger () self)
(method coerce: (aNumber) (aNumber asInteger))
```

### 10.7.5 Choice of representation: Natural numbers

The examples above show how to work with complex methods on representations that I have defined. To solidify your understanding, I recommend that you apply the same techniques to a representation that you can define: a representation of natural numbers (Exercise 37). Here I present two possible representations, with hints, ideas and private protocols that will help with the implementation.

A natural number is represented as a sequence of digits in some base  $b$  (Appendix B). The sequence may be represented as an array or as a list. I recommend

against using  $\mu$ Smalltalk's predefined class `List`; you are better off defining empty and nonempty lists of digits as subclasses of class `Natural1`. For this reason, I refer to the two representations as the “array representation” and the “subclass representation.” Each representation calls for its own private protocol to be used to implement the complex methods. But no matter which representation you choose, you can start with the template in Figure 10.23.

#### *Natural numbers: the array representation*

If a natural number is represented using an array of digits, I recommend giving it two instance variables: `degree` and `digits`. The representation invariant should be as follows: `digits` should be an array containing at least `degree + 1` integers, each of which lies in the range  $0 \leq x_i < b$ , where  $b$  is the base. If `digits` contains coefficients  $x_i$ , where  $0 \leq i \leq \text{degree}$ , then the abstraction function says that the object represents natural number  $X$ , where

$$X = \sum_{i=0}^{\text{degree}} x_i \cdot b^i.$$

With this array representation, I recommend the private protocol shown in Figure 10.24 on the following page.

- The `base` method on the class provides a single point of truth about  $b$ , which you choose.
- The digit-related methods are used to read, write, and iterate over digits.
- Methods `trim` and `degree` are used to keep the arrays as small as possible, so leading zeroes don't accumulate.
- Method `makeEmpty:` is used to initialize newly allocated instances.

My experience using the array representation suggests that it offers these trade-offs: Because you have easy access to any digit you like, you can treat Smalltalk as if it were a procedural language, like C. In particular, you can get away without thinking too hard about dynamic dispatch, because a lot of decisions can be made by looking at digits and at `degree`. But the individual methods are a little complicated, and you may not learn a whole lot—my array-based code uses objects only to hide information from client code, and it doesn't exploit dynamic dispatch or inheritance.

#### *Natural numbers: the subclass representation*

If a natural number is represented as a list of digits, I recommend defining two additional classes that inherit from `Natural`: `NatZero` and `NatNonzero`. An instance of class `NatZero` represents zero, and it doesn't need any instance variables. An instance of class `NatNonzero` represents the natural number  $x_0 + X' \cdot b$ , where  $x_0$  is a digit (a small integer) and  $X'$  is a natural number. Class `NatNonzero` needs instance variables for  $x_0$  and  $X'$ ; these might be called `x-0` and `other-digits`. The representation invariant is that `x-0` and `other-digits` are not both zero.

With the subclass representation, I recommend the private protocol in Figure 10.25 on page 681.

- As with arrays, class method `base` provides a single point of truth about  $b$ .

Private class method for class Natural:

<code>base</code>	Answers <i>b</i> .
-------------------	--------------------

Private instance methods for class Natural:

<code>digit: anIndex</code>	Upon receiving <code>digit: i</code> , answer $x_i$ . Should work for any nonnegative <i>i</i> , no matter how large.
<code>digit:put: anIndex aDigit</code>	On receiving <code>digit:put: i y</code> , mutate the receiver, making $x_i = y$ . Although Natural is not a mutable type (and therefore this method should never be called by clients), it can be quite useful to mutate individual digits while you are constructing a new instance.
<code>digits: aSequence</code>	Take a sequence of $x_i$ and use it to initialize digits and degree.
<code>doDigitIndices: aBlock</code>	For <i>i</i> from zero to degree, send value <i>i</i> to aBlock.
<code>trim</code>	Set degree on the receiver as small as possible, and answer the receiver.
<code>degree</code>	Answer the degree of the receiver.
<code>makeEmpty: aDegree</code>	Set digits to an array suitable for representing natural numbers of the specified degree. (Also change the degree of the receiver to aDegree.)

Figure 10.24: Suggested private methods for class Natural, array representation

- Class method `first:rest:` creates a new instance of one of the two subclasses. If both arguments are zero, it answers an instance of class `NatZero`. Otherwise, it answers an instance of class `NatNonzero`.
- Private methods `modBase`, `divBase`, and `timesBase`, together with public method `isZero` (Figure 10.19), are the protocol that allows a method to inspect its argument. If a natural number  $X$  is  $x_0 + X' \cdot b$ , then `modBase` answers  $x_0$  and `divBase` answers  $X'$ . (If a natural number is zero, it answers all of these messages with zero.)
- The comparison method simplifies the implementations of methods `<` and `=`, which are subclass responsibilities of class `Natural` (from the `Magnitude` protocol).
- Methods `plus:carry:` and `minus:borrow:` implement functions `adc` and `sbb`, which are explained in Appendix B.

My experience using the subclass representation suggests that it offers these tradeoffs: You have easy access only to the least significant digit of a natural number (using `modBase`). In each method, a lot of decisions about what to do next and when algorithms should terminate are made for you by dynamic dispatch: the action is determined by the class on which the method is defined. Each individual method is simpler than corresponding methods that use the array representation. For example, the `+` method defined on class `NatZero` simply returns its argument, and the `*` method simply returns zero. No conditionals, no scrutiny, end of story.

Private class method for class Natural:

<code>base</code>	Answers $b$ , the base of Natural numbers.
<code>first:rest: anInteger aNatural</code>	Answers a Natural number representing $\text{anInteger} + \text{aNatural} \cdot b$ .

§10.8

Technique IV:  
Object-oriented  
programming with  
invariants

681

Private instance methods for class Natural:

<code>modBase</code>	Answers a small integer whose value is the receiver modulo the base of Natural numbers.
<code>divBase</code>	Answers a Natural whose value is the receiver divided by the base of Natural numbers.
<code>timesBase</code>	Answers a Natural whose value is the receiver multiplied by the base of Natural numbers.
<code>compare:withLt:withEq:withGt: aNatural ltBlock eqBlock gtBlock</code>	Compares <code>self</code> with <code>aNatural</code> . If <code>self</code> is smaller than <code>aNatural</code> evaluate <code>ltBlock</code> . If they are equal, evaluate <code>eqBlock</code> . If <code>self</code> is greater, evaluate <code>gtBlock</code> .
<code>plus:carry: aNatural c</code>	Answer the sum <code>self + aNatural + c</code> , where <code>c</code> is a carry bit (either 0 or 1).
<code>minus:borrow: aNatural c</code>	Compute the difference <code>self - (aNatural + c)</code> , where <code>c</code> is a borrow bit (either 0 or 1). If the difference is nonnegative, answer the difference; otherwise, halt the program with a checked run-time error.

Figure 10.25: Suggested private methods for class Natural, subclass representation

But although the individual methods are simple, you won't understand the overall algorithm unless you understand dynamic dispatch.

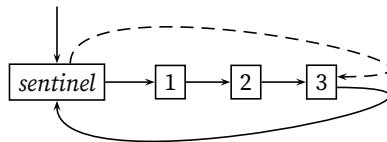
## 10.8 TECHNIQUE IV: OBJECT-ORIENTED PROGRAMMING WITH INVARIANTS

The program-design techniques described in Chapter 9 aren't just for abstract data types; they work just as well with objects. This section demonstrates the use of abstraction function and representation invariant to implement a linked list with a good cost model: linear-time traversal and constant-time access to first and last elements.

As in  $\mu$ Scheme, the representation uses cons cells. But unlike  $\mu$ Scheme code,  $\mu$ Smalltalk code never asks a list if it is null or cons. Instead, empty and nonempty lists are represented by objects of different classes, and decisions are made by dispatching to the right method.

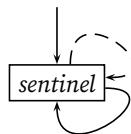
To support efficient insertion and deletion at either end of a list, I represent it using a *circular* list of cons cells. This representation relies on a sophisticated invariant: both the beginning and end of the list are marked by a special cons cell, which is called a *sentinel*. Using a sentinel is a standard technique that often simplifies the implementation of a data structure (Sedgewick 1988). Because I am programming in  $\mu$ Smalltalk, I can attach all the special-case end-of-list code to the sentinel, as methods of class `ListSentinel`. As a result, I implement lists without even one conditional that checks for end of list.

Just as in  $\mu$ Scheme, a cons cell holds two values: a car and a cdr. Unlike in  $\mu$ Scheme, the cdr *always* points to another cons cell. This invariant holds because every list is circularly linked—the other cons cell might be a sentinel. For example, here is a list containing the elements 1, 2, and 3:



A sentinel contains *two* pointers: the cdr, which it inherits from class Cons, points to the elements of the list, if any; the pred (shown as a dashed line), which is found only on objects of class ListSentinel, points to the sentinel's predecessor.

A sentinel's predecessor is normally the last element of its list, but if a list is empty then both fields of the sentinel point to the sentinel itself:



When the cdr points to the sentinel itself, the abstraction function maps the representation to the empty sequence. When the cdr points to another object, that object is a cons cell, and the abstraction function maps the representation to the sequence of objects stored in the cons cells pointed to by the cdr of the sentinel.

Each cons cell, including the sentinel, responds to the protocol shown in Figure 10.26 on the facing page.

The representation of a List stores a pointer to the sentinel in instance variable sentinel.

**682a.**  $\langle$ collection classes 665b $\rangle + \equiv$  (S560c)  $\triangleleft$  669d  
 $\langle$ classes that define cons cells and sentinels 684a $\rangle$   
 $\langle$ class List  
 [subclass-of SequenceableCollection]  
 [ivars sentinel]  
 (class-method new () ((super new) sentinel: (ListSentinel new)))  
 (method sentinel: (s) (set sentinel s) self) ; private  
 (method isEmpty () (sentinel == (sentinel cdr)))  
 (method last () ((sentinel pred) car))  
 (method do: (aBlock) ((sentinel cdr) do: aBlock))  
 (other methods of class List 682b)  
 )

The method addLast: mutates a list by adding an element to the end. This means inserting an element just after the predecessor of the sentinel. Similarly, addFirst: inserts just after the sentinel itself. Having a sentinel means there is no special-case code for an empty list.

**682b.**  $\langle$ other methods of class List 682b $\rangle \equiv$  (682a) 682c  $\triangleright$   
 (method addLast: (item) ((sentinel pred) insertAfter: item) self)  
 (method addFirst: (item) (sentinel insertAfter: item) self)  
 (method add: (item) (self addLast: item))

To remove the first element of a list, we remove the element after the sentinel. Removing the last element is left as Exercise 25.

**682c.**  $\langle$ other methods of class List 682b $\rangle + \equiv$  (682a)  $\triangleleft$  682b 683a  $\triangleright$   
 (method removeFirst () (sentinel deleteAfter))  
 (method removeLast () (self leftAsExercise))

<code>car</code>	Answer the car of the receiver.
<code>cdr</code>	Answer the cdr of the receiver.
<code>car: anObject</code>	Set the receiver's car and answer the receiver.
<code>cdr: anObject</code>	Set the receiver's cdr and answer the receiver.
<code>pred: aCons</code>	Notify the receiver that its predecessor is the cons cell <code>aCons</code> .
<code>deleteAfter</code>	Delete the cons cell that the receiver's cdr points to. Answer the car of that cons cell.
<code>insertAfter: anObject</code>	Insert a new cons cell after the receiver, letting the new cons cell's car point to <code>anObject</code> . Answer <code>anObject</code> .
<code>do: aBlock</code>	For each cons cell <code>c</code> in the receiver, excluding the sentinel, use a <code>value:</code> message to send ( <code>c car</code> ) to <code>aBlock</code> .
<code>rejectOne:ifAbsent:withPred: aBlock exnBlock aCons</code>	Starting at the receiver, search the list for a cons cell <code>c</code> such that ( <code>aBlock value: c</code> ) is true. If such a cell is found, remove it. Otherwise, answer ( <code>exnBlock value</code> ). As a precondition, the argument <code>aCons</code> <i>must</i> be the predecessor of the receiver.

In addition, a sentinel (and only a sentinel) can respond to this message:

<code>pred</code>	Answer the predecessor of the receiver.
-------------------	---

Figure 10.26: Protocol for cons cells

To remove a given element, we tell the first cons cell in the list to remove it.

**683a.** *(other methods of class List 682b)* +≡ (682a) ▷ 682c 683b ▷  
 (method `remove:ifAbsent: (oldObject exnBlock)`  
 ((sentinel cdr)  
   rejectOne:ifAbsent:withPred:  
   [block (x) (oldObject = (x car))]  
   exnBlock  
   sentinel))

Removal by key is left as an exercise.

**683b.** *(other methods of class List 682b)* +≡ (682a) ▷ 683a 683c ▷  
 (method `removeKey:ifAbsent: (n exnBlock) (self leftAsExercise)`)

List is a subclass of SequenceableCollection, so it must answer messages involving integer keys. The first key in a List is always 0.

**683c.** *(other methods of class List 682b)* +≡ (682a) ▷ 683b  
 (method `firstKey () 0`)  
 (method `lastKey () ((self size) - 1)`)  
 (method `at:put: (n value) [locals tmp]`  
 (set `tmp` (sentinel cdr))  
 ({(n = 0)} whileFalse:  
   {(set `n` (`n` - 1))  
    (set `tmp` (`tmp` cdr)))}  
   (`tmp car: value`)  
   `self`)

Method `at:put:` simply walks the car links  $n - 1$  times. If  $n$  is out of range, you can

get wrong answers. Wrong answers can be prevented in more than one way; Exercise 26 on page 729 asks you to build and compare two preventative mechanisms.

The hard work of implementing the list methods is done by methods defined on cons cells. We begin with the simple methods that expose the representation as a pair of car and cdr. The `pred:` method makes it possible to tell *any* cons cell what its predecessor is, but unless the cons cell is a sentinel, the information is thrown away. (The sentinel an instance of a subclass of `Cons`.)

**684a.** *(classes that define cons cells and sentinels 684a)≡* (682a) 685c▷

```
(class Cons
  [subclass-of Object]
  [ivars car cdr]
  (method car () car)
  (method cdr () cdr)
  (method car: (anObject) (set car anObject) self)
  (method cdr: (anObject) (set cdr anObject) self)
  (method pred: (aCons) nil)
  (more methods of class Cons 684b)
)
```

Insertion and deletion use the standard pointer manipulations for a circularly linked list. The only unusual bit is that when we delete or insert a node, we notify the successor node of its new predecessor.

**684b.** *(more methods of class Cons 684b)≡* (684a) 684c▷

```
(method deleteAfter () [locals answer]
  (set answer (cdr car))
  (set cdr (cdr cdr))
  (cdr pred: self)
  answer)
(method insertAfter: (anObject)
  (set cdr (((Cons new) cdr: cdr) car: anObject))
  ((cdr cdr) pred: cdr)
  anObject)
```

Iteration and removal really take advantage of object-oriented programming. By defining one method differently on class `Cons` and on class `ListSentinel`, I create code that iterates over a list without ever using an explicit `if` or `while`—all it does is method dispatch. To make the computation a little clearer, I present some of the methods of class `Cons` right next to the corresponding methods of class `ListSentinel`.

To iterate over all the cons cells in a list, we do the current cons cell, then continue with a tail call to the `do:` method of the next cons cell. The iteration terminates in the sentinel, whose `do:` method does nothing.

**684c.** *(more methods of class Cons 684b)+≡* (684a) <684b 685a▷

```
(method do: (aBlock)
  (aBlock value: car)
  (cdr do: aBlock))
```

**684d.** *(iterating methods of class ListSentinel 684d)≡* (685c) 685b▷

```
(method do: (aBlock) nil)
```

Similarly, to remove the first cons cell satisfying `aBlock`, we try the current cons cell, and if it works, we remove it using `(pred deleteAfter)`. Otherwise, we continue by trying the next cons cell. If we get to the sentinel, we haven't found what we're looking for, and we terminate the loop by sending the `value` message to the exception block.

```

685a. ⟨more methods of class Cons 684b⟩+≡ (684a) ▷ 684c
(method rejectOne;ifAbsent:withPred: (aBlock exnBlock pred)
 ((aBlock value: self) ifTrue;ifFalse:
  {(pred deleteAfter)}
  {(cdr rejectOne;ifAbsent:withPred: aBlock exnBlock self)}))

```

**685b.** ⟨iterating methods of class ListSentinel 684d⟩+≡ (685c) ▷ 684d

```

(method rejectOne;ifAbsent:withPred: (aBlock exnBlock pred)
 (exnBlock value))

```

The final instance methods of class ListSentinel expose the pred pointer. And class method new allocates a new sentinel, whose pred and cdr both point to itself. Such a sentinel represents an empty list.

**685c.** ⟨classes that define cons cells and sentinels 684a⟩+≡ (682a) ▷ 684a

```

(class ListSentinel
 [subclass-of Cons]
 [ivars pred]
 (method pred: (aCons) (set pred aCons))
 (method pred () pred)
 (class-method new ()
 [locals tmp]
 (set tmp (super new))
 (tmp pred: tmp)
 (tmp cdr: tmp)
 tmp)
 ⟨iterating methods of class ListSentinel 684d⟩)

```

§10.9  
Operational semantics  
685

## 10.9 OPERATIONAL SEMANTICS

The operational semantics of  $\mu$ Smalltalk is in the same family as the operational semantics of  $\mu$ Scheme: it's a big-step semantics in which each variable name stands for a mutable location. And both semantics use closures; a block in  $\mu$ Smalltalk works about the same way as a lambda expression in  $\mu$ Scheme. But there are several differences:

- Unlike a block, a  $\mu$ Smalltalk method does not evaluate to a closure; a method is represented as code plus a static superclass, with no other environment. When a message is dispatched to a method, the method's body is evaluated in an environment built from global variables, instance variables, message arguments, and local variables. Like arguments and local variables, the instance variables and global variables are not known until the message is sent. Because methods send messages that activate other methods, global variables must be available separately, to help build the environment of the next method. The globals' locations are therefore stored in their own environment  $\xi$ . All other variables' locations are stored in environment  $\rho$ .
- The semantics of the control operator return, which terminates the method in which it appears, cannot be expressed using the same judgment as the evaluation of an expression. The return expression is a “control operator” like those described in Chapter 3, and like those other control operators, it can be described using a small-step semantics with an explicit stack. But this would be a bad idea: unless you already understand how a language works, that kind of semantics is hard to follow. Instead, I add a new judgment form to our big-step semantics: with some parts omitted, the judgment looks like  $\langle e, \dots \rangle \uparrow \langle v, F; \dots \rangle$ , and it means that evaluating  $e$  causes stack frame  $F$  to return value  $v$ .

- Every value is an object, and every object has both a *class* and a *representation*. The operational semantics may have to look at both, so when necessary, a value  $v$  is written in the form  $v = \langle \text{class}, \text{rep} \rangle$ .
- The behaviors of literal integers, symbols, and arrays are defined by classes, and if class `SmallInteger`, `Symbol`, or `Array` is redefined, the behaviors of the associated literals change. For example, if you complete Exercises 36, 38, and 39, you will change the behavior of integer literals to provide a seamless, transparent blend of small- and large-integer arithmetic, all without touching the  $\mu$ Smalltalk interpreter. The power comes at a price: if you make a mistake redefining `SmallInteger`, for example, you could render your interpreter unusable. The dependence of behavior on the current definitions of classes like `SmallInteger` is reflected in the semantics.

The differences between  $\mu$ Smalltalk and  $\mu$ Scheme show up not just in the rules, but in the states that the semantics refers to. The semantics of  $\mu$ Scheme, which you might find worth reviewing at this point, use an initial state of  $\langle e, \rho, \sigma \rangle$ : the syntax being evaluated, the locations of all the variables, and the contents of all the locations. It's hard to imagine anything simpler. An initial state for  $\mu$ Smalltalk has all these components and more (Table 10.27 on the next page):

- Environment  $\xi$  holds the locations of the global variables. It's needed because unlike a  $\mu$ Scheme function, a method is given access to the global variables defined at the time it receives a message, not at the time it is defined.
- Class  $c_{\text{super}}$  tracks the *static* superclass of the method definition within which an expression is evaluated. This class, which is not the same as the superclass of the object that received the message (Exercise 9), is the class at which method search begins when a message is sent to `super`. It is associated with every method and every block, and for all the expressions and blocks of a single method, it remains unchanged.
- Stack frame  $F$  tracks the method activation which `return` should cause to return. Like  $c_{\text{super}}$ , it is associated with every method and every block, and for all the expressions and blocks of a single activation of a single method, it remains unchanged.
- Set  $\mathcal{F}$  records all the stack frames that have ever been used. It is used to ensure that every time a method is activated, the activation is uniquely identified with a new stack frame  $F$ . That device ensures in turn that if a `return` escapes its original activation, any attempt to evaluate that `return` results in a checked run-time error. The frame set  $\mathcal{F}$  is threaded throughout the entire computation, in much the same way as the store  $\sigma$ .

These additional components are needed to specify message send, sending to `super`, and `return`. Each individual component is used in a straightforward way, but the combination can be intimidating. No wonder most theorists prefer to work with functional languages!

The seven components listed in Table 10.27 form the initial state of an abstract machine for evaluating  $\mu$ Smalltalk:  $\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle$ . If expression  $e$  is evaluated successfully, the machine transitions, expressing one of two *behaviors*:

- If an expression terminates normally and produces a value  $v$ , this behavior is represented by a judgment of the form  $\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle$ . As usual for a language with imperative features, evaluating  $e$  can change the values of variables, so evaluation results in a new store  $\sigma'$ . And evaluation

Metavariable	What it stands for	
$e$	Expression	Expression being evaluated
$\rho$	Environment	Instance variables, arguments, and local variables
$c_{\text{super}}$	Class	Destination for messages to super
$F$	Stack frame	Method activation that is terminated by <code>return</code>
$\xi$	Environment	Global variables
$\sigma$	Store	Current values of all variables
$\mathcal{F}$	Frame set	Every activation of every method ever

§10.9  
*Operational semantics*

687

Table 10.27: Components of an initial state

may also send messages and allocate new stack frames, so evaluation also results in a new used-frame set  $\mathcal{F}'$ . Metavariables  $\sigma'$  and  $\mathcal{F}'$  capture the *effects* of evaluating  $e$ , not its main result, so to make the judgment a little easier to read, I separate them from  $v$  using a semicolon, not a comma.

- If an expression evaluates a `return`, it immediately terminates an activation of the method in which the `return` appears. I say it “returns  $v$  to frame  $F'$ ,” and the behavior is represented by a judgment of the form  $\boxed{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$ , with an arrow pointing up. Again, the main results are separated from the effects by a semicolon.

If a syntactic form contains an expression, its evaluation can end in `return` behavior. But while we are trying to learn the main part of the language, the `return` behaviors are too distracting. For that reason, the semantics of `return` are presented in their own section.

I also introduce new judgment forms for evaluating lists of expressions. Most semantics in this book deal with lists using informal notation with ellipses (“ $\dots$ ”). But in this chapter, to specify the behavior of `return`, we need more precision. The new judgments are

- $\boxed{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma', \mathcal{F}' \rangle}$   
Evaluating a list of expressions produces a list of values  $[v_1, \dots, v_n]$ .
- $\boxed{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$   
Evaluating a list of expressions returns  $v$  to  $F'$ .

The rules for the first judgment are worth giving right away. Formally, a list of expressions  $[e_1, \dots, e_n]$  either is  $[]$  or has the form  $e :: es$ . The result of evaluation either is  $[]$  or has the form  $v :: vs$ .

$$\frac{}{\langle [], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle []; \sigma, \mathcal{F} \rangle} \quad (\text{EMPTYLIST})$$

$$\frac{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\frac{\langle es, \rho, c_{\text{super}}, F, \xi, \sigma', \mathcal{F}' \rangle \Downarrow \langle vs; \sigma'', \mathcal{F}'' \rangle}{\langle e :: es, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v :: vs; \sigma'', \mathcal{F}'' \rangle}} \quad (\text{NONEMPTYLIST})$$

The rules for the second judgment are given with the rules for the other returns.

Finally, I introduce a special judgment form for the evaluation of a primitive. Most primitives are pure functions, but the `value` primitive evaluates a block, so it can do anything an expression can do. Therefore, every primitive gets access to

global variables, and a primitive may affect values of variables and may allocate new stack frames. When a primitive  $p$  is passed values  $v_1, \dots, v_n$ , I express its behavior using the new judgment form  $\boxed{\langle p, [v_1, \dots, v_n], \xi, \sigma, \mathcal{F} \rangle \Downarrow_p \langle v; \sigma', \mathcal{F}' \rangle}$ .

### 10.9.1 Operational semantics of expressions without return

This section presents the rules that apply to situations in which no `return` is evaluated. The semantics of return behavior is presented in Section 10.9.2. For reference, the concrete syntax of expressions is on page 637.

*Variables and assignment* As in Impcore, we have environments  $\rho$  and  $\xi$  for local and global variables. Both environments bind names to mutable locations. Aside from all the extra bookkeeping imposed by messages to `super` and by returns, there is nothing new here.

$$\frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \sigma(\rho(x)); \sigma, \mathcal{F} \rangle} \quad (\text{VAR})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \sigma(\xi(x)); \sigma, \mathcal{F} \rangle} \quad (\text{GLOBALVAR})$$

Assignment translates the name into a location, then changes the value in that location. As in  $\mu$ Scheme, we thread the store. The set of allocated stack frames is threaded in the same way; evaluating the right-hand side transitions that set from  $\mathcal{F}$  to  $\mathcal{F}'$ .

$$\frac{x \in \text{dom } \rho \quad \rho(x) = \ell \quad \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma' \{ \ell \mapsto v \}, \mathcal{F}' \rangle} \quad (\text{ASSIGN})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi \quad \xi(x) = \ell \quad \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma' \{ \ell \mapsto v \}, \mathcal{F}' \rangle} \quad (\text{ASSIGNGLOBAL})$$

Assignment to variables `self`, `super`, `true`, `false`, and `nil` is not permitted, but this restriction is enforced in the parser, so it need not be mentioned here.

*Self and super* In  $\mu$ Smalltalk, `self` is treated as an ordinary variable, so one of the variable rules applies. In most contexts, `super` behaves exactly like `self`.

$$\frac{\langle \text{VAR}(\text{self}), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle}{\langle \text{SUPER}, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle} \quad (\text{SUPER})$$

When a message is sent to `super`, it behaves differently from `self`; accordingly, there are two rules for message sends (page 689).

*Values* As in Impcore, a `VALUE` node evaluates to itself without changing the store.

$$\frac{}{\langle \text{VALUE}(v), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle} \quad (\text{VALUE})$$

*Literals* As noted above, a literal's behavior depends on the current definitions of `SmallInteger`, `Symbol`, and `Array`. A literal evaluates to a value. I give formal semantics only for integer and symbol literals.

$$\frac{}{\langle \text{LITERAL}(\text{NUM}(n)), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle \sigma(\xi(\text{SmallInteger})), \text{NUM}(n) \rangle; \sigma, \mathcal{F} \rangle} \quad (\text{LITERALNUMBER})$$

$$\frac{\langle \text{LITERAL}(\text{SYM}(s)), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle \sigma(\xi(\text{Symbol})), \text{SYM}(s) \rangle; \sigma, \mathcal{F} \rangle}{(\text{LITERALSMBOL})}$$

The class of a literal number or a literal symbol is taken from the current global environment  $\xi$ , which means its behavior can be changed by changing class `SmallInteger` or `Symbol`.

*Blocks* A block is much like a lambda abstraction, except that the body  $es$  is a sequence of expressions, not a single expression. The block creates a closure, which captures the current environment  $\rho$ , the static superclass  $c_{\text{super}}$ , and the current stack frame  $F$ . If the block is sent somewhere else and is evaluated inside another method, as in the `isEmpty` method on class `Collection` (chunk 666c), for example, its return still terminates frame  $F$ .

$$\frac{v = \langle \sigma(\xi(\text{Block})), \text{CLOSURE}(\langle x_1, \dots, x_n \rangle, es, \rho, c_{\text{super}}, F) \rangle}{\langle \text{BLOCK}(\langle x_1, \dots, x_n \rangle, es), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle} \quad (\text{MKCLOSURE})$$

*Sequential execution* BEGIN expressions are as in  $\mu$ Scheme: evaluate the expressions in sequence and take the last value.

$$\frac{\langle \text{BEGIN}(), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \text{nil}; \sigma, \mathcal{F} \rangle}{(\text{EMPTYBEGIN})}$$

$$\frac{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma', \mathcal{F}' \rangle}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v_n; \sigma', \mathcal{F}' \rangle} \quad (\text{BEGIN})$$

*Message send* A message send takes place in four stages: evaluate the receiver and the arguments, find the method to dispatch to, set up a new environment and frame, and evaluate the method's body. To express the method-dispatch algorithm, I introduce the judgment form

$$m \triangleright c @ imp$$

which should be pronounced “sending  $m$  to  $c$  is answered by  $imp$ .” The judgment means that sending a message  $m$  to an object of class  $c$  dispatches to the implementation  $imp$ . The idea is that the judgment  $m \triangleright c @ imp$  is provable if and only if  $imp$  is the *first* method named  $m$  defined either on class  $c$  or on one of  $c$ 's superclasses. The formal semantics is left as Exercise 40.

An ordinary message send tries  $m \triangleright c @ imp$  on the class of the receiver:

$$\frac{\begin{array}{c} e \neq \text{SUPER} \\ \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle \\ \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma_n, \mathcal{F}_n \rangle \\ m \triangleright c @ \text{METHOD}(-, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s) \\ \hat{F} \notin \mathcal{F}_n \\ \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \quad \ell'_1, \dots, \ell'_k \notin \text{dom } \sigma_n \\ \ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_k \text{ all distinct} \\ \rho_i = \text{instanceVars}(\langle c, r \rangle) \\ \rho_a = \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\ \rho_l = \{y_1 \mapsto \ell'_1, \dots, y_k \mapsto \ell'_k\} \\ \hat{\sigma} = \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n, \ell'_1 \mapsto \text{nil}, \dots, \ell'_k \mapsto \text{nil} \} \\ \langle e_m, \rho_a + \rho_l, \hat{\sigma}, \hat{F}, \xi, \hat{\sigma}, \mathcal{F}_n \cup \{\hat{F}\} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \\ \end{array}}{\langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle} \quad (\text{SEND})$$

The premise on the first line shows that this is a rule for an ordinary send, not a send to SUPER. The rest of the rule has much in common with the closure rule for  $\mu$ Scheme:

- The premises on the next two lines show the evaluation of the receiver  $e$  and of the arguments  $e_1, \dots, e_n$ . After these evaluations, we know we are sending message  $m$  to receiver  $r$  of class  $c$ , with actual parameters  $v_1, \dots, v_n$ . The method is evaluated with store  $\sigma_n$ .
- The premise  $m \triangleright c @ \text{METHOD}(\_, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s)$  shows that this send executes a method with formal parameters  $x_1, \dots, x_n$ , local variables  $y_1, \dots, y_k$ , body  $e_m$ , and static superclass  $s$ .
- The next three lines show the allocation of a fresh stack frame and of fresh locations to hold the message arguments and the local variables of the method.
- The equations for  $\rho_i$ ,  $\rho_a$ , and  $\rho_l$  create environments for the receiver's instance variables, the method's formal parameters, and the method's local variables, respectively.
- The equation for  $\hat{\sigma}$  initializes the formal parameters and the local variables.
- Finally, the last premise shows the evaluation of the body of the method  $e_m$ , in the new environment created by combining environments for instance variables, actual parameters, local variables, and returning to stack frame  $\hat{F}$ .

I do not specify `instanceVars` formally. Calling `instanceVars(⟨c, r⟩)`, which is defined in chunk 698a, takes the representation  $r$  of an object and returns an environment mapping the names of that object's instance variables to the locations containing those instance variables, and also mapping the name `self` to a location containing the object itself.

If a message is sent to SUPER, the action is almost the same, except the method search takes place on  $c_{\text{super}}$ , the static superclass of the current method, not on the class of the receiver. I show the new parts in black and the repeated parts in gray.

$$\begin{aligned}
 & \langle \text{SUPER}, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle \\
 & \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma_n, \mathcal{F}_n \rangle \\
 & m \triangleright c_{\text{super}} @ \text{METHOD}(\_, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s) \\
 & \quad \hat{F} \notin \mathcal{F} \\
 & \quad \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \quad \ell'_1, \dots, \ell'_k \notin \text{dom } \sigma_n \\
 & \quad \ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_k \text{ all distinct} \\
 & \quad \rho_i = \text{instanceVars}(\langle c, r \rangle) \\
 & \quad \rho_a = \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\
 & \quad \rho_l = \{y_1 \mapsto \ell'_1, \dots, y_k \mapsto \ell'_k\} \\
 & \hat{\sigma} = \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n, \ell'_1 \mapsto \text{nil}, \dots, \ell'_k \mapsto \text{nil} \} \\
 & \quad \langle e_m, \rho_i + \rho_a + \rho_l, s, \hat{F}, \xi, \hat{\sigma}, \mathcal{F} \cup \{\hat{F}\} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \\
 & \quad \langle \text{SEND}(m, \text{SUPER}, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \quad (\text{SENDSUPER})
 \end{aligned}$$

*Primitives* When a PRIMITIVE expression is evaluated, it evaluates its arguments, then passes them to the primitive named by  $p$ .

$$\begin{aligned}
 & \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma', \mathcal{F}' \rangle \\
 & \quad \langle p, [v_1, \dots, v_n], \xi, \sigma', \mathcal{F}' \rangle \Downarrow_p \langle v; \sigma'', \mathcal{F}'' \rangle \\
 & \quad \langle \text{PRIMITIVE}(p, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma'', \mathcal{F}'' \rangle \quad (\text{PRIMITIVE})
 \end{aligned}$$

$$\boxed{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}{\langle e :: es, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\begin{array}{c} \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \\ \langle es, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma'', \mathcal{F}'' \rangle \end{array}}{\langle e :: es, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma'', \mathcal{F}'' \rangle}$$

§10.9  
Operational  
semantics

691

$$\boxed{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{RETURN}(e), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}{\langle \text{BEGIN}(e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}{\langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle}{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

$$\frac{}{\langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle}$$

Figure 10.28: Rules for propagation of returns (other RETURN rules appear in the text)

Each primitive  $p$  has its own rule. The most interesting one is the value primitive, which evaluates a block. Its rule resembles the rule for sending a message, except unlike a method, a block has no local variables or instance variables of its own. And the body of a block is evaluated using its stored return frame  $F_c$ , not the frame of the calling context.

$$\frac{\begin{array}{c} \ell_1, \dots, \ell_n \notin \text{dom } \sigma \quad \ell_1, \dots, \ell_n \text{ all distinct} \\ \hat{\sigma} = \sigma \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n \} \end{array}}{\langle \text{BEGIN}(es), \rho_c + \{ x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n \}, s_c, F_c, \xi, \hat{\sigma}, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}$$

$$\frac{\langle \text{value}, [\langle \langle c, \text{CLOSURE}(\langle x_1, \dots, x_n \rangle, es, s_c, \rho_c, F_c) \rangle, v_1, \dots, v_n], \xi, \sigma, \mathcal{F} \rangle \Downarrow_p \langle v; \sigma', \mathcal{F}' \rangle}{(\text{VALUEPRIMITIVE})}$$

### 10.9.2 Operational semantics of returns

If subexpressions are being evaluated, and if any subexpression returns, the containing expression also returns. This statement, which is expressed by the rules in Figure 10.28, covers the vast majority of rules for returns, but it doesn't address the purpose of a return, which is to terminate a given method. If a method body

executing in frame  $\hat{F}$  returns to that frame, the result of the return becomes the result of the SEND that activated the method. As with messages to super, the gray parts of the rule are the same as in SEND, and the black parts are different.

$$\begin{array}{c}
 \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle \\
 \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma_n, \mathcal{F}_n \rangle \\
 m \triangleright c @ \text{METHOD}(\_, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s) \\
 \quad \quad \quad \hat{F} \notin \mathcal{F} \\
 \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \quad \ell'_1, \dots, \ell'_k \notin \text{dom } \sigma_n \\
 \ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_k \text{ all distinct} \\
 \rho_i = \text{instanceVars}(\langle c, r \rangle) \\
 \rho_a = \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\
 \rho_l = \{y_1 \mapsto \ell'_1, \dots, y_k \mapsto \ell'_k\} \\
 \hat{\sigma} = \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n, \ell'_1 \mapsto \text{nil}, \dots, \ell'_k \mapsto \text{nil} \} \\
 \frac{\langle e_m, \rho_i + \rho_a + \rho_l, s, \hat{F}, \xi, \hat{\sigma}, \mathcal{F} \cup \{\hat{F}\} \rangle \uparrow \langle v, \hat{F}; \sigma', \mathcal{F}' \rangle}{\langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle} \quad (\text{RETURNTO})
 \end{array}$$

If method body  $e_m$  tries to return somewhere else, to  $F'$ , the whole SEND operation returns to  $F'$ .

$$\begin{array}{c}
 \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle \\
 \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma_n, \mathcal{F}_n \rangle \\
 m \triangleright c @ \text{METHOD}(\_, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s) \\
 \quad \quad \quad \hat{F} \notin \mathcal{F} \\
 \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \quad \ell'_1, \dots, \ell'_k \notin \text{dom } \sigma_n \\
 \ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_k \text{ all distinct} \\
 \rho_i = \text{instanceVars}(\langle c, r \rangle) \\
 \rho_a = \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\
 \rho_l = \{y_1 \mapsto \ell'_1, \dots, y_k \mapsto \ell'_k\} \\
 \hat{\sigma} = \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n, \ell'_1 \mapsto \text{nil}, \dots, \ell'_k \mapsto \text{nil} \} \\
 \frac{\langle e_m, \rho_i + \rho_a + \rho_l, s, \hat{F}, \xi, \hat{\sigma}, \mathcal{F} \cup \{\hat{F}\} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle \quad F' \neq \hat{F}}{\langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle} \quad (\text{RETURNPAST})
 \end{array}$$

### 10.9.3 Operational semantics of definitions

The top-level, persistent state of a  $\mu$ Smalltalk machine has three of the seven components listed in Table 10.27: a global environment  $\xi$ , a store  $\sigma$ , and a set of used stack frames  $\mathcal{F}$ . Evaluating a definition  $d$  may change all three; I use the judgment  $\langle d, \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle$ .

*Global variables* As in  $\mu$ Scheme, a VAL binding for an existing variable assigns to that variable's location. The right-hand side is evaluated using the judgment form  $\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle$ . This form requires additional context, including an environment  $\rho$  and a putative superclass and stack frame. I use the empty environment, and as the superclass, I use the root class Object. We use the original definition of Object taken from the initial global environment  $\xi_0$ , not whatever definition of Object happens to be current. (In practice, the identity of the superclass is irrelevant. If a message is sent to super, the abstract machine tries to look up self in the empty environment, and it gets stuck.) The frame is one not previously allocated.

$$\frac{x \in \text{dom } \xi \quad \xi(x) = \ell \quad \hat{F} \notin \mathcal{F} \\ \langle e, \{\}, \xi_0(\text{Object}), \hat{F}, \xi, \sigma, \{\hat{F}\} \cup \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{VAL}(x, e), \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi, \sigma' \{ \ell \mapsto v \}, \mathcal{F}' \rangle} \quad (\text{DEFINEOLDGLOBAL})$$

$$\frac{x \notin \text{dom } \xi \quad \ell \notin \text{dom } \sigma \quad \hat{F} \notin \mathcal{F} \\ \langle e, \{\}, \xi_0(\text{Object}), \hat{F}, \xi, \sigma, \{\hat{F}\} \cup \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{VAL}(x, e), \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi \{ x \mapsto \ell \}, \sigma' \{ \ell \mapsto v \}, \mathcal{F}' \rangle} \quad (\text{DEFINENEWGLOBAL})$$

§10.10  
The interpreter  
693

*Top-level expressions* A top-level expression is syntactic sugar for a binding to it.

$$\frac{\langle \text{VAL(it, e)}, \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle}{\langle \text{EXP}(e), \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle} \quad (\text{EVALEXP})$$

*Block definition* DEFINE is syntactic sugar for creating a block.

$$\frac{\langle \text{VAL}(f, \text{BLOCK}(\langle x_1, \dots, x_n \rangle, e)), \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle \\ \langle \text{DEFINE}(f, \langle x_1, \dots, x_n \rangle, e), \xi, \sigma, \mathcal{F} \rangle \rightarrow \langle \xi', \sigma', \mathcal{F}' \rangle}{\langle \text{CLASSD}(d, \xi, \sigma, \mathcal{F}) \rightarrow \langle \xi, \sigma \{ \ell \mapsto v \}, \mathcal{F} \rangle} \quad (\text{DEFINEBLOCK})$$

*Class definition* The processing of a class definition is complex; the interpreter creates an object to represent the class. The details are hidden in the function newClassObject, which I don't specify formally, but you can consult the code in chunk 703a.

$$\frac{x \in \text{dom } \xi \quad \xi(x) = \ell \\ v = \text{newClassObject}(d, \xi, \sigma)}{\langle \text{CLASSD}(d, \xi, \sigma, \mathcal{F}) \rightarrow \langle \xi, \sigma \{ \ell \mapsto v \}, \mathcal{F} \rangle} \quad (\text{DEFINEOLDCLASS})$$

$$\frac{x \notin \text{dom } \xi \quad \ell \notin \text{dom } \sigma \\ v = \text{newClassObject}(d, \xi, \sigma)}{\langle \text{CLASSD}(d, \xi, \sigma, \mathcal{F}) \rightarrow \langle \xi \{ x \mapsto \ell \}, \sigma \{ \ell \mapsto v \}, \mathcal{F} \rangle} \quad (\text{DEFINENEWCLASS})$$

## 10.10 THE INTERPRETER

$\mu$ Smalltalk's interpreter is big, and here I present only a few key parts: evaluation, metaclasses, literals, and primitives. All these parts are influenced by decisions about representation, of which the most important is the representation of objects.

Although we usually use the word “object,” the ML type of the thing that eval returns is value, just as in every other interpreter in this book. An object is represented in two parts: its class, which determines its response to messages, and its internal representation, which has ML type rep.

type class	694c
type rep	694a

693.  $\langle$ definitions of value and method for  $\mu$ Smalltalk 693 $\rangle \equiv$   
withtype value = class \* rep

(S547a) 694d ▷

The rep part of an object exposes one of the biggest differences between  $\mu$ Smalltalk and Smalltalk-80. In Smalltalk-80, every object owns a collection of mutable locations, called “instance variables,” each of which can be filled either with an ordinary object or with a sequence of bytes. But because  $\mu$ Smalltalk is implemented in ML, it doesn’t work with byte sequences; an object’s representation is defined by an ML datatype. In  $\mu$ Smalltalk, that datatype can identify an ordi-

nary, user-defined object, which is represented by a collection of named locations. Or it can identify an array, a number, a symbol, a closure, a class object, or a compiled method:

**694a.** *(definitions of exp, rep, and class for  $\mu$ Smalltalk 694a)*  $\equiv$

(S547a) 694c  $\triangleright$

```
datatype rep
  = USER      of value ref env (* ordinary object *)
  | ARRAY     of value Array.array
  | NUM       of int
  | SYM       of name
  | CLOSURE   of name list * exp list * value ref env * class * frame
  | CLASSREP  of class
  | METHODDV  of method
```

Smalltalk and  
object-orientation  
10  
694

Each representation, except for a collection of named locations, can be created by evaluating some particular syntactic form in the source code: an array literal, a numeric literal, a literal symbol, a block, a class definition, or a method definition. Most representations (arrays, numbers, classes) can also be created by primitives.

A regrettable consequence of this design is that an object with a primitive representation has no instance variables besides `self`. By contrast, an object with a `USER` representation has all the instance variables dictated by its class's definition—always including an instance variable `self`, which is inherited from class `Object`.

**694b.** *(utility functions on  $\mu$ Smalltalk classes, methods, and values 694b)*  $\equiv$

(S547c)

```
fun instanceVars (_ , USER rep) = rep instanceVars : value -> value ref env
| instanceVars self = bind ("self", ref self, emptyEnv)
```

Internally, an object's class is represented by an ML value of type `class`. Its internal representation includes a superclass, instance-variable names, and methods. Except for the distinguished root class, `Object`, every class has a superclass. A class's `ivars` and `methods` lists include only the instance variables and methods defined in that class, not those of its superclass.

**694c.** *(definitions of exp, rep, and class for  $\mu$ Smalltalk 694a)*  $\doteqdot$

(S547a)  $\triangleleft$  694a 696a  $\triangleright$

```
and class
  = CLASS of { name    : name          (* name of the class *)
             , super    : class option    (* superclass, if any *)
             , ivars   : string list    (* instance variables *)
             , methods : method env ref (* both exported and private *)
             , class    : metaclass ref (* class of the class object *)
            }
and metaclass = META of class | PENDING
```

Every class is also an object, and as object, it is an instance of another class—its metaclass, which is also stored with the class. The location stored in the `class` field initially holds `PENDING`, and when the metaclass becomes available, `class` is updated.

A method has a name, formal parameters, local variables, and a body. A method also stores the superclass of the class in which it is defined, which it uses to interpret messages sent to `super`.

**694d.** *(definitions of value and method for  $\mu$ Smalltalk 693)*  $\doteqdot$

(S547a)  $\triangleleft$  693

```
and method = { name : name, formals : name list, locals : name list
              , body : exp, superclass : class
            }
```

These fundamental representations inform the representations of the elements of the abstract-machine state, as shown in Table 10.27 on page 687.

- An expression  $e$  or definition  $d$  is represented in the usual way, by a constructed value from an algebraic data type. The types are defined below.

- An environment  $\rho$  or  $\xi$  is represented, as usual, by an ML environment of type `value ref env`.
- A superclass  $c_{\text{super}}$  is represented by an ML value of type `class`.
- A stack frame  $F$  is represented by an ML value of type `frame`. We don't need to know exactly how a `frame` is represented; it's enough to know that a new frame can be allocated by calling `newFrame`, and that a frame is equal only to itself.
- The store  $\sigma$  and the set of used frames  $\mathcal{F}$  are both represented by mutable state of the ML program. Just as in the other interpreters,  $\sigma$  and  $\sigma'$  never coexist; instead, the interpreter updates its state, in effect replacing  $\sigma$  by  $\sigma'$ . The set  $\mathcal{F}$  is updated to  $\mathcal{F}'$  in the same way.

§10.10  
*The interpreter*  
695

The other key representation is the representation of behaviors.

- The behavior of producing a value, which is described by judgment form  $\langle e, \dots \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle$ , is represented by an ML computation that produces a value  $v$  of ML type `value`, while writing  $\sigma'$  and  $\mathcal{F}'$  over the previous  $\sigma$  and  $\mathcal{F}$  as a side effect.
- The behavior of a  $\mu$ Smalltalk `return`, which is described by judgment form  $\langle e, \dots \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle$ , is represented by an ML computation that raises the `ML Return` exception, again writing  $\sigma'$  and  $\mathcal{F}'$  as a side effect. The `Return` exception is defined here:

**695a.** *(definition of the Return exception 695a)*≡  
 exception  
   Return of { value : value, to : frame, unwound : active\_send list }

Fields `value` and `to` hold  $v$  and  $F$ . And in the unhappy event that a block tries to return after its frame has died, field `unwound` is used to print diagnostics.

I use an exception because raising an ML exception interrupts computation in exactly the same way as the propagated returns described in Figure 10.28 on page 691. That's not a coincidence; both exceptions and returns are language features that are designed to interrupt planned computations.

### 10.10.1 Abstract syntax

The syntax of definitions doesn't require any long explanations, so I begin with `def`. A definition may be one of our old friends `VAL` and `EXP`, a block definition (`DEFINE`), or a class definition (`CLASSD`). A class definition may include both instance methods and class methods.

**695b.** *(definition of def for  $\mu$ Smalltalk 695b)*≡  
 datatype def = VAL of name \* exp  
   | EXP of exp  
   | DEFINE of name \* name list \* exp  
   | CLASSD of { name : string  
     , super : string  
     , ivars : string list  
     , methods : method\_def list  
   }  
 and method\_flavor = IMETHOD (\* instance method \*)  
   | CMETHOD (\* class method \*)  
 withtype method\_def = { flavor : method\_flavor, name : name  
   , formals : name list, locals : name list, body : exp  
 }

type active_send	S551b
bind	312b
emptyEnv	311a
type env	310b
type exp	696a
type frame	S551b
type name	310a
type value	693

The syntax of expressions is a little more complicated. Most expressions are formed in the usual way; the abstract-syntax constructors VAR, SET, SEND, BEGIN, and BLOCK resemble the kinds of abstract syntax you see in other interpreters. (Although the concrete syntax of blocks has two forms—the general block and the special-purpose curly braces—the abstract syntax needs just the general one.) Forms RETURN, PRIMITIVE, and METHOD have no analogs in other interpreters, but they are also typical abstract syntax. And SUPER simply makes it easy to recognize super and give it the semantics it should have. But literal values are handled differently than in other interpreters.

A literal must ultimately stand for an object, whose representation has type value. That representation includes a class, but until the interpreter is bootstrapped, most classes aren't yet defined (Section 10.10.6). For example, until class Integer is defined, we can't make the class of a literal integer. To solve this problem, I store only a rep in a LITERAL expression. The class isn't computed until the LITERAL expression is evaluated.

The LITERAL form alone isn't enough: there are places in the interpreter—for example, the read-eval-print loop—where I must send a message to an object, which has both class and rep. To form an expression from such an object, I use VALUE.

**696a.** *(definitions of exp, rep, and class for  $\mu$ Smalltalk 694a)*  $\equiv$  (S547a)  $\triangleleft$  694c

```
and exp = VAR          of name
| SET      of name * exp
| SEND     of srcloc * exp * name * exp list
| BEGIN    of exp list
| BLOCK    of name list * exp list
| RETURN   of exp
| PRIMITIVE of name * exp list
| METHOD   of name list * name list * exp list
| SUPER
| LITERAL  of rep
| VALUE    of class * rep
```

One more part of this representation is not found in the concrete syntax: the srcloc field in the SEND node is a source-code location, which is used in diagnostic messages.

#### 10.10.2 Evaluating expressions, including dynamic dispatch

Just as in the operational semantics, a  $\mu$ Smalltalk expression is evaluated in a context that tells it about environments  $\rho$  and  $\xi$ ,<sup>14</sup> a static superclass to which messages to super are sent, and a frame which is terminated by return. The states  $\sigma$  and  $\sigma'$  represent states of the underlying ML interpreter, and the used-frame set  $\mathcal{F}$  is stored in a mutable variable, so they are not passed explicitly.

**696b.** *(evaluation, basis, and processDef for  $\mu$ Smalltalk 696b)*  $\equiv$  (S559a) 699c  $\triangleright$

<pre>eval: exp * value ref env * class * frame * value ref env -&gt; value ev : exp -&gt; value</pre>
<pre>fun eval (e, rho, superclass, F, xi) =   let &lt;i&gt;(definition of function evalMethod 698a)&lt;/i&gt;     &lt;i&gt;(function ev, the evaluator proper 697a)&lt;/i&gt;   in ev e   end</pre>

Internal function ev handles all the syntactic forms. I begin with the most interesting forms: RETURN and SEND.

<sup>14</sup>In the code, the Greek letter  $\xi$ , pronounced “ksee,” is spelled “xi.”

*Evaluating returns and sends* To interpret RETURN, we evaluate the expression to be returned, then return its value to frame F, by raising the Return exception.

**697a.** *(function ev, the evaluator proper 697a)  $\equiv$*  (696b) 697b  $\triangleright$   
 $\text{fun ev (RETURN e) = raise Return \{ value = ev e, to = F, unwound = [] \}}$

That Return exception is caught by the code that interprets message send. This code carries a lot of freight, implementing most of rules SEND, SENDSUPER, RETURNTO, and RETURNPAST. All these rules follow the same outline, so here, for reference, is SEND, with the first part highlighted:

§10.10  
The interpreter

697

$$\begin{aligned}
 & e \neq \text{SUPER} \\
 & \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle \\
 & \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma_n, \mathcal{F}_n \rangle \\
 & m \triangleright c @ \text{METHOD}(\_, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s) \\
 & \quad \hat{F} \notin \mathcal{F}_n \\
 & \quad \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \quad \ell'_1, \dots, \ell'_k \notin \text{dom } \sigma_n \\
 & \quad \ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_k \text{ all distinct} \\
 & \quad \rho_i = \text{instanceVars}(\langle c, r \rangle) \\
 & \quad \rho_a = \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\
 & \quad \rho_l = \{y_1 \mapsto \ell'_1, \dots, y_k \mapsto \ell'_k\} \\
 & \quad \hat{\sigma} = \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n, \ell'_1 \mapsto \text{nil}, \dots, \ell'_k \mapsto \text{nil} \} \\
 & \quad \langle e_m, \rho_i + \rho_a + \rho_l, s, \hat{F}, \xi, \hat{\sigma}, \mathcal{F}_n \cup \{\hat{F}\} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \\
 & \quad \langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \quad (\text{SEND})
 \end{aligned}$$

Each SEND rule begins in the same way: evaluate the receiver and the arguments using ev, then use the syntax of the receiver to identify the class on which method search begins. Message send dispatches on the receiver, which is used to find the method that defines message, *except* when the message is sent to super, in which case we use the superclass of the currently running method. At that point, because of tracing and returns, things start to get complicated, so let's look at the code, then focus on the function passed to trace:

**697b.** *(function ev, the evaluator proper 697a)  $\vdash$*  (696b)  $\triangleleft$  697a 699a  $\triangleright$

```

| ev (SEND (srcloc, receiver, msgname, args)) =
  let val obj as (class, rep) = ev receiver
  val vs = map ev args
  val startingClass =
    case receiver of SUPER => superclass | _ => class
    (definition of function trace S558a)
  in trace
    (fn () =>
      let val imp = findMethod (msgname, startingClass)
      val Fhat = newFrame ()
      in evalMethod (imp, obj, vs, Fhat)
      handle Return { value = v, to = F', unwound = unwound } =>
        if F' = Fhat then
          v
        else
          (reraise Return, adding msgname, class, and loc to unwound S551c)
    end)
  end

```

applyChecking-	
Overflow	S242b
checkpointLimit	S242b
type class	694c
evalMethod	698a
findMethod	698b
id	S263d
logging	S548a
logSend	S548a
type name	310a
newFrame	S551b
type rep	694a
restoreLimits	S242b
Return	695a
trace	S558a

The anonymous function passed to trace calls findMethod to implement  $m \triangleright c @ \text{imp}$ , and it allocates  $\hat{F}$  as Fhat. It then delegates the second part of the SEND rules to helper function evalMethod, except for the conditions surrounding RETURNTO and RETURNPAST. Those conditions are dealt with by code that catches every Return

exception. If the Return is meant to terminate this very SEND—that is, if  $F' = \hat{F}$ —then the ML code returns  $v$  as the result of the call to ev. If it's not, the ML code re-raises the exception, adding information to the unwind list.

What about function trace? It wraps the action of the anonymous function, so if findMethod results in a “message not understood” error, or if evalMethod results in some other run-time error, it can produce a stack trace. Function trace is defined in Appendix U.

The second part of the SEND rule is implemented by function evalMethod.

$$\begin{array}{c}
 e \neq \text{SUPER} \\
 \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle c, r \rangle; \sigma_0, \mathcal{F}_0 \rangle \\
 \langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma_0, \mathcal{F}_0 \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma_n, \mathcal{F}_n \rangle \\
 m \triangleright c @ \text{METHOD}(\_, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_k \rangle, e_m, s) \\
 \hat{F} \notin \mathcal{F}_n \\
 \ell_1, \dots, \ell_n \notin \text{dom } \sigma_n \quad \ell'_1, \dots, \ell'_k \notin \text{dom } \sigma_n \\
 \ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_k \text{ all distinct} \\
 \rho_i = \text{instanceVars}(\langle c, r \rangle) \\
 \rho_a = \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\} \\
 \rho_l = \{y_1 \mapsto \ell'_1, \dots, y_k \mapsto \ell'_k\} \\
 \hat{\sigma} = \sigma_n \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n, \ell'_1 \mapsto \text{nil}, \dots, \ell'_k \mapsto \text{nil} \} \\
 \langle e_m, \rho_i + \rho_a + \rho_l, s, \hat{F}, \xi, \hat{\sigma}, \mathcal{F}_n \cup \{\hat{F}\} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \\
 \hline
 \langle \text{SEND}(m, e, e_1, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle \quad (\text{SEND})
 \end{array}$$

Function evalMethod computes  $\rho_i$  as ivars,  $\rho_a$  as args, and  $\rho_l$  as locals. It also allocates and initializes locations  $\ell_1, \dots, \ell_n$  and  $\ell'_1, \dots, \ell'_k$ , then calls eval.

698a. ⟨definition of function evalMethod 698a⟩ ≡

(696b)

evalMethod : method * value * value list * frame -> value
---

```

fun evalMethod ({ name, superclass, formals, locals, body },
               receiver, vs, Fhat) =
  let val ivars = instanceVars receiver
      val args = mkEnv (formals, map ref vs)
      val locals = mkEnv (locals, map (fn _ => ref nilValue) locals)
  in eval (body, ivars <+> args <+> locals, superclass, Fhat, xi)
  end

```

All that remains is method search. If  $m \triangleright c @ imp$ , then calling findMethod ( $m, c$ ) returns  $imp$ . If there is no  $imp$  such that  $m \triangleright c @ imp$ , then the same call raises the RuntimeError exception.

698b. ⟨helper functions for evaluation 698b⟩ ≡

(S559a)

fun findMethod (name, class) =	findMethod : name * class -> method
	fm : class -> method

```

let fun fm (subclass as CLASS { methods, super, ... }) =
  find (name, !methods)
  handle NotFound m =>
    case super
    of SOME c => fm c
    | NONE   =>
        raise RuntimeError (className class ^
                             " does not understand message " ^ m)
in fm class
end

```

*Allocating and evaluating blocks* When a BLOCK form is evaluated, it captures the current environment, superclass, and stack frame in a closure.

$$v = \langle \sigma(\xi(\text{Block})), \text{CLOSURE}(\langle x_1, \dots, x_n \rangle, es, \rho, c_{\text{super}}, F) \rangle \quad (\text{MKCLOSURE})$$

$$\langle \text{BLOCK}(\langle x_1, \dots, x_n \rangle, es), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle$$

699a. *(function ev, the evaluator proper 697a)* +≡ (696b) ▷ 697b 699d ▷  
| ev (BLOCK (formals, body)) = mkBlock (formals, body, rho, superclass, F)

Unlike Scheme, Smalltalk does not have a syntactic form that evaluates code inside a closure. Instead, Smalltalk uses the value primitive. The value primitive is the only primitive that is mutually recursive with eval; it uses the function stored in applyClosureRef.

\$10.10  
The interpreter  
699

699b. *(ML code for remaining classes' primitives 699b)* ≡ (S548b) 707a ▷

```
type closure = name list * exp list * value ref env * class * frame
val applyClosureRef : (closure * value list * value ref env -> value) ref
= ref (fn _ => raise InternalError "applyClosureRef not set")
```

```
fun valuePrim ((_, CLOSURE clo) :: vs, xi) = !applyClosureRef (clo, vs, xi)
| valuePrim _ = raise RuntimeError "primitive 'value' needs a closure"
```

Once eval is defined, applyClosureRef can be initialized properly, to a function that implements this rule:

$$\frac{\ell_1, \dots, \ell_n \notin \text{dom } \sigma \quad \ell_1, \dots, \ell_n \text{ all distinct}}{\hat{\sigma} = \sigma \{ \ell_1 \mapsto v_1, \dots, \ell_n \mapsto v_n \}}$$

$$\frac{\langle \text{BEGIN}(es), \rho_c + \{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}, s_c, F_c, \xi, \hat{\sigma}, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{value}, [\langle c, \text{CLOSURE}(\langle x_1, \dots, x_n \rangle, es, s_c, \rho_c, F_c) \rangle, v_1, \dots, v_n], \xi, \sigma, \mathcal{F} \rangle \Downarrow_p \langle v; \sigma', \mathcal{F}' \rangle}$$

(VALUEPRIMITIVE)

699c. *(evaluation, basis, and processDef for μSmalltalk 696b)* +≡ (S559a) ▷ 696b 701c ▷  
| applyClosure : closure \* value list \* value ref env -> value

```
fun applyClosure ((formals, body, rho, superclass, frame), vs, xi) =
eval (BEGIN body, bindList (formals, map ref vs, rho), superclass,
      frame, xi)
handle BindListLength =>
raise RuntimeError ("wrong number of arguments to block; expected " ^
                     "(<block> " ^ valueSelector formals ^ " " ^ spaceSep formals ^ ")")
val () = applyClosureRef := applyClosure
```

*Evaluating value and literal nodes* A VALUE node stands for itself.

$$\langle \text{VALUE}(v), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle \quad (\text{VALUE})$$

699d. *(function ev, the evaluator proper 697a)* +≡ (696b) ▷ 699a 700a ▷  
| ev (VALUE v) = v

Most VALUE nodes are created internally, e.g., by the read-eval-print loop when it wants to send print to an object. Literal numbers and symbols are represented by LITERAL nodes. When we see one, we call mkInteger or mkSymbol to build the literal. It is unsafe to call these functions until we have read the initial basis and bootstrapped the interpreter (Section 10.10.6); integer or symbol literals in the initial basis had better appear only inside method definitions. Evaluating such a literal

<+>	312d
BEGIN	696a
bindList	312c
BindListLength	
	312c
BLOCK	696a
CLASS	694c
type class	694c
className	S549d
CLOSURE	694a
type env	310b
eval	696b
type exp	696a
find	311b
type frame	S551b
instanceVars	694b
InternalError	
	S366f
logFind	S548a
logging	S548a
mkBlock	S551d
mkEnv	312c
type name	310a
nilValue	704c
NotFound	311b
rho	696b
RuntimeError	S366c
spaceSep	S239a
superclass	696b
VALUE	696a
type value	693
valueSelector	S549c
xi	696b

calls `mkInteger` or `mkSymbol`, and if you revisit chunks 706a and 706b, you will see that it is safe to call `mkInteger` only after the interpreter is fully initialized.

$$\frac{\langle \text{LITERAL}(\text{NUM}(n)), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle \sigma(\xi(\text{SmallInteger})), \text{NUM}(n) \rangle; \sigma, \mathcal{F} \rangle}{(\text{LITERALNUMBER})}$$

Smalltalk and  
object-orientation  
10  
700

$$\frac{\langle \text{LITERAL}(\text{SYM}(s)), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \langle \sigma(\xi(\text{Symbol})), \text{SYM}(s) \rangle; \sigma, \mathcal{F} \rangle}{(\text{LITERALSYMBOL})}$$

**700a.** *(function ev, the evaluator proper 697a)+≡* (696b) ▷ 699d 700b▷

```
| ev (LITERAL c) =
  | case c of NUM n => mkInteger n
  | SYM s => mkSymbol s
  | _ => raise InternalError "unexpected literal")
```

*Reading and writing variables* The cases for VAR and SET are as we would expect, given that we have both local and global environments, just as in Impcore.

$$\frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \sigma(\rho(x)); \sigma, \mathcal{F} \rangle} \quad (\text{VAR})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \sigma(\xi(x)); \sigma, \mathcal{F} \rangle} \quad (\text{GLOBALVAR})$$

$$\frac{x \in \text{dom } \rho \quad \rho(x) = \ell \quad \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma'\{\ell \mapsto v\}, \mathcal{F}' \rangle} \quad (\text{ASSIGN})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi \quad \xi(x) = \ell \quad \langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle}{\langle \text{SET}(x, e), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma'\{\ell \mapsto v\}, \mathcal{F}' \rangle} \quad (\text{ASSIGNGLOBAL})$$

**700b.** *(function ev, the evaluator proper 697a)+≡* (696b) ▷ 700a 700c▷

```
| ev (VAR x) = !(find (x, rho) handle NotFound _ => find (x, xi))
| ev (SET (x, e)) =
  let val v = ev e
  val cell = find (x, rho) handle NotFound _ => find (x, xi)
  in cell := v; v
  end
```

SUPER, when used as an expression, acts just as self does.

$$\frac{\langle \text{VAR}(\text{self}), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle}{\langle \text{SUPER}, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma, \mathcal{F} \rangle} \quad (\text{SUPER})$$

**700c.** *(function ev, the evaluator proper 697a)+≡* (696b) ▷ 700b 700d▷

```
| ev (SUPER) = ev (VAR "self")
```

*Sequential evaluation* Evaluation of BEGIN is as in μScheme.

$$\frac{\langle \text{BEGIN}(), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle \text{nil}; \sigma, \mathcal{F} \rangle}{(\text{EMPTYBEGIN})}$$

$$\frac{\langle [e_1, \dots, e_n], \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle [v_1, \dots, v_n]; \sigma', \mathcal{F}' \rangle}{\langle \text{BEGIN}(e_1, e_2, \dots, e_n), \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v_n; \sigma', \mathcal{F}' \rangle} \quad (\text{BEGIN})$$

**700d.** *(function ev, the evaluator proper 697a)+≡* (696b) ▷ 700c 701a▷

```
| ev (BEGIN es) =
  let fun b (e::es, lastval) = b (es, ev e)
      | b ( [], lastval) = lastval
  in b (es, nilValue)
  end
```

*Evaluating primitives* To evaluate a primitive, find it on an association list, then applying it to its arguments. The value primitive calls `applyClosure`, which needs a global environment, so a global environment is also supplied.

**701a.** *(function ev, the evaluator proper 697a)* +≡ (696b) ▷ 700d 701b ▷

```
| ev (PRIMITIVE (p, args)) =
|   let val f = find (p, primitives)
|     handle NotFound n =>
|       raise RuntimeError ("There is no primitive named " ^ n)
|     in  f (map ev args, xi)
|   end
```

\$10.10

The interpreter

701

The list `primitives` is defined in the Supplement.

*Compiled methods* Evaluating a `compiled-method` involves no actual compilation. We just put the formal parameters, local variables, and body into an object.

**701b.** *(function ev, the evaluator proper 697a)* +≡ (696b) ▷ 701a

```
| ev (METHOD (xs, ys, es)) =
|   mkCompiledMethod { name = "", formals = xs, locals = ys, body = BEGIN es
|                      , superclass = objectClass }
```

### 10.10.3 Evaluating definitions

Most definitions are evaluated more or less as in other interpreters, but class definitions require a lot of special-purpose code for creating classes.

*Function evaldef, for evaluating definitions*

Evaluating a definition computes a new global environment. But in  $\mu$ Smalltalk, it also has a side effect on the state of the interpreter. The implementation is best understood in four steps:

1. Define auxiliary function `ev`, which evaluates an expression `e` as if in the context of a message sent to an instance of class `Object`. That means using an empty  $\rho$ , using `Object` as the superclass, and using the given `xi`.
2. Analyze the definition `d` to find the name `x` being defined and to compute the value `v` that `x` stands for. Depending on the form of the definition, `v` may be the result of evaluating an expression, or it may be a new class object.
3. Compute the new environment  $\xi'$ .
4. Return  $\xi'$  and `v`.

Here's the code:

**701c.** *(evaluation, basis, and processDef for  $\mu$ Smalltalk 696b)* +≡ (S559a) ▷ 699c

evaldef : def * value ref env -> value ref env * value
ev : exp -> value

```
fun evaldef (d, xi) =
  let fun ev e = eval (e, emptyEnv, objectClass, noFrame, xi)
       handle Return { value = v, unwound = frames, ... } =>
           (report return escapes frames S567b)
  in
    val (x, v) =
      case d
        of VAL (name, e)          => (name, ev e)
        | EXP e                  => ("it", ev e)
        | DEFINE (name, args, body) => (name, ev (BLOCK (args, [body])))
        | CLASSD (d as {name, ...}) => (name, newClassObject d xi)
    val xi' = optimizedBind (x, v, xi)
  in (xi', v)
  end
```

BEGIN	696a
BLOCK	696a
CLASSD	695b
DEFINE	695b
emptyEnv	311a
ev	697a
eval	696b
EXP	695b
find	311b
InternalError	S366f
LITERAL	696a
METHOD	696a
mkCompiledMethod	S557a
mkInteger	706a
mkSymbol	706a
newClassObject	703a
nilValue	704c
noFrame	S551b
NotFound	311b
NUM	694a
objectClass	704a
optimizedBind	S549b
PRIMITIVE	696a
primitives	S552a
Return	695a
rho	696b
RuntimeError	S366c
saveLiteralClasses	706b
SET	696a
SUPER	696a
SYM	694a
VAL	695b
VAR	696a
xi	696b

```

mkClass      : name -> metaclass -> class -> name list -> method list -> class
methodDefns : class * class -> method_def list -> method list * method list
findClass   : name * value ref env -> class * class
setMeta     : class * class -> unit
className  : class -> name
classId    : class -> metaclass ref
methodName : method -> name
methodsEnv : method list -> method env

```

Table 10.29: Utilities for manipulating classes (from the Supplement)

The VAL and EXP forms both evaluate the given expression and bind it to a name—either the given name or "it". DEFINE is syntactic sugar for a definition of a block. And evaluating a class definition binds a new class object, as described above.

#### 10.10.4 Class objects and metaclasses

If you look at the source code for a class definition, it looks like there are two kinds of methods: instance methods and class methods. But if you look at the operational semantics, or at the run-time representation of class, there are only “methods.” What’s up with that? Smalltalk has *one* mechanism for dynamic dispatch, regardless of whether a message is sent to a class or to an instance. The mechanism is enabled by an extra, hidden class for each class in the system: a *metaclass*. If all you want is to write Smalltalk code, you can do so happily without understanding metaclasses. But if you want to know how the mechanisms work, you can discover it here.

Here are the key invariants of the system:

- Every object is an instance of some class.
- Every class is also an object—and is therefore an instance of some class.
- A class whose instances are classes is called a *metaclass*.
- Classes and metaclasses are one to one: every class has a unique metaclass, and every metaclass has a unique instance.
- The instance methods of a class are stored in the class object.
- The class methods of a class are stored in the metaclass object.
- Every metaclass is an instance of class *Metaclass*.
- If a class *C* has a superclass, the metaclass of its superclass is the superclass of its metaclass. This invariant can be expressed as the algebraic law  $((C \text{ superclass}) \text{ metaclass}) = ((C \text{ metaclass}) \text{ superclass})$ .
- Class *Object* has no superclass, and the superclass of its metaclass is class *Class*.

These invariants dictate that classes and metaclasses be linked in memory in circular ways: because every class points both to its superclass and to its metaclass, the graph of class and metaclass objects has a cycle. If we consider just the “subclass-of” relation, there are no cycles, but the “instance-of” relation has a cycle, and the combined relation has an additional cycle. To implement the cycles, I create every class object with a PENDING metaclass, then update the metaclasses once the metaclass object is created.

Most classes and metaclasses are created by calling *newClassObject*, which is given a class definition and creates a class object and a metaclass object for that

class. The superclass and its metaclass are found by looking up the named superclass in environment  $\xi$ . Function `methodDefns` segregates the method definitions into class methods and instance methods, and it attaches the correct static superclass to each method. The class is built using `mkClass`, and its metaclass is built using `mkMeta`.

**703a.** *(definition of newClassObject and supporting functions 703a)*≡ (S559a)

```
fun newClassObject {name, super, ivars, methods} xi =
  let val (superMeta, super) = findClass (super, xi)
    handle NotFound s =>
      raise RuntimeError ("Superclass " ^ s ^ " not found")
  val (cmethods, imethods) = methodDefns (superMeta, super) methods
  val class = mkClass name PENDING super ivars imethods
  val () = setMeta (class, mkMeta class cmethods)
in classObject class
end
```

Function `classObject` uses `CLASSREP` to make the class a rep, then pairs it with the class of which it is an instance, that is, its metaclass. Any attempt to refer to an uninitialized metaclass results in a checked run-time error.

**703b.** *(metaclass utilities 703b)*≡ (S548b)

metaclass : class → class
classObject : class → value

```
fun metaclass (CLASS { class = ref meta, ... }) =
  case meta of META c => c
  | PENDING => raise InternalError "pending class"
```

```
fun classObject c = (metaclass c, CLASSREP c)
```

As noted above, a new metaclass inherits from the metaclass of its superclass, unless there is no superclass, in which case it inherits from `Class`. Internal representation `classClass` is defined below.

**703c.** *(metaclasses for built-in classes 703c)*≡ (S548b) 703d

metaSuper : class → class
---------------------------

```
fun metaSuper (CLASS { super = NONE, ... }) = classClass
  | metaSuper (CLASS { super = SOME c, ... }) = metaclass c
```

To make a metaclass, we need to know the class `c` of which the result is the metaclass, and the class `methods`. The metaclass has a name derived from `c`'s name and is an instance of class `Metaclass`. It has the superclass computed by `metaSuper`, no instance variables, and the given `methods`.

**703d.** *(metaclasses for built-in classes 703c)*+≡ (S548b) ▷ 703c 705▷

```
fun mkMeta c methods =
  mkClass ("class " ^ className c) (META metaclassClass) (metaSuper c)
  [] methods
```

CLASS	694c
classClass	704d
className	S549d
CLASSREP	694a
findClass	S551a
InternalError	S366f
META	694c
metaclassClass	704d
methodDefns	S550d
mkClass	S550a
NotFound	311b
PENDING	694c
RuntimeError	S366c
setMeta	S550c

### 10.10.5 The primitive classes

Function `newClassObject`, described immediately above, is what's called when a class definition is evaluated, and it is sufficient to create almost every class in  $\mu$ Smalltalk. Just four classes have to be primitive (that is, defined using ML code):

- `Object` is primitive because it has no superclass.
- `UndefinedObject` is primitive because it is `nil`'s class, and `nil` is primitive because a number of primitives need it, as does the evaluator.

- Class and Metaclass are primitive because they are needed to create new classes: every metaclass descends from class Class and is an instance of class Metaclass.

Every other class in the initial basis is defined by a class definition written in  $\mu$ Smalltalk itself.

Class Object is the ultimate superclass. To support the implementation of self, it has one instance variable, self. Putting the instance variable here ensures that every user-defined object has an instance variable called self, into which, when a new object is created, function newUserObject places a pointer to the object itself (chunk 707b).

**704a.**  $\langle$ built-in class Object 704a $\rangle \equiv$  (S548b)  
 val objectMethods =  
   internalMethods  $\langle$ methods of class Object, as strings (from chunk 644a) $\rangle$   
 val objectClass =  
   CLASS { name = "Object", super = NONE, ivars = ["self"]  
     , class = ref PENDING , methods = ref (methodsEnv objectMethods)  
   }

The  $\langle$ methods of class Object 644a $\rangle$  are defined throughout this chapter and in Appendix U, starting in chunk 644a.

Class UndefinedObject, whose sole instance is nil, redefines isNil, notNil, and print, as shown in chunks 663a and S538a.

**704b.**  $\langle$ built-in class UndefinedObject and value nilValue 704b $\rangle \equiv$  (S548b) 704c▷  
 val nilClass =  
   mkClass "UndefinedObject" PENDING objectClass []  
   (internalMethods  $\langle$ methods of class UndefinedObject, as strings (from chunk 663a) $\rangle$ )

The single instance of UndefinedObject, internally called nilValue, is created here, so it can be returned from some primitives.

**704c.**  $\langle$ built-in class UndefinedObject and value nilValue 704b $\rangle + \equiv$  (S548b) ▷ 704b  
 val nilValue =  
   let val nilCell = ref (nilClass, USER []) : value ref  
     val nilValue = (nilClass, USER (bind ("self", nilCell, emptyEnv)))  
     val \_ = nilCell := nilValue  
   in nilValue  
   end

Class Class is in the interpreter so that metaclasses can inherit from it, and Metaclass is here so that each metaclass can be an instance of it.

**704d.**  $\langle$ built-in classes Class and Metaclass 704d $\rangle \equiv$  (S548b)  
 val classClass =  
   mkClass "Class" PENDING objectClass []  
   (internalMethods  $\langle$ methods of class Class, as strings (from chunk 704e) $\rangle$ )  
  
 val metaclassClass =  
   mkClass "Metaclass" PENDING classClass []  
   (internalMethods  $\langle$ methods of class Metaclass, as strings (from chunk 704f) $\rangle$ )

Most of the methods of class Class are relegated to Appendix U, but here I show the default implementation of new.

**704e.**  $\langle$ methods of class Class 704e $\rangle \equiv$   
 (method new () (primitive newUserObject self))

For metaclasses, this default is overridden; a metaclass may not be used to instantiate new objects.

**704f.**  $\langle$ methods of class Metaclass 704f $\rangle \equiv$   
 (method new () (self error: 'a-metaclass-may-have-only-one-instance'))

Internal classes `classClass` and `metaclassClass` are used in the implementation of `mkMeta`, shown above in chunk 703d. Once `mkMeta` is defined, it can be used to create the metaclasses of classes `Object`, `UndefinedObject`, `Class`, and `Metaclass`.

705. *{metaclasses for built-in classes 703c}* +≡ (S548b) ↳ 703d  
`fun patchMeta c = setMeta (c, mkMeta c [])`  
`val () = app patchMeta [objectClass, nilClass, classClass, metaclassClass]`

### 10.10.6 Bootstrapping for literal and Boolean values

\$10.10

The interpreter

705

In most languages, literal integers, Booleans, and `nil` would be simple atomic values, and we could define them directly. But in Smalltalk, every value is an object, every object has a class, and relations among objects and classes include circular dependencies:

1. When the evaluator sees an integer literal, it must create an integer value.
2. That integer value must be of class `Integer`.<sup>15</sup>
3. Class `Integer` is defined by  $\mu$ Smalltalk code.
4. To evaluate  $\mu$ Smalltalk code, we need the evaluator.

Another circular dependency involves Boolean values and their classes:

1. Value `true` must be an instance of class `Boolean`.
2. Class `Boolean` must be a subclass of class `Object`.
3. Class `Object` has method `notNil`.
4. Method `notNil` must return `true` on class `Object`.

Each of these things depends on all the others, and we must find a way to break the cycle. Value `false` and method `isNil` participate in a similar cycle.

Each cycle is broken with a well-placed ref cell. The ref cell is initialized with an unusable value; the interpreter is bootstrapped by feeding it the definitions of the predefined classes; and the ref cell is assigned its proper value, closing the cycle.

The ref cells and the functions that update them are defined in the chunk *{support for bootstrapping classes/values used during parsing 706a}*. Primitives and built-in objects depend on these ref cells, as does the parser.

*Literals* An integer literal should be of class `SmallInteger`, which means that before we can evaluate an integer literal, we need to have parsed and evaluated the definition of `SmallInteger`. But to do that, we need the evaluator!

I break the circularity by reserving a mutable reference cell, `intClass`, to hold the representation of class `SmallInteger`. The cell is initially empty, but after the definition of `SmallInteger` is read, it is updated. The cell is used every time an integer literal is read: the interpreter calls function `mkInteger`, which tries to fetch the `SmallInteger` representation out of `intClass`. If `SmallInteger` has been read and `intClass` is not empty, `mkInteger` creates and returns a new instance of class `SmallInteger`. Otherwise, the interpreter crashes with an uncaught `InternalError` exception. Symbol literals and array literals are handled by similar functions and reference cells.

Because it's hard to reason about reference cells, I define them inside ML's local form. This form protects the reference cells from outside interference: the only functions whose behavior can depend on the contents of the reference cells are defined between `local` and `end`.

bind	312b
CLASS	694c
commaSep	S239a
emptyEnv	311a
internalMethods	S549a
logging	S548a
methodName	S549e
methodsEnv	S549e
mkClass	S550a
mkMeta	703d
PENDING	694c
setMeta	S550c
USER	694a
type value	693

<sup>15</sup>Our interpreter supports only `SmallInteger`, but full Smalltalk-80 supports literals that are too large to fit in `SmallInteger`.

**706a.** *(support for bootstrapping classes/values used during parsing 706a) ≡ (S547c) 706b ▷*

mkInteger : int → value
mkSymbol : string → value
mkArray : value list → value

```

local
  val intClass    = ref NONE : class option ref
  val symbolClass = ref NONE : class option ref
  val arrayClass  = ref NONE : class option ref
  fun badlit what =
    raise InternalError
      ("(bootstrapping) -- can't " ^ what ^ " in predefined classes")
in
  fun mkInteger n = (valOf (!intClass), NUM n)
    handle Option => badlit "evaluate integer literal or use array literal"

  fun mkSymbol s = (valOf (!symbolClass), SYM s)
    handle Option => badlit "evaluate symbol literal or use array literal"

  fun mkArray a = (valOf (!arrayClass), ARRAY (Array.fromList a))
    handle Option => badlit "use array literal"

```

Function `valOf` and exception `Option` are part of the initial basis of Standard ML.

Once the predefined class definitions have been read, classes are stored in the reference cells by function `saveLiteralClasses`, which takes one parameter, the global environment `xi`.

**706b.** *(support for bootstrapping classes/values used during parsing 706a) +≡ (S547c) ▷ 706a 706c ▷*

findClass : string * value ref env → class
--

```

fun saveLiteralClasses xi =
  findClass : string * value ref env → class
  ( intClass    := SOME (findClass ("SmallInteger", xi))
  ; symbolClass := SOME (findClass ("Symbol",           xi))
  ; arrayClass  := SOME (findClass ("Array",            xi))
  )
and findClass (name, xi) =
  case !(find (name, xi))
  of (_, CLASSREP c) => c
  | _ => raise InternalError ("class " ^ name ^ " isn't defined")
end

```

*Blocks* When eval interprets a block expression, it has to give it class `Block`, which also is not defined until its definition is read. Therefore, the same drill applies to blocks as applies to literal expressions. Appendix U defines these functions:

```

mkBlock : name list * exp list * value ref env * class * frame → value
saveBlockClass : value ref env → unit

```

*Booleans* I use the same technique for Booleans, except instead of using a mutable cell for each class, I use a mutable cell for each value.

**706c.** *(support for bootstrapping classes/values used during parsing 706a) +≡ (S547c) ▷ 706b*

mkBoolean : bool → value
--------------------------

```

local
  val trueValue  = ref NONE : value option ref
  val falseValue = ref NONE : value option ref
in
  fun mkBoolean b = valOf (!(if b then trueValue else falseValue))
    handle Option => raise InternalError "uninitialized Booleans"
  fun saveTrueAndFalse xi =
    ( trueValue  := SOME ( !(find ("true", xi)))
    ; falseValue := SOME ( !(find ("false", xi)))
    )
end

```

sameObject	className	addWithOverflow	value
class	protocol	subWithOverflow	printu
isKindOf	localProtocol	mulWithOverflow	printSymbol
isMemberOf	getMethod	+	newSymbol
error	setMethod	-	arrayNew
subclassResponsibility	removeMethod	*	arraySize
leftAsExercise	methodNames	div	arrayAt
newUserObject	newSmallInteger	<	arrayUpdate
superclass	printSmallInteger	>	

§10.11  
*Smalltalk as it  
really is*

707

Figure 10.30:  $\mu$ Smalltalk’s primitives

### 10.10.7 Primitives

$\mu$ Smalltalk’s primitives are listed in Figure 10.30. Each primitive’s specification should be suggested by its name, but if you are uncertain about what a primitive does, you can study how it is used in predefined classes.

A  $\mu$ Smalltalk primitive is almost the same thing as a  $\mu$ Scheme primitive function, and they are defined in the same way, by means of higher-order ML functions in the interpreter. For example, the value primitive is defined as function `valuePrim` in chunk 699b. There is no benefit to repeating the development in Chapter 5, so I discuss just two primitives here: `class`, which returns an object’s class, and `newUserObject`, which creates a new object.

The `class` primitive takes an object as its single argument. The object is represented by a pair that includes the object’s class, which is promoted to a full object by calling function `classObject` (chunk 703b).

**707a.** *(ML code for remaining classes’ primitives 699b)*  $\equiv$  (S548b) ▷ 699b 707b ▷  
`val classPrimitive = unaryPrim (fn (c, rep) => classObject c)`

The `newUserObject` primitive allocates fresh instance variables, each containing `nilValue`. It then allocates the object, and finally it assigns `self` to point to the object itself.

**707b.** *(ML code for remaining classes’ primitives 699b)*  $\equiv$  (S548b) ▷ 707a  

<code>mkIvars : class -&gt; value ref env</code>
<code>newUserObject : class -&gt; value</code>

```

local
  fun mkIvars (CLASS { ivars, super, ... }) =
    let val supervars = case super of NONE => emptyEnv | SOME c => mkIvars c
    in  foldl (fn (x, rho) => bind (x, ref nilValue, rho)) supervars ivars
    end
in
  fun newUserObject c =
    let val ivars = mkIvars c
        val self = (c, USER ivars)
    in  (find ("self", ivars) := self; self)
    end
end

```

ARRAY	694a
bind	312b
CLASS	694c
type class	694c
CLASSREP	694a
emptyEnv	311a
find	311b
InternalError	S366f
nilValue	704c
NUM	694a
SYM	694a
USER	694a
type value	693

## 10.11 SMALLTALK AS IT REALLY IS

The Smalltalk *language* is so small and simple that  $\mu$ Smalltalk can model almost all of it: assignment, message send, block creation, and nonlocal return. But Smalltalk-80 is not just a programming language. It is a programming *system*, unifying elements of programming language and operating system. And it is also

### 10.11.1 Programming language as operating system

Like all of the bridge languages,  $\mu$ Smalltalk fits comfortably into the development paradigm made popular by Unix:

- Programs are stored in files, which are managed by an underlying operating system.
- Whether a program is run by an interpreter or compiled by a compiler, it is invoked by an operating-system command, which finds the program in the filesystem. Usually, a person causes the command to run by typing at a *shell*.
- A program's variables have to be initialized when the program is run, and when the program terminates, their values are lost. Persistent state is stored only in the filesystem.
- Source code is edited and organized by programs that are unrelated to the interpreter or compiler, except they share a file system.
- When code changes, programs that use the code are restarted from the beginning.

Smalltalk-80 operates on a different paradigm:

- Programs are stored in an *image* of a running Smalltalk system. Operating-system abstractions like files are irrelevant to most Smalltalk source code.
- Persistent state is stored in Smalltalk variables, not in files. The state of the entire Smalltalk image, including the values of the variables of all the objects and classes that are defined—that is,  $\xi$  and  $\sigma$ —automatically persists.
- There are no programs or commands *per se*. Instead of “run a program,” the paradigm of execution is “send a message to an object.” Usually, a person causes the message to be sent by typing at a graphical user interface.
- Variables and objects don't need to be initialized before code runs; they retain their (persistent) state from the time they were first created. And when code finishes running—that is, when a message send terminates—the internal state of the relevant objects is (still) part of the image's state, so nothing is lost.
- Source code is just data, and it need not live in files. It is stored in instance variables of objects, and it is edited by sending messages to objects—usually by means of the graphical user interface.

- Unless something goes dramatically wrong, images don't terminate and aren't restarted. Sending a message to an object updates the image's state in much the same way that running a Unix command updates the filesystem's state. You are no more likely to restart an image than to restore a filesystem from backup.

This image-based paradigm has deep consequences for language design. Most dramatically, Smalltalk-80 doesn't have definition forms! If you want to create a new class, for example, you don't evaluate a definition form; instead, you send a message like `subclass:instanceVariableNames:` to the superclass of the class you wish to create. It responds with a class object, to which you add methods by sending `addSelector:withMethod:` or something similar. Or more likely, you manipulate a graphical user interface which sends these messages on your behalf.

### 10.11.2 Smalltalk-80's language

#### More literals and classes

Smalltalk-80 provides all the data forms in  $\mu$ Smalltalk and more besides. And more classes, including classes for characters and strings, can be instantiated using literal expressions:

Class	Literal
Integer	230
Character	\$a
Float	3.39e-5
Symbol	#hashnotquote
String	's p a c e s'
Array	#(\$a \$b \$c)

For symbols and arrays, Smalltalk-80 uses the hash mark, not our quote mark.

#### Syntax in context, with postfix, infix, and mixfix message sends

Smalltalk-80's syntax is meant to be displayed using a graphical user interface. But I'm stuck with ink on paper. I could show screen shots, but because they don't interact, they don't do justice to the experience. Instead, just to give you a feel for concrete syntax, I display code using the old "publication format" from the Smalltalk Blue Book (Goldberg and Robson 1983).

As an example, Figure 10.31 shows the publication format of the Shape class from Section 10.1.5. This format isn't source code; it's a display—it is tuned to printed paper in much the same way that real Smalltalk is tuned to two-dimensional graphics. Compare this display of Shape with my code from page 631. The protocols and computations are the same, but both the abstract syntax and the concrete syntax are different.

- As noted above, full Smalltalk doesn't use syntax for a class definition. The display at the top, which shows the class's name, superclass, and instance variables, could be obtained by sending messages `name`, `superclass`, and `instVarNames` to the class object.
- Instead of tagging each individual method as a class method or an instance method, Smalltalk-80 displays groups of methods. These displays will also have been computed by sending messages to the class object.

class name	Shape
superclass	Object
instance variable names	center radius
class methods	
instance creation	
<b>new</b>	
	↑super new center: (CoordPair x: 0 y: 0) radius: 1
instance methods	
observing locations	
<b>location: pointName</b>	
	↑center + ((pointVectors at: pointName) * radius)
<b>locations: pointNames</b>	
	locs
	pointNames do: [:point   locs add: (self location: point)].
	↑locs
mutators	
<b>adjustPoint: pointName to: location</b>	
	center ← center + (location - (self location: pointName))
<b>scale: k</b>	
	radius ← radius * k
drawing	
<b>drawOn: picture</b>	
	self subclassResponsibility
private	
<b>center: c radius: r</b>	
	center ← c.
	radius ← r

Figure 10.31: Class Shape, in Smalltalk-80 publication format

- The Smalltalk-80 display shows information about *subgroups* of methods: “instance creation,” “observing locations,” “mutators,” and “private.” These subgroups, which have no analog in  $\mu$ Smalltalk, serve as documentation. In Smalltalk-80 the subgroups are called *message categories*. More modern Smalltalks use the term *protocol* for a subgroup, as well as for the collection of messages as a whole.
- Each method definition is introduced by a line set in bold type; this line, which is called the *message pattern*, gives the name of the method and the names of its arguments. A method that expects no arguments has a name composed of letters and numbers. A method that expects one argument has a colon after the name; the name of the method is followed by the name of the argument. When a method expects more than one argument, the name of the method is split into *keywords*: each keyword ends with a colon and is

followed by the name of one argument. This concrete syntax matches the concrete syntax of message sends, as shown below.

When a method has two or more arguments, we name it by concatenating the keywords; thus, the name of the private message is “center:radius:,” just as in  $\mu$ Smalltalk.

- The display indents the bodies of the methods. The body of a method contains a sequence of *statements*, which are separated by periods. To return a value from a method, a Smalltalk-80 programmer must use `return`, which in full Smalltalk is written using the up arrow  $\uparrow$  or caret  $^$ . A method that doesn’t evaluate a `return` answers `self`, but it is likely to be evaluated only for its effects.
- The “ $\leftarrow$ ” symbol is used for assignment. (Modern implementations use `:=`.)
- Within a statement, the most significant difference between Smalltalk-80 syntax and  $\mu$ Smalltalk syntax is that when a keyword message is sent, the message’s arguments are *interleaved* with the parts of the message’s name. And Smalltalk-80 uses round brackets only where needed to disambiguate. For example, in  $\mu$ Smalltalk we write,

```
((Triangle new) adjustPoint:to: 'Southwest (s location: 'East))
```

but in Smalltalk-80, we write,

```
Triangle new adjustPoint: #Southwest :to (s location: #East)
```

- As in  $\mu$ Smalltalk, `+` and `-` are *binary messages*: a binary message is sent to a receiver and expects one argument. As with keyword messages, the code is written in receiver-message-argument form, for example, `3 + 4`. Binary messages have higher precedence than keyword messages; for example, in the `location:` method, to prevent `pointName` from being interpreted as the receiver of `*`, the message send `pointVectors at: pointName` is wrapped in parentheses.
- Blocks are written in square brackets, not in curly braces. ( $\mu$ Smalltalk’s curly braces come from Ruby, a close relative of Smalltalk.) And when a block takes arguments, its syntax is lighter weight than what  $\mu$ Smalltalk uses. Arguments are written after the opening “[” and before a vertical bar “|”; their names are preceded by colons. Thus, the block we write in  $\mu$ Smalltalk as

```
[block (x) (self add: x)]
```

is written in Smalltalk-80 as

```
[:x | self add: x].
```

Interestingly, only Objective C and Grace support Smalltalk’s interleaving of keyword and argument in concrete syntax. Several other languages do, however, allow you to simulate keyword syntax by passing a literal dictionary as an argument.

*More variables: class instance variables and class variables*

In Smalltalk-80, a class object has its own instance variables, which are distinct from the instance variables of its instances. These variables are called *class instance variables*. Class instance variables are specific to a class, and their values are not shared with subclasses—they are, quite simply, the instance variables of the

metaclass. Smalltalk-80 also has *class variables*, which, unlike instance variables or class instance variables, are shared: they are accessible to all the instances of a class and its subclasses, as well as the class itself and its subclasses. Class variables can make global variables unnecessary: because *every* class inherits from Object, the class variables of class Object can play the role of global variables. So in full Smalltalk, for example, the point-vectors dictionary from page 632 would be a class variable of class Shape.

### Restricted primitives

In  $\mu$ Smalltalk, a primitive expression may appear anywhere within a method, with any arguments. In Smalltalk-80, primitives are numbered, not named, and a primitive may appear only at the beginning of a method, before any statements or expressions. And the primitive has access only to the receiver and to the arguments of the message that activated the method.

With these restrictions, a Smalltalk-80 method may be defined as a numbered primitive followed by one or more expressions, possibly ending in a return:

*message-pattern*  
<primitive *n*>  
*expressions*

When the method is activated, it executes the primitive numbered *n*. If for any reason the primitive fails, the *expressions* are executed; if the primitive succeeds, its result is returned and the *expressions* are ignored.

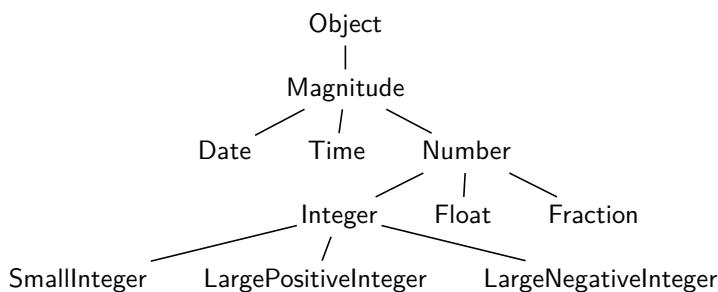
Application programmers never need to write a method that contains a primitive; as in  $\mu$ Smalltalk, primitives are used only in methods of predefined classes.

#### 10.11.3 Smalltalk-80's class hierarchy

Smalltalk-80 comes with many, many class definitions. This section sketches only Smalltalk-80's Number and Collection sub-hierarchies; these general-purpose classes are used by many programmers. Smalltalk-80 also includes many special-purpose classes, including some that support graphics, compilation, and process scheduling. They are beyond the scope of this book.

#### Smalltalk-80's numbers

In Smalltalk-80, the part of the class hierarchy that contains numeric classes, which is the model for  $\mu$ Smalltalk, looks like this:



Classes `Magnitude` and `Number` resemble  $\mu$ Smalltalk's versions, but Smalltalk-80's `Numbers` respond to more messages than  $\mu$ Smalltalk's `Numbers`. Classes `Float` and `Fraction` are likewise similar in both languages. Smalltalk-80 provides floating-point literals, and its `Integer` literals may be arbitrarily long; the class of an integer literal is determined by its magnitude and sign.

Like many languages, Smalltalk-80 supports mixed arithmetic, in such a way that the class of a result may depend on the classes of its arguments. For example, Smalltalk-80 can add a fraction to an integer, answering a fraction. But Smalltalk-80 is more flexible than languages in which the representation of a result is determined by a static type system; in Smalltalk-80, an arithmetic method conventionally answers an object of the *least general* class that can hold the result. Under this convention, the class of a result depends upon the *value* of a receiver or an argument, not just its class. For example, `10 raisedTo: 2` answers a `SmallInteger`, but `10 raisedTo: 20` answers a `LargePositiveInteger`. `SmallInteger` is considered less general than `LargePositiveInteger`, which is considered less general than `Float`. That is why `10 raisedTo: 20` does not answer a `Float`.

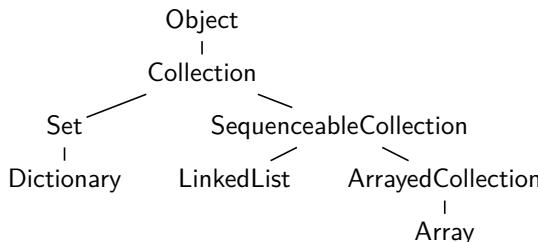
The implementations of the `Number` operations exploit the continue-after-failure semantics of Smalltalk-80's primitives. For example, the definition of `+` on `SmallInteger` uses primitive 1 to do the addition, but if the primitive fails, e.g., because the addition overflows, the succeeding statements automatically change over to large-integer arithmetic. The changeover is effected by the simple coercion `asLargeInteger`:

```
+ aSmallInteger
<primitive 1>
↑self asLargeInteger + aSmallInteger

asLargeInteger
self negative
ifTrue: [↑self asLargeNegativeInteger]
ifFalse: [↑self asLargePositiveInteger]
```

#### *Smalltalk-80's Collection classes*

Smalltalk-80's Collection hierarchy looks a bit different from the Collection hierarchy in  $\mu$ Smalltalk. The  $\mu$ Smalltalk hierarchy is derived from the one in Tim Budd's (1987) Little Smalltalk. Here is a part of the Smalltalk-80 collection hierarchy:



Both the Smalltalk-80 and the  $\mu$ Smalltalk collection hierarchies use iteration (`do:`) as the fundamental operation.

```
((List new) add: 1) add: 2) add: 3) // uSmalltalk
List:new():add(1):add(2):add(3) -- Lua
(new List).add(1).add(2).add(3) // inspired by C++
```

This useful convention extends to all sorts of mutable abstractions. But if you study the collection protocols for Smalltalk-80, you'll be surprised: the `add:` message does *not* answer the collection receiving the message; instead, it answers the object just added. What gives?

It turns out that the convention of sending a sequence of messages to a mutable abstraction is so useful that Smalltalk-80 provides special syntax for it: the *message cascade*. A receiver can be followed by a *sequence* of messages, separated by semi-colons, and each message is sent in sequence to the same receiver. Here is Squeak Smalltalk:

```
OrderedCollection new add: 1; add: 2; add: 3; yourself
```

This statement sends a cascade—a sequence of four messages—to the object answered by `OrderedCollection new`. The answer from the cascade is the answer from the last message. In this example, that's `yourself`, which causes the receiver to answer itself: a list containing the elements 1, 2, and 3.

### *Reflection*

In Smalltalk-80, aspects of a running image are themselves exposed as objects. An image can use these objects to manipulate itself; the ability of a program to manipulate itself is sometimes called *reflection*. Reflection makes it possible to implement class browsers, compilers, debuggers, garbage collectors, and so on in the same language used to write applications. In  $\mu$ Smalltalk, you can get a taste of reflection by using methods like `class` or `addSelector:withMethod:`, but in a full Smalltalk system, you can do much more.

For example, given a class object, you can add or remove methods, query for particular methods, and call methods by name. You can compile or decompile code associated with methods. You can add or remove methods. You can change the superclass and add or remove subclasses. You can get the names of all the instance variables, which you can also add and remove. You can get information about the representations of instances. And so on.

Even the “active contexts” (stack frames) used to implement message send are represented as objects. This means, for example, that a debugger or trace routine can visit all active contexts and display values of local variables and instance variables, all using reflection.

## 10.12 OBJECTS AND CLASSES AS THEY REALLY ARE

While most popular object-oriented languages organize objects into classes, classes aren't required. Object-oriented programming without classes is demonstrated most beautifully by the Self programming language. In Self, an object is a collection of named *slots*; a slot holds one object. Sending a slot name to an object answers the

value of the object in the slot, or if the slot holds a special “method object,” the answer is the result of running the method. An object is created by giving the names and contents of its slots. An object may also be created by *cloning* another object, called the *prototype*. Cloning is a shallow copying operation; it creates a new object whose slots hold the same objects as the original. Self’s model of object-orientation has been adopted by JavaScript, and it is also used in Lua.

Among popular object-oriented languages that do use classes, many are hybrids. The obvious way in which many languages are hybrids is that they include primitive types which act like abstract data types, not like objects. The classic example is the `int` type found in C++ and Java. A less obvious form of hybridization occurs when classes act as static types. In this form, a class behaves more like one of Molecule’s abstract data types than like one of Smalltalk’s classes. For example, in Modula-3 and C++, the arguments to methods have static types, and a class method can see the representation of any argument of the class. But the method interoperates only with objects of the class (or its subclasses); Smalltalk’s flexible, behavioral subtyping (“duck typing”) is lost.

One popular language that offers *both* forms of data abstraction is Java. A Java interface corresponds nicely to a Smalltalk protocol. And as long as the types of methods are written entirely using interfaces, you can do object-oriented programming much in the style of Smalltalk, with behavioral subtyping—different implementations of an interface interoperate with one another. But you must use Java classes *only* to manufacture new objects. The moment you allow a Java class to specify the type of an argument to a method, you are using that class as an abstract data type, not as a factory that builds objects which answer a given protocol. The method then gains access to the representation of the argument, but as a consequence, it loses some ability to interoperate: it can interoperate only with objects of the specified class, or of one of its subclasses.

### 10.13 SUMMARY

Smalltalk helped object-oriented languages take over the world. Object-orientation seems very effective for certain types of problems, including simulation, graphics, and user interfaces. Although classes, subclasses, and objects were first introduced in Simula 67, the compelling case for these ideas was made by Smalltalk. Apple uses many of the programming-environment ideas developed for Smalltalk, and for many years Apple’s devices were programmed primarily using the object-oriented language Objective C, a close relative of Smalltalk. (Apple’s more recent language Swift adds functional-programming ideas to the mix.) The wild ideas that Alan Kay thought might “amplify human reach” are now central to mainstream programming.

An object is a bundle of operations that share state. When one of the operations is invoked, the implementation is chosen dynamically, and it can be different for different executions of the same code. The bundle of operations and their behaviors are specified by a *protocol*, which describes an abstraction. In a context expecting such an abstraction, any object that responds to the protocol may be used, regardless of the class used to manufacture the object. This property of object-oriented programming is called *behavioral subtyping* or *duck typing*.

In a pure object-oriented language, an operation has access only to the representation of the object on which it is defined; each argument has to be treated abstractly, through an interface or protocol. An operation that wishes to look at the representation of an argument must somehow extend the argument’s interface—perhaps simply by defining additional methods, like  $\mu$ Smalltalk’s predefined numeric classes, or perhaps by using double dispatch, like the large-integer classes

in the exercises. In a hybrid language like Java or C++, the abstraction presented by an argument may be compromised: a class may act not only as a factory that builds new objects but also as an abstract data type. This sort of hybrid gains the ability to inspect an argument's representation directly, but it loses the ability to implement the argument's abstraction with a representation that is independent of its original class. The distinction between class as object factory and class as abstract type is subtle, and in popular object-oriented languages, it can be hard to untangle.

### 10.13.1 Key words and phrases

**DATA ABSTRACTION** The practice of characterizing a data type by its operations and their specifications, not by its **REPRESENTATION**.

**OBJECT** The unit of **DATA ABSTRACTION** in Smalltalk. It encapsulates mutable state represented as **INSTANCE VARIABLES** and code represented as **METHODS**. Only an object's own methods have access to its instance variables.

**AUTOGNOSTIC PRINCIPLE** Cook's (2009) term for a defining principle of object-oriented languages: an object has access only to its own representation, and it can access other objects only through their public interfaces.

**CLASS** A specification for making **OBJECTS**. In Smalltalk, a class is also itself an object.

**INSTANCE** When object  $O$  is made from the specification given by class  $C$ ,  $O$  is called an *instance* of  $C$ .

**ABSTRACT CLASS** A class used only to define methods, not to create instances. Other classes **INHERIT** from it.

**INSTANCE VARIABLE** Part of the state of a Smalltalk object. An object's instance variables are accessible only to that object's own methods.

**METHOD** Code that is executed in response to a **MESSAGE**. Sometimes called an **INSTANCE METHOD**. Like a function, a method takes formal parameters and returns a result. Unlike a function in Scheme, ML, or Molecule, a method is not a value. A method has access to the **INSTANCE VARIABLES** of its **RECEIVER**, but not to the instance variables of any of its arguments. An object's methods are determined by its **CLASS**.

**COMPLEX METHOD** A **METHOD** that wishes to inspect the representation of one or more **ARGUMENTS**, not just the **RECEIVER**.

**MESSAGE PASSING** Smalltalk's central control structure. Message passing sends a **MESSAGE** to an **OBJECT**. The message includes actual parameters, which are also objects. The receiver of the message first selects, and then evaluates, a **METHOD**, the result of which is **ANSWERED** in reply to the message.

**MESSAGE** A combination of a message name, which is used to select a **METHOD**, and actual parameters. Sent to an **OBJECT**. The name is called a **MESSAGE SELECTOR**.

**PROTOCOL** The set of **MESSAGES** to which an object can respond, together with the rules that say what actual parameters are acceptable and how the object is expected to behave in response.

**PRIVATE METHOD** A method that is to be invoked only by other methods of the same CLASS, or of its SUBCLASSES. We can also talk about an object's private PROTOCOL.

In Smalltalk, the private method is purely a programming convention, as it is also in Python. In object-oriented languages with static type systems, the type system usually provides a mechanism to make a method private.

**MESSAGE SELECTOR** The name that, together with arguments, constitutes a MESSAGE.

§10.13. Summary

717

**RECEIVER** In MESSAGE PASSING, the object to which a message is sent. A message need not have any actual parameters, but it is always sent to a receiver.

**METHOD DISPATCH** The algorithm used to determine what METHOD is evaluated in response to a MESSAGE. A form of DYNAMIC DISPATCH. In Smalltalk, method dispatch begins with the class of the RECEIVER, looking for a method with the same name as the message. If no such method is found, method dispatch continues with the receiver's superclass, and so on all the way up to class Object. If method dispatch fails, it does so with a "message not understood" error.

**ANSWER** Smalltalk's term for the response to a message. Corresponds to a "return value" in other languages. Smalltalkers sometimes refer to an *action* "to answer," which is analogous to the action "to return" found in other languages.

**CLASS METHOD** A method that responds to messages sent to a class, rather than to instances of that class. Used most often to create or initialize instances of the class, as with the message new.

**SUPERCLASS** A class named in a class definition, from which the defined class inherits instance variables and methods. Except for the primitive class Object, every class has a unique superclass. Class Object has no superclass. To confuse matters slightly, a superclass of a superclass is sometimes also called a superclass.

**SUBCLASS** A class that INHERITS from a superclass. A class may many subclasses or none at all. And a subclass of a subclass is sometimes also called a subclass.

**INHERITANCE** The mechanism by which a CLASS definition is combined with the definition of its SUPERCLASS to specify how objects of the class are made.

**SINGLE INHERITANCE** Describes a language in which a class inherits from exactly one superclass. Contrasted with MULTIPLE INHERITANCE. Smalltalk uses single inheritance.

**MULTIPLE INHERITANCE** Describes a language in which a class may inherit from more than one superclass. When different superclasses provide conflicting definitions, e.g., of the same method, the conflict must be resolved somehow. Multiple inheritance has had passionate advocates, but the success of Java has weighted the scales in favor of SINGLE INHERITANCE, and much of the passion has died down.

**OVERRIDE** When a class defines a method that is also defined in a superclass, every instance of that class uses the new definition, not the superclass definition. The superclass definition is said to be *overridden*. Overriding is sometimes called REDEFINING the superclass method.

**DELEGATION** “Delegation” is a term of art for an implementation technique in which a method is implemented by sending its message to another object. For example, if a large integer is represented by a sign and a magnitude, the `isZero` method on large integers might be implemented by sending the `isZero` message to the object that represents the large integer’s magnitude.

**self** A keyword used within a METHOD to refer to the RECEIVER of a MESSAGE. When a message is sent to `self`, METHOD DISPATCH begins with the class of the receiver, not the class in which the method is defined. In some object-oriented languages, “`self`” is written “`this`.”

**super** A special keyword that bypasses the normal METHOD DISPATCH algorithm and dispatches directly to a method defined in the superclass of the class in which `super` appears.

**OPEN RECURSION** When a method sends a message to `self`, the message may dispatch to a method defined by a SUBCLASS. If the message being sent has the same name as the method from which the message is sent, what looks like a “recursive call” may not be a recursive call. This ability of message passing to “intercept” or “override” the apparently recursive call is called *open recursion*. The phrase “open recursion” is also used to describe the more general phenomenon of a message dispatching to a subclass method, even when the message name and method name are different.

When open recursion is used extensively, as in the implementation of arithmetic, for example, understanding the sequence of actions taken by an algorithm requires you to piece together many methods, which can be difficult.

**BLOCK** An object much like a lambda abstraction, which is activated by sending it a message. It has no instance variables. Unlike a method, a block is a first-class value—an object with the same privileges as any other object. Smalltalk blocks are used as continuations to implement conditionals, loops, and exceptions.

**REPRESENTATION INVARIANT** A property that holds among all the instance variables of an object, and which the object’s methods rely on for proper functioning. Responsibility for maintaining representation invariants falls on an object’s own methods and on the methods of its subclasses. But unlike in Molecule, an object’s methods are not responsible for the invariants of actual parameters, even if they of the same class.

**ABSTRACTION FUNCTION** A function, used conceptually but not necessarily written in code, that maps the values of an object’s instance variables to the abstract thing the object represents.

**EXPOSE THE REPRESENTATION** Because Smalltalk methods are AUTOGNOSTIC, no method can ever inspect the representation of an argument—all a method can do with an argument is send messages to it. But an argument may define methods that *expose* the representation. The most extreme example is to define a “getter” and “setter” method for every instance variable. This kind of exposure makes it easy to implement COMPLEX METHODS, but it destroys data abstraction.

**REFLECTION** A feature of full Smalltalk whereby methods, usually defined on class Class or class Object, provide information about the implementations of classes or objects, or enable a Smalltalk program to alter itself. One simple

example is the method `subclassName:instanceVariableNames:`, which can be sent to any class object to create a new subclass of that class. Smalltalk-80 has many other reflective methods: all instances of a class, all instance variables of an object, and so on.  $\mu$ Smalltalk provides a cheap, plastic imitation of reflection, limited to such methods as `class`, `addSelector:withMethod:`, `compiledMethodAt:`, and a few others.

**COLLECTION** One of several Smalltalk classes whose objects act as containers for other objects. Predefined collections include dictionaries, sets, lists, and arrays. A collection can be defined by just a handful of methods, of which the most important is the `do:` method, which iterates over the objects contained in the collection. The collection then inherits a large number of useful methods from class `Collection`.

**MAGNITUDE** A quantity like a date, a time, or a number, of which it can be said that one precedes another, but that might not support arithmetic.

**NUMBER** In  $\mu$ Smalltalk, a machine integer, rational number, floating-point number, or arbitrary-precision integer. Machine integers are primitive, and arbitrary-precision integers are left as an exercise. Rational numbers and floating-point numbers are predefined.

**BEHAVIORAL SUBTYPING** A principle which says that if an object answers the same **PROTOCOL** as another object, and if it behaves in the same way, it may be considered to have the same *type* as the other object. Called DUCK TYPING by Ruby programmers, because “if it walks like a duck...”

### 10.13.2 Further reading

Smalltalk was invented to help create a computing experience that might deserve to be called “personal.” Engineers at the Xerox Palo Alto Research Center (PARC) were inventing and refining the one-person computer; the bitmapped display; the user-interface elements that we now call windows, menus, and pointing; the use of abstract data types in systems programming; and many other delights that we have long since taken for granted. Alan Kay and his group worked to create a software system that would make the computer “an amplifier for the human reach,” not just a tool for building software systems. Kay’s (1993) account of those heady days is well worth your time.

The basic reference on Smalltalk-80 is “blue book,” by Goldberg and Robson (1983), which is so called because of its predominantly blue cover. The blue book introduces the language and class hierarchy. It also presents five chapters on simulation and five chapters on the implementation and the virtual machine on which it is based. In designing  $\mu$ Smalltalk, I drew heavily on Goldberg and Robson, especially for blocks. The “red book” (Goldberg 1983) describes the Smalltalk-80 programming environment and its implementation. There is also a “green book,” which describes some of the history of Smalltalk (Krasner 1983).

Smalltalk was the topic of the August 1981 special issue of Byte Magazine (Byte 1981). The issue includes twelve articles on the system, written by the people who built it. Topics range from design principles to engineering details, with plenty of examples, plus a description of the programming environment. A narrow, deep introduction to the Smalltalk-80 language and programming environment is presented by Kaehler and Patterson (1986), who develop a single, simple example—the towers of Hanoi—all the way up to an animated graphical version.

If you want to use a Smalltalk system, Ingalls et al. (1997) describe Squeak, a free, portable implementation of Smalltalk that is written in Smalltalk itself. To get

started, you will want help with the integrated programming environment, which is otherwise overwhelming; Black et al. (2009) will help. Or you could try Pharo (Ducasse et al. 2016), a fork that looks better to modern eyes.

If you want to tackle something simpler than Smalltalk-80 but more ambitious than  $\mu$ Smalltalk, Budd (1987) describes “Little Smalltalk,” which is nearly identical to Smalltalk-80, but which lacks some extras, especially around graphics. Budd’s book is easier reading than the blue book, and interpreters for Little Smalltalk are available.

In the end, no book tells you everything you need to know to become an effective Smalltalk programmer. You must master the class hierarchy, and to gain mastery, you must read and write code. As you write, be guided by Beck (1997), who discusses coding style; Beck’s excellent book will help you make your Smalltalk code idiomatic.

The first object-oriented language was Simula 67, which introduced objects, classes, and inheritance (Dahl and Hoare 1972; Birtwistle et al. 1973). As its name suggests, Simula 67 was designed with discrete-event simulation in mind. Simula 67 inspired the designers of CLU, Smalltalk, and C++, among others.

Object-oriented languages without classes are rare. The first such language, and still one of the simplest and most innovative, is Self (Ungar and Smith 1987). Another such language, whose semantics is admittedly cluttered with strange behaviors and corner cases, is JavaScript. Underneath the clutter is a language with good bones, which come from Scheme and from Self; for a view of the good parts, see Crockford (2008). And even if your language doesn’t acknowledge objects, you can still write object-oriented programs, provided your language gives you associative arrays and first-class functions. For a nice tutorial on this idea and the corresponding programming techniques, see Ierusalimschy (2013).

Object-oriented programming has become so popular that many, many programming languages identify themselves as object-oriented or include object-oriented features. C++ (Stroustrup 1997) adds object-oriented features—and many others besides—to C. Objective C is another object-oriented extension of C; its object-oriented features look more like Smalltalk than like C++ (Cox 1986). Objective C was, for many years, the language most often used to write software for Apple products. Modula-3 (Nelson 1991) is an object-oriented extension of Modula-2. Modula-3 inspired the development of the scripting language Python (van Rossum 1998) and the general-purpose application language Java (Gosling, Joy, and Steele 1997). Another scripting language, Ruby (Flanagan and Matsumoto 2008), is more directly inspired by Smalltalk.

Many statically typed object-oriented languages, including C++, Modula-3, and Ada 95, mix object-oriented ideas with ideas from abstract data types. It’s often hard to tell what’s going on. For a clear, deep analysis of abstract data types, objects, and the differences between them, I recommend Cook (2009). Cook analyzes each set of ideas separately, then applies both to a couple of popular languages, including both Java and Smalltalk. For a foundational view of these two kinds of abstraction, consult Reynolds (1978).

To understand any language that claims to be both statically typed and object-oriented, you can precede Cook’s analysis with a simpler question: is subtyping distinct from subclassing? If not, then the designer has chosen to trade off some of the openness and expressivity of object-oriented programming in favor of some other good—like a simple type system. But there are type systems that can handle open systems, even those as flexible as Smalltalk. For example, Graver and Johnson (1990) describe a type system for Smalltalk in which subtyping and subclassing are

carefully distinguished. This type system is the basis for an optimizing Smalltalk compiler (Johnson, Graver, and Zurawski 1988).

Behavioral subtyping is described formally by Liskov and Wing (1994), who propose that any definition of subtyping should meet this criterion: if type  $\tau'$  is considered a subtype of  $\tau$ , then any property that is provable about objects of type  $\tau$  must also be true of objects of type  $\tau'$ . Liskov and Wing provide two definitions of subtyping that meet the criterion.

While inheritance helps build open, extensible systems, it comes with another tradeoff: loss of modularity. Because a subclass can mutate all of a class's representation and can send to any of its private methods, it must understand everything that is going on in the class, lest it violate the class's invariants. Stata (1996) presents the problem and proposes a new programming-language mechanism which expresses the interface between a class and its subclasses, restoring modular reasoning.

Object-orientation continues to inspire the design of new languages. Eiffel, which is described in the excellent book by Meyer (1992), is designed to support a particular object-oriented design methodology (Meyer 1997). Grace (Black et al. 2012) is designed especially for teaching novices. Swift is a successor to Objective C for programming Apple devices (Buttfield-Addison, Manning, and Nugent 2016). Of the languages above, only Objective C, Python, and Ruby share with Smalltalk the property of being dynamically typed; the others are statically typed, or in the case of Grace, optionally typed. Smalltalk itself can be extended with type checking (Johnson, Graver, and Zurawski 1988).

The object-oriented ideas popularized by Smalltalk were quickly adopted into Lisp. The Flavors system (Cannon 1979; Moon 1986) was developed for MIT's Lisp Machine as a "non-hierarchical" approach to object-oriented programming: a "flavor" takes the place of a class, and a new flavor can be defined by combining several existing flavors. Flavors influenced CommonLoops, a substrate that can be used to absorb ideas from Flavors and from Smalltalk into Common Lisp (Bobrow et al. 1986), and eventually the Common Lisp Object System, an object-oriented extension of Lisp usually called CLOS (Bobrow et al. 1988).

## 10.14 EXERCISES

The exercises are arranged by the skill or knowledge they call on, as shown in Table 10.32. More than in other chapters, the exercises call on knowledge of  $\mu$ Smalltalk's large initial basis. Here are some that I especially like:

- Smalltalk's method dispatch might seem inefficient. But many call sites dispatch to the same method over and over—a property that clever implementations exploit. Exercise 41 asks you to implement a simple *method cache* and measure its effectiveness. The results will surprise you.
- Exercises 37 to 39 are the most ambitious exercises in this chapter. You implement a classic abstraction: full integers of unbounded magnitude. I've split the project into three stages: natural numbers, signed large integers with arithmetic, and finally "mixed arithmetic," which uses small machine integers when possible and transparently switches over to large integers when necessary. You will boost your object-oriented programming skills, and you will understand, from the ground up, an implementation of arithmetic that should be available in every civilized programming language.

<i>Exercises</i>	<i>Sections</i>	<i>Notes</i>
1	10.1	Using shape classes.
2 to 5	10.1	Defining new shapes and new canvases.
6 and 7	10.1	Creating new classes from scratch (for random numbers, see also §10.4.6).
8 and 9	10.3.4	Messages to super: their use for initialization, and their semantics.
10	10.4.1	Equality and object identity.
11 and 12	10.4.2	Redefinition of existing methods using the protocol for all classes: print methods for pictures (§10.4.5); pictures with floating-point coordinates (§10.4.6).
13 to 15	10.4.2	New methods on existing classes: identify metaclasses (§10.10.4), condition on nil, hash.
16	10.5	Conditionals as dynamic dispatch: the implementation of class False.
17 to 31	10.4.5, 10.6	Modifying existing collections and defining new collections.
32 and 33	10.4.6, 10.6.1, 10.7.2	Make class Char a magnitude (§10.4.2); define new magnitudes.
34 to 39	10.4.6, 10.7	Numbers and arithmetic: reasoning about overflow; arithmetic between different classes of numbers; arbitrary-precision arithmetic.
40 and 41	10.9.1, 10.10.2	Method dispatch: understanding its semantics; improving its implementation using a method cache.
42	10.10.2	Object-oriented profiling: finding hot spots by measuring what objects receive what messages.

Table 10.32: Synopsis of all the exercises, with most relevant sections

#### 10.14.1 Retrieval practice and other short questions

- In the expression (c location: 'East), what is the receiver? What is the message selector? What is the argument?
- In the expression (c location: 'East), what is the role of the colon after the word location? If the colon were missing, what would go wrong?
- Why does class CoordPair have two protocols (a “class” protocol and an “instance” protocol)? What can you do with the messages in the class protocol? What can you do with the messages in the instance protocol?
- In a language like C++, new signifies a syntactic form. But in Smalltalk, new isn’t special. When the Smalltalk expression (List new) is evaluated, what happens?
- In the expression (c location: 'East), c is an object of class Circle, and class Circle doesn’t define a location: method. So when the expression is evaluated, what happens? How does it work?
- What objects can the message class be sent to? What objects can the message superclass be sent to?
- Once a class is defined, can you go back and add, change, or remove methods? How?

- H. Message `ifTrue:` is sent to a Boolean, but message `whileTrue:` is sent to a block. Why the difference?
- I. In the `drawPolygon:` method implemented on class `TikzCanvas`, what does the expression in square brackets evaluate to? When the resulting object is sent to `coord-list` as an argument to the `do:` message, what happens?
- J. A  $\mu$ Smalltalk list is just one form of *collection*—there are others. All collections understand messages that are analogous to the  $\mu$ Scheme list functions `map`, `app`, `filter`, `exists?`, and `fold`. What is the Smalltalk analog of each of these list functions?
- K. When message `at:ifAbsent:` is sent to a keyed collection, why is the second argument a block?
- L. In Smalltalk, every number is a magnitude, but not every magnitude is a number. What's an example of something you can do with a `Number` that you can't do with a `Magnitude`? What's an example of an abstraction that is a magnitude but not a number?
- M. How is the message `isNil` implemented without a conditional test?
- N. The `Magnitude` protocol offers six relational operators plus operations `min:` and `max::`. But no individual magnitude class has to implement all of them. What operations have to be implemented by subclasses of `Magnitude`, like `Integer` or `Fraction`? And how do things work with the other `Magnitude` operations, which aren't implemented by the subclass?
- O. Study how `return` is used in the implementations of methods `isEmpty` and `includes:` on class `Collection`. Each of these methods has a body, which contains one or more nested blocks, one of which contains a `return`. When the `return` is evaluated, what evaluation or evaluations are terminated?
- P. To iterate over a list in  $\mu$ Scheme, we have to ask the list how it was formed—is it empty or nonempty? Iteration in Smalltalk doesn't need to ask such questions. How is list iteration accomplished instead?
- Q. Method `=` on class `Fraction` needs to know the numerator and denominator of both the receiver and the argument. How does the method get the numerator and denominator of the receiver? How does it get the numerator and denominator of the argument?
- R. In the semantic judgment  $\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \Downarrow \langle v; \sigma', \mathcal{F}' \rangle$ , components  $e$ ,  $\rho$ ,  $\xi$ , and  $\sigma$  have their familiar roles. What is the role of the component  $c_{\text{super}}$ ?
- S. In the semantic judgment  $\langle e, \rho, c_{\text{super}}, F, \xi, \sigma, \mathcal{F} \rangle \uparrow \langle v, F'; \sigma', \mathcal{F}' \rangle$ , what are the roles of the two frames  $F$  and  $F'$ ? In particular, what happens if those two frames are identical?
- T. In the evaluation of an expression of the form `SEND(m, e, e1, ..., en)`, suppose that the evaluation of  $e_1$  returns  $v$  to frame  $F'$ . Are expressions  $e_2$  to  $e_n$  evaluated? What rules of the operational semantics determine the answer?
- U. A Smalltalk class is also an object, but in the interpreter, it is represented as an ML record. How does such an object know what its class is? What is the name for the class of such an object?
- V. A  $\mu$ Smalltalk literal like `1983` evaluates to an object of class `SmallInteger`. But class `SmallInteger` is itself defined using  $\mu$ Smalltalk code, and that code has to be read by the parser! How is this cyclical dependence resolved? Can anything go wrong?

### 10.14.2 Using shape classes

1. *Draw pictures.* By combining `adjustPoint:to:` with `location:`, you can create pictures without ever having to do vector arithmetic—all the arithmetic is encapsulated in the two methods. Using `adjustPoint:to:`, `location:`, and the other shape and picture methods, write Smalltalk code to draw each of these pictures:

(a) 

(b) 

(c)  (The radii of the circles are in the ratio 9 to 6 to 4.)

### 10.14.3 New shapes and canvases

2. *Implement a new shape.* Using `Square` as a model, implement class `Triangle`.
3. *Reuse code by defining and inheriting from a Polygon class.* The square and triangle are both polygons, and both can be drawn in the same way, only using different points. Define a new, abstract class `Polygon`, which is a subclass of `Shape`, and which includes a method `points`. When sent the `points` message, a polygon should answer a list of symbols that name the control points that should be drawn.
  - (a) Implement `Polygon`. Method `drawOn:` should use method `points`, which should be a subclass responsibility.
  - (b) Redefine `Square` and `Triangle` to be subclasses of `Polygon`. The new `Square` and `Triangle` classes should define only the `points` method—all other methods should be inherited from `Polygon`.
  - (c) As a subclass of `Polygon`, implement the new shape `Diamond`.
  - (d) As subclasses of `Polygon`, implement the new shapes `TriangleRight`, `TriangleLeft`, and `TriangleDown`, whose triangles point left, right, and down.
4. *A scalable canvas.* The `startDrawing` method of class `TikzCanvas` establishes the scale that one unit is rendered as “4pt,” which means four printer’s points, or in L<sup>A</sup>T<sub>E</sub>X, about  $\frac{4}{72}$  of an inch. Change class `TikzCanvas` so that the size of a unit is an instance variable of the class, and add a class method `withUnit:` that can specify the initial scale. Use a symbol, so that you can specify a scale using any of the units L<sup>A</sup>T<sub>E</sub>X understands, as in these examples:

```
(Tikzpicture withUnit: '12pt)  
(Tikzpicture withUnit: '1cm)  
(Tikzpicture withUnit: '0.5in)
```

The class method `new` will need to be changed, but plain `new` should continue to work with the default scale of 4pt.

5. *A new class of canvas.* In this problem, you replace `TikzCanvas` with a different back end.

- (a) Learn enough PostScript commands to draw pictures containing polygons and ellipses. For a whole picture, you'll need PostScript commands `%!PS`, `scale`, `setlinewidth`, and `showpage`. For polygons, you'll need `newpath`, `moveto`, `lineto`, `closepath`, and `stroke`. You can draw circles using `arc`, `closepath`, and `stroke`; to draw an ellipse, save the context using `gsave` and `grestore`, then use `scale` to adjust the *x* and *y* radii and draw a circle.
- (b) To draw PostScript pictures, define a new class `PSCanvas`. It should implement the same protocol as `TikzCanvas`, and it should produce PostScript.

#### 10.14.4 Creating new classes from scratch

6. *An object as a source of random numbers.* For random-number generation, define class `Rand`. An object of class `Rand` should have an internal *seed*, which is an integer, and should respond to the following protocol:

- Class method `new`: creates a new random-number generator with the given seed.
- Instance method `next` answers the random number and updates the seed.

To generate the next random number, use the algorithms from page 126.

7. *Strings.* Define a class `SymbolString` which represents a string of symbols.

- (a) Define a `^` method that concatenates a string or a symbol to a string of symbols. Should the class be mutable or immutable?
- (b) Now extend `Symbol` so that `^` is defined on symbols as well. Should the class be mutable or immutable?

#### 10.14.5 Messages to super

8. *Correct initialization of new pictures.* Section 10.3.4 says that sending `new` to a class should answer a newly allocated object whose state respects the private invariants of the class. For example, a private invariant of the class `Picture` is that instance variable `shapes` refers to a list of shapes. But the definition of `Picture` in Figure 10.4 simply inherits `new` from `Object`, and `new` returns an *uninitialized* `Picture` object. The class definition does not include `new` because in such an early example, I did not want to introduce the technique of sending messages to `super`. Fix the problem such that `new` creates a picture with an empty list of shapes, by sending `empty` to `self`. You will also need to change the `empty` class method to use `super`.
9. *Alternative semantics for super.* Here's a different way that `super` could be implemented: when sending message `m` to `super`, send it to `self`, but start the method search in the superclass of `self`'s class. Compare this implementation with the way `super` is actually implemented, and explain how they are different. Find an example from this chapter in which the incorrect version does the wrong thing.

You may find it expedient to implement the incorrect version.

### 10.14.6 Equality

10. *Object identity for keys in collections.* The collection classes compare keys with equivalence (`=`). What if you really wanted object identity (`==`)? You don't have to rewrite all the code. Instead, try the following:

- (a) Define a `Wrapped` class that has the following properties:

```
((Wrapped new: anObject) value)      == anObject  
(Wrapped new: x) = (Wrapped new: y)) == (x == y)
```

- (b) Define a new collection class that inherits from one of the existing classes, but which overrides methods to use wrapped keys.

### 10.14.7 Method redefinition

11. *Pictures with non-integer coordinates.* The `print` method in the `CoordPair` class sends the `print` message to instance variables `x` and `y`. As long as `x` and `y` are integers, this works fine, but if they are any other kind of numbers, the L<sup>A</sup>T<sub>E</sub>X TikZ package won't be able to interpret the results. In this exercise you modify both the `Float` and the `CoordPair` classes to support shapes and positions with non-integer values.

- (a) Redefine the `print` method on the `Float` class so that it prints `self` as an integer part, followed by a decimal point, followed by two decimal digits, as follows:

726. *<exercise transcripts 726>*≡  
728a▷  
-> (69 asFloat)  
69.00  
-> ((-3 / 4) asFloat)  
-0.75  
-> ((314 asFloat) / (100 asFloat))  
3.14

Send message `addSelector:withMethod:` to class `Float` with two arguments: the literal symbol '`print`' and a `compiled-method` expression with the new method definition.

- (b) Establish a new invariant for class `CoordPair`: that the values of instance variables `x` and `y` are always numbers of class `Float`. Modify only methods of class `CoordPair`, not methods of any other class.  
(c) Make sure the public methods of class `CoordPair` work with arguments of *any* numeric class, not just integers or `Float`s.

12. *More informative print method for pictures.* The `print` method of class `Picture` would be more interesting if it showed the shapes inside the picture, like so:

```
-> pic  
Tikzpicture( <Circle> <Square> <Triangle> )
```

Make it so. (You might wish to study the `print` method for class `Collection`.)

### 10.14.8 New methods on existing classes

The exercises in the next group explore changes, extensions, and additions to the predefined classes. To modify a predefined class, you have two choices:

- Edit the source code of the interpreter.
- Send the class the `addSelector:withMethod:` message, giving it the method name (as a literal symbol) and a compiled method. Exercise 11 on page 726 has an example.

### §10.14. Exercises

727

The exercises are as follows:

13. *Method `isMeta` for all classes.* In a full Smalltalk system, every class answers the message `isMeta`, saying if it is a metaclass. Define as many `isMeta` methods as you need to, then use reflection to update  $\mu$ Smalltalk's predefined classes so that every *class* object responds appropriately to `isMeta`.
14. *Method `ifNil:` for all objects.* In a full Smalltalk system, message `ifNil:` can be sent to any object with one argument, a block. The receiver answers itself, unless the receiver is `nil`, in which case it answers the result of sending `value` to the argument block. Define as many `ifNil:` methods as you need to, then use reflection to update  $\mu$ Smalltalk's predefined classes so that every object responds appropriately to `ifNil:`.
15. *Give every object a hash method.* Using reflection, update the predefined classes so that every object responds to the `hash` message with a small integer. Equal objects must hash to equal values. For bonus points, add a primitive method that enables a symbol to hash to something useful.

### 10.14.9 Conditional tests via dynamic dispatch

16. *Conditional operations on class `False`.* Study the implementation of class `True` in Section 10.5, and write the corresponding definition of class `False`.

### 10.14.10 Collections

17. *Pictures as collections.* A `Picture` is, among other things, a collection of shapes. Perhaps it should understand the entire `Collection` protocol?
  - (a) Redefine `Picture` to make it inherit from `Collection`. You could choose to inherit directly from `Collection` or from any of `Collection`'s (transitive) subclasses.
  - (b) Which is the better design: `Picture` as defined in Section 10.1, or `Picture` as a specialized collection? Justify your answer.
18. *A class Interval of integer sequences.* Define class `Interval` as a subclass of class `SequenceableCollection`. An object of class `Interval` represents a finite sequence of integers in arithmetic progression. Such a sequence takes the form  $[n, n + k, n + 2k, \dots, n + mk]$  for some  $n > 0$ ,  $k > 0$ , and  $m \geq 0$ . An interval is defined by  $n$ ,  $m$ , and  $k$ ; intervals are immutable. The instance protocol of class `Interval` is the same as other `SequenceableCollections`, except that it lacks `add:`. The class protocol should provide two initializing methods: `from:to:` and `from:to:by:`. Sending `(from:to:by: Interval n (n + mk) k)` results in an interval containing the sequence shown above; `(from:to: Interval n (n + m))` uses  $k = 1$ .

19. *Using Interval for array indices.* Use class `Interval` to implement methods `associationsDo:`, `collect:`, and `select:` on arrays, and to re-implement `do::`.
20. *Class method withAll: for arrays.* Add to class `Array` a class method `withAll: aCollection` that makes an array out of the elements of another collection. To add a class method using reflection, you send `addSelector:withMethod:` to `Array`'s *metaclass*, which you get by evaluating (`Array class`).
21. *Methods select: and collect: for arrays.* Add implementations of `select:` and `collect:` to class `Array`.
22. *Collection class Bag.* Define the collection class `Bag`. An object of class `Bag` is a grocery bag; it's an unordered collection of elements. Unlike a set, a bag may contain the same element multiple times, as in two cartons of milk; bags are sometimes called *multisets*. The protocol for `Bag` is the same as for `Set`, plus a method `count::`, which tells how many copies of a given object are stored in the bag. For example:

**728a.** *(exercise transcripts 726)* +≡

△726 728b ▷

```
-> (val B (Bag new))
-> (B add: 'milk)
-> (B add: 'milk)
-> (B add: 'macaroni)
-> (B count: 'milk)
2
```

Iterating through a bag should visit every item:

**728b.** *(exercise transcripts 726)* +≡

△728a

```
-> (B do: [block (x) ('Bagged: print) (space print) (x println)])
Bagged: milk
Bagged: milk
Bagged: macaroni
nil
```

Choose `Bag`'s superclass so as to require as little new code as possible, but do not change any existing classes.

23. *Arrays that can change size.* Fixed-size arrays are a nuisance. Using the design in Section 9.6.2 as a model, create an `ArrayList` class that inherits from `SequenceableCollection` and that represents arrays with indices from  $n$  to  $m$ , inclusive. It should also support the following new messages.

- The class method `new` should create a new, empty array with  $n = 1$  and  $m = 0$ .
- The instance method `addlo:` should decrease  $n$  by 1 and add its argument in the new slot. The instance method `addhi:` should increase  $m$  by 1 and add its argument in the new slot. The method `add:` should be a synonym for `addlo::`.
- The instance method `remlo` should remove the first element, increase  $n$  by 1, and answer the removed element. The instance method `remhi` should remove the last element, decrease  $m$  by 1, and answer the removed element.

All these operations should take constant time in the normal case, as should `at::`.

For suggestions about representation, look at the implementation of this abstraction using Molecule in Appendix T.

24. *Lists implemented by arrays.* The `ArrayList` class from Exercise 23 could be used to implement `List`.
  - (a) Estimate the space savings for large lists, in both the average case and the worst case.
  - (b) Write an implementation.

Redefine the `detect:ifNone:` method on `Collection` so as to avoid the final `if`.

25. *Complete class List.*
  - (a) Implement `removeLast` for class `List`.
  - (b) Implement `removeKey:ifAbsent:` for lists.
26. *Fix `at:put:` on class List.* If the index is out of range, the `at:put:` method on class `List` silently produces a wrong answer. What an embarrassment! Try the following:
  - (a) Supplement the existing implementation with an explicit range check.
  - (b) Throw away the existing code, and in its place devise a solution that delegates the work to classes `Cons` and `ListSentinel`.
  - (c) Compare the two solutions above, say which you like better, and why.

27. *Refactor keyed collections.* Increase code reuse in class `KeyedCollection` by introducing a private `detectAssociation:ifAbsent:` method that takes two blocks as arguments, then using it to implement other methods. Explain why this change might be a good thing even if it makes the code harder to understand. *Hint: what about implementations that use hash tables?*

Aggressive reuse often makes the code harder to follow, because the implementation is divided into more small pieces, which are probably distributed over the definitions of several classes. If every bit of duplicated code is abstracted out into its own method, every nontrivial computation appears to be composed of many methods, each of which does almost nothing. The learning curve for such a system can be quite high. This problem is characteristic of aggressively object-oriented systems.

Argue whether your change to `KeyedCollection` is an improvement.

28. *Hash tables.* Define a class `Hash` that implements a mutable finite map using a hash table. Objects of class `Hash` should respond to the same protocol as objects of class `Dictionary`, except that a `Hash` object is entitled to assume that the `hash` message may be sent to any key. A key object answers the `hash` message with a machine integer that never changes, even if the state of the key changes. Choose a suitable  $\mu$ Smalltalk class to inherit from, and use the representation and the invariants from the hash table in Exercise 34 in Chapter 9 (page 607).
29. *Binary-search trees.* Define a class `BST` that implements a mutable finite map using a binary search tree. Objects of class `BST` should respond to the same protocol as objects of class `Dictionary`, except that a `BST` object is entitled to assume that keys respond to the `Magnitude` protocol.

30. *Constant-time at:ifAbsent: for arrays.* Class `Array` inherits `at:ifAbsent:` from `SequenceableCollection`. This implementation costs time linear in the size of the array. Without defining any new primitives, build a new implementation that takes constant time:

- Using `removeSelector:`, remove the definition of `at:` from class `Array`, so it inherits `at:` from `SequenceableCollection`.
- On class `Array`, implement `at:ifAbsent:` using primitive `arrayAt` and the other methods of class `Array`.

31. *An iterator for locations in shapes.* Examine the algorithm for drawing polygons, and look at what kinds of intermediate data are allocated. Sending `locations:` to an object of class `Shape` allocates an intermediate list of locations, which is used in `drawPolygon:` on class `Tikzpicture`, then immediately thrown away. Eliminating this sort of allocation often improves performance.

Class `Shape` doesn't have to provide a *list* of the locations; it could equally well provide an *iterator*, with this protocol:

```
locationsDo:with: symbols aBlock
  For each symbol s in the collection symbols, send
    (aBlock value: l), where l is the location of the control point
    named by symbol s. If s does not name any control point, the
    result is a checked run-time error.
```

- Extend class `Shape` with a `locationsDo:with:` method.
- Suppose that `locations:` were eliminated, so that all client code had to use the `locationsDo:with:` method. What other classes would have to change, in what ways, to continue to draw polygons?
- Is there a way to change the other classes so that no intermediate list (or other collection) is allocated? If so, explain what other intermediate objects have to be allocated instead. If not, explain why not.
- If your answer to the previous part is “yes, the other classes can be changed,” then explain whether you prefer the new design or the original, and why.

#### 10.14.11 New magnitudes

32. *Characters as magnitudes.* In full Smalltalk-80, a character is a `Magnitude`, but in  $\mu$ Smalltalk, it's not. Redefine class `Char` to be a subclass of `Magnitude`. To be sure it implements the full protocol for magnitudes, write unit tests.

33. *More magnitudes.* In  $\mu$ Smalltalk, the only interesting `Magnitudes` are `Numbers`. Don't stop at `Char`—define more `Magnitudes`!

- Define `Date` as a subclass of `Magnitude`. A `Date` is given by a month, day, and year. Define a `+` method on `Date` that adds a number of days to the date. This method should know how many days are in each month, and it should also know the rules for leap years.
- Define `Time` as a subclass of `Magnitude`. A `Time` object represents a time on a 24-hour clock. Define a `+` method that adds minutes.

### 10.14.12 Numbers

34. *Arithmetic overflow in Fraction.* In class `Fraction`, could redefining `<` in terms of `-` reduce the possibility of overflow? Justify your answer.
35. *Arithmetic overflow in Float.* As defined in Section U.1.7, floating-point addition might overflow. Re-implement floating-point addition so that it neither overflows nor loses precision unnecessarily. For example, you shouldn't lose precision adding 0.1 to 1. Here are some suggestions:

- Make each exponent as small as it can be without overflowing.
- Then use the larger of the two exponents.

§10.14. Exercises

731

36. *Mixed arithmetic with integers and fractions.* This problem explores improvements in the built-in numeric classes, to try to support mixed arithmetic.

- (a) Use reflection to arrange the `Fraction` and `Integer` classes so you can add integers to fractions, subtract integers from fractions, multiply fractions by integers, etc. That is, make it possible to perform arithmetic by sending an integer as an argument to a receiver of class `Fraction`. Minimize the number of methods you add or change.
- (b) Use reflection to change the `Integer` class so you can add fractions to integers—that is, use a fraction as an argument to a `+` message sent to an integer. This requires much more work than part 36(a). You might be tempted to change the `+` method to test to see if its argument is an integer or a fraction, then proceed. A better technique is to use *double dispatch*: the `+` method does nothing but send a message to its argument, asking the argument to add `self`. The key is that the message encodes the type of `self`. For example, using double dispatch, the `+` method on `Integer` might be defined this way:

```
731. ⟨double dispatch 731⟩≡  
(method + (aNumber)  
  (aNumber addIntegerTo: self))
```

The classes `Integer`, `Float`, and `Fraction` would all then define methods for `addIntegerTo::`.

- (c) Complete your work on the `Integer` class so that all the arithmetic operations, including `=` and `<`, work on mixed integers and fractions. Minimize the number of new messages you have to introduce. You may wish to use reflection to change or remove some existing methods on `Integer` and `SmallInteger`.
- (d) Finish the job by making `Fraction`'s methods answer an integer if the denominator of the fraction is 1.
37. *Arbitrary-precision natural-number arithmetic.* An object of class `Natural` represents a natural number of *any size*; the protocol for natural numbers is shown in Figure 10.19 on page 662. Study the discussion of natural-number arithmetic in Appendix B, which starts on page S15. Then study the discussion of object-oriented implementations of natural numbers in Section 10.7.5 on page 678. Finally, implement natural-number arithmetic by completing class `Natural`, which is shown in Figure 10.23 on page 678. You might start by implementing `=` and `<` using the other methods of the class, which you can do independent of your choice of representation.

38. *Arbitrary-precision signed-integer arithmetic.* Using sign-magnitude representation, implement large signed integers, as described in Section 10.7.3. Define whatever methods are needed to answer not only the large-integer protocol but also the Integer, Number, Magnitude, and Object protocols, except for `div:` and `mod::`. You may find it most useful to focus on methods `print`, `isNegative`, `isNonnegative`, `isStrictlyPositive`, `negated`, `+`, `*`, and `sdiv::`.

So that small-integer arguments can be passed to large-integer methods, use reflection to add an `asLargeInteger` method to class `SmallInteger`.

39. *Transparent failover from machine-integer arithmetic to arbitrary-precision arithmetic.* Use reflection to change the definition of class `SmallInteger` such that if arithmetic overflows, the system automatically moves to large integers. Furthermore, make sure that arithmetic between small and large integers works transparently as needed, and that small integers support the same `sdiv:` and `smod:` messages supported by large integers.

You will need the following primitives:

- Evaluating (`primitive addWithOverflow self aSmallInteger ovBlock`) computes the sum of the receiver and the argument `aSmallInteger`. If this computation overflows, the result is `ovBlock`; otherwise it is a block that will answer the sum.
- Evaluating (`primitive subWithOverflow self aSmallInteger ovBlock`) computes the difference of the receiver and the argument `aSmallInteger`. If this computation overflows, the result is `ovBlock`; otherwise it is a block that will answer the difference.
- Evaluating (`primitive mulWithOverflow self aSmallInteger ovBlock`) computes the product of the receiver and the argument `aSmallInteger`. If this computation overflows, the result is `ovBlock`; otherwise it is a block that will answer the product.

These primitives require a level of indirection; instead of answering directly with a result, they answer a block capable of returning the result. This property is needed to arrange for the proper execution of the recovery code, but it makes the use of these primitives a bit strange. Here's an example, which includes the analog of the Smalltalk-80 code on page 713:

**732a.** *(example use of primitives with overflow recovery 732a)≡*

```
(SmallInteger addSelector:withMethod: '+'
  (compiled-method (aNumber) (aNumber addSmallIntegerTo: self)))
(SmallInteger addSelector:withMethod: 'addSmallIntegerTo:
  (compiled-method (anInteger)
    ((primitive addWithOverflow self anInteger
      {((self asLargeInteger) + anInteger)} value))))
```

A familiar way to get large integers is to compute factorials. Factorials make a poor test, because they test only multiplication, and they never multiply two large integers. But they make a good start, so here is a `Factorial` class you can use:

**732b.** *(large-arithmetic transcript 732b)≡*

```
-> (class Factorial
  [subclass-of Object]
  (class-method printUpto: (limit) [locals n nfac]
    (set n 1))
```

```

(set nfac 1)
({(n <= limit)} whileTrue:
  {(n print) (! print) (space print) ('= print)
   (space print) (nfac println)}
  (set n (n + 1))
  (set nfac (n * nfac)))))

-> (use mixed.solution)
-> (use bignum.solution)
-> (Factorial printUpto: 20)
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
nil

```

#### 10.14.13 Method dispatch

40. *Formal semantics for method dispatch.* In Section 10.9.1, I present the judgment form  $m \triangleright c @ imp$ , which describes method dispatch. We didn't give any proof rules for this form, but there is a function `findMethod` in chunk 698b which is supposed to implement it. Using the implementation as a guide, write inference rules for the judgment.
41. *Method cache.* Because Smalltalk does everything with method dispatch, method dispatch has to be fast. Since all computation, even integer arithmetic, is done by message-passing, unnecessary overhead hurts performance. The overhead can be reduced by *method caching*, as described by Deutsch and Schiffman (1984). Method caching works like this: with each SEND node in an abstract-syntax tree, associate a two-word cache. One word gives the class of the last object to receive the message sent, and the other gives the address of the method to which the message was last dispatched. When the message is sent, consult the cache; if the class of the current receiver is the same as the class of the cached receiver, then the cached method is the one to be executed, and no method search is needed. If the class of the current receiver is different, do the usual method search and update the cache.

Method caching saves time if each SEND in the source code usually sends to an object of a single class—and research suggests this is true about 95% of the

time. Conroy and Pelegri-Llopert (1982) report that adding a method cache made the early Berkeley Smalltalk system run 37% faster.

- (a) Extend the  $\mu$ Smalltalk interpreter with method caches.
- Define a type `classid` whose value can uniquely identify a class. Since the `class` field of each class is unique, you can start with  
`type classid = metaclass ref`
  - Add a field of type `(classid * method) ref` to the `SEND` node in the abstract syntax. This reference cell should hold a pair representing the unique identifier of the last class to receive the message at this call site, plus the method found on that class.
  - Change the parser to initialize each cache. As the initial class identifier, define  
`val invalidId : classid = ref PENDING`  
Value `invalidId` is guaranteed to be distinct from the `class` field of any class.
  - Change function `ev` in chunk 697b to use the cache before calling `findMethod`, and to update the cache afterward (when a method is found). Consider defining a function `findCachedMethod`, whose type might be `(classid * method) ref * name * class -> method`.
- (b) Add code to gather statistics about hits and misses, and measure the cache-hit ratio for a variety of programs.
- (c) Find or create some long-running  $\mu$ Smalltalk programs, and measure the total speedup obtained by caching.
- (d) If a  $\mu$ Smalltalk program uses the `setMethod` primitive, or any of the reflective facilities that copy methods, it could invalidate a cache. Either eliminate the primitive or correct your implementation so that every use of the primitive invalidates caches that depend on the relevant class. Don't forget that changing a method on class  $C$  may invalidate not only caches holding  $C$  but also caches holding  $C$ 's subclasses.  
Measure the cost of the additional overhead required to invalidate the caches.

#### 10.14.14 Object-oriented profiling

42. *Profiling.* Implement a message-send profiler that counts the number of times each message is sent to each class. Use this profiler to identify hot spots in the simulation code from Appendix U.

## Afterword

*It's a magical world, Hobbes, ol' buddy... let's go exploring!*

Calvin

Congratulations! You now have some solid skills using functions, types, modules, objects, and so on. You now have a cognitive framework that you can use to learn new programming languages, and if you're like my students, you'll be pleasantly surprised at how broadly your skills apply already. This afterword presents a sort of dessert menu of languages and topics that you might explore next. I mention some specific languages you might like to explore: the superlative, the unusual, and the popular. Some are languages that most educated people would agree are interesting or important, and some are just to my personal taste. Together, they offer a variety of jumping-off points, ranging from younger, less proven ideas to fashionable ideas drawn from recent headlines.

### 11.1 TYPEFUL PROGRAMMING

If you like types, try **Haskell**. Haskell is a *pure* functional language: even I/O interactions are represented as values with special IO types. And Haskell is pushing new type-system ideas into the mainstream far faster than any other language. By the time this book is published, there will be cool new features I haven't even heard of yet. Start with the basics: *type classes* and *monads* (Lipovača 2011). Definitely try QuickCheck (Claessen and Hughes 2000; Hughes 2016), which uses Prolog-like inference rules to do amazing things with random testing. And don't overlook QuickCheck's *shrink* function—although it's barely mentioned in the original paper, it's a crucial part. I'm not sure what should be next after that, but I admire the work of Stephanie Weirich and Richard Eisenberg.

With *dependent types*, you can express more properties of your data. The classic example is to keep track of how long a list is, without losing polymorphism. Interesting dependently typed languages include **Idris** and **Agda**. There's not as much activity going on around **Epigram**, but the papers and tutorials are very good (McBride 2004).

Types are also being used to manage memory: we can have safe, fast systems programming without (much) garbage collection. Look into **Rust**, and checkout earlier work on **Cyclone** (Grossman et al. 2005).

### 11.2 PROPOSITIONS AS TYPES

But at the start of the 21st century, proofs about correctness are built on a foundation that starts with the idea of *propositions as types*.

Table 11.1: Connectives and truth values of classical logic with their counterparts in types

Table 11.1 shows a correspondence between propositions in classical logic and types in a functional language. In logic, symbols  $A$  and  $B$  represent propositions; in language,  $A$  and  $B$  represent types. The correspondence goes deep; to explore it, let's look at *derivations* in both logic and type systems. For example, in logic, we prove  $A \wedge B$  by proving both  $A$  and  $B$ . In language, we introduce a term of type  $A \times B$  using terms of types  $A$  and  $B$ . The last steps of the derivations are the same: and-introduction corresponds to pair-introduction:

$$\frac{T \vdash A \quad T \vdash B}{T \vdash A \wedge B} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}$$

In logic, symbol  $T$  represents a *theory*, and it lists all the proposition whose truth is assumed. In language, symbol  $\Gamma$  represents a *typing context*, and it lists all the types whose inhabitation is assumed. (Because the context gives a variable of each type, we know the type is inhabited by the value of that variable.)

As another example, in logic, to prove  $A$  implies  $B$ , we assume  $A$ , and we use the assumption to prove  $B$ . In language, to create a value of type  $A \rightarrow B$ , we define a function that takes a value of type  $A$  and returns a value of type  $B$ . Again, the structure of the derivations is the same:

$$\frac{T, A \vdash B}{T \vdash A \supset B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A. e : A \rightarrow B}$$

Almost every classical logical connective enjoys a corresponding type constructor, except one: logical complement. Programming-language types correspond to what is called *constructive* or *intuitionistic* logic. Like Prolog, such a logic concerns itself not with what is true, but with what is provable. What this has to do with programming languages is that the provable is the computable.

- A *type* in a programming language corresponds to a *proposition* in logic.
- A *term* in a programming language corresponds to a *proof* in logic. But caution! For the proof to be any good, the term must evaluate to completion. To write such proofs, scientists use programming languages in which *all* evaluations are guaranteed to terminate. And you almost know such a language: if you remove `val-rec` and `letrec` from Typed  $\mu$ Scheme, then the evaluation of any well-typed expression is guaranteed to terminate.

The propositions-as-types principle has led to a host of automated theorem provers that are also programming languages. In the late 1980s, this idea was popularized by Nuprl (Constable et al. 1985); in the mid-2010s, much work is done in

**Coq.** These systems are generally considered “proof assistants” rather than “programming languages,” but the line is blurry; for example, similar proofs can be written in the dependently typed languages **Agda** and **Idris**.

For deep mathematical background on the connections between propositions and types, as well as connections to decidability and models of computation, read [Wadler \(2015\)](#). To use your knowledge of functional programming to start proving theorems for yourself in Coq, tackle *Software Foundations* (Pierce et al. 2016).

§11.3

*More functions*

739

### 11.3 MORE FUNCTIONS

Among functional languages, **OCaml** includes some interesting design experiments, and it’s cool among systems programmers, financial engineers, and many academics. **Clojure** is Lisp on the Java Virtual Machine, with interesting support for multithreading. There’s also Haskell of course, and you ought to play with **Racket**, which is pushing ideas of Scheme and macros to the limit. **Racket** isn’t just a language or a system; it’s a factory for making new languages (Flatt 2012).

### 11.4 MORE OBJECTS

Objects are everywhere. If you know Smalltalk, you also know **Ruby**, but the programming environment is very different. If you download Squeak Smalltalk, you get a lovely graphical programming environment, but it’s not really connected to anything else on your computer. It’s like having a fabulous house—on Mars. Ruby gives you the same fabulous stuff, but right next door.

Then there are the hybrids. **Objective C** seems most faithful to Smalltalk’s vision. If you prefer complexity to simplicity, you’re ready for **Python**. Add more static types and you have **Java**, **C#**, **C++**, and many other popular languages.

It’s more interesting to explore languages that do objects with *prototypes* instead of classes. **Self** is the original and a great place to start (Ungar and Smith 1987). Once you have the ideas, you can apply them in **JavaScript**, whose good parts combine ideas from Scheme and Self (Crockford 2008), and in **Lua**, a great scripting language mentioned below.

### 11.5 FUNCTIONS AND OBJECTS, TOGETHER

Lots of designers have been interested in getting functions and objects together in the same language. **OCaml** was an early player, combining what was then known about objects and functions into a single system. More ambitious efforts include **Scala** and **F#**, each of which integrates an existing object-oriented type system (respectively the Java Virtual Machine and the Microsoft Common Language Runtime) with ideas from functional programming.

### 11.6 FUNCTIONAL ANIMATION

A *functional reactive animation* is a program that implements animation or even interaction using pure functional code. A very nice exemplar is the **Elm** programming language. It runs right in your web browser, and there’s a great community.

## 11.7 SCRIPTING

My favorite scripting language is **Lua**. It is lean, fast, and simple—what Scheme might have been if it had used Pascal’s syntax and the central data structure were the mutable hash table instead of the cons cell (but without macros). Lua tables work like CLU arrays, except they are also hash tables, and they can even be (prototype-based) objects! Lua ships with a nice, simple string-processing library. And Lua is an *embedded* language: you can easily import any library written in C, and you can drop Lua into any C program and use it for scripting and control. Most of Adobe Lightroom, for example, is written in Lua. There is a fine book (Ierusalimschy 2013) and a nice paper about the design and history (Ierusalimschy, de Figueiredo, and Celes 2007).

## 11.8 PARALLEL AND DISTRIBUTED COMPUTATION

If you want programs that run parallel, or especially if you want programs that run distributed over multiple computers, try **Erlang**. It’s got a simple, effective message-passing model of concurrency, plus great ideas about recovering from failure—and if you’re distributing your computations, you have to worry about failure. Plus it’s easy to pick up, as it uses Prolog syntax and data, with a computational module somewhat like Scheme.

For more message-passing concurrency, but with a very different flavor, try **Go**. Go’s primary mission is to “make systems programming fun again,” and the concurrency model is sweet.

## 11.9 ONE WEIRD, COOL, DOMAIN-SPECIFIC LANGUAGE

The language **Inform7** is designed for writing interactive fiction, or as we used to call it, text adventure games (ATTACK DRAGON WITH BARE HANDS and all that). What’s cool is that an Inform7 program doesn’t really look like a program; it looks like a cross between a manual and a text adventure game. And it ships with a really interesting interactive programming environment. I’m still wondering exactly what the abstract syntax of Inform7 is—or if it even uses an abstract syntax—but if you have any interest in this domain, you have to try it.

## 11.10 STACK-BASED LANGUAGES

Chapter 3 uses an evaluation stack to implement  $\mu$ Scheme+, but the stack is hidden from the programmer. But stacks can be exposed. The classic stack-based language is **Forth**, which gets extraordinary power from a minimal design. A Forth environment can fit in a few kilobytes of RAM, and Forth has been used on many embedded systems. For a nice example, check out the Open Firmware project.

If you like your stacks with a few more data types, write some **PostScript**. Sweep away all the primitives related to fonts and graphics, and you’ll find a very nice stack-based language underneath. Even the current environment is represented as a stack, and by manipulating it, you can quickly change the meanings of names *en masse*.

## 11.11 ARRAY LANGUAGES

While John McCarthy was developing his list-based language, Ken Iverson was developing the array-based language **APL** (Iverson 1962).<sup>1</sup> Derived from notation used at the blackboard, APL evolved into an important language at IBM, where it was written by using a special typeball on IBM's Selectric typewriter. At a time when almost all IBM's mainframe business meant programming with punched cards, APL gave IBM a powerful interactive option. APL originated many ideas that were fully developed in functional languages, including maps, filters, and folds. You can also check out Iverson's successor language, **J**, which may look like line noise, but which runs on ordinary computers. Both APL and J enable you to define array functions that are polymorphic not just in the size of an array, but in the number of its dimensions. Slepak, Shivers, and Manolios (2014) explain this behavior using a modern, static type system of the sort you have mastered.

## 11.12 LANGUAGES BASED ON SUBSTITUTION

When you instantiate a polymorphic type in Typed  $\mu$ Scheme, you have to implement capture-avoiding substitution (Section 6.6.7 on page 380). That same substitution is the computational engine behind lambda calculus. Substitution is hard to get right, hard to reason about, and inefficient. So it's astounding how many eminent computer scientists—and uneducated hackers—have designed languages around substitution. Notable examples include TeX (used to typeset this book); Tcl/Tk (used to make GUIs quickly); the POSIX, Bourne, Korn, and Bourne Again shells (used to script Unix systems), and PHP (used to make web pages). Each of these languages does something well, but for general-purpose programming, they are no fun. For Facebook, PHP is so much not fun that Facebook is developing a language called Hack, whose mission is to embrace legacy PHP code while adding static types—a real language-design challenge.

A related language, based more on string rewriting, was TRAC (Mooers 1966); if you're intrigued by text macros, as opposed to syntax macros, it's worth a look. A clone of TRAC, MINT (for MINT Is Not TRAC), shipped as the extension language of Russ Nelson's Emacs clone, Freemacs.

## 11.13 STRING-PROCESSING LANGUAGES

Domain-specific languages for string processing are in decline, but **Icon** is worth looking at: it uses a *string scanning* technique that relies on a backtracking evaluation model just like the one used in Prolog. The great thing about string scanning is that, unlike regular expressions, you can extend string scanning with your own string-processing abstractions, and they compose nicely with the built-in abstractions.

For historical interest only, I recommend **Awk**, which was important early and for a long time, although it is both dominated and superseded by later languages. The code I used to write in Awk I now write in Lua.

Not quite relegated to historical interest is Perl, which has first-class functions like Scheme, multiple name spaces like Impcore, and string processing based on regular expressions. Happily, Perl has become unfashionable, but there is plenty of legacy Perl code out there—to which your skills apply.

---

<sup>1</sup>Another Turing Award.

## 11.14 CONCLUSION

It really is a magical world. Go exploring!

## **IV. BACK MATTER**



## *Key words and phrases*

- $\lambda$ -bound variable (nano-ML), 453  
call/cc ( $\mu$ Scheme+), 246, 247  
self (Smalltalk), 718  
super (Smalltalk), 718  
cons, car, cdr (Scheme), 174
- abstract class (Smalltalk), 716  
abstract data type (Molecule), 595  
abstract machine (Impcore), 71  
abstract syntax (Impcore), 69–71, 73  
abstract-machine semantics  
    ( $\mu$ Scheme+), 246  
abstraction (Molecule), 596  
abstraction function (Molecule), 596  
abstraction function (Smalltalk), 718  
algebraic data type ( $\mu$ Scheme in ML),  
    325, 326  
allocation pointer (Garbage collection),  
    295  
answer (Smalltalk), 717  
answered (Smalltalk), 716  
aPI (Molecule), 596  
argument (Smalltalk), 716  
at least as general (nano-ML), 453  
atom (Prolog), S102  
atom (Scheme), 174, 176  
autognostic principle (Smalltalk), 716,  
    718  
automatic memory management (Garbage collection), 294, 296
- basis (Impcore), 72  
behavioral subtyping (Smalltalk), 719  
big-step semantics ( $\mu$ Scheme+), 246  
big-step semantics (Impcore), 70  
block (Smalltalk), 718  
bound ( $\mu$ Scheme in ML), 327  
bound variable ( $\mu$ Scheme in ML), 327
- call stack ( $\mu$ Scheme+), 246, 247  
call stack (Garbage collection), 293  
capsule (Molecule), 596  
class (Smalltalk), 716, 717  
class method (Smalltalk), 717  
clausal definition ( $\mu$ Scheme in ML), 325  
clause (Prolog), S102  
client (Molecule), 596
- closure (Scheme), 174–176  
closures (Scheme), 175  
collection (Smalltalk), 719  
compacting collector (Garbage collection), 295  
compiler (Impcore), 73  
completeness (Prolog), S103  
complex method (Smalltalk), 716, 718  
concrete syntax (Impcore), 70, 71, 73  
cons (Scheme), 175, 176  
cons cell (Scheme), 174  
conservative collector (Garbage collection), 295  
constraint (nano-ML), 453  
constraint solver (nano-ML), 453  
continuation ( $\mu$ Scheme+), 247  
continuation semantics ( $\mu$ Scheme+),  
    247  
control ( $\mu$ Scheme+), 246  
control operator ( $\mu$ Scheme+), 246  
control operators ( $\mu$ Scheme+), 246, 247  
copying (Garbage collection), 296  
copying collector (Garbage collection),  
    295, 296
- data abstraction (Molecule), 595  
data abstraction (Smalltalk), 716  
dead (Garbage collection), 294, 295  
dead object (Garbage collection), 295  
dead variable (Garbage collection), 295  
decidability (Prolog), S103  
decidable (Prolog), S101  
definitional interpreter (Impcore), 71,  
    73  
delegation (Smalltalk), 718  
delimited continuation ( $\mu$ Scheme+),  
    247  
derivation (Impcore), 71  
difference list (Prolog), S103  
divergence ( $\mu$ Scheme+), 246  
duck typing (Smalltalk), 719  
dynamic dispatch (Smalltalk), 717
- elimination form (Type systems), 394  
elimination rule (Type systems), 394  
embedding ( $\mu$ Scheme in ML), 326  
encapsulate (Molecule), 596

- encapsulation (Molecule), 595  
environment (Impcore), 69–71  
equality constraint (nano-ML), 453  
evaluation (Impcore), 71  
evaluation context ( $\mu$ Scheme+), 246, 247  
evaluation stack ( $\mu$ Scheme+), 247  
evaluation stack (Garbage collection), 294  
evaluator (Impcore), 70, 73  
exception ( $\mu$ Scheme in ML), 324, 327  
exception handler ( $\mu$ Scheme in ML), 325  
exhaustive pattern match ( $\mu$ Scheme in ML), 325  
explicit memory management (Garbage collection), 294  
expose the representation (Smalltalk), 718  
expression-oriented language (Impcore), 72  
  
fact (Prolog), S103  
filter (Scheme), 174, 175  
first class (Scheme), 175  
first-class (Scheme), 174, 175  
first-class function (Scheme), 174  
first-order function (Scheme), 174, 175  
first-order logic (Prolog), S103  
fold (Scheme), 175  
formation rule (Type systems), 394  
forwarding pointer (Garbage collection), 296  
free ( $\mu$ Scheme in ML), 327  
free list (Garbage collection), 295  
free variable ( $\mu$ Scheme in ML), 327  
free variable (Scheme), 174, 175  
functor (Prolog), S102  
  
garbage collection (Garbage collection), 294, 295  
garbage collector (Garbage collection), 295  
generalization (nano-ML), 453  
generalizing (nano-ML), 453  
generational collector (Garbage collection), 296  
generational hypothesis (Garbage collection), 296  
generativity (Type systems), 394  
generic module (Molecule), 596  
generic read-eval-print loop ( $\mu$ Scheme in ML), 325  
goal (Prolog), S102  
grammar (Impcore), 70  
ground term (Prolog), S102  
  
heap (Garbage collection), 294, 295  
heap allocation (Garbage collection), 294, 295  
heap graph (Garbage collection), 294, 295  
heap object (Garbage collection), 294–296  
higher-order (Scheme), 175  
higher-order function (Scheme), 174, 175  
higher-order functions (Scheme), 175  
hole ( $\mu$ Scheme+), 247  
  
implementation (Molecule), 596  
inference rule (Impcore), 71  
inhabitant (Type systems), 393  
inherit (Smalltalk), 716, 717  
inheritance (Smalltalk), 717  
initial basis (Impcore), 72  
instance (nano-ML), 453  
instance (Smalltalk), 716  
instance method (Smalltalk), 716  
instance variable (Smalltalk), 716  
instantiation (nano-ML), 453  
instantiation (Type systems), 393, 394  
interactivity ( $\mu$ Scheme in ML), 327  
interface (Molecule), 596  
introduction form (Type systems), 394  
introduction rule (Type systems), 394  
  
judgment (Impcore), 69–71  
judgment form (Impcore), 69–71  
  
kind (Type systems), 394  
  
lambda (Scheme), 174  
lambda abstraction (Scheme), 174–176  
let (Scheme), 174  
let binding (Scheme), 175  
let-bound variable (nano-ML), 453  
list constructor ( $\mu$ Scheme in ML), 325  
live (Garbage collection), 295, 296  
live data (Garbage collection), 294, 295  
live object (Garbage collection), 294, 295  
location (Scheme), 175, 176  
location semantics (Scheme), 175, 176  
locations (Scheme), 175, 176  
logic programming (Prolog), S101  
logical variable (Prolog), S102  
loop invariant (Molecule), 597  
  
magnitude (Smalltalk), 719  
managed heap (Garbage collection), 294, 295  
map (Scheme), 175  
maps (Scheme), 175  
mark bit (Garbage collection), 296  
mark-and-sweep (Garbage collection), 295

mark-and-sweep collection (Garbage collection), 296  
 mark-and-sweep collector (Garbage collection), 295, 296  
 memory safety (Garbage collection), 294  
 message (Smalltalk), 716–718  
 message passing (Smalltalk), 716, 717  
 message selector (Smalltalk), 716, 717  
 metalanguage ( $\mu$ Scheme in ML), 326,  
     327  
 metatheoretic (Impcore), 71  
 metatheoretic proof (Impcore), 72  
 metatheory (Impcore), 72  
 metavariable (Impcore), 69, 71  
 method (Smalltalk), 716–718  
 method dispatch (Smalltalk), 717, 718  
 model (Prolog), S103  
 module (Molecule), 596  
 monomorphic (Scheme), 175, 176  
 monomorphic (Type systems), 393  
 monomorphic type system (Type systems), 393  
 monotype (nano-ML), 453  
 monotype (Type systems), 394  
 most general (nano-ML), 453  
 multiple inheritance (Smalltalk), 717  
 mutable (Scheme), 175  
 mutable reference cell ( $\mu$ Scheme in ML), 326  
 mutator (Garbage collection), 294, 295  
 mutual recursion (code) ( $\mu$ Scheme in ML), 326  
 mutual recursion (data) ( $\mu$ Scheme in ML), 326  
 natural deduction (Impcore), 71  
 nested (Scheme), 174, 175  
 nested function (Scheme), 175  
 number (Smalltalk), 719  
 object (Prolog), S101  
 object (Smalltalk), 716  
 object graph (Garbage collection), 295  
 object language ( $\mu$ Scheme in ML), 326  
 object-language ( $\mu$ Scheme in ML), 326,  
     327  
 object-orientation (Molecule), 595  
 occurs check (Prolog), S103  
 open recursion (Smalltalk), 718  
 operational semantics (Impcore), 70, 71  
 override (Smalltalk), 717  
 parametric polymorphism (Type systems), 393, 394  
 parser (Impcore), 73  
 parsing (Impcore), 71  
 pattern ( $\mu$ Scheme in ML), 325  
 pattern match ( $\mu$ Scheme in ML), 325  
 pattern matching ( $\mu$ Scheme in ML), 325  
 polymorphic ( $\mu$ Scheme in ML), 326  
 polymorphic (Scheme), 175, 176  
 polymorphic (Type systems), 393  
 polymorphic type ( $\mu$ Scheme in ML), 326  
 polymorphic type system (Type systems), 393  
 polymorphism (Type systems), 393  
 polytype (nano-ML), 453  
 polytype (Type systems), 394  
 predefined function (Impcore), 72  
 predicate (Prolog), S102  
 predicate logic (Prolog), S101  
 primitive function (Impcore), 72  
 principal type (nano-ML), 453  
 principal type scheme (nano-ML), 453  
 private method (Smalltalk), 717  
 procedural programming (Impcore), 72  
 projection ( $\mu$ Scheme in ML), 327  
 proof rule (Impcore), 70, 71  
 property (Prolog), S102  
 proposition (Prolog), S102  
 propositional logic (Prolog), S101  
 protocol (Smalltalk), 716, 717, 719  
 quantified type (Type systems), 394  
 query (Prolog), S101  
 reachability (Garbage collection), 294  
 reachable (Garbage collection), 294  
 read-eval-print loop ( $\mu$ Scheme in ML),  
     327  
 read-eval-print loop (Impcore), 72  
 receiver (Smalltalk), 716–718  
 redefining (Smalltalk), 717  
 redex ( $\mu$ Scheme+), 246  
 reduction semantics ( $\mu$ Scheme+), 246  
 redundant pattern ( $\mu$ Scheme in ML),  
     325  
 reference counting (Garbage collection),  
     294, 296  
 reflection (Smalltalk), 718  
 relation (Prolog), S102  
 rely-guarantee reasoning (Molecule),  
     597  
 representation (Molecule), 595, 596  
 representation (Smalltalk), 716  
 representation invariant (Molecule), 596  
 representation invariant (Smalltalk), 718  
 root (Garbage collection), 294–296  
 rule (Prolog), S103  
 s-expression (Scheme), 176  
 s-expressions (Scheme), 174  
 scrutinee ( $\mu$ Scheme in ML), 325  
 shared mutable state (Scheme), 176  
 short-circuit (Scheme), 176

*Key words and phrases*

747

- short-circuit conditional ( $\mu$ Scheme in ML), 326  
short-circuit evaluation (Scheme), 176  
simple equality constraint (nano-ML), 453  
single inheritance (Smalltalk), 717  
small-step semantics ( $\mu$ Scheme+), 246  
small-step semantics (Impcore), 71  
soundness (Prolog), S103  
stack allocation (Garbage collection), 294  
static scoping (Scheme), 175  
store (Scheme), 176  
structural induction (Impcore), 72  
structured operational semantics ( $\mu$ Scheme+), 246  
subclass (Smalltalk), 717, 718  
subclasses (Smalltalk), 717  
subgoal (Prolog), S102  
substituting (nano-ML), 453  
substitution (nano-ML), 453  
substitution (Prolog), S103  
subtyping (Molecule), 596  
superclass (Smalltalk), 717  
symbol table (Impcore), 70  
syntactic category (Impcore), 70  
syntactic form (Impcore), 70  
syntactic proof (Impcore), 71  
syntactic proof system (Impcore), 71  
syntactic sugar (Impcore), 70  
syntactic sugar (Scheme), 176
- term (Prolog), S102  
term (Type systems), 393  
the cut (Prolog), S103  
theory (Impcore), 69, 71  
tricolor marking (Garbage collection), 296  
type (nano-ML), 452, 453  
type (Type systems), 393, 394  
type abbreviation ( $\mu$ Scheme in ML), 326  
type abstraction (Type systems), 394  
type application (Type systems), 394  
type checker (Type systems), 393  
type constructor (Type systems), 394  
type inference (nano-ML), 452  
type reconstruction (nano-ML), 452  
type scheme (nano-ML), 452, 453  
type system (Type systems), 393  
type variable ( $\mu$ Scheme in ML), 326  
type-application (Type systems), 394  
  
undelimited continuation ( $\mu$ Scheme+), 247  
unification (nano-ML), 453  
unification (Prolog), S102  
unreachable (Garbage collection), 295  
  
value constructor ( $\mu$ Scheme in ML), 324, 325  
value semantics (Scheme), 175, 176  
  
work per allocation (Garbage collection), 295

## Bibliography

- Harold Abelson and Gerald Jay Sussman. 1985. *Structure and Interpretation of Computer Programs*. McGraw-Hill, New York.
- Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, MA.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools*. Pearson.
- Alfred V. Aho and Jeffrey D. Ullman. 1972. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-914556-7.
- Hassan Aït-Kaci. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.
- Lloyd Allison. 1986. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, Cambridge, England.
- ANSI. 1998. *American National Standards for Information Systems, INCITS 319-1998, Information Technology - Programming Languages - Smalltalk*. American National Standard Institute, New York.
- Andrew W. Appel. 1989 (February). Simple generational garbage collection and fast allocation. *Software—Practice & Experience*, 19(2):171–183.
- Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge.
- Andrew W. Appel. 1998. *Modern Compiler Implementation*. Cambridge University Press, Cambridge, UK. Available in three editions: C, Java, and ML.
- Zena M. Ariola, Huge Herbelin, and David Herman. 2011 (May). A robust implementation of delimited control. In *Theory and Practice of Delimited Continuations Workshop*, pages 6–19.
- Kenichi Asai and Oleg Kiselyov. 2011 (September). Introduction to programming with shift and reset. Notes for tutorial given at the ACM SIGPLAN Continuation Workshop.
- John Backus. 1978 (August). 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.
- Henry G. Baker. 1978 (April). List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294.
- W. Barrett, R. Bates, D. Gustafson, and J. Couch. 1986. *Compiler Construction: Theory and Practice*. SRA, Chicago, 2nd edition.
- Joel F. Bartlett. 1988 (February). Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC WRL, 100 Hamilton Avenue, Palo Alto, California 94301.
- Marianne Baudinet and David MacQueen. 1985 (December). Tree pattern matching for ML (extended abstract). Unpublished manuscript, AT&T Bell Laboratories.
- Kent Beck. 1997. *Smalltalk Best Practice Patterns*. Prentice Hall.
- Jon Bentley. 1983 (December). Programming pearls: Writing correct programs. *Communications of the ACM*, 26(12):1040–1045. Reprinted in (Bentley 2000, Chapter 4).
- Jon Bentley. 2000. *Programming Pearls*. Addison-Wesley, Reading, Mass, second edition.
- Jon L. Bentley and Robert Sedgewick. 1997. Fast algorithms for sorting and searching strings. In ACM, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, January 5–7, 1997*, pages 360–

- 369, New York, NY 10036, USA. ACM Press. ISBN 0-89871-390-0.
- Jon L. Bentley and Robert Sedgewick. 1998 (April). Ternary search trees. *Dr. Dobb's Journal*.
- Nick Benton and Andrew Kennedy. 2001 (July). Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410. ISSN 0956-7968. URL <http://dx.doi.org/10.1017/S0956796801004099>.
- Richard Bird and Philip Wadler. 1988. *Introduction to Functional Programming*. Prentice Hall, New York.
- G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. 1973. *Simula Begin*. Van Nostrand-Reinhold, New York, 1 edition.
- Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Markus Denker. 2009. *Squeak by Example*. Square Bracket Associates. Available under a Creative Commons license.
- Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2012*, pages 85–98, New York, NY, USA. ACM.
- Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, and Damien Pollet. 2010. *Pharo by Example (Version 2010-02-01)*. Square Bracket Associates. ISBN 978-3-9523341-4-0.
- Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004 (June). Myths and realities: The performance impact of garbage collection. *SIGMETRICS Performance Evaluation Review*, 32(1):25–36.
- D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. 1988 (September). Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI). ISSN 0362-1340.
- Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, M. Stefik, and F. Zdybel. 1986 (November). Common Loops, merging Lisp and object-oriented programming. *ACM SIGPLAN Notices*, 21(11):17–29. ISSN 0362-1340.
- Hans-Juergen Boehm and Mark Weiser. 1988 (September). Garbage collection in an uncooperative environment. *Software—Practice & Experience*, 18(9):807–820. Hans Boehm continues to maintain this collector, and a link can usually be found on his web pages, e.g., at [http://www.hpl.hp.com/personal/Hans\\_Boehm/gc](http://www.hpl.hp.com/personal/Hans_Boehm/gc) as of January 2003.
- George Boole. 1847. *The mathematical analysis of logic*. Macmillan, Barclay, & Macmillan, Cambridge. Project Gutenberg ebook number 36884.
- Per Brinch Hansen. 1994 (June). Multiple-length division revisited: A tour of the minefield. *Software—Practice & Experience*, 24(6):579–601.
- Frederick P. Brooks, Jr. 1975. *The Mythical Man-Month*. Addison Wesley, Reading, MA.
- Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996 (May). Representing control in the presence of one-shot continuations. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 31(5):99–107. ISSN 0362-1340.
- Timothy Budd. 1987. *A Little Smalltalk*. Addison-Wesley, Reading, MA. ISBN 0-201-10698-1.
- W. H. Burge. 1975. *Recursive Programming Techniques*. Addison-Wesley, Reading. ISBN 0-201-14450-6.
- R. M. Burstall, J. S. Collins, and R. J. Popplestone. 1971. *Programming in POP-2*. Edinburgh University Press, Edinburgh, UK.
- Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. 1980 (August). Hope: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143. ACM, ACM.
- Paris Buttfield-Addison, Jon Manning, and Tim Nugent. 2016. *Learning Swift: Building Apps for OS X and iOS*. O'Reilly.
- Lawrence Byrd. 1980. Understanding the control flow of Prolog programs. In S.-A. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–138. See also University of Edinburgh technical report 151.
- Byte. 1981 (August). Special issue on Smalltalk. *Byte Magazine*, 6(8). ISSN 0360-5280. Can be downloaded from [archive.org](http://archive.org).

- Howard I. Cannon. 1979. Flavors: A non-hierarchical approach to object-oriented programming. Unnumbered, draft technical report, variously attributed to the MIT AI Lab or to Symbolics, Inc.
- Luca Cardelli. 1987 (April). Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172. ISSN 0167-6423. URL <http://research.microsoft.com/Users/lucu/Papers/BasicTypechecking.ps>.
- Luca Cardelli. 1989 (February). Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State of the Art Reports Series. Springer-Verlag. Also appeared as DEC SRC Research Report 45.
- Luca Cardelli. 1997. Type systems. In Jr. Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL.
- Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. 1992 (August). Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42.
- Craig Chambers. 1992 (March). *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, California. Tech Report STAN-CS-92-1420.
- Craig Chambers and David Ungar. 1989 (July). Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, pages 146–160.
- C. J. Cheney. 1970. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–78.
- Koen Claessen and John Hughes. 2000 (September). QuickCheck: a lightweight tool for random testing of Haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, in *SIGPLAN Notices*, 35(9):268–279.
- William Clinger, Anne H. Hartheimer, and Eric M. Ost. 1999. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45.
- William D. Clinger. 1998 (May). Proper tail recursion and space efficiency. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 33(5):174–185. ISSN 0362-1340. URL <http://www.acm.org:80/pubs/citations/proceedings/pldi/277650/p174-clinger/>.
- W. F. Clocksin and C. S. Mellish. 2013. *Programming in Prolog*. Springer, 5th edition. ISBN 3540006788.
- Jacques Cohen. 1988 (January). A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36. ISSN 0001-0782. URL <http://www.acm.org/pubs/toc/Abstracts/0001-0782/35045.html>.
- A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. 1973 (November). Un système de communication homme-machine en Français. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.
- Thomas J. Conroy and Eduardo Pelegri-Llopert. 1982. An assessment of method-lookup caches for Smalltalk-80 implementations. In *Smalltalk-80: Bits of History, Words of Advice*, chapter 13, pages 239–247. Addison-Wesley, Reading, MA.
- RL Constable, SF Allen, HM Bromley, WR Cleaveland, JF Cremer, RW Harper, DJ Howe, TB Knoblock, NP Mendler, P Panangaden, JT Sasaki, and SF Smith. 1985. *Implementing mathematics with The Nuprl Proof Development System*. Prentice-Hall.
- William R. Cook. 2009 (October). On understanding data abstraction, revisited. *OOPSLA 2009 Conference Proceedings*, in *SIGPLAN Notices*, 44(10):557–572.
- Brad J. Cox. 1986. *Object-Oriented Programming - an Evolutionary Approach*. Addison-Wesley, Reading, 1 edition. ISBN 0-201-10393-1.
- Marcus Crestani and Michael Sperber. 2010 (September). Experience Report: Growing programming languages for beginning students. *Proceedings of the Fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, in *SIGPLAN Notices*, 45(9):229–234.

- Douglas Crockford. 2008. *JavaScript: The Good Parts*. O'Reilly.
- Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare. 1972. *Structured Programming*. Academic Press, London and New York.
- Ole-Johan Dahl and C. A. R. Hoare. 1972. Hierarchical program structures. In *Structured Programming*, chapter 3, pages 175–220. Academic Press.
- Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–12, New York. ACM Press.
- Olivier Danvy. 2006 (October). *An Analytical Approach to Programs as Data Objects*. DSc thesis, BRICS Research Series, University of Aarhus, Aarhus, Denmark.
- Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 151–160, New York, NY, USA. ACM.
- Peter Deutsch and Alan M. Schiffman. 1984 (January). Efficient implementation of the Smalltalk-80 system. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, ACM.
- Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. 2000 (5). Abstract machines for programming language implementation. *Future Generation Computer Systems*, 2000 (nr. 7):739–751.
- Edsger W. Dijkstra. 1968 (March). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11 (3):147–148.
- Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.
- Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Stoffens. 1978 (November). On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975.
- S. Drew, K. John Gough, and J. Ledermann. 1995. Implementing zero overhead head exception handling. Technical Report 95-12, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia.
- Derek Dreyer, Karl Crary, and Robert Harper. 2003 (January). A type system for higher-order modules. *Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages*, in *SIGPLAN Notices*, 38(1):236–249.
- Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular type classes. In *Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages*, pages 63–70, New York, NY, USA. ACM.
- Stéhane Ducasse, Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. 2016. *Pharo by Example 5*. Square Bracket Associates. Available under a Creative Commons license.
- R. Kent Dybvig. 1987. *The SCHEME Programming Language*. Prentice-Hall, Upper Saddle River, NJ 07458, USA. ISBN 0-13-791864-X (paperback).
- R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1992 (December). Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326. ISSN 0892-4635.
- Sebastian Egner. 2002 (June). Notation for specializing parameters without currying. SRFI 26, in the Scheme Requests for Implementation series.
- Matthias Felleisen. 1988. The theory and practice of first-class prompts. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, New York, NY, USA. ACM Press.
- Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. MIT Press.
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, second edition. ISBN 0262534800, 9780262534802.
- Matthias Felleisen and Daniel P. Friedman. 1997 (December). *The Little MLer*. MIT Press.
- Robert R. Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(10):666–672.

Andrzej Filinski. 1994. Representing monads. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, POPL ’94, pages 446–457, New York, NY, USA. ACM.

Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER ’14, pages 35–42, New York, NY, USA. ACM.

David Flanagan and Yukihiro Matsumoto. 2008. *The Ruby Programming Language*. O’Reilly. ISBN 9780596516178.

Matthew Flatt. 2012 (January). Creating languages in racket. *Communications of the ACM*, 55(1):48–56.

Matthew Flatt. 2016 (January). Bindings as sets of scopes. *Conference Record of the 43rd Annual ACM Symposium on Principles of Programming Languages*, in *SIGPLAN Notices*. To appear.

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that work together. *Journal of Functional Programming*, 22:181–216.

Matthew Flatt and Matthias Felleisen. 1998 (May). Units: Cool modules for HOT languages. *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 33(5):236–248.

Christopher W. Fraser and David R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings.

Daniel P. Friedman and Matthias Felleisen. 1996. *The Little Schemer*. MIT Press, fourth edition. ISBN 0-0262-56099-2.

Daniel P. Friedman, Christopher T. Haynes, and Eugene E. Kohlbecker. 1984. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 263–274. Springer-Verlag.

Emden Gansner and John Reppy, editors. 2002. *The Standard ML Basis Library*. Cambridge University Press, New York, NY.

Martin Gasbichler and Michael Sperber. 2002. Final shift for call/cc: Direct implementation of shift and reset. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (ICFP’02), pages 271–282. ACM.

Jeremy Gibbons and Geraint Jones. 1998 (September). The under-appreciated unfold. *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, in *SIGPLAN Notices*, 34(1):273–279.

A. Goldberg. 1983. *Smalltalk 80: The Interactive Programming Environment*. Addison-Wesley.

Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.

James Gosling, Bill Joy, and Guy Steele. 1997. *The Java Language Specification*. The Java Series. Addison-Wesley. ISBN 0-201-63451-1. URL <http://www.aw.com/cp/gosling-joy-et.al.html>.

Paul Graham. 1993. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0130305529. May be downloadable from <http://www.paulgraham.com/onlisp.html>.

Justin O. Graver and Ralph E. Johnson. 1990. A type system for Smalltalk. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 136–150, New York, NY, USA. ACM.

David Gries. 1981. *The Science of Programming*. Springer-Verlag.

Ralph E. Griswold and Madge T. Griswold. 1996. *The Icon Programming Language*. Peer-to-Peer Communications, San Jose, CA, third edition. ISBN 1-57398-001-3.

Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. 2005 (January). Cyclone: a type-safe dialect of C. *C/C++ Users Journal*, 23(1).

Dirk Grunwald and Benjamin G. Zorn. 1993 (August). CustoMalloc: Efficient synthesized memory allocators. *Software—Practice & Experience*, 23(8): 851–869. ISSN 0038-0644. URL [http://www.cs.colorado.edu/homes/grunwald/public\\_html/SPE93-customalloc.ps](http://www.cs.colorado.edu/homes/grunwald/public_html/SPE93-customalloc.ps).

David R. Hanson. 1996. *C Interfaces and Implementations*. Addison Wesley.

Robert Harper. 1986 (September). Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for the

## Bibliography

753

- Foundations of Computer Science, Edinburgh University.
- Robert Harper. 2011. Programming in Standard ML. Book draft licensed under Creative Commons. Best sought from Harper's home page at Carnegie Mellon University.
- Robert Harper and Mark Lillibridge. 1994 (January). A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137.
- Robert Harper and Christopher Stone. 2000. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Sterling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press.
- Robert W. Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.
- Brian Harvey and Matthew Wright. 1994. *Simply Scheme: Introducing Computer Science*. MIT Press, Cambridge, Massachusetts, U.S.A. ISBN 0-262-08226-8.
- Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. 1984. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 293–298. ACM.
- Greg Hendershott. 2014. Fear of macros. URL <http://www.greghendershott.com/fear-of-macros/>. Tutorial from the author's web site.
- Peter Henderson. 1980. *Functional Programming: Application and Implementation*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J.
- Matthew Hertz and Emery D. Berger. 2005 (October). Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313–326. ISSN 0362-1340. URL <http://doi.acm.org/10.1145/1103845.1094836>.
- Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. 1990 (June). Representing control in the presence of first-class continuations. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 25(6):66–77. ISSN 0362-1340.
- J. Roger Hindley. 1969. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60.
- Ralf Hinze. 2003. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, chapter 12, pages 245–262. Palgrave Macmillan.
- C. A. R. Hoare. 1972. Proof of correctness of data representations. *Acta Informatica*, 1: 271–281.
- C. A. R. Hoare. 2009 (August). Null references: The billion dollar mistake. Presentation delivered at QCon.
- C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. 1987 (August). Laws of programming. *Communications of the ACM*, 30(8):672–686. ISSN 0001-0782. URL <http://www.acm.org/pubs/toc/Abstracts/0001-0782/27653.html>. See corrigendum in the September 1987 CACM.
- C. J. Hogger. 1984. *Introduction to Logic Programming*. Academic Press, London. ISBN 0-12-352090-8.
- John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA.
- Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. 1993 (December). Some useful Modula-3 interfaces. Research Report 113, Digital Systems Research Center, Palo Alto, CA.
- Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. 2007. A history of haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 1–55. ACM.
- John Hughes. 1989 (April). Why functional programming matters. *The Computer Journal*, 32(2):98–107.
- John Hughes. 1995. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, LNCS volume 925, pages 53–96. Springer Verlag.
- John Hughes. 2016. Experiences with QuickCheck: Testing the hard stuff and

- staying sane. In *A List of Successes That Can Change the World — Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 169–186. Springer.
- Galen C. Hunt and James R. Larus. 2007 (April). Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49.
- Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heillard. 1979 (June). Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6b):1–261. ISSN 0362-1340. URL <http://doi.acm.org/10.1145/956653.956654>.
- Roberto Ierusalimschy. 2013 (December). *Programming in Lua*. Lua.org, third edition. ISBN 85-903798-5-X.
- Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 2007 (June). The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, pages 2-1–2-26.
- Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997 (October 5–9). Back to the future: The story of Squeak - a practical Smalltalk written in itself. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 318–326, New York. ACM Press.
- Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-471430-14-5.
- Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. 1988. TS: An optimizing compiler for Smalltalk. In Norman Meyrowitz, editor, *OOPSLA'88: Object-Oriented Programming Systems, Languages and Applications: Conference Proceedings*, pages 18–26. ISBN 0-89791-284-5.
- Mark S. Johnstone and Paul R. Wilson. 1998 (October). The memory fragmentation problem: Solved? *Proceedings of the First International Symposium on Memory Management*, in *SIGPLAN Notices*, 34(3):26–36.
- Mark P. Jones. 1993 (November). Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA.
- Mark P. Jones. 1999 (October). Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Paris, France. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht. Additional resources at <http://www.cse.ogi.edu/~mpj/thih>.
- Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC.
- Richard Jones and Rafael Lins. 1996. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, New York, NY, USA. ISBN 0-471-94148-4. URL [http://www.ukc.ac.uk/computer\\_science/Html/Jones/gc.html](http://www.ukc.ac.uk/computer_science/Html/Jones/gc.html). Reprinted in 1999 with improved index and corrected errata.
- Ted Kaehler and Dave Patterson. 1986. *A Taste of Smalltalk*. W. W. Norton and Company, New York.
- Gilles Kahn. 1987. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, LNCS volume 247, pages 22–39. Springer-Verlag.
- Samuel N. Kamin. 1990. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, Reading, MA.
- A. C. Kay. 1993 (March). The early history of Smalltalk. *SIGPLAN Notices*, 28(3):69–95. ISSN 0362-1340.
- Richard Kelsey, William Clinger, and Jonathan Rees. 1998 (September). Revised<sup>5</sup> report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76. ISSN 0362-1340.
- Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition.
- Oleg Kiselyov. 2012 (August). An argument against call/cc. URL <http://okmij.org/ftp/continuations/against-callcc.html>. Referenced in August 2015.
- Donald E. Knuth. 1965 (December). On the translation of languages from left to right. *Information and Control*, 8(6):607–639.
- Donald E. Knuth. 1973. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading,

- MA, second edition.
- Donald E. Knuth. 1974 (December). Structured programming with **go to** statements. *ACM Computing Surveys*, 6(4):261–301.
- Donald E. Knuth. 1981. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition.
- Donald E. Knuth. 1984. Literate programming. *The Computer Journal*, 27(2):97–111.
- Andrew Koenig. 1994. An anecdote about ML type inference. In *Proceedings of the USENIX 1994 Very High Level Languages Symposium*, VHLLS'94, pages 1–1, Berkeley, CA, USA. USENIX Association.
- Peter M. Kogge. 1990. *The Architecture of Symbolic Computers*. McGraw-Hill, New York. ISBN 0070355967.
- Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA. ACM. ISBN 0-89791-200-4. URL <http://doi.acm.org/10.1145/319838.319859>.
- Robert A. Kowalski. 1974. Predicate logic as a programming language. *Proc. IFIP 4*, pages 569–574.
- Robert A. Kowalski. 1979. *Logic for Problem Solving*. The Computer Science Library, Artificial Intelligence Series. North Holland, New York. As of July 2015, available online from Kowalski's site at Imperial College London, although with the chapters out of order.
- Robert A. Kowalski. 1988 (January). The early years of logic programming. *Communications of the ACM*, CACM, 31(1):38–43.
- Robert A. Kowalski. 2014. *Logic for Problem Solving, Revisited*. Books on Demand.
- Glenn Krasner, editor. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Longman, Boston, Mass. ISBN 0-201-11669-3.
- P. J. Landin. 1966. The next 700 programming languages. *Communications of the ACM*, 9(3):157–66.
- Peter J. Landin. 1964 (January). The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320.
- Konstantin Läufer and Martin Odersky. 1994 (September). Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430.
- Xavier Leroy. 1994 (January). Manifest types, modules, and separate compilation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 109–122.
- Xavier Leroy. 1999 (September). Objects and classes vs. modules in Objective Caml. URL <http://pauillac.inria.fr/~xleroy/talks/icfp99.ps.gz>. Invited talk delivered at the 1999 International Conference on Functional Programming (ICFP).
- Xavier Leroy. 2000 (May). A modular module system. *Journal of Functional Programming*, 10(3):269–303.
- Henry Lieberman and Carl Hewitt. 1983 (June). A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429.
- Miran Lipovača. 2011. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press.
- Barbara Liskov. 1996. A history of CLU. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming Languages—II*, pages 471–510. ACM, New York, NY, USA.
- Barbara Liskov and John Guttag. 1986. *Abstraction and Specification in Program Development*. MIT Press / McGraw-Hill.
- Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977 (August). Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576. Also published in/as: In “Readings in Object-Oriented Database Systems” edited by S.Zdonik and D.Maier, Morgan Kaufman, 1990.
- Barbara Liskov and Stephen Zilles. 1974 (March). Programming with abstract data types. *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, in *SIGPLAN Notices*, 9(4):50–59.
- Barbara H. Liskov and Alan Snyder. 1979 (November). Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558.

- Barbara H. Liskov and Jeannette M. Wing. 1994 (November). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841. ISSN 0164-0925.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005 (June). Pin: Building customized program analysis tools with dynamic instrumentation. *Proceedings of the ACM SIGPLAN '05 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 40(6):190–200.
- Luc Maranget. 2007. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421.
- Luc Maranget. 2008. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA. ACM.
- Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. 2008 (June). Parallel generational-copying garbage collection with a block-structured heap. In *ISMM '08: Proceedings of the 7th International Symposium on Memory Management*. ACM.
- Moe Masuko and Kenichi Asai. 2009. Direct implementation of shift and reset in the MinCaml compiler. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, pages 49–60, New York, NY, USA. ACM.
- Conor McBride. 2004. Epigram: Practical programming with dependent types. In *International School on Advanced Functional Programming*, pages 130–170. Springer.
- Conor McBride and Ross Paterson. 2008 (January). Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13.
- John McCarthy. 1960 (April). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195.
- John McCarthy. 1962. *Lisp 1.5 Programmer's Manual*. Cambridge, Mass.
- Sandi Metz. 2013. *Practical Object-oriented Design in Ruby: An Agile Primer*. Addison-Wesley professional Ruby series. Addison-Wesley. ISBN 9780321721334.
- Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice Hall International, London.
- Bertrand Meyer. 1997. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, second edition.
- Jan Midgaard, Norman Ramsey, and Bradford Larsen. 2013 (September). Engineering definitional interpreters. In *Proceedings of the 15th International Symposium on Principles and Practice of Declarative Programming (PPDP'13)*, pages 121–132.
- Todd Millstein, Colin Bleckner, and Craig Chambers. 2004. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems*, 26(5): 836–889.
- Robin Milner. 1978 (December). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.
- Robin Milner. 1983. How ML evolved. *Polymorphism—The ML/LCF/Hope Newsletter*, 1(1).
- Robin Milner. 1999 (May). *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press. ISBN 0 521 64320 1 (hc), 0 521 65869 1 (pbk).
- Robin Milner and Mads Tofte. 1991. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts.
- Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts.
- Marvin L. Minsky. 1963 (December). A lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58, Project MAC, MIT, Cambridge, MA.
- John C. Mitchell and Gordon D. Plotkin. 1988 (July). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502.
- Calvin N. Mooers. 1966 (March). TRAC, A procedure-describing language for the reactive typewriter. *Communications of the ACM*, 9(3):215–219. ISSN 0001-0782.
- David A. Moon. 1986 (October). Object-oriented programming with flavours.

- In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8.
- George C. Necula, Scott McPeak, and Westley Weimer. 2002 (January). CCured: Type-safe retrofitting of legacy code. *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages*, in *SIGPLAN Notices*, 37(1):128–139.
- Greg Nelson, editor. 1991. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ.
- Nicholas Nethercote and Julian Seward. 2007a. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE ’07, pages 65–74, New York, NY, USA. ACM.
- Nicholas Nethercote and Julian Seward. 2007 (June)b. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN ’07 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 42(6):89–100. ISSN 0362-1340. URL <http://doi.acm.org/10.1145/1273442.1250746>.
- Nils J. Nilsson. 1980. *Principles of artificial intelligence*. Tioga/Morgan Kaufman, 476 pages; ISBN 0-934613-10-9; US&Canada \$54.95 CALL NUMBER: Q335 .N515.
- James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Mark S. Miller. 2016. The left hand of equals. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 224–237, New York, NY, USA. ACM.
- Kristen Nygaard and Ole-Johan Dahl. 1981. The development of the simula languages. In Richard L. Wexelblat, editor, *History of Programming Languages I*, pages 439–480. ACM, New York, NY, USA.
- Martin Odersky, Lex Spoon, and Bill Venners. 2019. *Programming in Scala*. Artima Press, Walnut Creek, CA, fourth edition. ISBN 098153161X.
- Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55.
- Melissa E. O’Neill. 2009 (January). The genuine sieve of Eratosthenes. *Journal of Functional Programming*, 19(1):95–106. ISSN 0956-7968.
- Derek C. Oppen. 1980 (October). Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483.
- David Lorge Parnas. 1972 (December). On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.
- Laurence C. Paulson. 1996. *ML for the working programmer*. Cambridge University Press, New York, second edition.
- Nigel Perry. 1991. *The implementation of practical functional programming languages*. PhD thesis, Imperial College, London.
- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82.
- Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.
- Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. 2016 (May). Software foundations. URL <https://www.cis.upenn.edu/~bcpierce/sf/>. Version 4.0.
- Rob Pike. 1990 (July). The implementation of newsqueak. *Software—Practice & Experience*, 20(7):649–659. URL <http://plan9.bell-labs.com/cm/cs/doc/88/1-a.ps.gz>.
- Gordon D. Plotkin. 1981 (September). A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Aarhus, Denmark.
- François Pottier and Didier Rémy. 2005. *The Essence of ML Type Inference*, chapter 10, pages 389–489. MIT Press.
- François Pottier and Yann Régis-Gianas. 2006 (March). Towards efficient, typed LR parsers. In *ACM Workshop on ML*, pages 155–180.

- Todd A. Proebsting. 1997 (June). Simple translation of goal-directed evaluation. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 32(5):1–6.
- Norman Ramsey. 1990 (April). Concurrent programming in ML. Technical Report TR-262-90, Department of Computer Science, Princeton University.
- Norman Ramsey. 1994 (September). Literate programming simplified. *IEEE Software*, 11(5):97–105.
- Norman Ramsey. 1999. Eliminating spurious error messages using exceptions, polymorphism, and higher-order functions. *Computer Journal*, 42(5):360–372.
- Norman Ramsey. 2005 (September). ML module mania: A type-safe, separately compiled, extensible interpreter. In *ACM SIGPLAN Workshop on ML*, pages 172–202.
- Norman Ramsey. 2011 (November). Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 21(6):585–615. A preliminary version of this paper appeared in *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003.
- Norman Ramsey, João Dias, and Simon L. Peyton Jones. 2010 (September). Hoopl: A modular, reusable library for dataflow analysis and transformation. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell 2010)*.
- Norman Ramsey, Kathleen Fisher, and Paul Govereau. 2005 (September). An expressive language of signatures. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 27–40. Selected as one of the best papers of ICFP 2005.
- Jonathan Rees and William Clinger. 1986 (December). Revised<sup>3</sup> report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79.
- E. Reingold and R. Reingold. 1988. *Pascal Algorithms: An Introduction to Programming*. Scott, Foresman, Illinois.
- John Reynolds. 1972 (August). Definitional interpreters for higher-order programming languages. In *Proceedings of the 25<sup>th</sup> ACM National Conference*, pages 717–740. ACM.
- 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.
- John C. Reynolds. 1978. User-defined types and procedural data structures as complementary approaches to data abstraction. In David Gries, editor, *Programming Methodology, Texts and Monographs in Computer Science*, pages 309–317, New York, NY. Springer.
- John C. Reynolds. 1993 (November). The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248. ISSN 0892-4635. URL <ftp://e.ergo.cs.cmu.edu/histcont/draft.ps.Z>.
- John C. Reynolds. 1998 (December). Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397. Reprinted from the proceedings of the 25th ACM National Conference (1972).
- E. S. Roberts. 1986. *Thinking Recursively*. John Wiley & Sons, New York.
- Eric S. Roberts. 1989 (March). Implementing exceptions in C. Technical Report Research Report 40, Digital Systems Research Center, Palo Alto, CA.
- J. Alan Robinson. 1965 (January). A machine-oriented logic based on resolution principle. *Journal of the ACM*, 12(1):23–49.
- J. Alan Robinson. 1983. Logic programming - past, present and future. *New Generation Comput. (Japan)* ISSN: 0288-3635, 1(2): 107–24. QA 76 N 48.
- J. S. Rohl. 1984. *Recursion via Pascal*, volume 19 of *Cambridge Computer Science Texts*, Editors: E. S. Page and C. M. Reeves and D. E. Conway. Cambridge University Press, Cambridge, UK. ISBN 0-521-26329-8 (hardcover), 0-521-26934-2 (paperback).
- Raúl Rojas. 2015. A tutorial introduction to the lambda calculus. *CoRR*, abs/1503.09060. URL <http://arxiv.org/abs/1503.09060>.
- Colin Runciman and David Wakeling. 1993 (April). Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–246.
- David A. Schmidt. 1986. *Denotational Semantics: A Methodology for Language Development*.

- ment. Allyn and Bacon.
- Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009 (August). Complete and decidable type inference for GADTs. *Proceedings of the Fourteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, in *SIGPLAN Notices*, 44(9):341–352.
- Kevin Scott and Norman Ramsey. 2000 (May). When do match-compilation heuristics matter? Technical Report CS-2000-13, Department of Computer Science, University of Virginia.
- Robert Sedgewick. 1988. *Algorithms*. Addison-Wesley, second edition.
- Manuel Serrano and Hans-Juergen Boehm. 2000 (September). Understanding memory allocation of Scheme programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, in *SIGPLAN Notices*, 35(9):245–256. ISSN 0362-1340. URL <http://www.acm.org/pubs/citations/proceedings/fp/351240/p245-serrano/>; <http://www.acm.org/pubs/articles/proceedings/fp/351240/p245-serrano/p245-serrano.pdf>.
- Alex Shinn, John Cowan, and Arthur A. Gleckler. 2013. Revised<sup>7</sup> Report on the algorithmic language Scheme. Technical report, R7RS Working Group 1. URL <https://small.r7rs.org/attachment/r7rs.pdf>.
- Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *23rd European Symposium on Programming (ESOP 2014)*, pages 27–46.
- Frederick Smith and Greg Morrisett. 1999 (March). Comparing mostly-copying and mark-sweep conservative collection. *Proceedings of the First International Symposium on Memory Management (ISMM'98)*, in *SIGPLAN Notices*, 34(3):68–78.
- Elliott Soloway. 1986 (September). Learning to program=learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858.
- Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised<sup>6</sup> Report on the algorithmic language Scheme. *Journal of Functional Programming*, 19 (Supplement S1):1–301. URL <http://dx.doi.org/10.1017/S0956796809990074>.
- Mike Spivey. 1990 (June). A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42.
- T. A. Standish. 1980. *Data Structure Techniques*. Addison-Wesley, Reading, MA.
- Raymie Stata and John V. Guttag. 1995 (October). Modular reasoning in the presence of subclassing. *OOPSLA '95 Conference Proceedings*, in *SIGPLAN Notices*, 30(10):200–214.
- Raymond Paul Stata. 1996. *Modularity in the Presence of Subclassing*. PhD thesis, Massachusetts Institute of Technology.
- Guy Lewis Steele, Jr. 1977 (October). Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, LAMBDA: The ultimate GOTO. In *Proceedings ACM Annual Conference*, pages 153–162.
- Guy Lewis Steele, Jr. 1984. *Common Lisp: The Language*. Digital Press.
- Guy Lewis Steele, Jr and Gerald Jay Sussman. 1976 (March). Lambda: The ultimate imperative. Technical Report AIM-353, Massachusetts Institute of Technology. URL <ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-353.tiff.tar.gz>.
- Guy Lewis Steele, Jr and Gerald Jay Sussman. 1978 (May). The art of the interpreter or, the modularity complex (parts zero, one, and two). Technical Report AIM-453, Massachusetts Institute of Technology. URL <ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-453.ps>.
- L. Sterling and E. Shapiro. 1986. *The Art of Prolog*. MIT Press, Cambridge, Mass. ISBN 0-262-19250-0.
- Joseph E. Stoy. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.
- Christopher Strachey and Christopher P. Wadsworth. 2000 (April). Continuations: A mathematical semantics for handling full jumps. *Higher Order Symbol. Comput.*, 13(1-2):135–152. Oxford monograph from 1974 republished in a special issue of HOSC devoted to Christopher Strachey.
- Bjarne Stroustrup. 1997. *The C++ Programming Language*. Addison-Wesley, Reading,

MA, third edition. ISBN 0-201-88954-4.

Gerald Jay Sussman and Guy Lewis Steele, Jr. 1975 (December). Scheme: An interpreter for extended lambda calculus. MIT AI Memo No. 349, reprinted in *Higher-Order and Symbolic Computation* 11(4):405–439, Dec 1998.

Gerald Jay Sussman and Guy Lewis Steele, Jr. 1998 (December). Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11 (4):405–439. Originally MIT AI Lab MEMO AIM-349, December 1975.

Mads Tofte. 2009 (August). Tips for computer scientists on Standard ML (revised). Linked from <http://www.itu.dk/people/tofte>.

D. S. Touretzky. 1984. *LISP: A Gentle Introduction to Symbolic Computation*. Harper & Row, Pubs., New York, NY. ISBN 0-06-046657-X.

Jeffrey D. Ullman. 1997. *Elements of ML Programming, ML97 Edition*. Prentice-Hall, Englewood Cliffs, NJ.

David Ungar. 1984 (May). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, in *SIGPLAN Notices*, 19 (5):157–167.

David Ungar and Randall B. Smith. 1987. Self: The power of simplicity. *OOPSLA '87 Conference Proceedings*, in *SIGPLAN Notices*, pages 227–242.

Peter Van der Linden. 1994. *Expert C programming: deep C secrets*. Prentice Hall Professional.

Guido van Rossum. 1998. A tour of the Python language. In R. Ege, M. Singh, and B. Meyer, editors, *Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS-23*, page 370, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. IEEE Computer Society Press. ISBN 0-8186-8383-X.

David Vengerov. 2009. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the 2009 International Symposium on Memory Management, ISMM '09*, pages 1–9, New York, NY, USA. ACM.

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalised. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation, TLDI '10*, pages 39–50. ACM.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011 (September). OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412. ISSN 0956-7968.

Philip Wadler. 1999. A prettier printer. Unpublished note available from the author's Web site.

Philip Wadler. 2015 (November). Propositions as types. *Communications of the ACM*, 58(12):75–84.

Philip Wadler and Stephen Blott. 1989 (January). How to make *ad-hoc* polymorphism less *ad hoc*. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76.

Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997 (October). The Zephyr Abstract Syntax Description Language. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 213–227, Santa Barbara, CA.

David H. D. Warren. 1983 (October). An abstract Prolog instruction set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA.

R. L. Wexelblat, editor. 1981. *History of Programming Languages*. Academic Press, New York. ISBN 0-12-745040-8.

Robert Wilensky. 1986. *Common LISPcraft*. W. W. Norton & Co., New York, NY, USA. ISBN 0-393-95544-3.

Paul R. Wilson. 1992 (September). Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42, St. Malo, France. Published as Lecture Notes in Computer Science 637, Springer-Verlag.

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995 (September). Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International*

## Bibliography

761

- Workshop on Memory Management, LNCS volume 986, Kinross, Scotland. Springer-Verlag. URL <ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps>.
- Terry Winograd. 1972. *Understanding Natural Language*. Academic Press, New York.
- P. H. Winston and B. K. P. Horn. 1984. *Lisp Second Edition*. Addison-Wesley, Reading. ISBN 0-201-08372-8.
- Patrick H. Winston. 1977. *Artificial Intelligence*. Addison Wesley Publishing Co., Reading, Mass.
- Niklaus Wirth. 1971 (April). Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227. ISSN 0001-0782. URL <http://www.acm.org/classics/dec95/>.
- Niklaus Wirth. 1977 (November). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823.
- Niklaus Wirth. 1982. *Programming in Modula-2*. Springer, Berlin. ISBN 0-387-11674-5.
- Robert S. Wolf. 2005. *A Tour through Mathematical Logic*. Mathematical Association of America.
- Andrew K. Wright. 1995 (December). Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355.
- William A. Wulf, R. L. London, and Mary Shaw. 1976 (December). An introduction to the construction and verification of alphard programs. *IEEE Transactions on Software Engineering*, 2(4):253–265.
- Hongwei Xi and Frank Pfenning. 1998 (May). Eliminating array bound checking through dependent types. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 33(5):249–257. ISSN 0362-1340. URL <http://www.acm.org:80/pubs/citations/proceedings/pldi/277650/p249-xi/>.
- Benjamin Zorn. 1993 (July). The measured cost of conservative garbage collection. *Software—Practice & Experience*, 23(7):733–756.

# Concept index

- E*, 572
- N*<sub>10</sub>, 83
- D*, 572
- T*, 572
- $\kappa$ , 363
- $\lambda$ -bound variable
  - key word or phrase, 453
- $\sigma$ , 418
- $\tau$ , 342, 366, 418, 572
- $\tau_f$ , 342
- $d$ , 83, 572
- $*\text{name}$ , 205, 362, 414, 479, 546
- C*, 572
- $\pi$ , 572
- def*, 95, 205, 362, 402, 414, 479
- exp*, 95, 205, 362, 414, 479
- formals*, 95, 205, 362, 414
- type-formals*, 362
- unit-test*, 414, 479, 547
- $*\text{name}$ , 18, 95
- numeral*, 18, 95
- atom, functor*, S55
- C*, 431
- call/cc
  - key word or phrase, 247
- self
  - key word or phrase, 718
- super
  - key word or phrase, 718
- cons, car, cdr
  - key word or phrase, 174
- A*, S13
- A*
  - as automatically generated function in identifier cross-reference, 39
- abbreviation
  - type, 539
- absolute access path, 569
- Abstract class
  - key word or phrase, 716
- abstract class, 632
  - Collection, 651
- Abstract data type
  - key word or phrase, 595
- Abstract machine
  - key word or phrase, 71
- key word or phrase, 71
- Abstract syntax
  - key word or phrase, 70
- Abstract-machine semantics
  - key word or phrase, 246
- abstraction, 537
  - syntactic, 173
- Abstraction function
  - key word or phrase, 596, 718
- abstraction, functional, 36
- access path
  - of a module or component, 569
  - relative, 583
- accessor functions
  - for records, 109
- accumulating parameters
  - method of, 103
- ad hoc polymorphism*, 134
- ad-hoc polymorphism, 554
- Ada, 11, 596, 598
- Algebraic data type
  - key word or phrase, 325
- algebraic laws, 100
  - of append, 101
  - soundness of
    - mentioned but not explained, 66
- Algol-60, 92
  - own variables, 126
- Algol 60, 11
- all.h
  - automatic inclusion of, S289
  - collecting all interfaces, S289
- Allocation pointer
  - key word or phrase, 295
- allof, 552, 562, 566
- Alphard, 596, 598
- amortized analysis
  - of a purely functional queue, 184
  - of garbage-collector performance, 274
- and
  - in ML, it's different, 310
- Answer
  - key word or phrase, 717
- answer
  - key word or phrase, 71

- nonstandard English in Smalltalk, 622
- answer type
  - in continuation-passing style, 140
- API
  - key word or phrase, 596
- appended
  - Prolog predicate, S76
- application
  - of types, 367
- arity
  - in  $\mu$ Smalltalk, 637
- array-element*, 638
- ascription
  - of a module type to a module, 575
- association list, \$107, 655, S77
  - in sets, 134
- associative array
  - also called table, finite map, 107
- At least as general
  - key word or phrase, 453
- at least as general
  - used to relate two types, 420
- Atom
  - key word or phrase, 174, S102
- atom
  - in Prolog, S47
  - in  $\mu$ Scheme, 94
  - $\mu$ Scheme concept, 94
- Autognostic principle
  - key word or phrase, 716
- autognostic principle, 636
- Automatic memory management
  - key word or phrase, 294
- axiom
  - in Prolog, S50
- B*
  - as basis function in identifier
    - cross-reference, 39
- Backus-Naur Form, 17
- base types, 352
- Basis
  - key word or phrase, 72
- basis
  - defined, 26
  - in Impcore, 28
  - initial, 26
- bearskins
  - used to implement abstract data types, 598
- Behavioral subtyping
  - key word or phrase, 719
- behavioral subtyping, 637
- better-move, Prolog predicate, S92
- Big-step semantics
  - key word or phrase, 70
- binary-functor*, S55
- binary-predicate*, S55
- black magic
  - decision procedure for, S49
- Block
  - key word or phrase, 718
- block
  - in Smalltalk, 628
- blocks world, \$S87
- BNF, 17
- body
  - of a Prolog clause, S56
- bound type variable
  - defined, 380
- Bound variable
  - key word or phrase, 327
- bound variable, 322
- brackets
  - square and round, 119
- Brown, Troy, 644
- bug
  - in  $\mu$ Scheme interpreter
    - false reports of, 156
- BugInTypeInference
  - in nano-ML, 421
- C
  - own variables static, 126
  - scope rules, 75
- C (programming language), 11
- c{a, d}+r, Scheme functions, 98
- calculational proof, 116
- calculus
  - as a form of abstract machine, 243
- Call stack
  - key word or phrase, 247, 293
- call stack, 66
- call/cc, real Scheme function, \$174
- Calvin and Hobbes, 742
- capture
  - of a type variable, 378
  - of type variables, 383
- capture of type variables
  - in type-lambda, 374
- capture-avoiding substitution, 167
  - for type variables, 374
- careless persons
  - misusing unspecified values, S318
- cascade
  - of messages in Smalltalk, 714
- case expression, 470
- catch
  - in  $\mu$ Scheme+, 206
- catching an exception, 210
- category, syntactic, *see* syntactic category
- characteristic function
  - representation of sets, 189
- check-assert

- explained, 23
- check-expect**
  - implementation in  $\mu$ Smalltalk using `=`, 644
- check-error**
  - explained, 23
- check-expect**
  - explained, 23
- checked run-time error, 40
- checkoverflow** functions, S197
- choice**, 471, 479
- Church-Rosser theorem, 248
- Class**
  - key word or phrase, 716
- class**
  - abstract, 632
  - definition of, 624
- Class method**
  - key word or phrase, 717
- classifier**
  - using  $k$  nearest neighbors, S148
- Clausal definition**
  - key word or phrase, 325
- clausal definition**, 492
- clausal definitions**, 526
- Clause**
  - key word or phrase, S103
- clause**, S55
- clause-or-query**, S55
- Client**
  - key word or phrase, 596
- client code**, 536
- close**
  - over free type variables, 425
- closed-world assumption**, S29
- closing**
  - over free type variables, 425
- Closure**
  - key word or phrase, 174
- closure**
  - introduced, 124
- CLU**
  - goal of program correctness, 595
  - mutable types
    - vs. immutable types, 555
- Clue**
  - board game, S106
- Cobol**, 11
- coerce:** method
  - examples of use, 672
- Collection**
  - key word or phrase, 719
- Common Lisp**, 91, 176, 721
- CommonLoops**, 721
- Compacting collector**
  - key word or phrase, 295
- Compiler**
  - key word or phrase, 73
- Completeness**
  - key word or phrase, S103
- Complex method**
  - key word or phrase, 716
- complex operation**
  - in Smalltalk, 671
- comprehension**
  - list
    - in a semantics for Prolog, S118
- conclusion**
  - of an inference rule, 71
- Concrete syntax**
  - key word or phrase, 70
- concurrent garbage collection**, 291
- conditional**
  - deprecated in object-oriented programming, 663
- cons cell**
  - defined, 97
- Conservative collector**
  - key word or phrase, 295
- conservative extension**, S331
- conservative garbage collection**, 288
- Constraint solver**
  - key word or phrase, 453
- constructed value**, 469
- constructor**
  - for values of algebraic data type, 469
  - of Moleculevalues, 543
  - pitfalls as a technical term, 113
  - smart
    - in binary trie, 523
- constructor functions**
  - for records, 110
- constructors of abstract syntax**
  - constraints on the choice of names, 43
- Continuation**
  - key word or phrase, 247
- continuation**, 138
  - defined by evaluation context, 244
  - entering, 244
  - failure, S371
  - first-class, 244
  - undelimited, 245
- continuation-passing style**, 140
  - in Smalltalk, 620, 649
  - versus direct style, 140
- continuations**
  - implemented as blocks in Smalltalk, 648
- contravariance**
  - of function arrow in subtyping, 615
- Control operator**
  - key word or phrase, 246
- conversion specification**, 47

conversion specifier, 47  
Copying collector  
    key word or phrase, 295  
core language, 216  
    in  $\mu$ Scheme, 122  
crash  
     $\mu$ Scheme interpreter  
    probable cause of, 156  
creators, 113  
creators, producers and observers  
    relationship to type systems, 356  
cross-reference  
    explanation of, 39  
Curry-Howard isomorphism  
    *see* propositions as types, 356

Data abstraction  
    key word or phrase, 595, 716  
data abstraction, §555  
    introduced, 535  
database  
    of Prolog clauses, S50  
De Morgan's laws  
    for `exists?` and `all?`, 133  
    in solving Boolean formulas, 191  
Dead object  
    key word or phrase, 295  
Dead variable  
    key word or phrase, 295  
debugging  
     $\mu$ Scheme code  
        by printing more information  
        about closures, S322  
*dec*, 547  
Decidability  
    key word or phrase, S103  
*def*, 18, 205, 339, 402, 484, 546, 547, 624,  
    638, S39  
*define*  
    explained, 22  
define a type  
    in ML  
        by type abbreviation, 310  
definition form  
    extended, 19  
    true, 19  
Definitionnal interpreter  
    key word or phrase, 73  
Delegation  
    key word or phrase, 718  
delegation  
    in the implementation of Set, S546  
Delimited continuation  
    key word or phrase, 247  
dependent function type, 551  
dependent types, 392  
Derivation  
    key word or phrase, 71

derivation, 56  
design patterns  
    factory, S170  
desugaring  
    defined, 68  
    of `define` in  $\mu$ Scheme, 122  
    of `val-rec` in Typed  $\mu$ Scheme, 404  
dictionary  
    also called table, finite map, 107  
diff appended, Prolog predicate, S80  
Difference list  
    key word or phrase, S103  
different, Prolog predicate, S91  
direct style  
    contrasted with continuation-passing style, 140  
dispatch  
    dynamic, 633  
    of a method, 620  
distinct names  
    checking for at parse time, 158  
Divergence  
    key word or phrase, 246  
duck typing, 637  
dynamic dispatch, 641  
    in Smalltalk, 633  
dynamic scoping, 150

EBNF, 17

effects  
    including errors, 112  
    including nontermination, 112  
Eiffel (object-oriented language), 721  
Einstein, Albert, S17  
either  
    alternative elimination form for  
    sum types, 359  
elaboration, 461, 581  
    and operator overloading, 586  
Elimination form  
    key word or phrase, 394  
elimination forms  
    *see also* “introduction and elimination forms, 356  
elimination rules, 354  
Embedding  
    key word or phrase, 326  
embedding, 315  
    of ML primitives into  $\mu$ Scheme  
    primitives, 319  
emptyset  
    Scheme function, 189  
Encapsulation  
    key word or phrase, 595  
encapsulation, 625  
Environment  
    key word or phrase, 70  
environment lookup

## Concept index

in Molecule, 583	
equality	
of association lists, 134	
of types	
how to check, 340	
Equality constraint	
key word or phrase, 453	
equality type	
in Standard ML, S240	
equational reasoning, 116	
equivalence	
observational, 644	
of objects in $\mu$ Smalltalk, 644	
equivalent	
vs identical when looking in collections, 655	
erasure	
of types, 389	
error	
run-time, 47	
syntax, 47	
etymology	
folk	
of snocced, S75	
Evaluation	
key word or phrase, 71	
Evaluation context	
key word or phrase, 246	
evaluation context, 243	
Evaluation stack	
key word or phrase, 247, 294	
Evaluator	
key word or phrase, 73	
Exception	
key word or phrase, 324	
exception	
catching, 210	
raising, 210	
throwing, 210	
Exception handler	
key word or phrase, 325	
exceptions, 592	
execute	
as a synonym for evaluate, 620	
Exhaustive pattern match	
key word or phrase, 325	
exhaustiveness check	
for case expressions, 511	
existential quantification	
in algebraic data types, S27	
$exp$ , 15, 18, 205, 339, 397, 470, 546, 547, 620, 627, 638	
Explicit memory management	
key word or phrase, 294	
exponent	
of a floating-point number, S543	
exported names, 536	
exporting module, 549	
Expose the representation	
key word or phrase, 718	
expression	
type-level, 361, 366	
Expression-oriented language	
key word or phrase, 72	
expression-oriented language, 17	
Extended Backus-Naur Form, 17	
extended definition, 19	

767

- defined, 380
- Free variable
  - key word or phrase, 175, 327
- free variable, 322
- free variables
  - in  $\mu$ Scheme expressions, S319
- fresh location
  - in  $\mu$ Schemefunction application, 149
- fresh names
  - stripping numeric suffix from, S240
- fresh variable, 167
  - doesn't change evaluation, 197
  - in  $\mu$ Scheme Exercises, 198
- freshness
  - in a name, S240
- function
  - primitive
    - in Impcore environment  $\phi$ , 30
  - user-defined
    - in Impcore environment  $\phi$ , 30
- function environment  $\phi$ 
  - in Impcore, 30
- functional abstraction, 36
- functions
  - first-order, 124
  - higher-order, 123
- Functor
  - key word or phrase, S102
- functor
  - in Prolog, S47
  - in Prolog, S56
- GADT, S27
  - abbreviation, S34
- Garbage collection
  - key word or phrase, 295
- garbage collection
  - analyzing performance of, 274
- generality
  - of types, 420
- Generalization
  - key word or phrase, 453
- generalization
  - of types in nano-ML, 433
- Generational collector
  - key word or phrase, 296
- Generational hypothesis
  - key word or phrase, 296
- generative syntactic form, 495
- Generativity
  - key word or phrase, 394
- generativity
  - defined, 495
  - from sealing in Molecule, 549
  - in Molecule, 569
- generic
  - meaning defined by recursion over types, 134
  - meaning parametrically polymorphic, 134
- Generic module
  - key word or phrase, 596
- generic module, 550
- Gentzen, Gerhard
  - and natural deduction, 71
- Goal
  - key word or phrase, S102
- goal
  - in Prolog, S53
- goal, S55
- goals, S55
- Grammar
  - key word or phrase, 70
- ground clause
  - in Prolog, S56
- Ground term
  - key word or phrase, S102
- ground term
  - in Prolog, S56
- ground type, 418
- head
  - of a Prolog clause, S56
  - of a Prolog rule, S102
- headroom
  - in garbage collection, 263, 275
- Heap
  - key word or phrase, 294
- heap
  - leftist, 565
- Heap allocation
  - key word or phrase, 294
- Heap object
  - key word or phrase, 294
- Heidi
  - broadcast on NBC, 536
- hexp, S93
- Higher-order function
  - key word or phrase, 175
- higher-order functions, 123
- Hoare triple
  - as metaphor for moves in blocks world, S88
- Hole
  - key word or phrase, 247
- Hope (functional language), 177
- Horn clause, S49
- htype, S93
- hygiene, 168
  - defined, 173
  - in full Scheme macros, 173
  - in translations for  $\mu$ Scheme, 164
- hygienic substitution, 168

idempotent	of a template for syntactic sugar, 168
substitution, 422	
identical	of an ML type scheme, 420
vs equivalent when looking in collections, 655	of generic modules in Molecule, 551
identifier cross-reference	of inference rules, 57
explanation of, 39	of polymorphic types in ML, 310
idiom	instantiation (type application)
inside joke about, S261	typing rule for, 374
IFX	instantiation of generic modules
why it isn't called IF, 43	typing rule for, 584
immutable abstraction, 555	Interactivity
Impcore	key word or phrase, 327
relation to C, 21, 22, 75	Interface
scope rules, 75	key word or phrase, 596
semantics, §20	interface, 536
syntax, §17	InternalError
imperative features, 413	exception, S366
Implementation	intersection types
key word or phrase, 596	subtyping of, 575
impredicative, 550	introduction and elimination forms
impredicative polymorphism, 587	relationship to creators, producers, and observers, 356
in modules, 588	Introduction form
impure	key word or phrase, 394
language, 413	introduction rules, 354
induction	isKindOf:
avoided with difference lists, S80	how to avoid using incorrectly, 647
induction step	isMemberOf:
in a recursive algorithm, 100	how to avoid using incorrectly, 647
inductively defined data, 118	it
specified by recursion equations, 119	automatic binding in Impcore, 54
inference rules, 31	automatic binding in $\mu$ Scheme, 162
inhabitants	iterator
of a subtype or supertype, 571	as a kind of observer, 654
Inheritance	in Smalltalk, 654
key word or phrase, 717	
Initial basis	Judgment
key word or phrase, 72	key word or phrase, 69
initial basis	Judgment form
defined, 26	key word or phrase, 69
initialization	Kind
of a Smalltalk object, 626	key word or phrase, 394
Instance	kind
key word or phrase, 453, 716	of a type, 363
instance	kind, 479
of a generic module, 551	kinds
of a template, 168	not needed in nano-ML, 419
of an inference rule, 56	slogan about classifying types, 365
relation between types, 420	
Instance variable	Lambda abstraction
key word or phrase, 716	key word or phrase, 175
Instantiation	lambda expression
key word or phrase, 394, 453	introduced, 122
instantiation	lambda-bound variables, 425
common mistakes using, 367	$\lambda$ -calculus
of a quantified type	
defined, 367	

- inspiration for Lisp, 91
- LAMBDA
  - why it isn't called LAMBDA, 43
- laws, algebraic, 100
- lazy sweeping, 271
- leftist heap, 565
- Let-bound variable
  - key word or phrase, 453
- let-bound variables, 425
- let-keyword*, 95, 205, 362, 414, 546
- letrec
  - less polymorphic than expected, 426
- LETX
  - why it isn't called LET, 43
- lexical analysis, S201
- lexical scoping, 150
- lies, damn lies, and the logical interpretation of Prolog, S67
- linear congruential method for generating random numbers, 126
- linear store, 146
- Lisp, §91
  - inspired by  $\lambda$ -calculus, 91
  - recursion in, 92, 100
- list, real Lisp function, 98
- list comprehensions
  - how to implement, S121
  - in a semantics for Prolog, S118
- List constructor
  - key word or phrase, 325
- literal
  - in  $\mu$ Scheme, 97
- literal*, 18, 95, 205, 339, 362, 414, 479, 546, 638
- Live data
  - key word or phrase, 294
- Live object
  - key word or phrase, 294
- locals*, 638
- Location
  - key word or phrase, 175
- location
  - mutable, 124
- Location semantics
  - key word or phrase, 175
- Logic programming
  - key word or phrase, S101
- logic programming, S101
- Logical variable
  - key word or phrase, S102
- logical variable, S49
- Loop invariant
  - key word or phrase, 597
- Lord Darcy
  - investigator for the Anglo-French empire, S49
- macro hygiene
  - defined, 173
- macros
  - hygienic, 173
- Magnitude
  - key word or phrase, 719
- Make
  - information hiding example, 535
- Managed heap
  - key word or phrase, 294
- manifest type, 39
  - XXX, 540
- mantissa
  - of a floating-point number, S543
- Map
  - key word or phrase, 175
- Mark bit
  - key word or phrase, 296
- Mark-and-sweep collector
  - key word or phrase, 295
- mark-compact collection, 288
- marker method
  - in Smalltalk, 646
- materialization, 181
- mathematical logic, S46
  - as a programming language, S\$46
- member, Prolog predicate, S76
- member\_variant, Prolog predicate, S76
- member?
  - Scheme function, 189
- memory management, S261
  - copying garbage collection, S276
  - mark-and-sweep garbage collection, S271
  - reference counting, S288
- Memory safety
  - key word or phrase, 294
- memory safety, 242, 261
- Message
  - key word or phrase, 716
- message cascade
  - in Smalltalk, 714
- message categories, 623
  - in Smalltalk-80, 710
- Message passing
  - key word or phrase, 716
- message pattern
  - in Smalltalk-80, 710
- Message selector
  - key word or phrase, 717
- message selector, 641
  - in Smalltalk, 622
- metaclass, 640, 702
  - in the  $\mu$ Smalltalk interpreter, 694
- Metalanguage
  - key word or phrase, 326
- metalanguage, 315
- Metatheoretic proof

key word or phrase, 72  
Metatheory  
    key word or phrase, 72  
Metavariable  
    key word or phrase, 69  
metavariables  
    as distinct from program variables, 19  
    in operational semantics, 30  
    that talk about syntax, 19  
Method  
    key word or phrase, 716  
Method dispatch  
    key word or phrase, 717  
method dispatch  
    in Smalltalk, 633  
    introduced, 620  
*method-definition*, 638  
*method-definition*  $\Rightarrow$ , 624  
methods, 619  
Milner, Robin, 326  
mini-index  
    explanation of, 39  
mixed arithmetic  
    in Smalltalk, 660  
    in Smalltalk-80, 713  
*mode-change*, S55  
Model  
    key word or phrase, S103  
*motype*, 547  
Modula-2, 11, 596, 598  
modular type checking, 568  
Module  
    key word or phrase, 596  
module  
    exporting  
        in Molecule, 549  
Molecule  
    relation to Smalltalk, 625  
Monomorphic  
    key word or phrase, 175  
monomorphic, 340, 353  
monomorphic function  
    defined, 133  
Monomorphic type system  
    key word or phrase, 393  
Monotype  
    key word or phrase, 394, 453  
monotype, 418  
more general  
    used to relate two types, 420  
Most general  
    key word or phrase, 453  
Multiple inheritance  
    key word or phrase, 717  
Mustard, Colonel, S106  
mutability, 555  
    of queues, S159  
Mutable  
    key word or phrase, 175  
mutable abstraction, 555  
mutable location, 124  
Mutable reference cell  
    key word or phrase, 326  
mutable reference cell, 190  
mutable state  
    of a machine, 11  
mutant superpowers, 245  
Mutator  
    key word or phrase, 294  
Mutual recursion (code)  
    key word or phrase, 326  
Mutual recursion (data)  
    key word or phrase, 326  
my head hurts, 129  
name  
    of value constructors for abstract syntax, 43  
    qualified, 538, 550  
name equivalence  
    deprecated, 496  
Natural deduction  
    key word or phrase, 71  
natural deduction, 32  
Nested function  
    key word or phrase, 175  
nested patterns, 472  
nil  
    different meanings in different languages, 639  
*not\_member*, Prolog predicate, S91  
nullary type constructors, 352  
Number  
    key word or phrase, 719  
*numeral*, 205, 362, 414, 479, 546  
  
Obj.magic, 409  
Object  
    key word or phrase, 716, S101  
object  
    in Prolog, S47  
object identity, 644  
Object language  
    key word or phrase, 326  
object language, 315  
Object-orientation  
    key word or phrase, 595  
observational equivalence, 112, 644  
    in Impcore exercise, 79  
observers, 113  
    see also “creators, producers, and observers”, 356  
OCaml, 409  
occurrence equivalence  
    deprecated, 496

- Occurs check
  - key word or phrase, S103
- occurs check
  - in ML type inference, 440
  - in Prolog, S64
- Open recursion
  - key word or phrase, 718
- Operational semantics
  - key word or phrase, 70
- operational semantics
  - big-step, 31
  - natural-deduction, 32
  - reduction, 217
  - small-step, 217
- operator overloading, 553
- ordered, Prolog predicate, S78
- overflow detection, S197
- overlapping patterns, 511
- overload resolution
  - in Molecule, 554
- overloaded names
  - resolution of
    - in Molecule, 554
- overloaded operators, 553
- overloading, 134
  - distinct from parametric polymorphism, 392
- Override
  - key word or phrase, 717
- overriding
  - of a Smalltalkmethod, 624
  - of a method, 641
- “own” variables, §126
- P*
  - as primitive function in identifier cross-reference, 39
- pair?
  - Scheme primitive, 113
- parameters
  - accumulating
    - method of, 103
- Parametric polymorphism
  - key word or phrase, 393
- parametric polymorphism, 134
  - of the list abstraction, 118
- Parser
  - key word or phrase, 73
- Parsing
  - key word or phrase, 71
- parsing, S201
  - using shift functions and reduce functions, S204
- partial application
  - of a curried function, 128
- partitioned, Prolog predicate, S79
- Pascal
  - as procedural language, 11
- Pattern
  - key word or phrase, 325
- pattern*, 471, 479, 546
- Pattern matching
  - key word or phrase, 325
- pattern matching
  - in ML code
    - explained, 314
- patterns
  - nested, 472
- permutation, Prolog predicate, S78
- phantom types, S39
- phase distinction, S382
- Pierce, Benjamin, 56
- Poisson distribution
  - in discrete-event simulation, S156
- Polymorphic
  - key word or phrase, 176
- polymorphic, 341, 353
- polymorphic function
  - defined, 133
- Polymorphic type
  - key word or phrase, 326
- Polymorphic type system
  - key word or phrase, 393
- polymorphic, recursive functions
  - in Typed  $\mu$ Scheme, 368
- Polymorphism
  - key word or phrase, 393
- polymorphism, 134
  - ad hoc*, 134
  - ad-hoc*, 554
  - parametric, 134
  - predicative, 550
  - subtype, 134
  - three kinds of, 134
- Polytype
  - key word or phrase, 394, 453
- polytype, 418
- Pop-2 (functional language), 177
- Predefined function
  - key word or phrase, 72
- predefined function
  - defined, 26
- Predicate
  - key word or phrase, S102
- predicate
  - in Prolog, S48
- predicate*, S55
- Predicate logic
  - key word or phrase, S101
- predicative
  - form of polymorphism, 550
- predicative polymorphism, 587
- premises
  - of a Prolog clause, S56
  - of an inference rule, 71
- primitive*, 205

Primitive function  
 key word or phrase, 72

primitive function  
 defined, 26  
 in Impcore environment  $\phi$ , 30

principal module type  
 how to compute, 578

principal module types, 576

Principal type  
 key word or phrase, 453

principal-type  
 testing, 412

printing primitives, 23

printu, 23

private  
 Smalltalkmethod, 626

private message  
 in Smalltalk, 633  
 in  $\mu$ Smalltalk collection classes, 667

private messages  
 recommended for class Natural, 679

Private method  
 key word or phrase, 717

private method  
 in Smalltalk, 717

private methods  
 used to expose representations of numbers, 673

Procedural programming  
 key word or phrase, 72

procedural programming, 11

producers  
 see also “creators, producers, and observers”, 356

product type, 477

program variables  
 as distinct from metavariables, 19

program verification, 563

Projection  
 key word or phrase, 327

projection, 315

Prolog, §S45  
 arity, S53  
 best-known logic programming language, S46  
 difference lists, S79  
 logical interpretation, §S59  
 occurs check, S103  
 primitive predicates, §S72  
 procedural interpretation, §S63  
 differs from logical interpretation, S101  
 real, §S96  
 the cut, S103  
 semantics, §S97  
 syntax, §S96

running programs “backward”, S60

semantics, §§58

syntax, §§56

unification, S64

proofs  
 by calculation, 116

proper tail call, 172

proper tail recursion, 172

Property  
 key word or phrase, S102

property-based testing, 115

Proposition  
 key word or phrase, S102

Propositional logic  
 key word or phrase, S101

propositions as types, 356

Protocol  
 key word or phrase, 716

protocol  
 of an object, 623

pseudorandom numbers  
 generated using the linear congruential method, 126

pure  
 expression, 112  
 language, 413

pure expression, 112

purity  
 of an expression, 112

qualified name, 538, 550

*qualified-name*, 546

quantification  
 existential  
 in algebraic data types, S27

Quantified type  
 key word or phrase, 394

Query  
 key word or phrase, S102

query  
 in Prolog, S50

*query*, S55

queue  
 as a purely functional data structure, 183

quicksorted, Prolog predicate, S79

raise an exception, 593

random numbers  
 generated using the linear congruential method, 126

Reachability  
 key word or phrase, 294

Read-eval-print loop  
 key word or phrase, 72, 327

realization  
 of a module type, 577

Receiver

- key word or phrase, 717
- records
  - how to simulate in Molecule, 543
  - in  $\mu$ Scheme, 109
- recursion
  - avoided with difference lists, S80
- recursion equations, 119
- recursive, polymorphic functions
  - in Typed  $\mu$ Scheme, 368
- redefinition
  - of a Smalltalk method, 624
- Redex
  - key word or phrase, 246
- reduce
  - in parsing, S204
- reduction
  - in a calculus, 243
- Reduction semantics
  - key word or phrase, 246
- Redundant pattern
  - key word or phrase, 325
- redundant pattern, 511
- reference cell
  - mutable, 190
- Reference counting
  - key word or phrase, 296
- refinement
  - of finite maps, 332
- Reflection
  - key word or phrase, 718
- reflection
  - in Prolog, S99
  - in Smalltalk, 640
  - in Smalltalk-80, 714
- Relation
  - key word or phrase, S102
- Rely-guarantee reasoning
  - key word or phrase, 597
- rely-guarantee reasoning
  - to satisfy invariants, 563
- remembered set, 292
- renaming
  - of variables, S62
- Representation
  - key word or phrase, 596
- Representation invariant
  - key word or phrase, 596, 718
- representation invariant
  - of  $\mu$ Smalltalk floating-point numbers, 661
  - of  $\mu$ Smalltalk fractions, 673
- reserved words, 19
- resolution
  - of overloaded names, 554
  - of overloaded names in Molecule, 554
- resolution of overloaded names
  - in Molecule, 554
- reveal
  - type identity, 540
- Root
  - key word or phrase, 294
- root
  - of a module type in an environment, 573
  - of an absolute access path in Molecule, 569
- rose tree, 183
- Rule
  - key word or phrase, S102
- rule
  - of Prolog, S49
- rules of inference, 31
- run-time error, 47
- S-exp*, 95, 205, 362, 414, 479, 546
- S-expression
  - key word or phrase, 176
- S-expressions
  - fully general, 96
  - ordinary, 94
- safety
  - guaranteed by type systems, 336
- satisfying assignment
  - to logical variables in Prolog, S53
- Scheme
  - continuation, 174
  - data abstraction in, 598
  - own variables implemented in, 126
  - Smalltalk blocks related, 647
- scoping
  - dynamic, 150
  - lexical, 150
  - static, 150
- scrutinee, 470, 471, 480
  - evaluated in a case expression, 503
- sealing, 549
  - in the subtype relation, 576
  - introduces new type constructor, 545
- Self
  - computer programming language, 619
- sentinel
  - in  $\mu$ Smalltalk lists, 681
- sets, §106
- Shape, real Smalltalk-80 class, 709
- Shared mutable state
  - key word or phrase, 176
- shared mutable state
  - in a resettable counter, 126
- shift-reduce parsing, S204
- Short-circuit conditionals
  - key word or phrase, 326
- Short-circuit evaluation
  - key word or phrase, 176

side effect, 16  
 side effects, 20, 352  
 simple  
     not the same as easy, 133  
 Simula 67, 596, 597  
     inspiration for Smalltalk, 715, 720  
 Simula-67, 619  
 Single inheritance  
     key word or phrase, 717  
 single-threaded store, 146, 156  
     implemented by update in place, 146, 156  
 Skolem type, S33  
 Small-step semantics  
     key word or phrase, 71, 246  
 Smalltalk, 7, 596, §619  
     abstract class, 632, 654, 655, 657, 669  
     answer, 620  
     full, §707  
         binary messages, 711  
         class hierarchy, 635, 651, 712  
         class variables, 712  
         Collection hierarchy, §713  
         literals, 709  
         numbers, §712  
         precedence, 711  
         semantics, §711  
         syntax, 634, §709  
         UndefinedObject class, §663  
     instance variable, 624, 625, 640  
     message, 620, 641  
     method, 620, 641  
     method dispatch, 641  
     method search, 642, 663, 733  
     mutable types, 651, 657, S545  
     polymorphism, 655  
     receiver, 620  
     relation to Molecule, 625  
     self, 641  
     Simula 67 inspiration for, 715, 720  
     simulation in, §S151  
     super, 642  
     super, 641, 642  
     syntax, §637  
 $\mu$ Smalltalk  
     arity of message names, 637  
 smart constructor  
     in binary trie, 523  
 snoeced  
     folk etymology, S75  
 snoeced, Prolog predicate, S75  
 sorting, \$103  
 Soundness  
     key word or phrase, S103  
 soundness  
     as applied to inference rules, 429  
     of a type system, 359  
     of algebraic laws  
     mentioned but not explained, 66  
 soundness proof  
     for inference rules using explicit substitutions, 429  
 square brackets  
     recommended usage, 119  
 Stack allocation  
     key word or phrase, 294  
 static scoping, 150  
 stone knives  
     used to implement abstract data types, 598  
 Store  
     key word or phrase, 176  
 store  
     single-threaded, 146, 156  
     implemented by update in place, 146, 156  
 streams  
     used in  $\mu$ Scheme to read definitions from a file, S369  
 strengthening, 569, 580  
 structural equivalence  
     in type systems, 495  
 structured control-flow constructs, 11  
 Structured operational semantics  
     key word or phrase, 246  
 stubborn, S178  
 Subclass  
     key word or phrase, 717  
 Subgoal  
     key word or phrase, S102  
 subgoals  
     of a Prolog clause, S56  
 Substitution  
     key word or phrase, 453, S102  
 substitution  
     as the essential operation on quantified types, 365  
     capture-avoiding, 374  
     in the implementation of syntactic sugar, 167  
     in translations for  $\mu$ Scheme, 164  
     of equals for equals, 115  
     while avoiding capture, 167  
 subtype polymorphism, 134  
 Subtyping  
     key word or phrase, 596  
 subtyping  
     behavioral, 637  
     rules for, 575  
 suggest, Prolog predicate, S92  
 sum of products, 477  
     in Smalltalk, 634  
 sum type, 477  
 sum types

- poor support in C, 42
- super**
  - messages to, 642
- Superclass**
  - key word or phrase, 717
- superclass**
  - Shape as an example of, 630
- superpowers**
  - of call/cc, 245
- suspension, S249**
- symbol**
  - in Scheme, 94
- Symbol table**
  - key word or phrase, 70
- syntactic abstraction, 173**
- Syntactic category**
  - key word or phrase, 70
- syntactic category, 16**
  - types, 338
- Syntactic form**
  - key word or phrase, 70
- syntactic form, 15**
- Syntactic proof**
  - key word or phrase, 71
- Syntactic sugar**
  - key word or phrase, 70, 176
- syntactic sugar, 26, 67**
  - define described as, 122
  - for while\*, do-while, and for, 69
- syntax error, 47**
- syntax, context-free, §S13**
- System F, 394**
- table**
  - also called finite map, 107
- tail call, 172**
- Term**
  - key word or phrase, 393, S102
- term**
  - a theoretician's name for an expression, 342
- term, S55**
- terms**
  - in Prolog, S47
- testing**
  - property-based, 115
- The cut**
  - key word or phrase, S103
- the zipper**
  - in μML, 520
- Theory**
  - key word or phrase, 71
- throwing an exception, 210**
- thunk, S249**
- Tiger, tiger, tiger, S388**
- tokens, S201**
- toplevel, S13**
- tracing**
- Smalltalk, S558**
- transforms, Prolog predicate, S91**
- tree traversal, §109, 271**
- Tricolor marking**
  - key word or phrase, 296
- Trotsky, Leon, 138**
- true definition, 19**
- Type**
  - key word or phrase, 393, 452
- type, 339**
- Type abbreviation**
  - key word or phrase, 326
- type abbreviation, 539**
  - in ML, 310
- Type Abstraction**
  - key word or phrase, 394
- type abstraction**
  - typing rule for, 374
- Type Application**
  - key word or phrase, 394
- type application, 367**
  - typing rule for, 374
- Type Checker**
  - key word or phrase, 393
- type checking, 335**
  - modular, 568
- Type Constructor**
  - key word or phrase, 394
- type constructor**
  - introduced by sealing, 545
- type equality**
  - dire warning about, 340
  - in Molecule, 570
- type equivalence, 376**
- type erasure, 389**
  - converting Typed μScheme into μScheme, 389
- Type inference**
  - key word or phrase, 452
- type inference, 335**
- type name, S425**
- type parameter, 365**
- type predicates**
  - built into μScheme, 94
  - for records, 110
- type reconstruction, 335**
- Type scheme**
  - key word or phrase, 452
- type scheme, 418**
- type signatures**
  - in Haskell, S36
- Type System**
  - key word or phrase, 393
- Type variable**
  - key word or phrase, 326
- type variable**
  - captured by a naïve instantiation, 383

key to understanding thereof, 365  
*type-exp*, 362, 414, 479, 546  
 type-formation rules, 354  
 type-level expression, 361, 366  
 type-lambda  
     in Typed  $\mu$ Scheme, 361, 368  
 typing  
     of definition, 375

$\mu$ ML  
     syntax, §479  
 unchecked run-time error, 40  
     from using an unspecified value, 153  
     in printing functions, 47  
     in `getline_`, S178

Undelimited continuation  
     key word or phrase, 247

unfold, S251  
 Unicode code point, 23  
 Unification  
     key word or phrase, 453, S102  
 unification  
     in type inference, 428  
 unifier, 428  
 uninhabited type, 342  
 union-find  
     relation to type inference, 441  
*unit-test*, 18, 95, 205, 339, 362, 546, 638, S55  
 Unreachable  
     key word or phrase, 295  
 unsafe functions  
     for printing, 47  
 unspecified  
     in Typed  $\mu$ Scheme semantics, 390  
     in  $\mu$ Scheme semantics, 153  
 unspecified values  
     preventing misuse by careless persons, S318  
 unwinding  
     a call stack, 593  
*use*, S55  
*use*  
     explained, 22  
 user-defined function  
     in Impcore environment  $\phi$ , 30

val  
     explained, 22  
 Valgrind, S332  
 Value constructor  
     key word or phrase, 325  
 value constructor, 469  
     equivalent in Prolog, S56  
     in Molecule, 543  
     misspelled  
         consequences of, 482  
 value constructors for abstract syntax constraints on the choice of names, 43  
 value restriction, S578  
     in ML, S84  
 Value semantics  
     key word or phrase, 176  
 value variable  
     contrasted with value constructor, 482  
*value-constructor-name*, 479, 546  
 variable  
     meaning in Smalltalk-80, 624  
     value variable, 482  
*variable*, S55  
 variable capture  
     in type-lambda, 374  
     of type variables, 383  
 variables  
     program variables vs metavariables, 19  
 variadic function  
     defined, S189  
 variadic functions  
     used for extensible printers, S189  
 verification  
     of invariants, 563

WHILEX  
     why it isn't called WHILE, 43

wildcard pattern  
     in ML, 314

Work per allocation  
     key word or phrase, 295

zipper  
     in  $\mu$ ML, 520



# B

## Arithmetic

In the 21st century, many programmers take numbers for granted. Computer-science students rarely get more than a week's worth of instruction in the properties of floating-point numbers, and many programmers are barely aware that machine integers have limited precision. So many languages provide arbitrary-precision arithmetic on integers or rational numbers that you don't even need to know how the tricks are done. This supplemental chapter, together with Exercises 49 and 50 in Chapter 9 and Exercises 37 and 38 in Chapter 10, will teach you. And if you do both sets of exercises, you'll see how abstract data types compare with objects: when inspecting representations of multiple arguments, abstract data types make the abstractions easier to code but less flexible in use.

In programming as in math, numbers start with integers. You may not think of `int` as an abstract type, but it is. It is, however, an unsatisfying abstraction. Values of type `int` aren't true integers; they are *machine integers*. Although machine integers get bigger as hardware gets bigger—a typical machine integer occupies a machine word or half a machine word—they are always limited in precision. A 32-bit or 64-bit integer is good for many purposes, but some computations need more precision; examples include some cryptographic computations as well as exact rational arithmetic. *Arbitrary-precision* integer arithmetic is limited only by the amount of memory available on a machine. It is supported in many languages, and in highly civilized languages like Scheme, Smalltalk, and Python, arbitrary precision is the default.

Arbitrary-precision arithmetic makes a fine case study in information hiding. The concepts and algorithms are explained below, and I encourage you to implement them using both abstract data types (Chapter 9) and objects (Chapter 10). The similarities and differences among implementations illuminate what abstract data types are good at and what objects are good at.

Arbitrary-precision arithmetic begins with natural numbers—the nonnegative integers. Basic arithmetic includes addition, subtraction, multiplication, and division. An interface for natural numbers, written in Molecule, is shown in Figure B.1 on page S16. There are just a couple of subtleties:

- The difference of two natural numbers isn't always a natural number; for example,  $19 - 83$  is not a natural number. If `-` is used to compute such a difference, it halts the program with a checked run-time error. If you want such a difference not to halt your program, you can use continuation-passing style (Section 2.10): calling `(cps-minus n1 n2 ks kf)` computes the difference  $n_1 - n_2$ , and when the difference is a natural number, `cps-minus` passes it to success continuation `ks`. Otherwise, `cps-minus` calls failure continuation `kf` without any arguments.
- For efficiency, we compute quotient and remainder together. (This is true even in hardware.) Storing quotient and remainder is the purpose of record type `QR.pair`.

```
S16a. ⟨nat.mcl S16a⟩≡
  (module-type NATURAL
    (exports [abstype t]
      [of-int : (int -> t)] ; creator
      [+ : (t t -> t)] ; producer
      [- : (t t -> t)] ; producer
      [* : (t t -> t)] ; producer
      [module [QR : (exports-record-ops pair
        ([quotient : t]
         [remainder : int]))]]
        [sdiv : (t int -> QR.pair)] ; producer
        [compare : (t t -> Order.t)] ; observer
        [decimal : (t -> (@m ArrayList Int).t)] ; observer
          ; decimal representation, most significant digit first
        [cps-minus : (t t (t -> unit) (-> unit) -> unit))])
          ; subtraction, using continuations
```

Figure B.1: An abstraction of natural numbers

- Long division—that is, division of a natural number by another natural number—is beyond the scope of this book. Instead, we divide a natural number only by a (positive) machine integer. This “short division” is implemented by function sdiv.

A natural number can be represented easily and efficiently as a sequence of *digits* in a given *base*. The algorithms for basic arithmetic, which you may have learned in primary school, work digit by digit. In everyday life, we use base  $b = 10$ , and we write the most significant digit  $x_n$  on the left. In hardware, our computers famously use base  $b = 2$ ; the word “bit” is a contraction of “binary digit.” Regardless of base, a single digit  $x_i$  is an integer in the range  $0 \leq x_i < b$ . In arbitrary-precision arithmetic, we pick as large a  $b$  as possible, subject to the constraint that every arithmetic operation on digits must be doable in a single machine operation.

As taught to schoolchildren, arithmetic algorithms use base  $b = 10$ , but the algorithms are independent of  $b$ , as should be your implementation. The algorithms do depend, however, on the representation of a sequence of digits. I discuss two representations:

- We can represent a sequence as a list of digits, which is either empty or is a digit followed by a sequence of digits. If  $X$  is a natural number, one of the following two equations holds:

$$\begin{aligned} X &= 0 \\ X &= x_0 + X' \cdot b \end{aligned}$$

where  $x_0$  is a digit and  $X'$  is a natural number. (It is possible to begin with  $x_n$  instead of  $x_0$ , but the so-called “little-endian” representation, with the least-significant digit on the left, simplifies all the computations.) A suitable representation might use an algebraic data type (Chapters 8 and 9):

**S16b.** ⟨representation of natural numbers as a list of digits S16b⟩≡
 

```
(data t
  [ZERO : t]
  [DIGIT-PLUS-NAT-TIMES-b : (int t -> t)])
```

Another possibility is to use objects: a class `NatZero` with no instance variables, and a class `NatNonzero` with instance variables  $x_0$  and  $X'$ .

Mathematicians and physicists often multiply quantities simply by placing one next to another; for example, in the famous equation  $E = mc^2$ ,  $m$  and  $c^2$  are multiplied. But in a textbook on programming languages, this notational convention will not do. First, it is better for multiplication to be visible than to be invisible. And second, when one name is placed next to another, it usually means function application—at least that’s what it means in ML, Haskell, and the lambda calculus.

Among the conventional infix operators,  $*$  is more suited to code than to mathematics, and the  $\times$  symbol is better reserved to denote a Cartesian product in a type system. In this book, on the rare occasions when we need to multiply numbers, I write an infix  $\cdot$ , so Einstein’s famous equation would be written  $E = m \cdot c^2$ .

A good invariant, no matter what the representation, is that for either  $(\text{DIGIT-PLUS-NAT-TIMES-b } x_0 X')$  or  $\text{NatNonzero}$ ,  $x_0$  and  $X'$  are not both zero. The abstraction function is

$$\begin{aligned}\mathcal{A}(\text{ZERO}) &= 0 \\ \mathcal{A}((\text{DIGIT-PLUS-NAT-TIMES-b } x_0 X')) &= x_0 + X' \cdot b\end{aligned}$$

- Alternatively, we can represent a sequence as an array of digits, that is,  $X = x_0, \dots, x_n$ . The abstraction function is

$$\mathcal{A}(X) = \sum_{i=0}^n x_i \cdot b^i$$

In both representations, every digit  $x_i$  satisfies the invariant  $0 \leq x_i < b$ .

Here are the design tradeoffs: Using the list representation, the algorithms are easy to code, but the representation requires roughly double the space of the array representation. Using the array representation, not all the algorithms are as easy to code, but the representation requires half the space of the list representation. The rest of this section shows algorithms for both representations.

## B.1 ADDITION

Adding two digits doesn’t always produce a digit. For example, if  $b = 10$ , the sum  $3 + 9$  is not a digit. To express the sum, we say that it *carries out* 1, which we write  $3 + 9 = 2 + 1 \cdot 10^1$ . The carried 1 is added to the sum of the next digits, at which time it is called a “carry in,” as in this example:

$$\begin{array}{r} 1 \\ 73 \\ +89 \\ \hline 162 \end{array}$$

The small 1 over the 7 is the “carry out” from adding 3 and 9, and it is “carried in” to the sum of 7 and 8, producing 16.

To turn the example into an algorithm, we start with the list representation, and we consider how to add nonzero natural numbers  $X = x_0 + X' \cdot b$  and

$b$	Base of multiprecision arithmetic
$X, Y$	A natural number that is added, subtracted, multiplied, or divided by
$x_0, y_0$	Least-significant digit ( $X \bmod b, Y \bmod b$ )
$x_i, y_i$	Digit $i$ of a natural number
$X', Y'$	Sequence of most-significant digits ( $X \div b, Y \div b$ )
$Z$	Sum, difference, or product
$z_i$	Digit $i$ of $Z$
$c_i$	Carry in at position $i$
$c_{i+1}$	Carry out at position $i$ (also carry in at position $i + 1$ )
$d$	Divisor
$Q$	Quotient
$q_0$	Least-significant digit of quotient ( $Q \bmod b$ )
$q_i$	Digit $i$ of quotient, $0 \leq q_i < b$
$Q'$	Most-significant digits of quotients ( $Q \div b$ )
$r$	Remainder, always $0 \leq r < d$
$r'_i$	“Remainder in” at digit $i$ , $0 \leq r'_i < d$
$r_i$	“Remainder out” at digit $i$ , $0 \leq r_i < d$

Table B.2: Metavariables used to describe multiprecision arithmetic

$Y = y_0 + Y' \cdot b$ . We first add the two least-significant digits  $x_0 + y_0$ , then add any resulting carry out to  $X' + Y'$ . To specify the algorithm precisely, we resort to algebra.

The sum of  $X$  and  $Y$  can be expressed as

$$X + Y = (x_0 + X' \cdot b) + (y_0 + Y' \cdot b) = (x_0 + y_0) + (X' + Y') \cdot b.$$

Because sum  $x_0 + y_0$  might be too big to fit in a digit, this right-hand side does not immediately determine a valid representation of the sum. To get a valid representation, we calculate the least-significant digit  $z_0$  of the sum and the carry out  $c_1$ :

$$\begin{aligned} z_0 &= (x_0 + y_0) \bmod b \\ c_1 &= (x_0 + y_0) \div b \end{aligned}$$

Now  $x_0 + y_0 = z_0 + c_1 \cdot b$ , and we can rewrite the sum as

$$X + Y = z_0 + (X' + Y' + c_1) \cdot b.$$

This right-hand side *does* immediately determine a good representation:  $z_0$  can be represented as a digit, and the sum  $X' + Y' + c_1$  can be represented as a natural number. The right-hand side also suggests that the general form of addition should compute sums of the form  $X + Y + c$ . Such sums can be expressed using a three-argument “add with carry” function,  $adc(X, Y, c)$ . Function  $adc$  is specified by these equations:

$$\begin{aligned} adc(0, Y, c_0) &= Y + c_0 \\ adc(X, 0, c_0) &= X + c_0 \\ adc(x_0 + X' \cdot b, y_0 + Y' \cdot b, c_0) &= z_0 + (X' + Y' + c_1) \cdot b, \\ &\quad \text{where } z_0 = (x_0 + y_0 + c_0) \bmod b \\ &\quad \quad c_1 = (x_0 + y_0 + c_0) \div b \end{aligned}$$

In the example shown above, where we add 73 and 89,

$$x_0 = 3 \quad X' = 7 \quad y_0 = 9 \quad Y' = 8 \quad c_0 = 0 \quad z_0 = 2 \quad c_1 = 1$$

Given an  $X$  and a  $Y$  represented as lists, function *adc* is most easily implemented recursively, using case expressions to scrutinize the forms of  $X$  and  $Y$ . It needs an auxiliary function to compute  $Y + c_0$  and  $X + c_0$ , the specification of which is left as Exercise 11.

When  $X$  and  $Y$  are represented as arrays, function *adc* is not as easy to implement. A better approach instead loops on an index  $i$ ; at each iteration, the loop computes one digit  $z_i$  and one carry bit  $c_{i+1}$ :

$$\begin{aligned} z_i &= (x_i + y_i + c_i) \bmod b \\ c_{i+1} &= (x_i + y_i + c_i) \text{ div } b \end{aligned}$$

The initial carry in  $c_0$  is zero.

If  $X$  has  $n$  digits and  $Y$  has  $m$  digits, we require

$$X + Y = Z = \sum_{i=0}^{\max(m,n)+1} z_i \cdot b^i.$$

The computations of  $z_i$  and  $c_{i+1}$  are motivated by observing

$$\begin{aligned} X + Y &= \left( \sum_{i=0}^n x_i \cdot b^i \right) + \left( \sum_{j=0}^m y_j \cdot b^j \right) \\ &= \sum_{i=0}^{\max(m,n)} x_i \cdot b^i + y_i \cdot b^i \\ &= \sum_{i=0}^{\max(m,n)} (x_i + y_i) \cdot b^i \end{aligned}$$

and

$$x_i + y_i + c_i = z_i + c_{i+1} \cdot b.$$

In the example shown above, where we add 73 and 89,

$$\begin{aligned} z_0 + c_1 \cdot b &= x_0 + y_0 + c_0, \quad \text{where } x_0 = 3, y_0 = 9, c_0 = 0, z_0 = 2, c_1 = 1 \\ z_1 + c_2 \cdot b &= x_1 + y_1 + c_1, \quad \text{where } x_1 = 7, y_1 = 8, c_1 = 1, z_1 = 6, c_2 = 1 \\ z_2 + c_3 \cdot b &= x_2 + y_2 + c_2, \quad \text{where } x_2 = 0, y_2 = 0, c_2 = 1, z_2 = 1, c_3 = 0 \end{aligned}$$

## B.2 SUBTRACTION

The algorithm for subtraction resembles the algorithm for addition, but the carry bit is called a “borrow,” and it works a little differently. If  $Z = X - Y$ , then digit  $z_i$  is computed from the difference  $x_i - y_i - c_i$ , where  $c_i$  is a borrow bit. If this difference is negative, you must borrow  $b$  from a more significant digit, exploiting the identity

$$z_{i+1} \cdot b^{i+1} + z_i \cdot b^i = (z_{i+1} - 1) \cdot b^{i+1} + (z_i + b) \cdot b^i.$$

If no more significant digit is available to borrow from, the difference is negative and therefore is not representable as a natural number—and the subtraction function must transfer control to a failure continuation (or halt with a checked run-time error).

An algorithm that uses the array representation can loop on  $i$ , just as for addition, and it can keep track of the borrow bit  $c_i$  at each iteration. An algorithm that uses the list representation can use a recursive function  $sbb$  (subtract with borrow), which is specified by these equations for  $sbb(X, Y, c) = X - Y - c$ :

## B Arithmetic

S20

$$\begin{aligned}
 sbb(X, 0, 0) &= X \\
 sbb(X, 0, 1) &= X - 1 \\
 sbb(0, y_0 + Y' \cdot b, c) &= 0, && \text{if } y_0 = 0 \text{ and } Y' = 0 \text{ and } c = 0 \\
 sbb(0, y_0 + Y' \cdot b, c) &= \text{error}, && \text{if } y_0 \neq 0 \text{ or } Y' \neq 0 \text{ or } c \neq 0 \\
 sbb(x_0 + X' \cdot b, y_0 + Y' \cdot b, c) &= x_0 - y_0 - c + sbb(X', Y', 0) \cdot b, && \text{if } x_0 - y_0 - c \geq 0 \\
 sbb(x_0 + X' \cdot b, y_0 + Y' \cdot b, c) &= b + x_0 - y_0 - c + sbb(X', Y', 1) \cdot b, && \text{if } x_0 - y_0 - c < 0
 \end{aligned}$$

The specification of an algorithm for computing  $X - 1$  is left as Exercise 11 in Chapter 9.

### B.3 MULTIPLICATION

To compute the product of two natural numbers  $X$  and  $Y$ , we compute the partial products of all the pairs of digits, then add the partial products. Here's an example:

$$\begin{array}{r}
 7 \quad 3 \\
 8 \quad 9 \\
 \hline
 2 \quad 7 \\
 2 \quad 4 \\
 1 \quad 6 \quad 3 \\
 5 \quad 6 \\
 \hline
 6 \quad 4 \quad 9 \quad 7
 \end{array}$$

As in the case of addition, the product of two digits  $x_i \cdot y_i$  might not be representable as a digit, so we compute

$$\begin{aligned}
 z_{hi} &= (x_i \cdot y_i) \text{ div } b \\
 z_{lo} &= (x_i \cdot y_i) \text{ mod } b \\
 x_i \cdot y_i &= z_{lo} + z_{hi} \cdot b,
 \end{aligned}$$

and both  $z_{hi}$  and  $z_{lo}$  are representable as digits.

To multiply two natural numbers represented as lists, we use these equations:

$$\begin{aligned}
 X \cdot 0 &= 0 \\
 0 \cdot Y &= 0 \\
 (x_0 + X' \cdot b) \cdot (y_0 + Y' \cdot b) &= z_{lo} + (z_{hi} + x_0 \cdot Y' + X' \cdot y_0) \cdot b + (X' \cdot Y') \cdot b^2, \\
 &\text{where } z_{hi} = (x_0 \cdot y_0) \text{ div } b \\
 &z_{lo} = (x_0 \cdot y_0) \text{ mod } b
 \end{aligned}$$

That last equation unpacks into these steps:

1. Turn each single digit  $z_{lo}$ ,  $z_{hi}$ ,  $x_0$ , or  $y_0$  into a natural number, by forming  $z_{lo} = z_{lo} + 0 \cdot b$ , and so on.
2. Use recursive calls to multiply natural numbers  $x_0 \cdot Y'$ ,  $X' \cdot y_0$ , and  $X' \cdot Y'$ .
3. Add up natural numbers  $z_{hi}$ ,  $x_0 \cdot Y'$ , and  $X' \cdot y_0$  into an intermediate sum  $S$ , then multiply  $S \cdot b$  by forming the natural number  $0 + S \cdot b$ .
4. Compute  $(X' \cdot Y') \cdot b^2$  by forming the natural number  $0 + (0 + (X' \cdot Y') \cdot b) \cdot b$ .
5. Add the three natural-number terms of the right-hand side.

§B.4

*Short division*

S21

To multiply two natural numbers represented as arrays, we compute

$$\begin{aligned} X \cdot Y &= \left( \sum_i x_i b^i \right) \cdot \left( \sum_j y_j b^j \right) \\ &= \sum_i \sum_j (x_i \cdot y_j) \cdot b^{i+j} \end{aligned}$$

Again, to satisfy the representation invariant, each partial product  $(x_i \cdot y_j) \cdot b^{i+j}$  has to be split into two digits  $((x_i \cdot y_j) \bmod b) \cdot b^{i+j} + ((x_i \cdot y_j) \text{ div } b) \cdot b^{i+j+1}$ . Then all the partial products are added.

## B.4 SHORT DIVISION

Long division, in which you divide one natural number by another, is beyond the scope of this book. Consult Hanson (1996) or Brinch Hansen (1994). But short division, in which you divide a big number by a digit, is within the scope of the book, and it is used to implement `print`: to convert a large integer to a sequence of decimal digits, we divide it by 10 to get its least significant digit (the remainder), then recursively convert the quotient.

Here is an example of short division in decimal. When 1528 is divided by 7, the result is 218, with remainder 2:

$$\begin{array}{r} 0 \ 2 \ 1 \ 8 \\ 7 \overline{) 1 \ 5 \ 1 \ 2 \ 5 \ 8} \text{ remainder } 2 \end{array}$$

Short division works from the most-significant digit of the dividend down to the least-significant digit:

1. We start off dividing 1 by 7, getting 0 with remainder 1. Quotient 0 goes above the line (producing the most-significant digit of the overall quotient), and the remainder is multiplied by 10 and added to the next digit of the dividend (5) to produce 15.
2. When 15 is divided by 7, quotient 2 goes above the line (producing the next digit of the overall quotient), and remainder 1 is combined with the next digit of the dividend (2) to produce 12.
3. When 12 is divided by 7, quotient 1 goes above the line (producing the next digit of the overall quotient), and remainder 5 is combined with the next digit of the dividend (8) to produce 58.
4. When 58 is divided by 7, quotient 8 goes above the line (producing the final digit of the overall quotient), and remainder 2 is the overall remainder.

To turn the example into an algorithm, we consider large-integer dividend  $X$  divided by small-integer divisor  $d$ , from which we compute large-integer quotient  $Q$  and small-integer remainder  $r$ , satisfying

$$X = Q \cdot d + r \quad 0 \leq r < d.$$

The algorithm is easiest to specify when  $X$  is represented as a list of digits.

If  $X$  is zero, both  $Q$  and  $r$  are also zero. If  $X$  is nonzero, then it has the form  $x_0 + X' \cdot b$ , and we start with the most-significant digits  $X'$ . We recursively divide  $X'$  by  $d$ , giving quotient  $Q'$  and remainder  $r'$ . To get the final quotient  $Q = q_0 + Q' \cdot b$  and remainder  $r$ , we divide machine integer  $x_0 + r' \cdot b$  by  $d$ :

$$\begin{aligned} X = x_0 + X' \cdot b &= (q_0 + Q' \cdot b) \cdot d + r \\ \text{where } q_0 &= (x_0 + r' \cdot b) \text{ div } d \\ r &= (x_0 + r' \cdot b) \text{ mod } d \end{aligned}$$

In our example above,

$$\begin{array}{lll} X = 1528 & d = 7 & q_0 = 8 \\ x_0 = 8 & Q' = 21 & Q = 218 \\ X' = 152 & r' = 5 & r = 2 \end{array}$$

When  $X$  is represented as an array, the algorithm loops *down* over index  $i$ , starting with  $i = n$  and going down to  $i = 0$ . At each iteration, the algorithm computes a digit  $q_i$  of the quotient, and it computes an intermediate remainder  $r_i$ . That remainder is then named  $r'_{i-1}$ , where it is combined with digit  $x_{i-1}$  to be divided by  $d$ . Here are the equations:

$$\begin{array}{lll} q_i = (r'_i \cdot b + x_i) \text{ div } d & & r = r_0 \\ r_i = (r'_i \cdot b + x_i) \text{ mod } d & & r'_{i-1} = r_i \\ & & r'_n = 0 \end{array}$$

In the example on page S21,

$$\begin{array}{lll} x_3 = 1 & d = 7 & q_3 = (0 \cdot 10 + 1) \text{ div } 7 = 0 \\ x_2 = 5 & r'_3 = 0 & r_3 = (0 \cdot 10 + 1) \text{ mod } 7 = 1 \\ x_1 = 2 & & q_2 = (1 \cdot 10 + 5) \text{ div } 7 = 2 \\ x_0 = 8 & & r_2 = (1 \cdot 10 + 5) \text{ mod } 7 = 1 \\ & & q_1 = (1 \cdot 10 + 2) \text{ div } 7 = 1 \\ & & r_1 = (1 \cdot 10 + 2) \text{ mod } 7 = 5 \\ & & q_0 = (5 \cdot 10 + 8) \text{ div } 7 = 8 \\ & & r_0 = (5 \cdot 10 + 8) \text{ mod } 7 = 2 \end{array}$$

## B.5 CHOOSING A BASE OF NATURAL NUMBERS

The algorithms above are independent of the base  $b$ . This base should be hidden from client code, so you can choose any base that you want. What base should you choose? For best performance, choose the largest  $b$  such that every intermediate value of every computation can be represented as an atomic value.

**S23.** *(Molecule's predefined module types S23) ≡*

```
(module-type INT
  (exports [abstype t]
    [+ : (t t -> t)] [

```

§B.6  
*Signed-integer arithmetic*

Figure B.3: An interface to integer arithmetic

S23

Should you find yourself working with assembly code or with machine instructions, your atomic value would be a machine word. You would have access to a hardware “flag” or other register that could hold a carry bit or borrow bit, and also to an “extended multiply” instruction that would provide the full two-word product of two one-word multiplicands. The result of every intermediate computation would be right there in the hardware, and you would choose  $b = 2^k$ , where  $k$  would be the number of bits in a machine word.

When you’re working with a high-level language, your atomic value is a value of type `int`. But you probably *don’t* have access to an add-with-carry instruction or an extended-multiply instruction. More likely, you are stuck with an `int` that has only 32 or 64 bits—or in some cases, even fewer bits. You have to choose  $b$  small enough so that an `int` can represent any possible intermediate result:

- To implement addition and subtraction, you must be able to represent a sum which may be as large as  $2 \cdot b - 1$ .
- To implement multiplication, you must be able to represent a partial product which may be as large as  $(b - 1)^2$ .
- To implement division, you must be able to represent the combination of a remainder with a digit, which may be as large as  $(d - 1) \cdot b + (b - 1)$ . If  $d \leq b$ , this combination may be as large as  $b^2 - 1$ .

Depending on niceties of signed versus unsigned arithmetic, and whether values of type `int` occupy 32 bits or 64, you can usually get good results with  $b = 2^{15}$  or  $b = 2^{31}$ . (Using a power of 2 makes computations mod  $b$  and div  $b$  easy and fast.)

## B.6 SIGNED-INTEGER ARITHMETIC

Arithmetic on natural numbers can be leveraged to implement arithmetic on full, signed integers. One possible interface, written in Molecule, is shown in Figure B.3. While machine arithmetic typically uses a two’s-complement representation of integers, for arbitrary-precision arithmetic, I recommend a representation that tracks the *sign* and *magnitude* of an integer. If you’re using Molecule, here are three good representations:

- Represent the magnitude and sign independently.
- Define an algebraic data type that encodes the sign in a value constructor, and apply the value constructor to the magnitude, as in (`NEGATIVE mag`).

- Define an algebraic data type with *three* value constructors: one each for positive numbers, negative numbers, and zero. A value constructor for a positive or negative number is applied to a magnitude. The value constructor for zero is an integer all by itself.

If you're using  $\mu$ Smalltalk, there's only one sensible choice: as described in Section 10.7, use classes `LargePositiveInteger` and `LargeNegativeInteger`.

Sign and magnitude can also be used to specify the abstraction, and if you do so, you can specify most operations using algebraic laws. Some examples:

S24

$$\begin{array}{ll}
 +N ++M = +(N + M) & +N < +M = N < M \\
 +N - -M = +(N - M), \text{ when } N \geq M & +N < -M = \#f \\
 +N - -M = -(M - N), \text{ when } N < M & \text{negated}(+N) = -N \\
 +N + 0 = +N & \text{negated}(0) = 0
 \end{array}$$

The implementation of these laws depends on the programming language. If we're using abstract data types in Molecule, our code can inspect the representations of two integers at once, and the signed-integer operations can be implemented by pattern matching on pairs. If we're using objects in  $\mu$ Smalltalk, our code will have to identify some representations using double dispatch (Section 10.7.3).