### Preface contents

<table>
<thead>
<tr>
<th>What you will learn and how</th>
<th>vi</th>
<th>What to expect in each chapter</th>
<th>xvi</th>
</tr>
</thead>
<tbody>
<tr>
<td>The parts of the book</td>
<td>viii</td>
<td>Software infrastructure</td>
<td>xvi</td>
</tr>
<tr>
<td>Foundations</td>
<td>viii</td>
<td>How realistic are the bridge languages?</td>
<td>xvii</td>
</tr>
<tr>
<td>The Big Three</td>
<td>x</td>
<td>Designing a course to use this book</td>
<td>xix</td>
</tr>
<tr>
<td>Bonus features</td>
<td>xii</td>
<td>Final words for learners</td>
<td>xxi</td>
</tr>
<tr>
<td>What else you will find</td>
<td>xiii</td>
<td>Acknowledgments</td>
<td>xxi</td>
</tr>
<tr>
<td>Thematic threads</td>
<td>xiii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The scope of this book</td>
<td>xv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Preface

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 anonymous student evaluations

This book is about programming languages, but 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 will be longer, uglier, less robust, and harder to debug than it should be. If you know about programming languages but you don’t program, what is your knowledge for? To know what’s in a language is good, but to use the language is better.

People who know about programming languages also know some math. But if math is what you want most, you want another book. This book goes deep into programming practice and only lightly into math.

To know about programming languages, what should you learn? Don’t try to see as many languages as possible, or even to see each major family of languages. Unless you are interested in history, a comprehensive study of programming languages or programming-language features is not in your best interest. Instead, master language-design ideas of lasting value. Learn the very best the field has to offer. Learn what the great ideas are, in what guises they appear, how to recognize them, and how to use them. Great language-design ideas don’t appear just in today’s programming languages: they come from languages of the past, and they will continue to appear in languages of the future.

As a field, 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, to choose and use programming languages intelligently, and to place the languages of the future into a context that will enable you to use them effectively. But unless you are mathematically inclined, a book that is organized around formalism first and foremost—and there are some great ones—is daunting. This book is organized around languages, and formalism is a servant, not your master.
What you will learn and how

*Programming Languages: Build, Prove, and Compare* serves three primary goals:

- To help you use different programming languages effectively
- To help you describe programming languages precisely
- To help you understand and enjoy the diversity of programming languages

These goals are served through case studies: you experiment with and compare code written in different languages. I want you to use important programming-language features to write interesting code, to understand how each feature is implemented, and to develop a sense of how different languages are similar and how each one is distinctive. To have such experiences, you’re best off avoiding full implementations of real programming languages: the implementations are too big and complicated to learn from, and real programming languages differ in such inessential ways that they are too hard to compare. This book provides an alternative; I have distilled five 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. Putting these features into bridge languages, instead of whole languages, makes deep learning possible.

Each bridge language is small enough to learn, but big enough to act as a bridge to the real thing. To foster learning, almost 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 implemented by an interpreter, which is presented in depth. The interpreters are carefully crafted and documented, and the code is worth studying. An interpreter 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. An interpreter also makes it possible for you to learn language design in an eye-opening way: instead of just studying other people’s language designs, you can create your own. 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 that 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 learn by building and modifying programs. You write code in the bridge languages, and you modify the interpreters. Each chapter includes many exercises, and each set of exercises is accompanied by a short *guide*. Whether you are using the book on your own or you are using it to teach others, the guide will help you find exercises that fit you.

**Prove, to keep things simple and precise** This book is not about proofs, but if you are studying programming languages, you should get a little experience with proof. Try some exercises in Chapter 1 (language metatheory), Chapter 2 (equational reasoning), Chapters 6 and 7 (type-system metatheory), and Chapter 10 (more equational reasoning).

Proofs aren’t just for you; they’re also for me: to make proofs possible, I have to keep things simple and precise. Each bridge language simple enough to be understood completely, and each is described carefully, both formally and informally.
Compare, and find several ways to understand You learn more easily when you compare new ideas with existing ideas 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 you can say in a language.) 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 a simple 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 particular examples, you will learn general concepts: by seeing many recursive functions, you will learn about recursion; by using higher-order functions built into Scheme, you will learn about higher-order functions; by writing programs in three polymorphic languages, you will learn about polymorphism; and so on.

Finally, doing the 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.

If you work through this book, especially the exercises, you will be able to apply programming-language ideas in many situations, and you will also develop some essential technique: abstract syntax, operational semantics, equational reasoning, type systems, continuations, and garbage collection.

Programming exercises require languages in which to write programs. The five bridge languages—µScheme, µCLU, µML, µSmalltalk, and µProlog—are rich enough to write programs that are interesting.¹ 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 is expressive enough for writing really interesting programs. Typed µScheme introduces polymorphic type systems, and it’s very expressive, but it requires so much bookkeeping that programming is tiresome. And nano-ML introduces type inference. Although you can write interesting programs in nano-ML, you’re better off using µML, in which you can write even more.

If the sketch sounds good, the rest of the preface tells you enough for you to decide if this is the book for you. The next section describes the parts and chapters of the book in the order in which they appear. Another section presents concerns that cut across parts and chapters: themes, scope, organization, and software. The preface ends with an assessment of the bridge languages, so you can tell if they are realistic enough for your needs, plus some guidelines for teachers and for learners.

¹The bridge languages are distilled from languages whose greatness is widely acknowledged. For example, designers behind Scheme, CLU, ML, and Smalltalk have all won ACM Turing Awards. A Turing Award is the highest professional honor a computer scientist can receive.
The parts of the book

Whether you are teaching yourself or you plan to teach others, you'll want to choose a combination of languages and chapters that work for you. Here, in a half-dozen pages, you'll find what is available to choose from. You'll find many chapters introduced using jargon like “operational semantics,” “polymorphism,” or “garbage collection.” To the expert, such jargon says exactly and concisely what’s here. If you’re not expert yet, don’t worry—longer explanations tell you what’s going on. Just start at the bottom of Figure 1 and work up.

Foundations

The book, like the field, is founded on 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 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 present many examples: simple recursive functions, higher-order functions, standard higher-order functions on lists, continuation-passing style, and equational reasoning. All 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. In Figure 1, the first part is shown above and to the right of $\mu$Scheme. $\mu$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 are implemented, and you'll see a semantics that can model interaction and nontermination. You can also extend the interpreter to implement delimited and undelimited continuations. 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. All the other languages in the book rely on it. 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, and the only implementation language you need is C.

Independent of $\mu$Scheme+ and garbage collection, you can proceed directly from $\mu$Scheme to type systems, which are shown in Figure 1 above and to the left of $\mu$Scheme. 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 for sophisticated interpreters. For these kinds of tools, you want a language that provides algebraic data types. 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.

A type system can show that in a given context $\Gamma$, an expression $e$ has type $\tau$: I write $\Gamma \vdash e : \tau$. 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 (page 410). To solidify your understanding of type systems, I guide you in writing type checkers for Typed Impcore and Typed $\mu$Scheme. A type checker is given
a \Gamma that includes the type of every formal parameter and local variable, and it is given \epsilon, and it produces \tau.

The type system of Typed \(\mu\)Scheme is more expressive than any in common use. It is, in essence, what type theorists call System F. When you understand it, you’ll see that type systems in real languages, from the simple Hindley-Milner types of Standard ML through the complexities of features like Haskell type classes or Java generics, are easily expressed as special cases or extensions of Typed \(\mu\)Scheme.

When you write programs in Typed \(\mu\)Scheme—and you should write one or two polymorphic functions—you’ll quickly discover that you don’t want to. Like System F, Typed \(\mu\)Scheme requires that every polymorphic function be instantiated explicitly. Agony. As an alternative, Chapter 7 offers type inference for the language nano-ML.

Nano-ML, which is derived from \(\mu\)Scheme, is just as type-safe as a language like Typed \(\mu\)Scheme, but as a programmer, you never write a type—the type of every parameter, variable, and function is inferred. Nano-ML can even infer polymorphic types, so it’s almost as convenient as a dynamically typed language like \(\mu\)Scheme. 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. Because conjunction is associative and commutative, the constraint-based algorithm is much easier to implement than Damas and Milner’s original Algorithm W.

Types play a central role in programming languages, and you will want to study them. But you don’t have to do everything. Students do find it very satisfying to implement type inference, but it’s not necessary—you can use both nano-ML and \(\mu\)ML just as a programmer, and you will learn.

The Big Three

Operational semantics, functions, and types are everywhere. These foundations serve you whether you are learning a new language or you are reading the latest research from POPL and ICFP. Building on the foundations, the second part of the book presents great designs. More great designs exist than can fit between the covers of one book, and I have chosen just three, each of which showcases ideas of proven, lasting value: data abstraction and modules; algebraic data types and pattern matching; and objects and inheritance. These designs are shown at the top of Figure 1. You can explore any or all, but if you want to explore more than one, the chapters are set up to be explored from left to right.

\(\mu\)CLU (Chapter 8) demonstrates the ideas that make large systems possible: data abstraction and information hiding. \(\mu\)CLU is based on CLU, which is among the best safe, imperative languages ever implemented. Data abstraction is expressed using a simple module-like construct called the operation cluster. Abstraction is enforced by compile-time type checking; like Typed \(\mu\)Scheme and nano-ML, \(\mu\)CLU uses a polymorphic type system. By comparing these three languages, you will get a good idea of the design space of statically typed, polymorphic languages. \(\mu\)CLU’s core is imperative, but unlike Impcore, \(\mu\)CLU provides a rich set of data types from which abstractions can be built. \(\mu\)CLU also provides iteration abstraction and a simple, elegant form of operator overloading.

\(\mu\)CLU is the largest language in this book—as you can see from Table 2 on the facing page, \(\mu\)CLU packs more ideas into its design than any other bridge language. For that reason, \(\mu\)CLU’s type theory and operational semantics warrant separate treatment in a chapter of their own (Chapter 9). If you want only the type theory of data abstraction,
Chapter 9 opens with two core calculi that show how \( \mu \text{CLU}' \)'s modules are typechecked. These calculi use a variation on the standard interpretation of abstract types as existential types (Mitchell and Plotkin 1988): because every type in CLU has a name, each abstract type is referred to by its name, not by an existentially quantified type variable. Beyond these core calculi, Chapter 9 offers an opportunity to express, in formal notation, not just individual programming-language features, but an integrated design. As noted by Hoare (1989), finding good ideas about the design of language features is easy; integrating good features into a good design is not. The primary purpose of Chapter 9 is to provide an in-depth case study of how multiple features can be integrated into a single language with a single operational semantics and type system.

\( \mu \text{ML} \) (Chapter 10) evolves \( \mu \text{Scheme} \) and nano-ML into a language that includes what’s at the core of every higher-order, typed language: inductively defined algebraic data types. \( \mu \text{ML} \) is simpler and more uniform than nano-ML, and yet it can do more. Values of algebraic data type are examined using case expressions and pattern matching. Using syntactic sugar, \( \mu \text{ML} \) also adds pattern matching to all of the variable-binding constructs of nano-ML. \( \mu \text{ML} \) thereby offers almost all the features of core ML, omitting only local definitions and exceptions. It can be used as a destination in its own right or as a bridge to Standard ML. (Proficiency with Standard ML will enable you to extend the interpreters found in Chapters 5 to 12, as suggested in many of the programming exercises.)

To understand these design ideas more deeply, compare them. \( \mu \text{ML} \) and \( \mu \text{CLU} \) solve similar design problems in different ways. Both \( \mu \text{CLU} \)'s clusters and \( \mu \text{ML} \)'s algebraic data types offer a form of isorecursive type, but the details of the designs differ. \( \mu \text{CLU} \)'s tag-case statement offers pattern matching with “flat” patterns; \( \mu \text{ML} \)'s case expression offers full, nested patterns. Both \( \mu \text{CLU} \) and \( \mu \text{ML} \) can encode abstraction using existentially quantified types, but in \( \mu \text{CLU} \), an existential quantifier is attached to an every operation cluster, whereas in \( \mu \text{ML} \) an existential quantifier is attached only at the programmer’s discretion, and only to individual value constructors. All these advanced topics should be considered optional, but if you want them, \( \mu \text{CLU} \) and \( \mu \text{ML} \) provide concrete examples.
\(\mu\text{Smalltalk}\) (Chapter 11) demonstrates class-based object-orientation in a dynamically typed setting. 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 they offer extraordinary expressive power. You’ll see how easy it is to harness that power to implement families of related abstractions, including collections and numbers, which you can extend yourself. You’ll also see how hard it can be to understand your algorithms when they are implemented by cooperating methods that are spread out over half a dozen classes. Effective programming in Smalltalk is the biggest challenge in the book.

\(\mu\text{Smalltalk}\) and \(\mu\text{CLU}\) also merit comparison. They both provide encapsulation, but in different ways. Both hide representations, but in different ways. Both can expose operations. In \(\mu\text{Smalltalk}\), all operations are exposed, even those the programmer designates as “private.” Such exposure seems necessary in order to support code reuse through inheritance. By contrast, \(\mu\text{CLU}\) offers complete control over which operations are exposed and which are private, but it supports code reuse only through delegation. As above, \(\mu\text{CLU}\) and \(\mu\text{Smalltalk}\) provide concrete examples you can use to compare designs.

The Big Three are ordered not by complexity or by size but by ease of use. \(\mu\text{CLU}\), although it is the biggest language, and it has the most complicated implementation, semantics, and type theory, is the easiest to learn: its imperative constructs are familiar, and because types and polymorphism are explicit, everything that’s going on is visible in the source code. \(\mu\text{ML}\) is the next most easy to learn: it’s all about pattern matching. And \(\mu\text{ML}\)’s pattern matching can be viewed as an extension of \(\mu\text{CLU}\)’s tag-case statement. Types of functions and names are inferred, but the types of value constructors are given explicitly by the programmer. \(\mu\text{Smalltalk}\) is the most difficult to learn. The mechanisms of inheritance and dynamic dispatch are simple, but they have far-reaching consequences. Using Smalltalk well requires new ways of thinking, and when algorithms are spread out over multiple methods, they can be hard to write and harder to debug.

**Bonus features**

If you’re studying programming languages, you should study a weird one. *Build, Prove, and Compare* offers Prolog, a simple language inspired by mathematical logic. Mathematical logic is naturally connected to programming-language theory: operational semantics and type systems are expressed using logical rules. In \(\mu\text{Prolog}\) (Chapter 12), logical rules are the only form of code. Prolog also demonstrates an evaluation model that is wildly different from what is found in most languages, offering backtracking for control and unification for binding values to variables.

Chapter 12 develops an elegant, compact implementation of \(\mu\text{Prolog}\) using continuation-passing style. But this implementation isn’t good enough for all \(\mu\text{Prolog}\) programs, some of which demand industrial-strength compiler optimizations. So for compatibility with industrial implementations, \(\mu\text{Prolog}\) uses the Edinburgh Prolog syntax, not the Lisp syntax used in the other bridge languages.

The other bonus feature is not a single language or topic; it’s a sort of dessert menu of languages and topics that you might explore next. While Chapters 1 to 12 focus on proven ideas of lasting value, Chapter 13 offers a variety of jumping-off points, ranging from younger, less proven ideas to fashionable ideas drawn from recent headlines.
What else you will find

In a book like this one, which is organized by language, not topic, important themes in programming languages are threaded throughout the text, not gathered each in its own section. Such themes are listed below. You will also find information about scope, organization, and software infrastructure.

Thematic threads

The three major themes of the book are programming, semantics, and types.

- **Idiomatic programming** conveys the effective use of proven features that are found in many languages. Features that are explored in depth include functions in μScheme and μML (Chapters 2 and 10), abstract data types in μCLU (Chapter 8), algebraic data types in μML, objects and inheritance in μSmalltalk (Chapter 11), imperative and procedural programming in μCLU, and logic programming in μProlog (Chapter 12).

- **Big-step semantics** expresses the meaning of programs in a way that is easily connected to interpreters, and which, with practice, becomes easy to read and write. Big-step semantics are given for Impcore, μScheme, nano-ML, μCLU, μML, and μSmalltalk (Chapters 1, 2, 7, 9, 10, and 11). A nondeterministic semantics is given for μProlog (Chapter 12); a deterministic semantics is left as Exercise 37 on page 1062.

- **Type systems** guide the construction of correct programs, help document functions, and guarantee that language features like polymorphism and data abstraction are used as the designers intended. Type systems are given for Typed Impcore, Typed μScheme, nano-ML, μCLU, and μML (Chapters 6, 7, 9, and 10).

In addition to these major themes, the book supports several minor themes. Some are threaded throughout the text; others are supported by only one or two chapters.

- **Parametric polymorphism** enables one function to operate on data of many types; it’s as simple as writing a single `length` function that works with any linked list. Polymorphism begins with μScheme in Section 2.9 on page 125, and it is first enforced by a static type system in Typed μScheme (Section 6.6 on page 414). Making polymorphism fit for civilized use requires either the Damas-Milner type inference used in nano-ML and μML (Chapters 7 and 10), in which the compiler automatically instantiates every polymorphic value, or the large-scale instantiation of μCLU (Chapters 8 and 9), in which a single instantiation specializes an entire cluster of operations.

- **Implementation of type systems** connects theory and practice and helps build understanding of inference rules. You can complete my type checker for Typed Impcore (Exercise 2 on page 455 of Chapter 6), write a complete type checker of your own for Typed μScheme (Exercise 23 on page 464 of Chapter 6), implement type inference for nano-ML (Exercises 18 and 19 on pages 531 and 532 in Chapter 7), and complete my type checker for μCLU (Exercise 11 on page 723 of Chapter 8).

- **Metatheory** illustrates the power of operational semantics and type systems, primarily through exercises. In exercises, you use metatheory to prove results that describe all programs or all executions. Results range from the very simple (evaluating an expression cannot add a new global variable, Exercise 17 on page 77) to the moderately simple (evaluation is deterministic, Exercise 19 on page 77; the type of a well-typed
expression has kind $\ast$, Exercise 18 on page 460), to the slightly ambitious (Impcore can be evaluated using a stack of mutable environments, Exercise 24 on page 78; evaluation of an expression depends only on the bindings of the expression’s free variables, Exercise 1 on page 380). In this book, metatheory is touched on but lightly; classic metatheoretical results, like type soundness or strong normalization, must be found in another book.

- **Equational reasoning** can prove that functions are correct; it relies on algebraic laws, not directly on operational semantics. (The justification of such laws is explored by Exercise 43 on page 214.) Equational reasoning is both powerful and easy to use, but the simple form shown in this book works only in settings without imperative features. Because of its limited applicability, equational reasoning appears primarily in Chapter 2, where it is illustrated by examples and by a generous set of exercises. The examples and exercises emphasize proofs of simple algebraic laws, often requiring structural induction. Algebraic laws are also used for specification in Chapters 1, 7, and 8.

- **Free variables, bound variables, variable capture, and substitution** are the core mechanisms that define computation in the lambda calculus. This book does not discuss lambda calculus, but it does discuss these mechanisms. Variable capture is introduced in Section 2.16.3 on page 179, which describes how a short-circuit conditional or operation is desugared into a \texttt{let} expression. Discussion of free and bound variables is deferred to Section 5.10, which defines free and bound variables in $\mu$Scheme. The discussion is accompanied by an exercise that uses information about free variables to change the representation of closures in an interpreter for $\mu$Scheme (Exercise 2 on page 381). Capture-avoiding substitution is used not to compute beta-redexes but to instantiate polymorphic types; worked examples and a detailed specification appear in Section 6.6.7. Part of the implementation is left as an exercise.

- **Observational equivalence** is the principle that justifies all compilation and optimization techniques: two things are equivalent if and only if no program (or no “computational context”) can distinguish them. Because observational equivalence is used primarily in programming-language theory, not in programming practice, it is mentioned only briefly. Chapter 6 uses the principle of observational equivalence to justify renaming of bound variables, and Section 8.3.3 on page 565 appeals to observational equivalence to define equality, similarity, and copying in $\mu$CLU.

- **Continuations** are our most powerful tool for managing unusual control flow; we use them in semantics, in interpreters, and in programming practice. Continuations are used to implement solvers in Section 2.10.1 on page 133, including Exercise 21 on page 205; for control flow in $\mu$Smalltalk (Section 11.3.3 on page 879); and for search in $\mu$Prolog (Section 12.5.3 on page 1027), as well as for $\mu$Prolog’s cut (Exercise 44 on page 1065 of Chapter 12).

- **Memory management** makes civilized programming languages possible: when you don’t have to worry about each pointer and each word of memory, you can use your brainpower for other things. In Build, Prove, and Compare, memory management is mostly taken for granted. But Exercise 24 on page 78 of Chapter 1 shows that in Impcore, all formal parameters and local variables can be kept on a stack. Chapter 2 shows how formals and locals can be captured in a closure, requiring allocation on the heap. And Chapter 4 shows two ways of implementing heap allocation and garbage collection.
The propositions-as-types principle exposes a deep connection between type theory and mathematical logic. Details are beyond the scope of this book, but so important an idea must at least be mentioned. It is mentioned briefly in the sidebar on page 410, and there is a slightly longer sketch, including a couple of examples, in Chapter 13.

The scope of this book

*Programming Languages: Build, Prove, and Compare* is limited in both scope and topic. Its limits are described here.

If you don’t learn any theory, you’re not really studying programming languages. But you have to decide what place theory has in your world. In *Build, Prove, and Compare*, theory is a language in which we express and communicate ideas. You’ll learn to use theory to communicate your own ideas, and you’ll be able to read about new ideas in the professional literature. In the broader world, theory is also used to establish facts about programs and about languages, by proving theorems. You can prove a few theorems here, and you should—but you’ll do it to develop understanding, not to establish facts. And you’ll discover that although proving theorems can be satisfying, you often develop more understanding from trying to prove something that isn’t true.

Anyone interested in the foundations of programming languages should study the lambda calculus. But lambda calculus is a poor fit for *Build, Prove, and Compare*. While it’s easy to encode simple data structures, it’s not so easy to write interesting functions or programs, and when you do write functions, the results they return are hard to read. And it’s hard to find good exercises—implementations of almost every interesting lambda-calculus function are easily found on the Web. For these reasons, I decided not to include a chapter on lambda calculus.

But maybe that’s a bad decision. Lambda calculus, like Prolog, illustrates a language with no values, only terms. It illustrates term-rewriting semantics. It illustrates nondeterministic semantics, and its two popular reduction strategies (normal-order reduction and applicative-order reduction) underlie two major families of functional languages (lazy and eager). Lambda calculus serves as a medium with which to explore fundamental techniques like capture-avoiding substitution and fixed-point construction. And it is the simplest and most beautiful Turing-complete model of computation. For all these reasons, my own students study lambda calculus. They even use an interpreter, in which they can implement capture-avoiding substitution and two reduction strategies. If you would like these opportunities for yourself or for your own students, perhaps in the form of a chapter on the Web, please write to me.

Denotational semantics is a beautiful subject and is intimately connected to compiler construction. But it is now so far outside the main stream of programming-language theory that a chapter in this book no longer makes sense. If you wish to explore the subject, however, it is a natural one for definitional interpreters, and the control operators in µScheme+ and µCLU lend themselves well to a continuation-passing semantics.

Beyond theory, I have deliberately excluded one important area from *Build, Prove, and Compare*: concurrency and parallelism play an important and growing role in the design and implementation of programming languages, but these topics are too difficult and too ramified to be handled well in an introductory book. Concurrency brings with it enough new mathematics, new language constructs, and new programming technique that it needs a book of its own.

I have also omitted two elegant, influential programming models. One is the pure, lazy language; a good exemplar is Haskell, with its pure, monadic I/O. The other is the
prototype-based object-oriented language. Today’s obvious exemplar is Javascript, which runs in any Web browser, but I prefer Self, which, among object-oriented languages, offers the ultimate combination of power and simplicity. If you want to see a $\mu$Haskell or a $\mu$Self developed into a chapter to appear on the Web, please write to me.

**What to expect in each chapter**

Each chapter explores a different language or topic, but to facilitate learning and comparison, most explorations are structured in similar ways and use similar elements. The most important elements are programming examples, a programmer’s description, a theorist’s description, an implementation, and exercises.

- Small *programming examples* appear early, before a complete description of the language. Larger examples appear after a language description.

- A *programmer’s description* presents concrete syntax and some informal explanations about what each syntactic form does. It’s the sort of thing you would see in a manual intended for programmers.

- A *theorist’s description* presents abstract syntax and some formal inference rules that explain how each syntactic form behaves. If there is a static type system involved, more rules explain how each syntactic form is given a type. The theorist’s description is the sort of thing you would see in a paper from the professional literature.

  You’ll recognize the inference rules by their Greek letters and horizontal lines. The rules are “informal” inference rules, of the sort that are used in “pencil-and-paper” proofs, not in “machine-checked” proofs.

- An *implementation* presents the intellectually interesting parts of an interpreter for the language, and it connects those parts to the theorist’s description. Other code needed to make the interpreter work is relegated to the book’s appendices.

  Also relegated to appendices is a topic of some intellectual interest but of little importance in a book about programming languages: parsing. In the main text, I assume that all programs have been parsed into abstract-syntax trees, and the code uses abstract syntax, not concrete syntax. Abstract syntax is not just a useful representation for interpreters; it is the correct way to think about the syntactic structure of languages.

The programmer’s description, the theorist’s description, and the implementation provide different views of the same truth. In many chapters, connections between views are summarized in a table (pages 28, 152, 355, 398, 432, 508, 775, and 782). When you begin your studies, you can expect to be comfortable reading implementations, not inference rules. But as you continue to compare implementations and programmer’s descriptions with inference rules and theorist’s descriptions, you will discover that inference rules give you exactly the information you need to know, compactly, without distractions. You will come to like reading inference rules. When you get there, celebrate.

- *Exercises* are what you need to do if you hope to learn anything. You will find over 400 exercises. A typical chapter offers exercises that dig into the language, that dig into the theory, and that dig into the interpreter. Most chapters offer exercises in difficulties that range from simple finger exercises to weeklong projects. Each chapter also includes a guide to the exercises; the guide explains the idea behind each exercise, and it will help you choose which exercises you want to do.
HOW REALISTIC ARE THE BRIDGE LANGUAGES?

If you're teaching, you can get solutions to many exercises. For details, please see http://www.cs.tufts.edu/~nr/build-prove-compare.

Most chapters also explain, at the beginning, what makes the language interesting or worthy of study, often with some historical context. And near the end, each chapter presents some aspects of the parent language as it really is, including its native concrete syntax and some interesting features not included in the bridge language. Most chapters also include a list of key words and phrases, which help you remember what you’ve read. They also form a tiny cognitive framework you can use to organize your knowledge—that’s why they’re organized by topic, not alphabetically. And each chapter has suggestions for further reading.

In three chapters, I focus not on a language but on an implementation technique. Chapter 4 extends the interpreter of Chapter 2 by adding garbage collection. Chapter 5 introduces ML by providing a new implementation, written in ML, of the language described in Chapter 2. Chapter 6 presents type checkers for typed variants of the languages described in Chapters 1 and 2.

Software infrastructure

All the interpreters are available in machine-readable form. Actually, two machine-readable forms. One set is for learners, and in these interpreters, many of the interesting parts—garbage collectors, type checkers, constraint solvers, and so on—are left as exercises. The other set is for instructors, and in these interpreters, all the interesting parts are included. Instructors can compile them and make them available in execute-only form, without revealing solutions.

The instructors’ set of interpreters is also used to test the example code shown in the text, using the Noweb tool for literate programming. This tool guarantees that what appears in the text is consistent with the running code.

One of my most difficult decisions was choosing the language in which the interpreters themselves would be implemented. I decided that if you want to learn programming languages from a standing start, the best choice is not one language but two. Through Chapter 4, the interpreters are written in C. C is well suited to the implementation of garbage collectors, which are the topic of Chapter 4, and I hope you are one of the many programmers already familiar with C or C++. If not, Kernighan and Ritchie (1988) will teach you. The remaining interpreters are written in Standard ML. If you are not familiar with ML, it is a powerful language that is ideally suited to writing interpreters, and there are several fine, open-source implementations available. With the help of the material in Chapters 2 and 10, learning ML will be easy.

How realistic are the bridge languages?

If you already know Scheme, CLU, ML, Smalltalk, or Prolog, you may wonder if the bridge versions are good enough for what you have in mind. If you don’t know them, you may wonder if crucial parts are missing. This section explains briefly what’s there, what’s not, and why.

μScheme is a stripped-down version of Scheme. You get a couple forms of \texttt{define}, a \texttt{lambda}, and three “let” forms. You get S-expressions that include symbols, Booleans, machine integers, and \texttt{cons} cells (including list functions). μScheme is big enough that you can write plenty of interesting functional programs, but as the first language in the book that includes features not often found in introductory courses, it’s small enough to learn quickly. It is a lot smaller than full Scheme.
In particular, μScheme has no numeric tower; it has only machine integers. The cond form is available only as syntactic sugar; it’s not in the abstract syntax. (Although cond is superior, if is both universal and familiar. For consistency, all the bridge languages use if.) There is no local form for introducing definitions inside expressions; I avoid mutual recursion between the syntax of definitions and the syntax of expressions. Finally, there are no macros. Macros are a unique and powerful aspect of Scheme, but because they are an advanced technique found only in the Lisp/Scheme family, I cannot justify including them in an introductory book.

μCLU is close to full CLU; the only important, missing feature is exceptions. Exceptions are missing because the language is already large, and because they add significant complexity to the syntax of function types and function definitions, as well as to the type checker. And like all the bridge languages, μCLU omits some minor conveniences in the interest of simplicity; for example, there are no type sets, and clusters may take only types as parameters, not also compile-time values.

But μCLU implements the full story about data abstraction, with static type checking, arbitrary representations, recursive clusters, and parametric polymorphism bounded with where clauses. Like full CLU, μCLU uses the static type system to overload operators. μCLU also lets you typecheck and develop against separately typechecked interfaces, even if they don’t yet have implementations. And although μCLU doesn’t have full CLU’s character, string, or real-number types, all the other primitive types and type constructors are there: function types, iterator types, mutable and immutable array types, mutable and immutable record types, and mutable and immutable sum types.

μML is a good facsimile of core ML. You have type inference, algebraic data types, case expressions, and matching of nested patterns. Via syntactic sugar, pattern matching is extended to other forms, including clause definitions. Using these forms you can get close to the full core-ML experience, but not modules. Again, some features are omitted in the interest of simplicity. There is no local form for making definitions visible only in the context of other definitions. To keep the semantics and the type theory simple, there are no exceptions, no mutable reference cells, and no value restriction.

μSmalltalk, the language, includes almost all of Smalltalk-80. The data and evaluation model realize Alan Kay’s vision that everything is an object. Even classes are objects. Control flow is expressed via message passing; as in Smalltalk-80, conditionals and loops are implemented by sending messages and blocks to Booleans, which is a form of continuation-passing style. As in Smalltalk-80, the name of a method determines its arity, and arity errors are detected at compile time. μSmalltalk does lack Smalltalk-80’s nonlocal return, but it can be added as a programming exercise. Also, for consistency with the other bridge languages, μSmalltalk uses global variables, not class variables on class Object.

Three-quarters of Smalltalk is class hierarchy, and μSmalltalk’s class hierarchy is significantly smaller than what is found in Smalltalk-80—and it is organized a bit differently. In addition to blocks and Booleans, you will find a small Collection hierarchy, the leaves of which include Set, Dictionary, List, and Array. You will also find Magnitude and Number, which has subclasses for fractions, software floating-point numbers, and machine integers. Extending the hierarchy to arbitrary-precision integers, with seamless failover, is one of the better programming exercises in Chapter 11. The most notable missing elements are reflective features: in particular, it is not possible to ask a μSmalltalk object for the class of which it is an instance. This omission enables me to simplify the implementation of metaclasses, which reduces the amount of recursive knot-tying needed in the book’s most complicated interpreter. The omission of reflective features also militates toward the use of global variables instead of class variables on class Object.
µProlog provides a subset of full Prolog, using the standard Edinburgh syntax, including comments. Other lexical details are those of Build, Prove, and Compare, not of the ISO standard. µProlog provides no character or string type; numbers are machine integers. To avoid complicating the relationship between Prolog clauses and logical inference rules, µProlog does not include a disjunction operator. Non-logical features like the cut and not are left as exercises, as are reflective features like assert and retract. µProlog includes only a tiny subset of standard Prolog’s built-in predicates, operators, and functors: enough to recognize an atom, print a term, and do a little arithmetic.

Finally, the µProlog interpreter lacks the program analyses and optimizations found in a mature implementation like SWI Prolog or XSB Prolog. None of the interpreters is optimized, but for the others, it doesn’t matter—if they are compiled to native code, they perform well enough. Not µProlog. The µProlog interpreter can finish the examples in Chapter 12, but if you tackle larger search problems, including some of the exercises, µProlog may take days to work on a problem that SWI or XSB Prolog solves in seconds.

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, formal reasoning about the bridge languages, and implementation and extension of 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 µScheme+, and they can implement garbage collectors. Or if they can make a transition from µScheme to Standard ML, possibly via µML, 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 µScheme, transitions to Standard ML to work on type systems and type inference, then returns to the bridge languages to explore µSmalltalk, µProlog, and garbage collection.

If your students have seen interpreters and are comfortable with proof by induction, 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 “five languages in ten weeks.” That’s the Full Five: $\mu$Scheme, $\mu$CLU, $\mu$ML, $\mu$Smalltalk, and $\mu$Prolog. 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 five 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, $\mu$CLU, and $\mu$ML. They can compare how universally quantified types are used in three different designs (Typed $\mu$Scheme, nano-ML/$\mu$ML, and $\mu$CLU), and how existentially quantified types are used in two different designs ($\mu$CLU and $\mu$ML). They can also, through exercises, push $\mu$CLU close to a modern module system, and they can develop a precise, deterministic semantics for $\mu$Prolog.

• 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 Appendices A to E provide a powerful infrastructure that you can use to build more definitional interpreters. But apart from these appendices, 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 (Midtgaard, 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 a at least a little register allocation and code generation. For help designing and building such interpreters, you would need an additional book.

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, µScheme, 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.

Final words for learners

Using this book, you can learn about programming languages designed around functions, around types, around data abstraction, around pattern matching, around objects, and around logic. You don’t just learn about each language; you learn to use it. You build things, and you study how each language itself is built. You can also learn the rudiments of programming-language theory: you see not only how each language is built but also how that language is described mathematically—and you can compare each implementation with its specification. You learn to speak the mathematical language of our field, and maybe even to use it to express your own ideas. And possibly to prove a few things.

To master any of these languages or techniques, you must do some exercises. Most exercises ask you to build something, and many of the things you build are interesting. If you share your work with others, such as potential employers, I will be pleased and proud. My own students’ work is more than worthy of a professional portfolio. But if you choose to share your work using a public site like Github or Bitbucket, please share this part of your portfolio only with individuals that you name. Bluntly, don’t put your work in a public repository—for someone who may make a poor decision under time pressure, it’s too big a temptation.

The exercises and the bridge languages cover a wealth of programming styles and paradigms, and they illustrate some of the greatest ideas in programming. The bridge languages and their interpreters are the distinctive feature of this book, and they will help you learn not just to think intelligently about different programming languages, but to develop your intuition about what it is like to use different programming languages. And if you work with the interpreters, you will start to learn how languages work and what makes them possible. I hope you enjoy the experience.

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.

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.
Matthew Fluet and I spent two days working nearly nonstop on the design and type theory of \( \mu \text{CLU} \). Matthew’s insights and oversight were invaluable; without him, Chapters 8 and 9 would never have been completed. I’m also grateful to several members of IFIP Working Group 2.8, especially Stephanie Weirich, who helped me identify type-theoretic techniques for handling data abstraction.

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 \( \mu \text{Scheme} \), which I removed. Matthew also suggested the example used in Section 6.6.8, which shows that if variable capture is not avoided, Typed \( \mu \text{Scheme} \)’s type system can be subverted.

Benjamin Pierce taught me how to think about the roles of proofs in programming languages; the introduction to Section 1.6 explains his ideas as I understand them.

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

Simon Peyton Jones helped me understand the modern approach to type inference with existential value constructors. Beyond the individual help, Simon’s work and scholarship has greatly enhanced my understanding of the various ways and means of extending algebraic data types.

Ben Weitzman found the loophole exploited in Exercise 32 on page 467.

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

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 \texttt{check-expect} and \texttt{check-error}, which I have embraced and extended.

Cyrus Cousins found a subtle bug in \( \mu \text{Scheme}+ \).

Mike Hamburg and Inna Zakharevich spurred me to improve the concrete syntax of \( \mu \text{Smalltalk} \) and to provide better error messages.

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, Tony Hosking, Scott Johnson, and Juergen Kahrs 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, and Jan Vitek bravely used preliminary versions in their classes. Penny found far more errors and suggested many more improvements than anyone else; she has my profound thanks.

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

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-year 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. I am glad she is in my corner.

Norman Ramsey
Malden, Massachusetts
August, 2016
# Book contents

**Preface** v  
**Tables of judgment forms and important functions** xxx  
**Symbols and notation** xxxii

## I Foundations 1

1 **An imperative core** 3  
1.1 The Impcore language ........................................... 5  
1.2 Abstract syntax .................................................. 16  
1.3 Environments ...................................................... 17  
1.4 Operational semantics ........................................... 18  
1.5 The interpreter .................................................... 26  
1.6 Operational semantics revisited: Proofs ....................... 56  
1.7 Extending Impcore ................................................. 60  
1.8 Summary ........................................................... 62  
1.9 Exercises .......................................................... 66

2 **Scheme, S-expressions, and first-class functions** 85  
2.1 Overview of $\mu$Scheme ........................................... 86  
2.2 Language I: S-expressions and list primitives ............... 88  
2.3 Practice I: Recursive functions on lists .................... 91  
2.4 Interlude: Laws of applicative programming ............... 99  
2.5 Language II: Local variables and let ....................... 105  
2.6 Practice II: Recursive functions on binary trees .......... 106  
2.7 Language III: First-class functions ......................... 112  
2.8 Practice III: Higher-order functions on lists .......... 121  
2.9 Practice IV: Higher-order functions for polymorphism .... 125  
2.10 Practice V: Continuation-passing style .................... 131  
2.11 Syntax and values of $\mu$Scheme .............................. 138  
2.12 Operational semantics .......................................... 140  
2.13 The initial basis ............................................... 148  
2.14 The interpreter ............................................... 151  
2.15 Large example: A metacircular evaluator .................. 170  
2.16 Extending (and contracting) $\mu$Scheme ................... 176  
2.17 Scheme as it really is ......................................... 184  
2.18 Summary ......................................................... 188  
2.19 Exercises ........................................................ 192

3 **Control operators and reduction semantics** 227  
3.1 The $\mu$Scheme+ language ..................................... 228  
3.2 Imperative programming with control operators .......... 230  
3.3 Preparing for control operators: evaluation using a stack ... 239  
3.4 Operational Semantics for $\mu$Scheme+ ....................... 243  
3.5 The interpreter .................................................. 256  
3.6 Extending $\mu$Scheme+ with syntactic sugar ............... 277
| 3.7 | Control operators & semantics as they really are | 277 |
| 3.8 | Summary | 283 |
| 3.9 | Exercises | 287 |

4 Automatic Memory Management | 301 |
| 4.1 | What garbage is and where it comes from | 302 |
| 4.2 | Garbage-collection basics | 304 |
| 4.3 | The managed heap in \( \mu \text{Scheme} \) | 307 |
| 4.4 | Mark-and-sweep garbage collection | 312 |
| 4.5 | Copying garbage collection | 318 |
| 4.6 | Debugging a garbage collector | 327 |
| 4.7 | Reference counting | 331 |
| 4.8 | Garbage collection as it really is | 333 |
| 4.9 | Summary | 335 |
| 4.10 | Exercises | 341 |

5 Interlude: \( \mu \text{Scheme} \) in ML | 351 |
| 5.1 | Names and environments | 354 |
| 5.2 | Abstract syntax and values | 356 |
| 5.3 | Error detection and signaling | 359 |
| 5.4 | Evaluation | 360 |
| 5.5 | Primitives | 363 |
| 5.6 | Evaluating definitions | 366 |
| 5.7 | A reusable read-eval-print loop | 367 |
| 5.8 | Initializing and running the interpreter | 371 |
| 5.9 | Pulling the pieces together in the right order | 372 |
| 5.10 | Free and bound variables | 373 |
| 5.11 | Summary | 376 |
| 5.12 | Exercises | 380 |

6 Type Systems for Impcore and \( \mu \text{Scheme} \) | 385 |
| 6.1 | Typed Impcore: a statically typed imperative core | 388 |
| 6.2 | A typechecking interpreter for Typed Impcore | 397 |
| 6.3 | Extending Typed Impcore with arrays | 406 |
| 6.4 | Interlude: common type constructors | 411 |
| 6.5 | Type soundness | 413 |
| 6.6 | Polymorphic type systems and Typed \( \mu \text{Scheme} \) | 414 |
| 6.7 | Type systems as they really are | 450 |
| 6.8 | Summary | 451 |
| 6.9 | Exercises | 453 |

7 Nano-ML and type inference | 471 |
| 7.1 | Nano-ML: a nearly applicative language | 472 |
| 7.2 | Abstract syntax and values of nano-ML | 475 |
| 7.3 | Operational semantics | 476 |
| 7.4 | Type system for nano-ML | 479 |
| 7.5 | From type rules to type inference | 491 |
| 7.6 | The interpreter | 509 |
| 7.7 | Hindley-Milner as it really is | 524 |
| 7.8 | Summary | 524 |
II The Big Three

8 $\mu$CLU and data abstraction

8.1 Abstractions, representations, and invariants
8.2 Defining simple abstractions with $\mu$CLU
8.3 The $\mu$CLU language
8.4 Reference information for $\mu$CLU's primitive types
8.5 Iteration abstraction
8.6 Parametric polymorphism in CLU
8.7 Program design with abstract data types
8.8 Larger programming examples
8.9 $\mu$CLU, data abstraction, and modules as they really are
8.10 Summary
8.11 Exercises

9 Theory and implementation of $\mu$CLU

9.1 Essentials of $\mu$CLU's theory
9.2 Essentials of $\mu$CLU's implementation
9.3 Operational semantics and evaluation
9.4 Foundations of $\mu$CLU's type system
9.5 $\mu$CLU's type checking
9.6 Finishing the interpreter
9.7 Theory as it is elsewhere (with reading)
9.8 Exercises

10 $\mu$ML, Algebraic Data Types, and Pattern Matching

10.1 Case expressions and pattern matching
10.2 The $\mu$ML language
10.3 Large example: 2D-trees
10.4 Equational reasoning
10.5 Syntactic sugar: patterns everywhere
10.6 The theory and the interpreter
10.7 Extending algebraic data types
10.8 Algebraic data types as they really are
10.9 Further reading
10.10 Exercises

11 Smalltalk and object-orientation

11.1 Messages, classes, and inheritance
11.2 The $\mu$Smalltalk language
11.3 The initial basis
11.4 Extended example: Discrete event simulation
11.5 Implementations of built-in classes and objects
11.6 Interpreter and operational semantics
11.7 Smalltalk as it really is
11.8 Summary
11.9 Exercises
III  Bonus Features 987

12 Prolog and logic programming 989
12.1 Thinking in the language of logic 990
12.2 Using Prolog 995
12.3 The language 999
12.4 More small programming examples 1017
12.5 Implementation 1025
12.6 Larger example: The blocks world 1032
12.7 Larger example: Haskell type classes 1037
12.8 Prolog as it really is 1041
12.9 Summary 1046
12.10 Exercises 1049

13 What next? 1069
13.1 Typeful programming 1069
13.2 Propositions as types 1069
13.3 More functions 1071
13.4 More objects 1071
13.5 Functions and objects, together 1071
13.6 Functional animation 1071
13.7 Scripting 1072
13.8 Parallel and distributed computation 1072
13.9 One weird, cool, domain-specific language 1072
13.10 Stack-based languages 1072
13.11 Array languages 1072
13.12 Languages based on substitution 1073
13.13 String-processing languages 1073
13.14 Conclusion 1073

IV  Infrastructure 1075

A  Code for writing interpreters in C 1077
A.1 Streams 1077
A.2 The extensible printer 1090
A.3 Error functions 1094
A.4 Test processing and reporting 1097
A.5 Stack-overflow detection 1098
A.6 Unicode support 1099

B  Parsing parenthesized phrases in C 1101
B.1 Planning an extensible parser 1102
B.2 Components, reduce functions, and form codes 1105
B.3 Parser state and shift functions 1107
B.4 Representing and parsing tables and rows 1111
B.5 Parsing tables and functions 1113
B.6 Error detection and handling 1116
B.7 Extending Impcore with syntactic sugar 1120
<table>
<thead>
<tr>
<th>C Supporting discriminated unions in C</th>
<th>1123</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Lexical analysis</td>
<td>1124</td>
</tr>
<tr>
<td>C.2 Abstract syntax and parsing</td>
<td>1125</td>
</tr>
<tr>
<td>C.3 Interface to a general-purpose prettyprinter</td>
<td>1127</td>
</tr>
<tr>
<td>C.4 C types</td>
<td>1129</td>
</tr>
<tr>
<td>C.5 Prettyprinting C types</td>
<td>1130</td>
</tr>
<tr>
<td>C.6 Creating C types from sums and products</td>
<td>1131</td>
</tr>
<tr>
<td>C.7 Creating constructor functions and prototypes</td>
<td>1133</td>
</tr>
<tr>
<td>C.8 Writing the output</td>
<td>1135</td>
</tr>
<tr>
<td>C.9 Implementation of the prettyprinter</td>
<td>1137</td>
</tr>
<tr>
<td>C.10 Putting everything together</td>
<td>1139</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D Code for writing interpreters in ML</th>
<th>1143</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 Reusable utility functions</td>
<td>1144</td>
</tr>
<tr>
<td>D.2 Representing error outcomes as values</td>
<td>1148</td>
</tr>
<tr>
<td>D.3 Unit testing</td>
<td>1150</td>
</tr>
<tr>
<td>D.4 Polymorphic, effectful streams</td>
<td>1153</td>
</tr>
<tr>
<td>D.5 Tracking and reporting source-code locations</td>
<td>1160</td>
</tr>
<tr>
<td>D.6 Further reading</td>
<td>1162</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E Lexical analysis, parsing, and reading using ML</th>
<th>1165</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1 Stream transformers, which act as parsers</td>
<td>1167</td>
</tr>
<tr>
<td>E.2 Lexical analyzers: transformers of characters</td>
<td>1175</td>
</tr>
<tr>
<td>E.3 Parsers: reading tokens and source-code locations</td>
<td>1178</td>
</tr>
<tr>
<td>E.4 Streams that lex, parse, and prompt</td>
<td>1186</td>
</tr>
<tr>
<td>E.5 Further reading</td>
<td>1189</td>
</tr>
</tbody>
</table>

| F EBNF                                             | 1191 |

| V The Supporting Cast                              | 1193 |

<table>
<thead>
<tr>
<th>G Supporting code for Impcore</th>
<th>1195</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.1 Running unit tests</td>
<td>1195</td>
</tr>
<tr>
<td>G.2 Printing functions</td>
<td>1198</td>
</tr>
<tr>
<td>G.3 Implementation of function environments</td>
<td>1201</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H Supporting code for μScheme</th>
<th>1205</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.1 μScheme code not included in Chapter 2</td>
<td>1205</td>
</tr>
<tr>
<td>H.2 Implementation of μScheme environments</td>
<td>1205</td>
</tr>
<tr>
<td>H.3 Parsing μScheme code</td>
<td>1207</td>
</tr>
<tr>
<td>H.4 Implementation of μScheme’s value interface</td>
<td>1213</td>
</tr>
<tr>
<td>H.5 μScheme’s unit tests</td>
<td>1219</td>
</tr>
<tr>
<td>H.6 Parse-time error checking</td>
<td>1222</td>
</tr>
<tr>
<td>H.7 Support for an exercise: Concatenating names</td>
<td>1222</td>
</tr>
<tr>
<td>H.8 Print functions for expressions</td>
<td>1223</td>
</tr>
<tr>
<td>H.9 Support for μScheme+</td>
<td>1226</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I Supporting code for μScheme+</th>
<th>1229</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.1 The stack of evaluation contexts</td>
<td>1229</td>
</tr>
<tr>
<td>I.2 Updating lists of expressions within contexts</td>
<td>1233</td>
</tr>
</tbody>
</table>
## I. Options and diagnostic code
I.3 Options and diagnostic code ........................................... 1235
I.4 Parsing ........................................................................... 1236
I.5 Finding free variables ...................................................... 1237

## J. Supporting code for garbage collection
J.1 Basic support for the two collectors ................................. 1241
J.2 GC debugging, with or without Valgrind ......................... 1248
J.3 Code that is changed to support garbage collection .......... 1251
J.4 Placeholders for exercises .............................................. 1254

## K. Supporting code for \( \mu \text{Scheme} \) in ML
K.1 Lexical analysis and parsing ........................................... 1257
K.2 Unit tests for \( \mu \text{Scheme} \) ........................................... 1263
K.3 Unspecified values ...................................................... 1264
K.4 Further reading .......................................................... 1264

## L. Supporting code for Typed Impcore
L.1 Unit testing ................................................................... 1267
L.2 Printing types and values .............................................. 1269
L.3 Parsing ......................................................................... 1270
L.4 Evaluation ..................................................................... 1272

## M. Supporting code for Typed \( \mu \text{Scheme} \)
M.1 Printing types and values .............................................. 1277
M.2 Parsing ......................................................................... 1277
M.3 Evaluation .................................................................... 1280
M.4 Primitives of Typed \( \mu \text{Scheme} \) ................................. 1282
M.5 Predefined functions .................................................... 1283
M.6 Unit testing ................................................................... 1285

## N. Supporting code for nano-ML
N.1 Printing types and constraints ....................................... 1289
N.2 Parsing .......................................................................... 1290
N.3 Unit testing ................................................................... 1292
N.4 Predefined functions ..................................................... 1295
N.5 Cases and code for Chapter 10 ...................................... 1297

## O. Supporting code for \( \mu \text{CLU} \)
O.1 Implementations of \( \mu \text{CLU} \)'s primitive types ............... 1299
O.2 Evaluation ..................................................................... 1321
O.3 Type checking .............................................................. 1323
O.4 Giving every cluster a print operation ......................... 1335
O.5 Lexical analysis and parsing ........................................ 1335
O.6 Parsing ......................................................................... 1338
O.7 Unit testing ................................................................... 1347
O.8 Miscellaneous error messages ..................................... 1350
O.9 Printing stuff ............................................................... 1352

## P. Supporting code for \( \mu \text{ML} \)
P.1 Parsing .......................................................................... 1359
P.2 S-expression reader ....................................................... 1365
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.3</td>
<td>More predefined functions</td>
<td>1366</td>
</tr>
<tr>
<td>P.4</td>
<td>Useful $\mu$ML functions</td>
<td>1368</td>
</tr>
<tr>
<td>P.5</td>
<td>Drawing red-black trees with dot</td>
<td>1370</td>
</tr>
<tr>
<td>P.6</td>
<td>Printing values, patterns, types, and kinds</td>
<td>1372</td>
</tr>
<tr>
<td>P.7</td>
<td>Unit testing</td>
<td>1374</td>
</tr>
<tr>
<td>P.8</td>
<td>Support for datatype definitions</td>
<td>1375</td>
</tr>
<tr>
<td>P.9</td>
<td>Syntactic sugar for $\text{implicit-data}$</td>
<td>1377</td>
</tr>
<tr>
<td>P.10</td>
<td>Error cases for elaboration of type syntax</td>
<td>1378</td>
</tr>
<tr>
<td>Q</td>
<td>Supporting code for $\mu$Smalltalk</td>
<td>1381</td>
</tr>
<tr>
<td>Q.1</td>
<td>Lexing and parsing</td>
<td>1381</td>
</tr>
<tr>
<td>Q.2</td>
<td>Support for tracing</td>
<td>1386</td>
</tr>
<tr>
<td>Q.3</td>
<td>Unit testing</td>
<td>1389</td>
</tr>
<tr>
<td>R</td>
<td>Supporting code for $\mu$Prolog</td>
<td>1393</td>
</tr>
<tr>
<td>R.1</td>
<td>Substitution</td>
<td>1393</td>
</tr>
<tr>
<td>R.2</td>
<td>Unit testing</td>
<td>1394</td>
</tr>
<tr>
<td>R.3</td>
<td>String conversions</td>
<td>1396</td>
</tr>
<tr>
<td>R.4</td>
<td>Lexical analysis</td>
<td>1396</td>
</tr>
<tr>
<td>R.5</td>
<td>Parsing</td>
<td>1400</td>
</tr>
<tr>
<td>R.6</td>
<td>Command line</td>
<td>1408</td>
</tr>
</tbody>
</table>

VI Back matter                  1409

Key words and phrases            1411
References                        1415
Concept index                     1433
Author index                      1451
Code index                        1457
### Tables of judgment forms and important functions

#### Evaluation judgments

<table>
<thead>
<tr>
<th>Language</th>
<th>Expression or related form</th>
<th>Page</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impcore</td>
<td>⟨e, ξ, φ, ρ⟩ ↪ ⟨v, ξ', φ', ρ'⟩</td>
<td>19</td>
<td>(d, ξ, φ) → ⟨ξ', φ'⟩</td>
<td>25</td>
</tr>
<tr>
<td>µScheme</td>
<td>⟨e, ρ, σ⟩ ↪ ⟨v, σ'⟩</td>
<td>140</td>
<td>(d, ρ, σ) → ⟨ρ', σ'⟩</td>
<td>147</td>
</tr>
<tr>
<td>µScheme+</td>
<td>⟨e/v, ρ, σ, S⟩ → ⟨e'/v', ρ', σ', S'⟩</td>
<td>243</td>
<td>(d, ρ, σ) → ⟨ρ', σ'⟩</td>
<td>255</td>
</tr>
<tr>
<td>Typed Impcore</td>
<td>(as in Impcore)</td>
<td>19</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Typed µScheme</td>
<td>⟨e, ρ, σ⟩ ↪ ⟨v, σ'⟩</td>
<td>447</td>
<td>(as in Impcore)</td>
<td>147</td>
</tr>
<tr>
<td>nano-ML</td>
<td>⟨e, ρ⟩ ↪ v</td>
<td>476</td>
<td>(d, ρ) → ρ'</td>
<td>476</td>
</tr>
<tr>
<td>µCLU</td>
<td>⟨e, ρ, σ⟩ ↪ ⟨v, σ'⟩</td>
<td>669</td>
<td>(d, ρ, σ) → ⟨ρ', σ'⟩</td>
<td>684</td>
</tr>
<tr>
<td>µML</td>
<td>⟨e, ρ⟩ ↪ v</td>
<td>774</td>
<td>(as in nano-ML)</td>
<td>476</td>
</tr>
<tr>
<td>µSmalltalk</td>
<td>⟨e, ρ, c_SUPER, ξ, σ⟩ ↪ ⟨v, σ'⟩</td>
<td>933</td>
<td>(d, ρ, σ) → ⟨ρ', σ'⟩</td>
<td>937</td>
</tr>
<tr>
<td>µProlog</td>
<td>D ⊢ θ(gs) (query)</td>
<td>1006</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Typing judgments

<table>
<thead>
<tr>
<th>Language</th>
<th>Expression or related form</th>
<th>Page</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typed Impcore</td>
<td>Γξ, Γφ, Γρ ⊢ e : τ</td>
<td>394</td>
<td>(d, Γξ, Γφ) → (Γξ', Γφ')</td>
<td>396</td>
</tr>
<tr>
<td>Typed µScheme</td>
<td>Δ, Γ ⊢ e : τ</td>
<td>428</td>
<td>(d, Γ) → Γ'</td>
<td>430</td>
</tr>
<tr>
<td>nano-ML</td>
<td>Γ ⊢ e : τ (nondeterministic)</td>
<td>486</td>
<td>(d, Γ) → Γ'</td>
<td>489</td>
</tr>
<tr>
<td></td>
<td>θΓ ⊢ e : τ (with substitutions)</td>
<td>491</td>
<td>(d, Γ) → Γ'</td>
<td>489</td>
</tr>
<tr>
<td></td>
<td>C, Γ ⊢ e : τ (with constraints)</td>
<td>491</td>
<td>(d, Γ) → Γ'</td>
<td>489</td>
</tr>
<tr>
<td>Mono. dot calc.</td>
<td>E ⊢ e : τ</td>
<td>640</td>
<td>(d, E) → E'</td>
<td>640</td>
</tr>
<tr>
<td>Poly. dot calc.</td>
<td>E ⊢ e : τ</td>
<td>649</td>
<td>(d, E) → E'</td>
<td>649</td>
</tr>
<tr>
<td>µCLU</td>
<td>E ⊢ e : τ</td>
<td>700</td>
<td>(d, E) → E'</td>
<td>708</td>
</tr>
<tr>
<td></td>
<td>E ⊢ s : P (statement)</td>
<td>703</td>
<td></td>
<td></td>
</tr>
<tr>
<td>µML</td>
<td>(as in nano-ML)</td>
<td>491</td>
<td>(as in nano-ML)</td>
<td>489</td>
</tr>
<tr>
<td></td>
<td>Γ, Γ' ⊢ p : τ (pattern)</td>
<td>779</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Well-formedness judgments

<table>
<thead>
<tr>
<th>Language</th>
<th>Form</th>
<th>Judgment</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typed Impcore</td>
<td>Type</td>
<td>τ is a type</td>
<td>394</td>
</tr>
<tr>
<td>Typed µScheme</td>
<td>Kind</td>
<td>κ is a kind</td>
<td>418</td>
</tr>
<tr>
<td>Mono. dot calc.</td>
<td>Type</td>
<td>Δ ⊢ τ ::: κ</td>
<td>418</td>
</tr>
<tr>
<td></td>
<td>Environment</td>
<td>E ⊢ ⊥</td>
<td>640</td>
</tr>
<tr>
<td>Polymorphic CLU dot calculus</td>
<td>Environment</td>
<td>E ⊢ ⊥</td>
<td>640</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>E ⊢ τ ::: *</td>
<td>640</td>
</tr>
<tr>
<td></td>
<td>Constraint</td>
<td>E ⊢ C^x WF</td>
<td>645</td>
</tr>
</tbody>
</table>
# Evaluation and type-checking functions

<table>
<thead>
<tr>
<th>Language</th>
<th>Evaluation</th>
<th>Type checking and elaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exp. Page</td>
<td>Def. Page</td>
</tr>
<tr>
<td>Impcore</td>
<td>eval 43</td>
<td>evaldef 51</td>
</tr>
<tr>
<td>μScheme</td>
<td>eval 155</td>
<td>evaldef 160</td>
</tr>
<tr>
<td>μScheme+</td>
<td>eval 260</td>
<td>evaldef 160</td>
</tr>
<tr>
<td>μScheme (in ML)</td>
<td>eval 360</td>
<td>evaldef 366</td>
</tr>
<tr>
<td>Typed Impcore</td>
<td>eval 1273</td>
<td>evaldef 1274</td>
</tr>
<tr>
<td>Typed μScheme</td>
<td>eval 1280</td>
<td>evaldef 1282</td>
</tr>
<tr>
<td>nano-ML</td>
<td>eval 517</td>
<td>evaldef 518</td>
</tr>
<tr>
<td>μCLU</td>
<td>eval 1321</td>
<td>evaldef 684</td>
</tr>
<tr>
<td>μML</td>
<td>eval 775</td>
<td>evalDataDef 778</td>
</tr>
<tr>
<td>μSmalltalk</td>
<td>eval 950</td>
<td>evaldef 957</td>
</tr>
<tr>
<td></td>
<td>typeof 398</td>
<td>elabdef 402</td>
</tr>
<tr>
<td></td>
<td>typeof (\xi) 433</td>
<td>elabdef (\xi) 433</td>
</tr>
<tr>
<td></td>
<td>typeof 514</td>
<td>elabdef 516</td>
</tr>
<tr>
<td></td>
<td>typeof 699</td>
<td>elabdef 709</td>
</tr>
<tr>
<td></td>
<td>ty 781</td>
<td>elabDataDef 794</td>
</tr>
</tbody>
</table>

## Other judgments

<table>
<thead>
<tr>
<th>Language</th>
<th>Concept</th>
<th>Judgment</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>μScheme</td>
<td>Primitive equality</td>
<td>(v_1 \equiv v_2)</td>
<td>146</td>
</tr>
<tr>
<td>μScheme+</td>
<td>Tail position</td>
<td>(e) is in tail position</td>
<td>290</td>
</tr>
<tr>
<td>μScheme</td>
<td>Free term variable</td>
<td>(y \in \text{fv}(e))</td>
<td>374</td>
</tr>
<tr>
<td>Typed μScheme</td>
<td>Free type variable</td>
<td>(\alpha \in \text{ftv}(\tau))</td>
<td>438</td>
</tr>
<tr>
<td>Typed μScheme</td>
<td>Type equivalence</td>
<td>(\tau \equiv \tau')</td>
<td>436</td>
</tr>
<tr>
<td>Typed μScheme</td>
<td>Capture-avoiding substitution</td>
<td>(\tau'[\alpha \mapsto \tau] \equiv \tau'')</td>
<td>441</td>
</tr>
<tr>
<td>Nano-ML</td>
<td>Constraint satisfied</td>
<td>(C) is satisfied</td>
<td>503</td>
</tr>
<tr>
<td>Polymorphic CLU dot calculus</td>
<td>Constraint satisfied</td>
<td>(E : C^\times)</td>
<td>645</td>
</tr>
</tbody>
</table>
Symbols and notation, in order of appearance

Impcore

::= defines a syntactic category in a grammar, page 6
| separates alternatives in a grammar, page 6
{ ⋯ } repeatable syntax in a grammar, page 6
ρ value environment ("row"), page 17
x object-language variable, page 17
⇒ shows binding in function or environment, page 17
v value, page 17
y object-language variable, page 17
{} empty environment, page 17
ξ global-variable environment ("ksee"), page 18
φ function environment ("fee"), page 18
d definition, page 19
e expression, page 19
v value, page 17
y object-language variable, page 17
{} empty environment, page 17
ξ global-variable environment ("ksee"), page 18
φ function environment ("fee"), page 18
⇒ relates initial and final states in big-step evaluation judgment ("yields"), page 19
dom domain of an environment or function, page 21
∈ membership in a set, page 21
f name of object-language function, page 23
→ relates initial and final states in evaluation of definitions, page 25
A for an automatically generated function, used as index in lieu of a page number, page 32
≡ defines syntactic sugar, page 62
[ ⋯ ] brackets used to wrap syntax ("Oxford brackets"), page 74
[ ⋯ ] optional syntax in a grammar, page 80

μScheme

S S-expression, page 88
() empty list, page 88
#t literal true, page 88
#f literal false, page 88
〚 introduces literal S-expression in Scheme, page 89
P in a mini-index, marks a primitive function ("primitive"), page 89
X a list of X’s, page 93
O( ⋯ ) asymptotic complexity, page 93
k a key in an association list, page 97
a an attribute in an association list, page 97
{ ⋯ } justification of a step in an equational proof, page 102
∪ set union, page 107
| separates element from condition in set comprehension ("such that"), page 107
∧ logical conjunction ("and"), page 107
{ ⋯ } a closure, page 115
ℓ a location, page 116
° function composition ("composed with"), page 119
:: infix notation for cons ("cons"), page 122
∨ disjunction ("or"), page 133
¬ Boolean complement ("not"), page 133
σ the store: a mapping of locations to values ("sigma"), page 140
⊆ the subset relation, reflexively closed ("subset"), page 195

**µScheme**+

- a hole in an evaluation context ("hole"), page 239
- [] an empty stack ("empty"), page 240
- F evaluation context ("frame"), page 240
- S evaluation stack, page 240
- → the reduction relation in a small-step semantics ("steps to"), page 243
- e/v an abstract-machine component that is either an expression or a value, page 243
- →* the reflexive, transitive closure of the reduction relation ("normalizes to"), page 244
- λ the Greek way of writing `lambda`, page 279
- C an evaluation context in a traditional semantics, page 282

**Garbage collection**

- H the size of the heap, page 304
- L the amount of live data, page 305
- γ the ratio of heap size to live data ("gamma"), page 307

**Type systems**

- τ a type ("tau"), page 393
- Γ type environment, mapping name of term variable to its type ("gamma"), page 393
- → in a function type, separates argument types from the result type ("arrow"), page 393
- × in a function type, separates the types of the arguments ("cross"), page 393
- ⊢ in a judgment, separates context from conclusion ("turnstile"), page 394
- e : τ ascribes type τ to term e ("e has type τ"), page 394
- → relates type environments before and after elaboration of definition, page 396
- µ a type constructor ("mew"), page 410
- × Used to form pair types or product types ("cross"), page 411
- + used to form sum types, page 412
- [τ] the set of values associated with type τ, page 414
- κ a kind, which classifies types ("kappa"), page 417
- * the kind ascribed to types that classify terms ("type"), page 417
- ⇒ used to form kinds of type constructors ("arrow"), page 417
- τ :: κ ascribes kind κ to type τ ("τ has kind κ"), page 417
- Δ a kind environment ("delta"), page 418
- α, β, γ type variables ("alpha, beta, gamma"), page 420
- ∀ used to write quantified, polymorphic types ("for all"), page 420
- τ[α → τ'] τ with τ' substituted for α ("τ with α goes to τ'"), page 420
- (τ₁, ..., τₙ) τ applied to type parameters τ₁, ..., τₙ, page 422
- ≡ type equivalence, page 436
- ∩ set intersection, page 441
- ∅ the empty set ("empty"), page 441
Notation

Type inference

$\sigma$ a type scheme (“sigma”), page 480
$\theta$ a substitution (“THAYT-uh”), page 481
$\prec$ the instance relation (“instance of”), page 482
$\theta_I$ the identity substitution, page 483
$\tau \sim \tau'$ simple type-equality constraint (“$\tau$ must equal $\tau'$”), page 491
$C$ type-equality constraint, page 491
$T$ the trivial type-equality constraint, page 494
$\equiv$ equivalence of constraints, page 508

[· · ·] wrap optional syntax in a grammar, page 558
PART I
FOUNDATIONS
Chapter contents

1.1 The Impcore language 5
   1.1.1 Concrete syntax 6
   1.1.2 What the syntax and the primitives do 7
   1.1.3 Examples 8
   1.1.4 Beyond interactive computation 12
   1.1.5 Primitive, predefined and basis; the initial basis 15

1.2 Abstract syntax 16

1.3 Environments 17

1.4 Operational semantics 18
   1.4.1 Judgments and rules of inference 19
   1.4.2 Literal values 20
   1.4.3 Variables 21
   1.4.4 Assignment 21
   1.4.5 Control Flow 22
   1.4.6 Function Application 23
   1.4.7 Rules for evaluating definitions 25

1.5 The interpreter 26
   1.5.1 Interfaces 30
   1.5.2 Implementation of the evaluator 42
   1.5.3 Implementation of main 52
   1.5.4 Implementation of names 54
   1.5.5 Implementation of environments 54

1.6 Operational semantics revisited: Proofs 56
   1.6.1 Proofs about evaluation: Theory 56
   1.6.2 Proofs about derivations: Metatheory 57
   1.6.3 Why bother with semantics, proofs, theory, and metatheory? 59

1.7 Extending Impcore 60

1.8 Summary 62
   1.8.1 Key words and phrases 62
   1.8.2 Further reading 66

1.9 Exercises 66
   1.9.1 Language design: learning to think about syntactic structure 69
   1.9.2 The difficulties of theory 69
   1.9.3 Programming in Impcore 70
   1.9.4 Digging into proof theory and metatheory 73
   1.9.5 Working with Impcore’s semantics 77
   1.9.6 Modifying the Impcore interpreter 80
Chapter 1

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 pure imperative language such as Ada 83, Algol 60, C, Cobol, Fortran, Modula-2, or Pascal. Or you may have used a language that mixes imperative and 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 imperatively.

You may think of imperative programming as “just programming,” but imperative programming is a well-developed style with identifiable characteristics:

- Programs process data one word at a time. Source code mentions the computation and movement of each individual item 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.

- Programs use many side effects. Computation continually changes the “mutable state” of the machine—the values contained in the various machine words—by assignment.

- 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 an imperative language can often be built by adapting a machine-level debugger.

The imperative style can be embodied in a language. In this chapter, that language is Impcore. Impcore offers standard imperative constructs—assignments, loops, conditionals, and procedures—which are ubiquitous in programming languages. They are found not only in “imperative” or “procedural” languages but also in many “functional” and “object-oriented” languages.

More important than the constructs, which should be familiar, are the intellectual tools this book uses to describe, understand, and implement languages:

- **Concrete syntax** is the form in which people write programs. Because we don’t want to get into the parsing techniques used to recognize concrete syntax, we use a simple, parenthesized-prefix syntax, which is easy to recognize.

- **Abstract syntax** expresses the essential structure of programs.

- **Operational semantics** explains the meaning of the abstract syntax by explaining how it should be evaluated.

- An interactive **interpreter** parses concrete syntax, converts it to abstract syntax, and evaluates it according to the operational semantics. Experimenting with interpreters builds insights and skills that can’t be obtained by mere discussion. The first few interpreters in this book are written in C; the rest in Standard ML.

- **Environments** determine the meanings of identifiers, both in the operational semantics and in the interpreter.

The first four of these tools—concrete syntax, abstract syntax, operational semantics, and an interpreter—are intimately connected, and the center of the connection is the abstract syntax. Abstract syntax says what a language is. A language can be simple only if its syntax is simple, and to make syntax simple, one good trick is to use as few syntactic categories as possible.

A syntactic category is a kind of a phrase in a language, like an “expression,” a “definition,” or a “statement.” Operationally, two phrases are in the same syntactic category if and only if they can be exchanged for each other without changing syntactic correctness. That is, if a program parses without a syntax error, and you replace one phrase with a different phrase that is in the same syntactic category, the new program will also parse without a syntax error. For example, in the C programming language, here are several phrases in the syntactic category of expressions:

\[ x + 1 \quad p->next \quad f(17) \]

Any of these phrases can appear in any context where an expression is expected, and the resulting C program will be grammatical.\(^1\)

\(^1\)“Grammatical” is not the same as “correct.” For example, replacing \(x + 1\) with \(p->next\) will break a program if \(x\) is declared but \(p\) is not, or if \(x + 1\) and \(p->next\) have incompatible types.
Most programming languages have at least four syntactic categories:

- A **declaration** introduces a new named thing without providing complete information about that thing. For example, the declaration of a variable might give its name and possibly its type, but not its initial value. The declaration of a function might give the types of its arguments and result, but not its body.

- A **definition** introduces a new named thing and provides all the information needed to program with that thing. For example, the definition of a variable might give its name and its initial value. The definition of a function might give its name, the names of its parameters, and its body.

- A **statement** is executed for **side effect**. For example, it might print output, it might change the value of a variable, or it might change some value in memory.

- An **expression** is evaluated to produce a value. In some languages, an expression can produce a value and have a side effect.

What does it mean, then, to have as few syntactic categories as possible? In this book,

- When possible, I avoid declarations: where a declaration could appear, I almost always require a definition. In late chapters, I make a couple of exceptions, allowing declarations of local variables or local procedures.

- I use a single syntactic category that encompasses both side-effecting phrases and value-producing phrases. Because the value-producing aspect is more important, I call that category **expressions**. This convention might surprise you: for example, a while loop isn’t usually an expression! But it can be, and there’s precedent; in the programming language Icon, for example, all looping constructs are expressions (Griswold and Griswold 1996).

When a language has only expressions, instead of both statements and expressions, there are fewer design decisions to make, and the language is simpler. We avoid gratuitous differences like this one between C and Pascal: in Pascal, an assignment is a statement, but in C, an assignment is an expression. And we avoid duplicating syntactic forms; for example, C has both a conditional **statement**, written with if and **else**, and also a conditional **expression**, written with ? and ::. To use C effectively, you have to know both.

### 1.1 The Impcore language

An Impcore program is a sequence of **definitions**. Definitions can be entered interactively or loaded from a file. Definitions have 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)). Other definition forms include **unit tests**, such as (check-expect (square 5) 25). Finally, because it’s wonderfully convenient to be able to type in an expression, have it evaluated, and see the result, we also count 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 twenty years.

Impcore’s interpreter “remembers” all definitions; if you are used to compiling programs at a command line, “remembering a definition” corresponds roughly to “compiling a program.” (And at the end of each Impcore program or file, the interpreter runs all the unit
tests.) And “evaluating an expression” (the special syntax for the definition of it) corresponds roughly to “running a program.”

Impcore’s syntax, Impcore’s semantics, and examples of Impcore programs are presented below.

1.1.1 Concrete syntax

In this book, concrete syntax is specified using Extended Backus-Naur Form (EBNF). In this notation, text in typewriter font should be typed literally. Names in italic font stand for syntactic categories; the name of a category appears on the left, followed by a special symbol pronounced “produces” (::=), followed by the ways of producing phrases in the category. Alternatives are separated by vertical bars, as (one thing | another); each alternative is called a syntactic form. A phrase that can be repeated is written in braces, as {phrase}, which means that phrase may appear any number of times in sequence, or not at all. The notation is explained more fully in Appendix F.

Impcore’s concrete syntax wraps almost every syntactic form in parentheses. The forms are divided into syntactic categories, of which the most important are definitions, written def, and expressions, written exp. A unit-test is a special form of definition.

\[
\text{def} ::= \text{val} \text{ variable-name} \text{ exp} \\
| \text{exp} \\
| (\text{define} \text{ function-name} \text{ (formals)} \text{ exp}) \\
| (\text{use} \text{ file-name}) \\
| \text{unit-test}
\]

\[
\text{unit-test} ::= (\text{check-expect} \text{ exp} \text{ exp}) \\
| (\text{check-error} \text{ exp})
\]

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

\[
\text{formals} ::= \{\text{variable-name}\}
\]

\[
\text{literal} ::= \text{numeral}
\]

\[
\text{numeral} ::= \text{sequence of digits, possibly prefixed with a plus or minus sign}
\]

\[
\text{*\text{-name}} ::= \text{sequence of characters that is not a numeral and does not contain} \\
(, ), [ ], ;, \text{ or whitespace}
\]

A semicolon introduces a comment, which continues to the end of the line. A name may use any other characters except parentheses, brackets, whitespace, and null. But not every combination of such characters is a name; words define, val, set, if, while, begin, check-expect and check-error are reserved—they cannot be used to name functions or variables. Similarly, a numeral always stands for a number, and it 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
1.1. THE IMPCORE LANGUAGE

form the initial basis. These concepts—primitive, predefined, and basis—are explained in Section 1.1.5 on page 15.

Now that we know about names, let us look at the syntactic forms, starting with definitions. The definition form \((\text{val } x \ e)\) 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. And the definition form \(\text{(define function-name (formals) exp)}\) defines a new function. The \text{val}, \text{exp}, and \text{define} forms are the true definitions.

The remaining definition forms are extended definitions. Evaluating \((\text{use file-name})\) reads the definitions in the named file as if they had been typed directly to the interpreter. Evaluating \text{check-expect} or \text{check-error} remembers a test that the interpreter runs once it has read the file in which the test appears.

Expressions are fully parenthesized. They use prefix syntax, in which each operator precedes its arguments. You are probably more accustomed to infix syntax, in which binary operators appear between their arguments. In infix syntax, order of evaluation is determined by “operator precedence.” In prefix syntax, order of evaluation is determined by parentheses; there is no need for operator precedence. For example, the C assignment \(i = 2*j + i - k/3\), which is written in infix syntax, would be written in Impcore as \((\text{set } i (- (+ (* 2 j) i) (/ k 3)))\). The parentheses eliminate any possible ambiguity.

If you're used to infix syntax, you may find prefix 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 luckily, you’ll rarely see an expression as complex as the example above.

1.1.2 What the syntax and the primitives do

The meaning of a definition or expression is operational: we know what a thing means if we know what happens when it is evaluated. To explain the meanings of Impcore's syntactic forms, I want to draw on your experience and intuition, so I use informal English. But informal English lacks precision, so I also provide a precise, formal semantics, which appears in Section 1.4 on page 18.

The meaning of an expression is explained by saying what happens when the expression is evaluated and what result is returned. In Impcore, all values are integers; as in C, \text{if} and \text{while} use their conditions by interpreting zero as false and nonzero as true.

\((\text{if } e_1 \ e_2 \ e_3)\) — Evaluate \(e_1\); if it is nonzero, evaluate \(e_2\) and return the result, otherwise evaluate \(e_3\) and return the result.

\((\text{while } e_1 \ e_2)\) — Evaluate \(e_1\); if it is not zero, evaluate \(e_2\), then start evaluating the loop again with \(e_1\). Continue until \(e_1\) evaluates to zero. When \(e_1\) evaluates to zero, looping ends, and the result of the \text{while} loop is zero. (The result of evaluating a \text{while} expression is always zero, but because \text{while} is typically evaluated for its side effects, we usually don’t care.)

\((\text{set } x \ e)\) — Evaluate \(e\), assign its value to the variable \(x\), and return its value. Variable \(x\) must be either a function parameter or a global variable defined with \text{val}.

\((\text{begin } e_1 \cdots \ e_n)\) — Evaluate \(e_1, \ldots, e_n\), in that order, and return the value of \(e_n\).

\((\text{f } e_1 \cdots \ e_n)\) — Evaluate \(e_1, \ldots, e_n\), in that order, calling the results \(v_1, \ldots, v_n\). Apply function \(f\) to \(v_1, \ldots, v_n\) and return the result. Function \(f\) may be primitive or user-defined; if \(f\) is user-defined, find its definition, let the names in the \text{formals} stand for \(v_1, \ldots, v_n\), and return the result of evaluating \(f\)'s body.
Each primitive takes two arguments, except the printing primitives, which take one each. The arithmetic operators +, -, *, and / do machine arithmetic on integers, as in C. Each of the comparison operators <, >, and = does a comparison: if the comparison is true, the operator returns one; otherwise, it returns zero.

Like every bridge language, Impcore offers three primitives for printing. 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 λ. Some useful code points are shown in Table 2.1 on page 89.

Because the interpreter automatically prints the value of each expression you enter, you won't use the printing primitives often—I use them primarily for debugging. I typically use println, but when I want fancier output, a combination of printu and print can be the way to go.

What about the names of variables? Like C, Impcore has global variables and formal parameters. If a variable-name like x occurs in a function definition, and if the function has a formal parameter named x, then x refers to that formal parameter; otherwise x refers to a global variable. A variable-name that occurs in a top-level expression, outside of any function definition, necessarily refers to a global variable. (Like Awk, Impcore has no local variables; to provide them is the object of Exercise 30 on page 80.)

Like the meaning of an expression, the meaning of a definition is explained by saying what happens when the definition is evaluated. But in the case of a definition, 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.

(val x e) — If there is not already a global variable x, create one. Then evaluate e and assign its value to x.

(e) — Evaluate e and store its value in the global variable it.

(define f (x \cdot \cdot \cdot x_n) e) — Remember f as a function that takes arguments x_1, . . . , x_n and returns e.

(use my.imp) — File my.imp should contain a sequence of Impcore definitions; read them and evaluate them. After reading the file, run any unit tests it contains.

(check-expect e1 e2) — Remember this test: at the end of the file containing the test, evaluate both e1 and e2. If their values are equal, the test passes; if not, it fails.

(check-error e) — Remember this test: at the end of the file containing the test, evaluate e. If evaluating e triggers an error, like dividing by zero or passing the wrong number of arguments to a function, the test passes; if not, it fails.

1.1.3 Examples

As indicated above, you can enter definitions and expressions interactively. The interpreter remembers each definition. When the definition is an expression, the interpreter evaluates it, prints its value, and stores the value in the global variable it, which need not be defined in
1.1. THE IMPCORE LANGUAGE

advance. I demonstrate the interpreter by evaluating such expressions, using only primitive functions. 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:

9a \begin{verbatim}
(transcript 9a)≡
-> 3
-> (+ 4 7)
11
-> it
11
-> (val x 4)
4
-> (+ x x)
8
\end{verbatim}

In the examples just shown, the value defined or computed is printed automatically by the interpreter. This automatic printing happens even when evaluating the definition also prints. Don’t be baffled by effects like these:

9b \begin{verbatim}
(transcript 9a)+≡
-> (println x)
4
4
\end{verbatim}

The 4 is printed twice because println is called with actual parameter 4 (the value of x), and println both 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).

9c \begin{verbatim}
(transcript 9a)+≡
-> (val y 5)
5
-> (begin (println x) (println y) (* x y))
4
5
20
\end{verbatim}

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

9d \begin{verbatim}
(transcript 9a)+≡
-> (begin (print x) (print y) (* x y))
4520
\end{verbatim}

I show another strange example below, using printu, but for that example I need control structures and a user-defined function.

Our first control structure is an if expression:

9e \begin{verbatim}
(transcript 9a)+≡
-> (if (> y 0) 5 10)
5
\end{verbatim}

\footnote{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 9a identify chunks of examples or code, and pointers such as “9c ⊲” point to subsequent chunks. A fuller explanation of Noweb appears in Section 1.5 on page 27.}
Next a \texttt{while} expression whose body (that is, its second component) is a \texttt{begin} expression.

\begin{verbatim}
(transcript 9a)+≡
  -> (while (> y 0)
       (begin
         (set x (+ x x))
         (set y (- y 1)))
     0
  -> x
  128
\end{verbatim}

The interpreter allows expressions to spread over more than one line; if the interpreter sees an incomplete expression, it prompts with three spaces and waits for more input.

Next are two function definitions, and one call to each. After reading the definition of a function, the interpreter echoes its name.

\begin{verbatim}
(transcript 9a)+≡
  -> (define add1 (x) (+ x 1))
      add1
  -> (add1 4)
      5
  -> (define double (x) (+ x x))
       double
  -> (double (+ 3 4))
      14
\end{verbatim}

The call to a user-defined function works much as in C: first the arguments are evaluated; those values are the \textit{actual parameters}. Next then the function’s body is evaluated with each actual parameter “bound to” (which is to say, named by) the corresponding \textit{formal parameter} from the \textit{formals} in the function’s definition. In the first example, the actual parameter is 4, and (+ x 1) is evaluated with 4 bound to x. In the second example, the actual parameter is 7, and (+ x x) is evaluated with 7 bound to x.

Have you noticed that in some examples, x refers to a global variable, and in others, x refers to a formal parameter? Just as in C, we can use the same name in multiple contexts. And also as in C, we know from the context exactly what each occurrence of a name refers to: if the context is the body of a function definition that has a formal parameter of the given name, then the name refers to the formal parameter; otherwise, the name refers to a global variable. Let’s contrive an example:

\begin{verbatim}
(transcript 9a)+≡
  -> x
  128
  -> (define addx (x y) (set x (+ x y)))
      addx
  -> (addx x 1)
      129
  -> x
  128
\end{verbatim}

Within the body of \texttt{addx}, the two occurrences of x refer to the formal parameter. But in the top-level expression \texttt{x} and \texttt{(addx x 1)}, x refers to the global variable. And in the body of \texttt{addx}, where x is set, changing the formal x does not affect anything in the calling context; we say that Impcore passes parameters \textit{by value}. No assignment to a formal parameter can ever change the value of a global variable.
1.1. THE IMPCORE LANGUAGE

Here's another example of a function that takes multiple parameters, and that also uses `printu`. The call to `printu` uses the Unicode code point for an ordinary hyphen, which is 45; the function prints phone numbers.

```
-> (define println-phone (area-code exchange nnnn)
  (begin (print area-code) (printu 45)
    (print exchange) (printu 45)
    (println nnnn)))
```

```
-> (println-phone 607 797 7742)
607-797-7742
```

In the example, the body of `println-phone` is evaluated with 607 bound to `area-code`, 797 bound to `exchange`, and 7742 bound to `nnnn`. Evaluating the body has the effect of printing the phone number, and it returns the last value in the `begin` expression, which is 7742. As above, the things my function prints are followed by the value of the expression, which the interpreter prints.

In Impcore, a function call is valid 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.

```
-> (addx 17)
Run-time error: in (addx 17), expected 2 arguments but found 1
```

```
-> (println-phone 2124502027)
Run-time error: in (println-phone 2124502027), expected 3 arguments but found 1
```

The examples above show function definitions, function calls, a conditional and a `while` loop. To conclude, I show another loop and compare it with recursion. In particular, I compare iterative and recursive versions Euclid's algorithm for computing the greatest common denominator of two numbers, in both Impcore and C. Here is iterative C code:

```
int gcd(int m, int n) {
  int r;

  while ((r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

And here is iterative Impcore code, using predefined functions `!=` and `mod`. Because Impcore has no local variables, I use a global `r`:

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

```
-> (gcd 6 15)
3
```

3From here on, I omit the names that interpreters print after function definitions.
Here is recursive C code:

\[
\text{(gcd-recur.c 12a)≡}
\]
\[
\begin{align*}
\text{int gcd(int m, int n) \{} \\
\text{\hspace{1em} if (n == 0) } \\
\text{\hspace{2em} return m; } \\
\text{\hspace{1em} else } \\
\text{\hspace{2em} return gcd(n, m \% n); }
\end{align*}
\]

And here is the corresponding recursive Impcore code:

\[
\text{(transcript 9a)+≡} \]
\[
\text{→ (define gcd (m n)} \\
\text{\hspace{1em} (if (= n 0) } \\
\text{\hspace{2em} m } \\
\text{\hspace{1em} (gcd n (mod m n))))}
\]

### 1.1.4 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 keep for a while, the Impcore interpreter, like all my interpreters, enables you to put code in a file. And when you put code in a file, you can add unit tests. Here’s the \text{gcd} example in file form:

\[
\text{(gcd.imp 12c)≡} \]
\[
\text{(val r 0)} \\
\text{(define gcd (m n)} \\
\text{\hspace{1em} (begin } \\
\text{\hspace{2em} (while (!= (set r (mod m n)) 0) } \\
\text{\hspace{2em} (begin } \\
\text{\hspace{3em} (set m n) } \\
\text{\hspace{3em} (set n r))) } \\
\text\hspace{2em} n))}
\]

\[
\text{(check-expect (gcd 6 15) 3)}
\]

The code is unchanged, but since it’s in a file, you don’t see arrow prompts. And \text{check-expect} is new. It says if we call \text{(gcd 6 15)}, we expect the result to be 3. Here are some more unit tests:

\[
\text{(gcd.imp 12c)+≡} \]
\[
\text{(check-expect (gcd 15 15) 15)} \\
\text{(check-expect (gcd 14 15) 1)} \\
\text{(check-expect (gcd 14 1) 1)} \\
\text{(check-expect (gcd 72 96) 24)} \\
\text{(check-error (gcd 14 0))}
\]

The last unit test differs from the others. It says that if we evaluate \text{(gcd 14 0)}, we should expect a run-time error.

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

\[
\text{(transcript 9a)+≡} \]
\[
\text{→ (use gcd.imp)} \\
\text{0} \\
\text{gcd} \\
\text{All 6 tests passed.}
1.1. THE IMPCORE LANGUAGE

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.

\[
\begin{align*}
1 & \quad = \quad * \\
3 & \quad = \quad * \\
& \quad \quad ** \\
6 & \quad = \quad * \\
& \quad \quad * \\
& \quad \quad **
\end{align*}
\]

You can compute a triangular number using the \textit{sigma} function in Exercise 4 on page 70, but there’s a shortcut:

\begin{verbatim}
(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))
\end{verbatim}

When writing \textit{triangle}, I botched my first attempt. My unit tests caught the botch. Here’s what it looks like:

\begin{verbatim}
(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))
\end{verbatim}

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.
Sidebar: Designing “bridge” languages for learners

I know of two popular approaches to the study of programming languages. One is to study widely deployed industrial or research languages. Such languages are usually big and have interesting implementations, and you can write interesting programs in them. But it can be hard to discover what the languages actually are, and even harder to discover the rules that specify how programs behave.

Another approach is to study small, artificial languages, sometimes called “core calculi.” (The most famous such calculus is the lambda calculus.) Such languages are usually defined very precisely, making it easy to discover what they are and how programs behave. But it can be hard to write any interesting programs.

The languages in this book represent a design compromise: a bridge, if you will, from introductory programming to industrial practice. The bridge languages resemble core calculi more closely than industrial or research languages, but like the industrial and research languages, they are intended for actual programming. I won’t pretend you can write interesting programs in Impcore, but you can write interesting programs in μScheme, μML, μCLU, μSmalltalk, and μProlog. Nevertheless, the definitions of the languages in this book are mostly nailed down, so you always know exactly what the language is, and you almost always know the rules that specify how programs behave.

My design compromise is what led me to split the definition forms into two groups. The val, define, and top-level expression forms are true definitions. Each true definition has an operational semantics which nails down exactly how it behaves. Evaluating a true definition updates an environment, in a way that is specified by the semantics. The use, check-expect, and check-error forms are, by contrast, not true definitions. None of these forms is described by the operational semantics. And evaluating these forms doesn’t update an environment directly—instead, it either causes other definitions to be evaluated (use), or it saves test code to be run later (check-expect and check-error). These additional forms, together with the true definitions, constitute the extended definitions.

Here’s how I’d like you to think about true definitions and extended definitions: the true definitions are part of the language, and the extensions are there to make you more productive as a programmer. The use extension enables you to put code in files, and to load that code from files. The unit-test forms not only help you build confidence that your code is right; they help you document your code. (The best documentation is documentation that is checked by the interpreter.)

All the bridge languages use the same design compromise: every one has its true definitions, and every one includes extended definitions like use, check-expect, and check-error. I hope this compromise allows you to focus on ideas of lasting value, while also giving you the tools you need to write interesting programs and make them work.
1.1. Primitive, predefined and basis; the initial basis

A define may be used to redefine a primitive or predefined function, but it’s foolhardy.

In language design, programmers and implementors have competing desires. Programmers want lots of data types and operations. Implementors want to keep primitive functionality small and simple. (So do semanticists!) Designers have found two strategies that help satisfy these competing desires: 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. Section 1.7 on page 60 defines several such constructs, which you can add to Impcore in the Exercises. For example, you can extend Impcore with a for expression, which is defined by translation into begin and while.

The initial basis applies the same idea not to language constructs but to predefined functions. Some functions are primitive functions defined directly by C code in the interpreter. Other functions are also built into the interpreter, but are defined in terms of existing primitives and language constructs. All these functions are available to the programmer, and together they form the initial basis. Having a small set of primitives and a large initial basis makes things easy for everyone. Implementors can add to the initial basis just by writing ordinary code in the language, which is easier than defining new primitives. Figure 1.1 shows our predefined Impcore functions.

We write Boolean connectives using if expressions.

\[
\text{(define and (b c) (if b c b))}
\]

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

We add new arithmetic comparisons.

\[
\text{(define <= (x y) (not (> x y)))}
\]

We finally define modulus in terms of division.

\[
\text{(define mod (m n) (- m (* n (/ m n))))}
\]

The C code to install the initial basis is shown in chunk (install the initial basis in functions 53b), which is continued in chunk 53d.

Figure 1.1: Additions to Impcore’s initial basis

Before going on to the next sections, work some of the problems in Section 1.9.3 of the exercises (“Programming in Impcore”), which starts on page 70.
1.2 Abstract syntax

Throughout this book, we represent programs as abstract-syntax trees (ASTs), which are the best and simplest internal representation of source code. An abstract-syntax tree separates the internal representation of programs from the representation used to express the programs in written form. It reflects the structure of the original source code, without extraneous details. Abstract syntax is not just a great technique for implementing programming languages; it is the best way think about syntax.

As an example, a phrase in Impcore can be represented by an abstract-syntax tree that uses leaf nodes to represent atomic phrases of the language, such as names and numerals, and uses interior nodes to represent composite phrases, such as expressions formed using if and set.

We specify abstract syntax by naming each kind of node, and by saying what kind of children each node has. For example, the abstract representation of an Impcore definition is

\[
\text{Def} = \text{VAL} (\text{Name}, \text{Exp}) \\
| \text{EXP} (\text{Exp}) \\
| \text{DEFINE} (\text{Name}, \text{Namelist}, \text{Exp})
\]

The Def type includes only the true definitions, not the extended definitions; extended definitions are XDefs.

Similarly, the abstract representation of an expression is:

\[
\text{Exp} = \text{LITERAL} (\text{Value}) \\
| \text{VAR} (\text{Name}) \\
| \text{SET} (\text{Name}, \text{Exp}) \\
| \text{IF} (\text{Exp}, \text{Exp}, \text{Exp}) \\
| \text{WHILE} (\text{Exp}, \text{Exp}) \\
| \text{BEGIN} (\text{Explist}) \\
| \text{APPLY} (\text{Name}, \text{Explist})
\]

It’s worth comparing this description with the description of \textit{exp} in the concrete syntax (page 6).

As an example, the expression \texttt{(set i (- (+ (* 2 j) i) (/ k 3)))} has the following abstract-syntax tree:

```
SET
  i
  -
    APPLY
      Explist
    APPLY
      Explist
        +
        Explist
          APPLY
            Explist
              APPLY
                Explist
                  *
                  Explist
                    APPLY
                      Explist
                        LITERAL
                          k
                          3
                        LITERAL
                          2
                          j
```
1.3 ENVIRONMENTS

Abstract syntax does not say anything about how a program is written in a source file; it specifies only the structure of a language. For example, all that matters about an if node is that it has three children: a conditional expression, an expression to evaluate when the condition is true, and an expression to evaluate when the condition is false. It doesn’t matter how the expression is written using the concrete syntax; once we get to abstract syntax, the phrases “(if e₁ e₂ e₃),” “if e₁ then e₂ else e₃,” and even “e₁ ? e₂ : e₃” are indistinguishable—the concrete syntax is “abstracted away.”

The information in an abstract-syntax tree is also present in the (concrete) source, but the AST is much easier to analyze, manipulate, and interpret. 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.

The process of creating abstract syntax from concrete input is called parsing, and it is also a good way to find malformed phrases such as (if x 0) or (val y). There is a large body of literature devoted to parsing; Appel (1998) and Aho, Sethi, and Ullman (1986) provide textbook treatments. Appendix B presents a parser that you can easily extend to add new syntactic forms to Impcore and other bridge languages.

My interpreters convert concrete syntax to ASTs; then they manipulate the ASTs. ASTs haven’t always been a good choice: in the early days of programming-language implementation, memory was measured in kilobytes, so interpreters and compilers often avoided representing ASTs explicitly. Instead, they translated concrete syntax directly into a byte-code or intermediate form. As soon as machines started to have many megabytes of memory, let alone the quantities we enjoy today, the simplicity of explicit ASTs made them the better tradeoff, especially when using compiler-construction tools that automate the construction of ASTs.

1.3 Environments

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 (set x 1) depends on whether x is a formal parameter, a global variable, or something else entirely. In programming-language theory, the thing that defines the meanings of names is called an environment; it is usually considered to be a mapping from names to meanings. In implementations, environments are often realized as hash tables or search trees, and they are sometimes called symbol tables.

We write ρ(x) to mean whatever is associated with the name x in the environment ρ. We write ρ{x ↦ v} to mean the environment ρ extended by a binding of the name x to v, whatever v is:

\[ ρ\{x ↦ v\}(y) = \begin{cases} v, & \text{when } y = x \\ ρ(y), & \text{when } y \neq x \end{cases} \]

We write {} to mean the empty environment, i.e., the environment that does not bind any names. And we write dom ρ to mean the set of names bound in environment ρ.
Knowing what can be in an environment tells you what kinds of things names can stand for, which is a good first step in understanding a new programming language. Impcore uses three environments: the environment $\xi$ ("ksee") contains the values of global variables, the environment $\rho$ ("row", rhymes with "dough") contains the values of formal parameters, and the environment $\phi$ ("fee") contains the definitions of functions. The following transcript cleverly uses $x$ in all three environments.

\begin{verbatim}
(transcript x) +≡ ⊳ 13d
  -> (val x 2)
  2
  -> (define x (y) (+ x y)) ; pushing the boundaries of knowledge...
  -> (define z (x) (x x))    ; and sanity
  -> (z 4)
  6
\end{verbatim}

The first definition introduces a global variable $x$ with value 2. The second defines a function $x$ that adds its argument to the global variable $x$. The third defines $z$, which-passes its formal parameter $x$ to the function $x$. This example is designed to push to you understand the rules of Impcore; it can be understood only by a person who knows not only that Impcore has three environments, but also how they are used. Of course, no sane person programs this way; production codes are written to be easy to understand, even by readers who may have forgotten some details of the language.

1.4 Operational semantics

Section 1.1.2 describes Impcore informally. Section 1.5 presents C code for an interpreter that implements Impcore. The informal description is short and readable but hard to make precise. The interpreter is precise, but it is much longer and harder to understand than an informal description. It also embodies many other decisions, including decisions about the representations of names, environments and abstract syntax. If we just want to know what an Impcore program means, these decisions are irrelevant and distracting—they are part of an implementation, not part of the language.

This section defines the meaning of Impcore using formal operational semantics, a description technique that is both short and precise. Operational descriptions specify the meanings of programs while ignoring details required for a running implementation. Although the notation can be intimidating at first, with practice you can read an operational description almost as easily as a description written in informal English.

An operational semantics is written by defining an abstract machine and rules for its execution. A good abstract machine is designed to show how to run programs written in the language being defined. We define the machine’s states, including its start state and its acceptable final states, and we present 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.”

Sometimes a machine reaches a state from which it cannot make forward 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.)
1.4. OPERATIONAL SEMANTICS

The state of the 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. When the machine is evaluating a definition \(d\), we write its state as \(\langle d, \xi, \phi \rangle\). When it is evaluating an expression \(e\), we write its state \(\langle e, \xi, \phi, \rho \rangle\). The state \(\langle d, \xi, \phi \rangle\) does not include an environment \(\rho\), because definitions do not appear inside functions, so while a definition is being evaluated, there are no formal parameters. When the machine is resting between evaluations, it remembers only the values of global variables and the definitions of functions, and we write its state \(\langle \xi, \phi \rangle\).

Value environments \(\xi\) and \(\rho\) map names to values; in Impcore, the values stored in \(\xi\) and \(\rho\) are integers. A function-definition environment \(\phi\) maps each name either to a primitive function or a user-defined function. We write a primitive function as \text{PRIMITIVE}(\oplus), where \(\oplus\) is the name of an operator like +, =, or \(*\). We write a user-defined function as \text{USER}((x_1, \ldots, x_n), e), where the \(x_i\)’s are the formal parameters and \(e\) is the body. These symbols and their code equivalents are summarized in Table 1.2 on page 28.

In the initial state of the Impcore machine, \(\xi = \emptyset\), 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 (see Figure 1.1 on page 15).

1.4.1 Judgments and rules of inference

The transition rules of the abstract machine are written in the form of judgments. There is one form of judgment for definitions and a different form for expressions. We write the judgment \(\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho' \rangle\) to mean “evaluating expression \(e\) produces value \(v\).” More precisely, the judgment means “in the environments \(\xi, \phi, \rho\), evaluating \(e\) produces a value \(v\), and it also produces new environments \(\xi'\) and \(\rho'\), while leaving \(\phi\) unchanged.”

We always use symbols \(e\) and \(e_i\) for expressions, \(v\) and \(v_i\) for values, and \(x\) and \(x_i\) for names.

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 \text{SET} and \text{WHILE}, which are typically evaluated only for side effects, must produce values.\(^5\)

- Evaluating an expression might change the value of a global variable (from \(\xi\)) or a formal parameter (from \(\rho\)).

- 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 17 on page 77).

---

\(^4\)A judgment is 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 non-deterministic. All the languages in this book are deterministic, but multithreaded languages, like Java or C\#, can be non-deterministic. 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 non-deterministic. Programs written in such languages can produce different answers when translated with different compilers. (See also Exercise 19 on page 77.)

\(^5\)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.
The form of the 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 heart of the interpreter in Section 1.5 is the function $\text{eval}$. Calling $\text{eval}(e, \xi, \phi, \rho)$ returns $v$ and has side effects on $\xi$ and $\rho$ such that $\langle e, \xi_{\text{before}}, \phi, \rho_{\text{before}} \rangle \Downarrow \langle v, \xi_{\text{after}}, \phi, \rho_{\text{after}} \rangle$. The judgment for definitions takes a simpler form. We write $\langle d, \xi, \phi \rangle \rightarrow \langle \xi', \phi' \rangle$ to mean “evaluating definition $d$ in the environments $\xi$ and $\phi$ yields new environments $\xi'$ and $\phi'$.”

To help distinguish this judgment from an expression judgment, I use a different arrow.

We can’t write just any judgment and expect it to be valid. 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 $\phi$, $\langle (+ 1 1), \xi, \phi, \rho \rangle \Downarrow \langle 4, \xi, \phi, \rho \rangle$ is clearly invalid. To tell which judgments are valid, an operational semantics uses rules of inference. Each rule has the form

$$\begin{align*}
\text{premises} & \quad \text{(NAME OF RULE)} \\
\text{conclusion} &
\end{align*}$$

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

For example, the rule

$$\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle$$

IfTrue

\[ \langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle, \]

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 $e_2$. (We can tell by looking 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 $e_2$.) The premises of this rule don’t even mention $e_3$, because if $v_1 \neq 0$, $e_3$ is never evaluated.

Once you have learned to read rules of inference, you’ll be able to translate them into recursive code. In Section 1.5, you can learn by example; I present a recursive implementation of $\text{eval}$, and I show each fragment of code together with the relevant rule or rules. My recursive implementation of $\text{eval}$ actually works bottom-up through rules. It is given a state $\langle e, \xi, \phi, \rho \rangle$, and it finds a new state $\langle v, \xi', \phi, \rho' \rangle$ such that $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$. It does so by looking at the form of $e$ and examining rules that have $e$’s of that form in their conclusions. For example, to evaluate an IF expression, it 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 can make 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 can then return $v_2$ and the modified environments.

The rest of this section gives all the rules in the semantics of Impcore.

### 1.4.2 Literal values

Literal values evaluate to themselves without changing any environments.

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

(LITERAL)

This rule has no premises, so at a LITERAL node, the recursive implementation of $\text{eval}$ terminates.
1.4. OPERATIONAL SEMANTICS

1.4.3 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. The phrase \( \text{dom} \rho \) stands for the domain of \( \rho \), i.e., the set of names bound by \( \rho \).

\[
\begin{align*}
\text{VAR}(x), \xi, \phi, \rho & \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle & \text{(FormalVAR)} \\
x \notin \text{dom} \rho & \quad x \in \text{dom} \xi & \text{(GlobalVAR)}
\end{align*}
\]

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

1.4.4 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.

\[
\begin{align*}
\text{e}, \xi, \phi, \rho & \Downarrow \langle \xi', \phi, \rho' \rangle & \text{(FormalAssign)} \\
x \notin \text{dom} \rho & \quad x \in \text{dom} \xi & \text{(GlobalAssign)}
\end{align*}
\]

The premises of both \text{set} rules involve evaluation judgments on the right-hand side \( e \), so at a \text{set} node, the recursive implementation of \text{eval} always makes a recursive call.

In Impcore, it is possible to assign only to previously defined variables; given a \text{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:

\[
\begin{align*}
\text{e}, \xi, \phi, \rho & \Downarrow \langle \xi', \phi, \rho' \rangle & \text{(GlobalAssign for Awk)}
\end{align*}
\]

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 \text{val} definition, as shown below in rule \text{DefineGlobal} (page 25).
1.4.5 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.

\[
\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle \quad \text{(IfTrue)}
\]

\[
\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0 \quad \langle e_3, \xi', \phi, \rho' \rangle \Downarrow \langle v_3, \xi'', \phi, \rho'' \rangle \quad \text{(IfFalse)}
\]

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.

\[
\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle \quad \langle \text{while}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle v_3, \xi''', \phi, \rho''' \rangle \quad \text{(WhileIterate)}
\]

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 \( v_3, \xi''', \phi, \rho''' \). 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.

\[
\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 = 0 \quad \langle \text{while}(e_1, e_2), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi', \phi, \rho' \rangle \quad \text{(WhileEnd)}
\]

In Exercise 16 on page 77, 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.

\[
(BEGIN(), \xi, \phi, \rho) \Downarrow (0, \xi, \phi, \rho)
\]  

(EMPTYBEGIN)

A nonempty BEGIN evaluates its expressions left to right.

\[
\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
\langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\
\vdots \\
\langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle
\]

(BEGIN)

As expected, the values \(v_1, \ldots, v_{n-1}\) 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 (see also Exercise 25 on page 79). The order of the premises themselves is irrelevant. For example, the BEGIN rule might equally well have been written this way:

\[
\langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\
\langle e_{n-1}, \xi_{n-2}, \phi, \rho_{n-2} \rangle \Downarrow \langle v_{n-1}, \xi_{n-1}, \phi, \rho_{n-1} \rangle \\
\vdots \\
\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle
\]

(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.

1.4.6 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.

\[
\phi(f) = \text{USER}((x_1, \ldots, x_n), e) \\
\text{all distinct} \\
\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
\vdots \\
\langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\
\langle \text{APPLY}(f, e_1, \ldots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle
\]

(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 \(x_1, \ldots, x_n\) 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 \( \rho_0, \ldots, \rho_n \).

• 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:

\[
\begin{align*}
\phi(f) &= \text{PRIMITIVE}(+) \\
\langle e_1, \xi, \phi, \rho_0 \rangle &\downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
\langle e_2, \xi_1, \phi, \rho_1 \rangle &\downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\
\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle &\downarrow \langle v_1 + v_2, \xi_2, \phi, \rho_2 \rangle \\
\end{align*}
\]

(ApplyAdd)

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 \( v_1 \neq v_2 \)):

\[
\begin{align*}
\phi(f) &= \text{PRIMITIVE}(=) \\
\langle e_1, \xi_0, \phi, \rho_0 \rangle &\downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
\langle e_2, \xi_1, \phi, \rho_1 \rangle &\downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\
\hline
\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle &\downarrow \langle 1, \xi_2, \phi, \rho_2 \rangle \\
\end{align*}
\]

(ApplyEqTrue)

\[
\begin{align*}
\phi(f) &= \text{PRIMITIVE}(=) \\
\langle e_1, \xi_0, \phi, \rho_0 \rangle &\downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
\langle e_2, \xi_1, \phi, \rho_1 \rangle &\downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\
\hline
\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle &\downarrow \langle 0, \xi_2, \phi, \rho_2 \rangle \\
\end{align*}
\]

(ApplyEqFalse)

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.

\[
\begin{align*}
\phi(f) &= \text{PRIMITIVE}(\text{println}) \\
\langle e, \xi, \phi, \rho \rangle &\downarrow \langle v, \xi', \phi, \rho' \rangle \\
\langle \text{APPLY}(f, e), \xi, \phi, \rho \rangle &\downarrow \langle v, \xi', \phi, \rho' \rangle \quad \text{while printing } v \\
\end{align*}
\]

(ApplyPrintln)

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 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.

1.4.7 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 1.5.2. Extended definitions don’t have semantics, but in Exercises 26 and 27 on pages 79 and 80, 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 $\text{DefineGlobal}$ rule is almost identical to the $\text{GlobalAssign}$ rule, but the $\text{DefineGlobal}$ 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 $\text{user}(\langle x_1, \ldots, x_n \rangle, e)$, then binds the package into the function-definition environment $\phi$. The $\text{DefineFunction}$ rule enforces the invariant that the names of formal parameters are not duplicated.

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{define}(f, \langle x_1, \ldots, x_n \rangle, e), \xi, \phi \rangle \rightarrow (\xi, \phi\{ f \mapsto \text{user}(\langle x_1, \ldots, x_n \rangle, e) \})} \quad \text{(DefineFunction)}$$
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 $e$.

$$\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$$

(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 explained in the sidebar on page 14, I don’t want to use formal semantics to say exactly what use, check-expect, 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 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 greatly complicate our formal semantics while distracting us from what formal semantics is really meant to do: help us understand and compare programming languages.

To understand for yourself what the issues are, try designing some operational semantics for extended definitions. Exercise 26 on page 79 asks for relatively easy judgments about unit tests. Exercise 27 asks for a harder judgment: how do you formalize the idea that check-error succeeds? At minimum, such a formalization needs a proof system that says exactly what it means for evaluation to halt with an error.

1.5 The interpreter

The point of the bridge languages is 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.) If you’re going to write programs, you need an implementation, but why include the implementation in the chapter? Why not bury it in an appendix or off on a web site somewhere? Because the implementation of each interpreter has something to tell you about a language and its semantics, and the code can speak to you in a way that nothing else can. Moreover, if you really want to master your subject, you’ll want to try your own experiments in language design, which you can do by building on and changing my code. To make such experiments possible, I can’t just hand you the code; I have to explain it.

How and where should the code be explained? Some parts are more important than others, so I’ve organized the presentation to make it easy for you to study what you need, when you need it.

• What you need most is the code that is most relevant to the study of programming languages: the data structures for the crucial abstractions (abstract syntax, names, and environments), and the functions that evaluate expressions and definitions. All that appears in this chapter. And so you can see how the pieces are connected, the main function used to run the interpreter also appears here.
• The crucial functions and abstractions are supported by general-purpose infrastructure: error handling, printing, test reporting, and so on. Most of the infrastructure appears in Appendix A. But there’s one part I’ve split out: the parser that converts concrete syntax to abstract syntax appears in Appendix B. The parser appears on its own because it’s relatively elaborate, and it’s elaborate because I’ve made it easy for you to extend. Appendix B contains not only a general-purpose parsing infrastructure but also the parser for Impcore—if you want to understand the infrastructure, you should see it presented alongside an example of its use.

• The last and least interesting pieces of the interpreter are special-purpose modules that are used only to implement Impcore, but are not sufficiently relevant to programming language 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. This code appears in Appendix G.

The organization of the code is elaborated further in Table A.2 on page 1079. With this organization, you can study every line of code used to implement Impcore. There’s no magic; everything is built on top of the standard C library, which as libraries go, is simple and low-level. You can understand as much or as little as you need, as suits your purposes.

Next, 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→u.ifx and return the result 45a⟩. The label “45a” shows what page the definition is on. At the definition, the label appears in the left margin. When multiple definitions appear on the same page, each one gets a distinct lower-case letter, which is appended to the page number. Each definition is shown using an ≡ 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 +≡ sign in place of the ≡ sign. When a chunk’s definition is continued, Noweb includes pointers to the previous and next definitions, written “⊳42d” and “50a⊲”; these pointers appear at the right margin. The notation “(43a)” 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 50, on the bottom outside margin you’ll see an alphabetical list of identifiers: a mini-index. The mini-index tells you, for example, that function filedefs is declared in chunk 36c on page 36. It also tells you that functions fopen and fclose are not defined anywhere in the book; they are part of the initial basis of the C language. (In practice, they are defined by the include file <stdio.h>, but we refer to any built-in function as a basis function.) In any mini-index, B stands for a basis function, and A stands for an automatically generated function. 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 three large indices at the back of the book. The “concept index” on page 1433 provides the sort of index you might find in any textbook. The “author index” on page 1451 lists all the authors whose works are cited. And the “code Index” on page 1457 allows you to look up the definition—or in the case of 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 1.2. 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 Greek letters and math symbols from Section 1.1.2.)

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; 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.
Sidebar: 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, Java, Lua, ML, Perl, Python, Ruby, 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 very interesting systems-programming languages are safe by default, but have unsafe features that can be turned on explicitly at need, usually by a keyword `UNSAFE`. Among these languages, the best known may be Cedar and Modula-3.

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 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.
• In if and loop conditions, I don’t always use explicit inequality comparisons. In C, given a variable \( p \) of pointer type, it is correct and idiomatic to write simply \( p \) instead of \( p \neq \text{NULL} \), and I sometimes take advantage. However, for equality comparisons like \( p == \text{NULL} \) or \( n == 0 \), I avoid the abbreviations \(!p\) and \(!n\); at 3:00 AM, it is too easy to miss the exclamation mark.

On to the code! The presentation begins with the interfaces in Section 1.5.1. These interfaces include not only the interfaces associated with the evaluator, but also some interfaces associated with general-purpose utility code from Appendix A. 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 1.5.2, which starts on page 42.

1.5.1 Interfaces

C provides poor support for separating interfaces from implementations. The best a programmer can do is put each interface in a .h file and use the C preprocessor to #include those .h files where they are needed. Ensuring that the right files are #include’d, that they are #include’d in the right order, and that no file is #include’d more than once are all up to the programmer; the C language and preprocessor don’t help. These problems are common, and C programmers have developed conventions to deal with them, but these conventions are better suited to large software projects than to small interpreters. I have therefore chosen simply to put all the interfaces into one header file, all.h. When Noweb extracts code from the book, it automatically puts #include "all.h" at the beginning of each C file.

File all.h, which includes all interfaces used in the interpreter, is split into six parts:

• Imports of header files from the standard C library

• Type definitions

• Structure definitions

• Function prototypes

• Arcana used in lexical analysis and parsing

Putting types, structures, and functions in that order makes it easy for functions or structures declared in one interface to use types defined in another. And because declarations and definitions of types always precede the function prototypes that use those types, we need not worry about getting things in the right order.
1.5. THE INTERPRETER

To make it possible to reuse the general-purpose interfaces in later interpreters, I also distinguish between shared and unshared definitions; a definition is “shared” if it is used in another interpreter later in the book.

```c
(all.h for Impcore 31)
#include <assert.h>
#include <ctype.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

(type definitions for Impcore 33c)
(shared type definitions 34a)

(structure definitions for Impcore 38)
(shared structure definitions 1081a)

(function prototypes for Impcore 32b)
(shared function prototypes 34b)

(macro definitions used in parsing 1106a)
(declarations of global variables used in lexical analysis and parsing 1113b)
```

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, calling the union `u` and the tag `alt` (for “alternative”). I name each alternative, and I use each name in two places: in an enumeration type, which lists the possible values of `alt`, and in a union type, which lists the possible members of the union. And in keeping with common C practice, when I use a name as an enumeration literal, I write it using all capital letters, and when I use the same name as a member of a union, I write the lowercase equivalent.
My representation of sum types is safe and systematic, but it requires notation that can get unwieldy. For example, to refer to the name of a function in a definition \( d \), I must write \( d->\text{u.define.name} \). Although something like \( d->\text{name} \) might be more convenient, referring to \( \text{u.define} \) enables the compiler to keep me from making mistakes; in particular, I can’t accidentally grab a \( \text{name} \) 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 \( \text{Def} \), a simplified version of which appears in chunk (simplified example of abstract syntax for Impcore 16a). A \( \text{Def} \) is a sum type with three alternatives, called \( \text{VAL}, \text{EXP}, \) and \( \text{DEFINE} \).

To make these structures easy to create, I define a creator function for each alternative in the sum, as well as for \( \text{Userfun} \).

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 C shows an ML program that generates such code from the following descriptions.

A valid \( \text{Userfun} \) satisfies the invariant that the names in \( \text{formals} \) are all distinct.
Here is a description of the abstract syntax for Exp, which you might wish to compare with the concrete syntax given for exp on page 6:

\[(exp.t\ 33a)\equiv\]
\[
\begin{align*}
\text{Exp}\ast & = \text{LITERAL} \ (\text{Value}) \\
& | \text{VAR} \ (\text{Name}) \\
& | \text{SET} \ (\text{Name name, Exp exp}) \\
& | \text{IFX} \ (\text{Exp cond, Exp truex, Exp falsex}) \\
& | \text{WHILEX} \ (\text{Exp cond, Exp exp}) \\
& | \text{BEGIN} \ (\text{Explist}) \\
& | \text{APPLY} \ (\text{Name name, Explist actuals})
\end{align*}
\]

The descriptions above are slightly elaborated versions of \((\text{simplified example of abstract syntax for Impcore}\ 16a)\). 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 build a strong cognitive connection between operational semantics and code. But as discussed in the sidebar on page 14, Impcore also has extended definitions, which include unit tests. If you like, you can just use extended definitions and not worry about how they are implemented. But if you want to understand their implementations, you'll need to start with these descriptions of how extended definitions and unit tests are represented:

\[(xdef.t\ 33b)\equiv\]
\[
\begin{align*}
\text{XDef}\ast & = \text{DEF} \ (\text{Def}) \\
& | \text{USE} \ (\text{Name}) \\
& | \text{TEST} \ (\text{UnitTest}) \\
\text{UnitTest}\ast & = \text{CHECK\_EXPECT} \ (\text{Exp check, Exp expect}) \\
& | \text{CHECK\_ERROR} \ (\text{Exp})
\end{align*}
\]

To remember all the unit tests in a file, I use a list.

\[(type\ \text{definitions\ for\ Impcore}\ 33c)\equiv\]
\[
\text{typedef \\ struct\ UnitTestlist \ *UnitTestlist; \ //\ list\ of\ UnitTest}
\]

A UnitTestList is list of pointers of type UnitTest. I use this naming convention in all my C code. List types are manifest, and their definitions are in the lists interface in chunk 38. I also define a type for lists of Exps.

\[(type\ \text{definitions\ for\ Impcore}\ 33c)\equiv\]
\[
\text{typedef \\ struct\ Explist \ *Explist; \ //\ list\ of\ Exp}
\]

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.

---

6Why 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->u.literal. And I can't name a structure field u.if or u.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.
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.

(\textit{shared type definitions} 34a)\equiv
\begin{align*}
\text{typedef struct Name *Name;} \\
\text{typedef struct Namelist *Namelist;} & \quad /\!\!/\text{list of \textit{Name}}
\end{align*}

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

(\textit{shared function prototypes} 34b)\equiv
\begin{align*}
\text{Name strtoname(const char *s);} \\
\text{const char *nametostr(Name x)};
\end{align*}

These functions satisfy the following algebraic laws:

\begin{align*}
\text{strcmp}(s, \text{nametostr}\left(\text{strtoname}(s)\right)) & = 0 \\
\text{strcmp}(s, t) & = 0 \text{ if and only if } \text{strtoname}(s) = \text{strtoname}(t)
\end{align*}

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

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

The \textit{Name} type is abstract; the interface does not give the members of \textit{struct Name}. A client should create new values of type \textit{Name} only by calling \text{strtoname}; to do so by casting other pointers is a violation of the type system and an unchecked run-time error.

\textbf{Interface to values}

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

(\textit{type definitions for Impcore} 33c)\equiv
\begin{align*}
\text{typedef int Value;} \\
\text{typedef struct Valuelist *Valuelist;} & \quad /\!\!/\text{list of Value}
\end{align*}

\textbf{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 $\text{USERDEF}(\langle x_1, \ldots, x_n \rangle, e)$, the interpreter represents a user-defined function as a pair containing formals and body.

The interpreter represents each primitive by its name.

(\textit{type definitions for Impcore} 33c)\equiv
\begin{align*}
\text{typedef struct Funlist *Funlist;} & \quad /\!\!/\text{list of Fun}
\end{align*}

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

(\textit{type and structure definitions for Impcore} 32a)\equiv
\begin{align*}
\text{typedef struct Fun Fun;} \\
\text{typedef enum \{ USERDEF, PRIMITIVE \} Funalt;} \\
\text{struct Fun \{ Funalt alt; union \{ Userfun userdef; Name primitive; \} u; \};}
\end{align*}
Also generated automatically are these prototypes for creator functions.

\[
\begin{align*}
\text{(function prototypes for Impcore 32b)} & \equiv \\
\text{Fun } & \text{mkUserdef(Userfun } userdef); \\
\text{Fun } & \text{mkPrimitive(Name } primitive); \\
\end{align*}
\] (31) <32b 35c>

### 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 \textbf{Value} and another for environments that hold a \textbf{Fun}, 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 \textbf{void*} pointer, define a single version of each function, and use type casting to convert a \textbf{void*} to a \textbf{Value*} or \textbf{Fun*} as needed. This choice duplicates no code, but it is unsafe; if we accidentally put a \textbf{Value*} in an environment intended to hold a \textbf{Fun*}, 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 \textit{polymorphism} to achieve type safety without duplicating code.

\[
\begin{align*}
\text{(type definitions for Impcore 33c)} & \equiv \\
\text{typedef struct Valenv } *\text{Valenv;} \\
\text{typedef struct Funenv } *\text{Funenv;}
\end{align*}
\] (31) <34d>

A new environment may be created by passing a list of names and a list of associated values or function definitions. For example, \texttt{mkValenv}(\langle \texttt{x}_1, \ldots, \texttt{x}_n \rangle, \langle \texttt{v}_1, \ldots, \texttt{v}_n \rangle) returns the environment \(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\). If the two lists are not the same length, it is a checked run-time error.

\[
\begin{align*}
\text{(function prototypes for Impcore 32b)} & \equiv \\
\text{Valenv } & \text{mkValenv(Namelist vars, Valuelist vals);} \\
\text{Funenv } & \text{mkFunenv(Namelist vars, Funlist defs);} \\
\end{align*}
\] (31) <35a 35d>

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

\[
\begin{align*}
\text{(function prototypes for Impcore 32b)} & \equiv \\
\text{Value } & \text{fetchval(Name name, Valenv env);} \\
\text{Fun } & \text{fetchfun(Name name, Funenv env);} \\
\end{align*}
\] (31) <35c 35e>

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

\[
\begin{align*}
\text{(function prototypes for Impcore 32b)} & \equiv \\
\text{bool } & \text{isvalbound(Name name, Valenv env);} \\
\text{bool } & \text{isfunbound(Name name, Funenv env);} \\
\end{align*}
\] (31) <35d 35f>

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

\[
\begin{align*}
\text{(function prototypes for Impcore 32b)} & \equiv \\
\text{void } & \text{bindval(Name name, Value val, Valenv env);} \\
\text{void } & \text{bindfun(Name name, Fun fun, Funenv env);} \\
\end{align*}
\] (31) <35e 36e>
These functions can be used to replace existing bindings or to add new ones.

**Interface to infrastructure: Streams of definitions**

The details of reading characters and converting them to abstract syntax are interesting, but they are more relevant to study of compiler construction than to study of programming languages. From the programming-language point of view, all we need to know is that we have a source of extended definitions. The details are relegated to Appendix A.

A source of extended definitions is called an XDefstream. To obtain the next definition from such a source, call `getxdef`. Function `getxdef` returns either a pointer to the next definition or, if the source is exhausted, the NULL pointer. And if there is some problem converting input to abstract syntax, `getxdef` may call `synerror` (page 41).

To create a stream of definitions, we need a source of lines. That source can be a string compiled into the program, or an external file. So that error messages can refer to the source, we need to give its name. And if the source is a file, we need to say whether to prompt for input. (Reading from an internal string never prompts.)

The evaluator's interface is not very interesting: it is the implementation, which starts on page 42, 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 three functions: `eval`, `evaldef`, and `readevalprint`. These functions evaluate expressions, true definitions, and extended definitions respectively.

Function `eval` corresponds to the ⇓ relation in our operational semantics. For example, `eval(e, ξ, φ, ρ)` finds a v, ξ′, and ρ′ such that `(e, ξ, φ, ρ) ⇓ (v, ξ′, φ, ρ′)`, assigns ρ := ρ′ and ξ := ξ′, and returns v. The function `evaldef` similarly corresponds to the → relation.

If the `echo_level` parameter to `evaldef` 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.
1.5. THE INTERPRETER

Function \texttt{readevalprint} consumes a stream of extended definitions. It evaluates each true definition, remembers each unit test, and calls itself recursively on each use. When the stream of extended definitions is exhausted, \texttt{readevalprint} runs the remembered unit tests.

\begin{verbatim}
37a (function prototypes for Impcore 32b)+≡
   void readevalprint(XDefstream s, Valenv globals, Funenv functions, Echo echo_level);
As with \texttt{evaldef}, the \texttt{echo_level} parameter controls whether \texttt{readevalprint} prints the values and names of top-level expressions and functions.
37b (shared type definitions 34a)+≡
   typedef enum Echo { NO_ECHOES, ECHOES } Echo;
\end{verbatim}
CHAPTER 1. AN IMPERATIVE CORE

Interface to lists

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.7

Lists are recursive data types. Either a list is empty, or it 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. We use the null pointer to represent the empty list.

(structure definitions for Impcore 38)≡

\begin{verbatim}
struct Parlist {
    Par hd;
    struct Parlist *tl;
};

struct UnitTestlist {
    UnitTest hd;
    struct UnitTestlist *tl;
};

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

struct Namelist {
    Name hd;
    struct Namelist *tl;
};

struct Valuelist {
    Value hd;
    struct Valuelist *tl;
};

struct Funlist {
    Fun hd;
    struct Funlist *tl;
};
\end{verbatim}

7Programmers who use C++ templates have made the same choice. Early implementations of templates were notorious for code bloat, not only because they had a separate implementation for each type, but also because they might have a dozen duplicate implementations of List<Exp> alone, one for every module in which List<Exp> is used.
These definitions are generated by a Lua script, which searches header files for lines of the form

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

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(1, n)` when `n ≥ lengthNL(1)` 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:

```plaintext
(int function prototypes for Impcore 32b)≡ ⊢ (31) <37a 50b>

int lengthPL(Parlist ps);
Par nthPL (Parlist ps, unsigned n);
Parlist mkPL (Par p, Parlist ps);
Parlist popPL (Parlist ps);
Printer printparlist;

int lengthUL(UnitTestlist us);
UnitTest nthUL (UnitTestlist us, unsigned n);
UnitTestlist mkUL (UnitTest u, UnitTestlist us);
UnitTestlist popUL (UnitTestlist us);
Printer printunittestlist;

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

int lengthNL(Namelist ns);
Name nthNL (Namelist ns, unsigned n);
Namelist mkNL (Name n, Namelist ns);
Namelist popNL (Namelist ns);
Printer printnamelist;

int lengthVL(Valuelist vs);
Value nthVL (Valuelist vs, unsigned n);
Valuelist mkVL (Value v, Valuelist vs);
Valuelist popVL (Valuelist vs);
Printer printvaluelist;

int lengthFL(Funlist fs);
Fun nthFL (Funlist fs, unsigned n);
Funlist mkFL (Fun f, Funlist fs);
Funlist popFL (Funlist fs);
Printer printfunlist;
```

Generating all this code automatically makes the replication 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 \texttt{printf}, but \texttt{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 \texttt{print} and \texttt{fprint}, which support direct printing of \texttt{Name}s, \texttt{Exp}s, and so on.

\begin{quote}
\begin{verbatim}
void print (const char *fmt, ...); // print to standard output
void fprint(FILE *output, const char *fmt, ...); // print to given file
\end{verbatim}
\end{quote}

Functions \texttt{print} and \texttt{fprint} use interfaces that are very similar to \texttt{printf} and \texttt{fprintf}: the \texttt{fmt} parameter is a “format string” that contains “conversion specifications” marked with percent signs. The conversion specifications resemble those used by \texttt{printf}, but to keep the implementation simple, they are much simpler. Our conversion specifications are two characters: the first is always \%, and the second is a character like \texttt{d} or \texttt{s}, which is called a \textit{conversion specifier}. Unlike conversion specifications in standard C, ours may not contain minus signs, numbers, periods, etc. The Impcore interpreter uses the conversion specifications shown here in Table 1.3.  

By convention, I use lowercase letters to print individual values and uppercase letters to print lists. To print a \texttt{Def}, I cannot use \%d, because \%d is too firmly established as a specification for printing print decimal integers. Instead I use \%t, for “top-level,” which is where a \texttt{Def} appears.

\begin{table}[h]
\begin{tabular}{ll}
\%d & Print an integer in decimal format \\
\%e & Print an \texttt{Exp} \\
\%f & Print a \texttt{Fun} \\
\%E & Print an \texttt{Explist} (list of \texttt{Exp}) \\
\%n & Print a \texttt{Name} \\
\%N & Print a \texttt{Namelist} (list of \texttt{Name}) \\
\%p & Print a \texttt{Par} (see Appendix G) \\
\%P & Print a \texttt{Parlist} (list of \texttt{Par}) \\
\%s & Print a \texttt{char*} (string) \\
\%t & Print a \texttt{Def} (t stands for “top-level”, which is where definitions appear) \\
\%v & Print a \texttt{Value} \\
\%V & Print a \texttt{Valuelist} (list of \texttt{Value}) \\
\end{tabular}
\caption{Conversion specifiers for impcore}
\end{table}

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

The implementations of \texttt{print} and \texttt{fprint} are \textit{extensible}; adding a new conversion specification is as simple as calling \texttt{installprinter}:

\begin{quote}
\begin{verbatim}
void installprinter(unsigned char c, Printer *take_and_print);
\end{verbatim}
\end{quote}

The conversion specifications listed above are installed when the interpreter launches, by code chunk (\textit{install conversion specifications for print and fprint} 1198c). The details, including the definition of \texttt{Printer}, are in Sections A.2 and G.2.

\footnote{The conversion specification is the whole string, beginning with the percent sign. The conversion specifier is the single character (a letter or a percent sign) that terminates the conversion specification.}
1.5. THE INTERPRETER

Interface to infrastructure: Error handling

When given a faulty Impcore program, the interpreter can’t just quit; it has to recover. To signal or recover from errors, the interpreter uses the error-handling interface.

Errors may be signaled from anywhere; an error is signaled by a call to `synerror` or `runerror`. There are two error functions because the interpreter distinguishes two kinds of errors:

- A syntax error, signaled by calling `synerror`, is one that occurs as the interpreter is translating input into an abstract-syntax tree.
- A run-time error, signaled by calling `runerror`, is one that occurs as the interpreter is evaluating an abstract-syntax tree—that is, as the interpreter is running Impcore code.

The idea that different things happen at different times is ubiquitous in programming languages, but it’s often overlooked. People who are educated in programming languages learn to distinguish phases of execution. The Impcore interpreter uses two phases: parsing, which translates concrete syntax to abstract syntax, and evaluation, which evaluates abstract syntax. In Chapter 6 I introduce a third phase, type checking, which happens after syntax is built but before evaluation.

Whether an error is signaled at parse time or run time, an error-signaling function doesn’t just say that an error occurred; it also produces an error message. For that reason, the error-signaling functions have interfaces that resemble `print`. But because different information is available at parse time and at run time, the interfaces are not identical.

The simpler of the two error-signaling functions is `runerror`. In its normal mode of operation `runerror` prints to standard error and then `longjumps` to `errorjmp`.

During unit testing, `runerror` operates in testing mode, and it behaves a little differently. The details are in Section A.3.1 on page 1095.

The `synerror` function 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. The `Sourceloc` values are taken care of by the parsing infrastructure described in Appendix B, which is the place from which `synerror` is called.

The possibility of printing source-code locations complicates the interface. When the interpreter is reading code interactively, printing source-code locations is silly—if there’s a syntax error, it’s in what you just typed. But if the interpreter is reading code from a file, it’s a different story—it’s useful to have the file’s name and the number of the line containing the bad syntax. But the error module doesn’t know where the interpreter is reading code from—only the `main` function in chunk 52b knows that. So the error module has to be told how syntax errors should be formatted: with locations or without.
42a (shared function prototypes 34b)+≡
void set_toplevel_error_format(ErrorFormat format);

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.

Interface to infrastructure: Helper functions for common errors

To help the interpreter detect errors, I define a couple of functions that are used in evaluating function calls and function definitions. Function checkargc is used to check the number of arguments to both user-defined and primitive functions. The first argument is an abstract-syntax tree representing the application being checked; if expected ≠ actual, checkargc calls runerror, passing a message that contains e.

42b (shared function prototypes 34b)+≡
void checkargc(Exp e, int expected, int actual);

The duplicatename function finds a duplicate name on a Namelist if such a name exists. It is used to check that formal parameters to user-defined functions all have different names. If the name list names contains a duplicate occurrence of any name, the function returns such a name; otherwise it returns NULL.

1.5.2 Implementation of the evaluator

The most interesting parts of the implementation are the evaluator (functions eval and evaldef) and the readevalprint loop.

Evaluating expressions

The function eval implements the ⇓ relation from the operational semantics. Calling eval(e, ξ, φ, ρ) finds a v, ξ′, and ρ′ such that (e, ξ, φ, ρ) ⇓ (v, ξ′, φ, ρ′), assigns ρ := ρ′ and ξ := ξ′, and returns v. Because it is unusual to use Greek letters in C code, I have chosen these names instead:

ξ globals
φ functions
ρ formals

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

42c (shared function prototypes 34b)+≡
Name duplicatename(Namelist names);

1.5.2 Implementation of the evaluator

The most interesting parts of the implementation are the evaluator (functions eval and evaldef) and the readevalprint loop.

Evaluating expressions

The function eval implements the ⇓ relation from the operational semantics. Calling eval(e, ξ, φ, ρ) finds a v, ξ′, and ρ′ such that (e, ξ, φ, ρ) ⇓ (v, ξ′, φ, ρ′), assigns ρ := ρ′ and ξ := ξ′, and returns v. Because it is unusual to use Greek letters in C code, I have chosen these names instead:

ξ globals
φ functions
ρ formals

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

42c (shared function prototypes 34b)+≡
void set_toplevel_error_format(ErrorFormat format);

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.

Interface to infrastructure: Helper functions for common errors

To help the interpreter detect errors, I define a couple of functions that are used in evaluating function calls and function definitions. Function checkargc is used to check the number of arguments to both user-defined and primitive functions. The first argument is an abstract-syntax tree representing the application being checked; if expected ≠ actual, checkargc calls runerror, passing a message that contains e.

42b (shared function prototypes 34b)+≡
void checkargc(Exp e, int expected, int actual);

The duplicatename function finds a duplicate name on a Namelist if such a name exists. It is used to check that formal parameters to user-defined functions all have different names. If the name list names contains a duplicate occurrence of any name, the function returns such a name; otherwise it returns NULL.

1.5.2 Implementation of the evaluator

The most interesting parts of the implementation are the evaluator (functions eval and evaldef) and the readevalprint loop.

Evaluating expressions

The function eval implements the ⇓ relation from the operational semantics. Calling eval(e, ξ, φ, ρ) finds a v, ξ′, and ρ′ such that (e, ξ, φ, ρ) ⇓ (v, ξ′, φ, ρ′), assigns ρ := ρ′ and ξ := ξ′, and returns v. Because it is unusual to use Greek letters in C code, I have chosen these names instead:

ξ globals
φ functions
ρ formals

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

42c (shared function prototypes 34b)+≡
void checkargc(Exp e, int expected, int actual);

The duplicatename function finds a duplicate name on a Namelist if such a name exists. It is used to check that formal parameters to user-defined functions all have different names. If the name list names contains a duplicate occurrence of any name, the function returns such a name; otherwise it returns NULL.

1.5.2 Implementation of the evaluator

The most interesting parts of the implementation are the evaluator (functions eval and evaldef) and the readevalprint loop.

Evaluating expressions

The function eval implements the ⇓ relation from the operational semantics. Calling eval(e, ξ, φ, ρ) finds a v, ξ′, and ρ′ such that (e, ξ, φ, ρ) ⇓ (v, ξ′, φ, ρ′), assigns ρ := ρ′ and ξ := ξ′, and returns v. Because it is unusual to use Greek letters in C code, I have chosen these names instead:

ξ globals
φ functions
ρ formals

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

42c (shared function prototypes 34b)+≡
Name duplicatename(Namelist names);
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.

\[
\text{(eval.e \text{ 42d})} + \equiv
\]

\[
\text{Value eval(Exp e, Valenv globals, Funenv functions, Valenv formals) } \{
\text{switch (e->alt) } \{
\text{case LITERAL: } \langle \text{evaluate e->u.literal and return the result 43b} \rangle
\text{case VAR: } \langle \text{evaluate e->u.var and return the result 44a} \rangle
\text{case SET: } \langle \text{evaluate e->u.set and return the result 44b} \rangle
\text{case IFX: } \langle \text{evaluate e->u.ifx and return the result 45a} \rangle
\text{case WHILEX: } \langle \text{evaluate e->u.whilex and return the result 45b} \rangle
\text{case BEGIN: } \langle \text{evaluate e->u.begin and return the result 46a} \rangle
\text{case APPLY: } \langle \text{evaluate e->u.apply and return the result 46b} \rangle
\}
\text{assert(0);}\}
\]

The last two lines of eval might seem superfluous, but it isn’t; it helps protect me, and you, from mistakes. Calling assert(0) ensures that 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.

\[
\langle \text{LITERAL(v), } \xi, \phi, \rho \rangle \downarrow \langle v, \xi, \phi, \rho \rangle
\]

The implementation simply returns the literal value.

\[
\text{(evaluate e->u.literal and return the result 43b)} \equiv
\]

\[
\text{return e->u.literal};
\]
Two rules have variables in their conclusions.

\[
\begin{align*}
&x \in \text{dom } \rho \\
&\frac{(\text{VAR}(x), \xi, \phi, \rho) \Downarrow (\rho(x), \xi, \phi, \rho)}{(\text{FormalVar})} \\
&x \notin \text{dom } \rho \quad x \in \text{dom } \xi \\
&\frac{(\text{VAR}(x), \xi, \phi, \rho) \Downarrow (\xi(x), \xi, \phi, \rho)}{(\text{GlobalVar})}
\end{align*}
\]

We know which rule to use by checking \( x \in \text{dom } \rho \), which in C is \text{isvalbound}(e->u.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.

The call to \text{runerror} illustrates the convenience of the extensible printer; I use \texttt{%n} to print a \texttt{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).

\[
\begin{align*}
&x \in \text{dom } \rho \\
&\frac{(\text{set}(x, e), \xi, \phi, \rho) \Downarrow (v, \xi', \phi, \rho')}{(\text{FormalAssign})} \\
&x \notin \text{dom } \rho \quad x \in \text{dom } \xi \\
&\frac{(\text{set}(x, e), \xi, \phi, \rho) \Downarrow (v, \xi', \phi, \rho')}{(\text{GlobalAssign})}
\end{align*}
\]

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

The call to \text{runerror} illustrates the convenience of the extensible printer; I use \texttt{%n} to print a \texttt{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).
To evaluate ifx, we again have two rules.

\[
\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \quad v_1 \neq 0 \\
\langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle
\]

(IfTRUE)

\[
\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle \\
\langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle
\]

(IfFALSE)

Both rules have the same first premise: \(\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle\). We can find \(v_1\), \(\xi'\), and \(\rho'\) by making the recursive call \(\text{eval}(e->u.ifx.cond, globals, functions, formals)\).

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 \(\text{eval}\). The expression \(e_2\) is \(e->u.ifx.truex\); \(e_3\) is \(e->u.ifx.falsex\).

A systematic translation of \(\langle \text{while}(e_1, e_2), \xi', \phi, \rho' \rangle \Downarrow \langle v, \xi'', \phi, \rho'' \rangle\), where it appears as a premise of the first rule, would produce a recursive call to \(\text{eval}(e..., ... - 1)\). 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.
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.

\[
\langle \text{BEGIN}(\epsilon), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi, \phi, \rho \rangle
\]

\[
\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle
\]

\[
\langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle
\]

\[\vdots\]

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

\[
\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle
\]

There are many rules for applying functions, but I divide the 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->u.apply.name`.

\[
\langle \text{evaluate } e->u.apply \text{ and return the result } \rangle \equiv
\]

\[
\{ \text{Fun } f; \langle \text{make } f \text{ the function denoted by } e->u.apply.name, \text{ or call runerror } \rangle
\]

\[
\langle \text{switch } (f.alt) \{ \text{case USERDEF:} \langle \text{apply } f.u.userdef \text{ and return the result } \rangle
\]

\[
\langle \text{case PRIMITIVE:} \langle \text{apply } f.u.primitive \text{ and return the result } \rangle
\]

\[
\langle \text{default:} \langle \text{assert(0); } \rangle
\]

\[
\langle \text{runerror: } \langle \text{call to undefined function } \%n \text{ in } \%e, e->u.apply.name, e; \rangle
\]

\[
\langle f = \text{fetchfun}(e->u.apply.name, functions); \rangle
\]
Applying a user-defined function has something in common with `begin`, because arguments $e_1, \ldots, e_n$ have to be evaluated. The difference is that `begin` keeps only result $v_n$ (in variable $v$ in chunk 46a), where function evaluation keeps all the result values, to bind into a new environment.

\[
\begin{align*}
\phi(f) &= \text{USER}((x_1, \ldots, x_n), e) \\
&\text{for } x_1, \ldots, x_n \text{ all distinct} \\
\langle e_1, \xi_0, \phi, \rho_0 \rangle &\Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
\vdots \\
\langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle &\Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\
\langle \text{APPLY}(f, e_1, \ldots, e_n), \xi_0, \phi, \rho_0 \rangle &\Downarrow \langle v, \xi', \phi, \rho_n \rangle
\end{align*}
\]

(APPLYUser)

To produce values $v_1, \ldots, v_n$, I define the auxiliary function `evalallist`, which is given $e_1, \ldots, e_n$ along with $\xi_0$, $\phi$, and $\rho_0$. It evaluates $e_1, \ldots, e_n$ in order, and it mutates the environments so that when it is finished, $\xi = \xi_n$ and $\rho = \rho_n$. Finally, `evalallist` returns the list $v_1, \ldots, v_n$.

\[
\begin{align*}
\text{static Valuelist evalallist} \equiv \\
\text{static Valuelist evallist(Explist es, Valenv globals, Funenv functions, Valenv formals)} \{} \\
\text{if (es == NULL)} \{} \\
\quad \text{return NULL;} \\
\text{else} \{} \\
\quad \text{Value } v = \text{eval(es->hd, globals, functions, formals);} \\
\quad \text{return mkVL(v, evalallist(es->tl, globals, functions, formals))} \\
\}\end{align*}
\]

The rules of Impcore require that `es->hd` be evaluated before `es->tl`. To ensure the correct order of evaluation, we must call `eval(es->hd, ...)` and `evalallist(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 formal parameters to $f$ be the same length as the list of actual parameters in the call. I let $xs$ represent the formals $x_1, \ldots, x_n$ and $vs$ represent the actuals $v_1, \ldots, v_m$. If the formals and actuals are the same length, so $m = n$, I use `mkValenv(xs, vs)` to create a fresh environment \{ $x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$ \} in which to evaluate the body.

\[
\begin{align*}
\text{(apply } f.u.userdef \text{ and return the result 47b)} \equiv \\
\{} \\
\text{Namelist xs = f.u.userdef.formals;} \\
\text{Valuelist vs = evalallist(e->u.apply.actuals, globals, functions, formals);} \\
\text{checkargc(e, lengthNL(xs), lengthVL(vs));} \\
\text{return eval(f.u.userdef.body, globals, functions, mkValenv(xs, vs))} \\
\}\end{align*}
\]
Impcore has few primitive operators, and they are simple. We handle \texttt{print} separately from the arithmetic primitives. More general techniques for implementing primitives, which are appropriate for larger languages, are shown as part of the implementation of $\mu$Scheme in Section 2.14.1 on page 153.

\begin{align*}
\text{(apply f.u.primitive and return the result 48a)} & \equiv \\
\{ & \text{Valuelist } vs = \text{evallist(e->u.apply.actuals, globals, functions, formals)}; \\
& \text{if (f.u.primitive == strtoname("print"))} \\
& \quad \langle \text{apply Impcore primitive print to } vs \text{ and return 48b} \rangle \\
& \text{else if (f.u.primitive == strtoname("println"))} \\
& \quad \langle \text{apply Impcore primitive println to } vs \text{ and return 48c} \rangle \\
& \text{else if (f.u.primitive == strtoname("printu"))} \\
& \quad \langle \text{apply Impcore primitive printu to } vs \text{ and return 48d} \rangle \\
& \quad \text{else} \\
& \quad \langle \text{apply arithmetic primitive to } vs \text{ and return 49a} \rangle \\
\}
\end{align*}

\begin{align*}
\text{(apply Impcore primitive print to } vs \text{ and return 48b)} & \equiv \\
\{ & \text{checkargc(e, 1, lengthVL(vs));} \\
& \quad \text{Value } v = \text{nthVL(vs, 0);} \\
& \quad \text{print("\%v", } v); \\
& \quad \text{return } v;
\}
\end{align*}

\begin{align*}
\text{(apply Impcore primitive println to } vs \text{ and return 48c)} & \equiv \\
\{ & \text{checkargc(e, 1, lengthVL(vs));} \\
& \quad \text{Value } v = \text{nthVL(vs, 0);} \\
& \quad \text{print("\%v\n", } v); \\
& \quad \text{return } v;
\}
\end{align*}

\begin{align*}
\text{(apply Impcore primitive printu to } vs \text{ and return 48d)} & \equiv \\
\{ & \text{checkargc(e, 1, lengthVL(vs));} \\
& \quad \text{Value } v = \text{nthVL(vs, 0);} \\
& \quad \text{print_utf8(} v); \\
& \quad \text{return } v;
\}
\end{align*}
Because the name of each arithmetic primitive has just one character, I switch on that character. This technique is reasonable only because the set of primitives is small and fixed.

\[(\text{apply arithmetic primitive to } \mathbf{vs} \text{ and return})\]

\[
\begin{array}{l}
\text{const char } *s = \text{nametostr}(f.u.primitive); \\
\text{Value } v, w; \\
\text{(check that } \mathbf{vs} \text{ has exactly two values, and assign them to } v \text{ and } w) \\
\text{assert} (\text{strlen}(s) == 1); \\
\text{switch } (s[0]) { \\
\text{case } '<': \\
\text{\hspace{1em}} \text{return } v < w; \\
\text{case } '>': \\
\text{\hspace{1em}} \text{return } v > w; \\
\text{case } '=': \\
\text{\hspace{1em}} \text{return } v == w; \\
\text{case } '+': \\
\text{\hspace{1em}} \text{return } v + w; \\
\text{case } '-': \\
\text{\hspace{1em}} \text{return } v - w; \\
\text{case } '*': \\
\text{\hspace{1em}} \text{return } v * w; \\
\text{case } '/': \\
\text{\hspace{1em}} \text{if } (w == 0) \\
\text{\hspace{2em}} \text{runerror("division by zero in } \%e\text{", } e); \\
\text{\hspace{2em}} \text{return } v / w; \\
\text{default:} \\
\text{\hspace{1em}} \text{assert}(0); \\
\text{}} \\
\end{array}
\]

The code also depends on the fact that Impcore shares C’s rules for the values of comparison expressions.

\[(\text{check that } \mathbf{vs} \text{ has exactly two values, and assign them to } v \text{ and } w)\]

\[
\begin{array}{l}
\text{checkargc}(e, 2, \text{lengthVL} (\mathbf{vs})); \\
\text{v} = \text{nthVL}(\mathbf{vs}, 0); \\
\text{w} = \text{nthVL}(\mathbf{vs}, 1); \\
\end{array}
\]

### Evaluating definitions

As noted on page 14, there are two types of definitions: 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`. As shown on page 33, the `XDef` type includes both ordinary and extended definitions, and an `XDefstream` provides a stream of `XDefs`, usually from a file or from a user’s input.
Responsibility for evaluating definitions is shared between two functions. Function \texttt{readevalprint} takes as input a stream of definitions. The extended definitions are handled directly in \texttt{readevalprint}:

- Each unit test is remembered and later run.
- A file mentioned in \texttt{use} is converted to a stream of extended definitions, then passed recursively to \texttt{readevalprint}.

The true definitions are passed on to \texttt{evaldef}.

```c
void readevalprint(XDefstream xdefs, Valenv globals, Funenv functions, Echo echo) {
    UnitTestlist pending_unit_tests = NULL; // to be run when xdefs is exhausted
    for (XDef d = getxdef(xdefs); d; d = getxdef(xdefs))
        switch (d->alt) {
            case TEST:
                pending_unit_tests = mkUL(d->u.test, pending_unit_tests);
                break;
            case USE:
                ⟨evaluate d->u.use, possibly mutating globals and functions⟩
                break;
            case DEF:
                evaldef(d->u.def, globals, functions, echo);
                break;
            default:
                assert(0);
        }
    process_tests(pending_unit_tests, globals, functions);
}
```

Function \texttt{process_tests}, defined in Section G.1 on page 1195, runs the \texttt{pending_unit_tests} in the order in which they appear in the source code.

```c
void process_tests(UnitTestlist tests, Valenv globals, Funenv functions);
```

On seeing \texttt{use}, we open the file named by \texttt{use}, build a stream of definitions, and through \texttt{readevalprint}, recursively call \texttt{evaldef} on all the definitions in that file. When reading definitions via \texttt{use}, the interpreter neither prompts nor echoes.

```c
const char *filename = nametostr(d->u.use);
FILE *fin = fopen(filename, "r");
if (fin == NULL)
    runerror("cannot open file " filename "", filename);
readevalprint(filexdefs(filename, fin, NO_PROMPTS), globals, functions, echo);
fclose(fin);
```

As noted in Exercise 35, this code can leak open file descriptors.
### 1.5. THE INTERPRETER

The function `evaldef` implements the $\rightarrow$ relation on the true definitions, from the operational semantics. That is, calling `evaldef(d, ξ, φ, echo)` finds a $\xi'$ and $\phi'$ such that $(d, ξ, φ) \rightarrow (\xi', \phi')$, 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 $d$.

```c
(expr 42d) +=
void evaldef(Def d, Valenv globals, Funenv functions, Echo echo) {
  switch (d->alt) {
    case VAL:
      ⟨evaluate d->u.val, mutating globals 51b⟩
      return;
    case EXP:
      ⟨evaluate d->u.exp and possibly print the result 51c⟩
      return;
    case DEFINE:
      ⟨evaluate d->u.define, mutating functions 52a⟩
      return;
  }
  assert(0);
}
```

The operational semantics dictates the cases for `VAL`, `EXP`, and `DEFINE`. A variable definition updates $\xi$.

$$\langle e, ξ, φ, \{ \} \rangle \Downarrow \langle v, ξ', φ, ρ' \rangle$$  \hspace{1cm} (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 $ρ$. 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 $v$.

```c
51b  \hspace{1cm} (evaluate d->u.val, mutating globals 51b)≡ \hspace{1cm} \langle 51a \rangle
\{
  Value v = eval(d->u.val.exp, globals, functions, mkValenv(NULL, NULL));
  bindval(d->u.val.name, v, globals);
  if (echo == ECHOES)
    print("%v\n", v);
\}
```

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

$$\langle e, ξ, φ, \{ \} \rangle \Downarrow \langle v, ξ', φ, ρ' \rangle$$  \hspace{1cm} (EvalExp)

```c
51c  \hspace{1cm} (evaluate d->u.exp and possibly print the result 51c)≡ \hspace{1cm} \langle 51a \rangle
\{
  Value v = eval(d->u.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.

$$\langle \text{define}(f, \langle x_1, \ldots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi\{f \mapsto \text{user}(\langle x_1, \ldots, x_n \rangle, e)\} \rangle$$

(DefineFunction)

The evaluator does not check to see that the $x_1, \ldots, x_n$ are all distinct—the $x_i$'s are checked when the definition is parsed, by function check_def_duplicates in chunk 1119e.

### 1.5.3 Implementation of main

The main function coordinates all the pieces and forms a working interpreter. Such an interpreter can operate in two modes:

- In **interactive** mode, the interpreter prompts for every input, and when it detects a syntax error, it does not print the source-code location.
- In **non-interactive** mode, the interpreter does not prompt for any input, and when it detects a syntax error, it prints the source-code locations.

Interactive mode is meant for interactive use, and non-interactive mode is meant for redirecting standard input from a file. The interpreter is in interactive mode by default, but if its given the option -q, for “quiet,” it operates in non-interactive mode.

```
(int main(int argc, char *argv[]) {
    bool interactive = (argc <= 1) || (strcmp(argv[1], "-q") != 0);
    Prompts prompts = interactive ? STD_PROMPTS : NO_PROMPTS;
    set_toplevel_error_format(interactive ? WITHOUT_LOCATIONS : WITH_LOCATIONS);

    Valenv globals = mkValenv(NULL, NULL);
    Funenv functions = mkFunenv(NULL, NULL);
    print(1198c)

    XDefstream xdefs = filexdefs("standard input", stdin, prompts);
    while (setjmp(errorjmp))
    {
        readevalprint(xdefs, globals, functions, ECHOES);
        return 0;
    }
}
```

Before entering its main loop, the interpreter performs these phases of initialization:

- It decides whether it is operating interactively or non-interactively, and it sets prompts and the error format accordingly.
- It initializes print and fprint (the code appears in Appendix G).
• It creates empty environments for functions and global variables, then populates the `functions` environment with functions from the initial basis.

• It creates a stream of `XDef`s from the standard input.

The main loop is in the `readevalprint` function, the call to which is preceded by a C idiom:

\[
\text{⟨idiomatic error handler 53a⟩ ≡ while (setjmp(errorjmp)) { ⟨recover from an error⟩ }}\]

This idiom uses `setjmp` to deal with errors. On the first loop test, `setjmp` initializes `errorjmp` and returns zero, so the code in `⟨recover from an error⟩` is not executed, and control continues following the `while` loop. If an error occurs later, the error routine calls `longjmp(errorjmp, 1)`, which returns control to the `setjmp` again, this time returning 1. At this point the body of the `while` is executed. (In the definition of `main` above, no work is needed to recover from an error, instead of a block containing the action `⟨recover from an error⟩`, I use an empty statement, which is written as a single semicolon.) On the next iteration through the `while` statement, the process starts over from the beginning, because `setjmp` resets the jump buffer and returns zero again.

The initial basis includes both primitives and user-defined functions. We install the primitives first.

\[
\text{⟨install the initial basis in functions 53b⟩ ≡ (52b) 53d⟩}
\]

\[
\{ \text{static const char *prims[]} = \\
\{ "+", "-", "*", "/", "<", ">", "=", "println", "print", "printu", 0 \}; \\
\text{for (const char **p = prims; *p; p++) { \\
\text{Name x = strtoname(*p); \\
\text{bindfun(x, mkPrimitive(x), functions}); \\
\text{}} } 
\}
\]

I represent the user-defined part of the initial basis as a single string, which is interpreted by `readevalprint`. These functions also appear in Figure 1.1 on page 15, from which this code is derived automatically.

\[
\text{⟨predefined Impcore functions, as strings 53c⟩ ≡ (53d)}
\]

\[
\text{const char *basis= "(define and (b c) (if b c b))\n" "(define or (b c) (if b b c))\n" "(define not (b) (if b 0 1))\n" "(define <= (x y) (not (> x y)))\n" "(define >= (x y) (not (< x y)))\n" "(define != (x y) (not (= x y)))\n" "(define mod (m n) (- m (* n (/ m n))))\n";}
\]

\[
\text{⟨install the initial basis in functions 53b⟩ + ≡ (52b) o53b}
\]

\[
\{ \text{const char *fundefs = ⟨predefined Impcore functions, as strings 53c⟩; \\
\text{if (setjmp(errorjmp)) \\
\text{assert(0); // if error in predefined function, die horribly \\
\text{readevalprint(stringxdefs("predefined functions", fundefs), globals, functions, NO_ECHOES); \\
\text{}} } 
\}
\]
1.5.4 Implementation of names

Because names and environments are core concepts in programming languages, their implementations are included in this chapter. The implementations are straightforward, and the techniques I use should be familiar.

Each name is associated with a string. I just store the string inside the name.

54a \[(name.c 54a)\] 54b\>

\[
\text{struct Name} \\
\quad \text{const char } *s; \\
\}
\]

Returning the string associated with a name is trivial.

54\> \((name.c 54a)+\equiv \quad 54c\)

\[
\text{const char* nametostr(Name np)} \\
\quad \text{assert(np } \neq \text{ NULL)}; \\
\quad \text{return np->s; }
\]

Finding the name associated with a string is harder. To meet the specification, if I get a string I have seen before, I must return the same name I returned before. To remember what I have seen and returned, I use the simplest possible data structure: \textit{all_names}, a list of all names we ever returned. Given a string \textit{s}, a simple linear search finds the name associated with it, if any.

54c \((name.c 54a)+\equiv \quad 54d\)

\[
\text{Name strtoname(const char } *s) \\
\quad \text{static Namelist all_names; } \\
\quad \text{assert(s } \neq \text{ NULL)}; \\
\]

\[
\quad \text{for (Namelist unsearched = all_names; unsearched; unsearched = unsearched->tl)} \\
\quad \quad \text{if (strcmp(s, unsearched->hd->s) } = \text{ 0)} \\
\quad \quad \quad \text{return unsearched->hd; }
\]

\((allocate a new name, add it to all_names, and return it 54d)\)

A faster implementation might use a search tree or a hash table, not a simple list. Hanson (1996, Chapter 3) shows such an implementation.

If the string \textit{s} isn’t associated with any name on the list \textit{all_names}, I make a new name and add it.

54c \((allocate a new name, add it to all_names, and return it 54d)\)\equiv

\[
\text{Name np = malloc(sizeof(*np)); } \\
\quad \text{assert(np } \neq \text{ NULL); } \\
\quad \text{np->s = malloc(strlen(s) + 1); } \\
\quad \text{assert(np->s } \neq \text{ NULL); } \\
\quad \text{strcpy((char*)np->s, s); } \\
\quad \text{all_names = mkNL(np, all_names); } \\
\quad \text{return np; }
\]

1.5.5 Implementation of environments

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 G.
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.

```c
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.

```c
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 G.

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

```c
static Value* findval(Name x, Valenv env) {
    Namelist xs;
    Valuelist vs;
    for (xs=env->xs, vs=env->vs; xs && vs; xs=xs->tl, vs=vs->tl)
        if (x == xs->hd)
            return &vs->hd;
    return NULL;
}
```

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

```c
bool isvalbound(Name name, Valenv env) {
    return findval(name, env) != NULL;
}
```

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

```c
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 24 on page 78.)

```c
void bindval(Name name, Value val, Valenv env) {
    Value *vp = findval(name, env);
    if (vp != NULL)
        *vp = val; // safe optimization
    else {
        env->xs = mkNL(name, env->xs);
        env->vs = mkVL(val, env->vs);
    }
}
```

### 1.6 Operational semantics revisited: Proofs

If the interpreter in Section 1.5 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 1.4. 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 your language does what you think it does.

#### 1.6.1 Proofs about evaluation: Theory

If we are to draw conclusions from proofs, we must be sure that we know what a proof is. In programming languages, we borrow from formal logic the representation of a proof as a derivation of a judgment. In a derivation, inference rules are instantiated and composed into a derivation tree. Just as in a single rule, the conclusion of the derivation appears on the bottom, below a horizontal line. Above the line are the premises needed to prove that conclusion. The premises must be related to the conclusion by some inference rule. Finally, each of the premises must be the conclusion of a derivation of its own. This logical style of proof is called natural-deduction style, and that gives the semantics in this chapter the rest of its name: it is a big-step, natural-deduction semantics.

A derivation is also called a proof tree; the root contains the conclusion, and each subtree is also a derivation. A leaf node represents the application of an inference rule that has no premises, like the Literal rule on page 20. Every time a call to the \( \text{eval} \) function terminates, that call corresponds to a leaf node in a derivation.
Unusually for computer scientists, we write a derivation tree with the root at the bottom. For example, here is part of a derivation tree for evaluating the sum of two squares, written as the expression

\((+ (* x x) (* y y))\),

in an environment where \(\rho\) binds \(x\) to 3 and \(y\) to 4.

\[
\begin{align*}
\text{FORMALVAR} & \quad x \in \text{dom} \rho \quad \rho(x) = 3 \\
\text{APPLY_MUL} & \quad \langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle \\
\text{FORMALVAR} & \quad x \in \text{dom} \rho \quad \rho(x) = 3 \\
\text{APPLY_ADD} & \quad \langle \text{APPLY}(\ast, \text{VAR}(x), \text{VAR}(x)), \xi, \phi, \rho \rangle \Downarrow \langle 9, \xi, \phi, \rho \rangle \\
& \quad \langle \text{APPLY}(\ast, \text{VAR}(y), \text{VAR}(y)), \xi, \phi, \rho \rangle \Downarrow \langle 25, \xi, \phi, \rho \rangle
\end{align*}
\]

Each node in the tree 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

\[\langle \text{APPLY}(\ast, \text{VAR}(y), \text{VAR}(y)), \xi, \phi, \rho \rangle \Downarrow \langle 16, \xi, \phi, \rho \rangle.\]

To save space, we can take some liberties with the notation. In particular, instead of writing abstract syntax like \(\text{APPLY}(\ast, \text{VAR}(y), \text{VAR}(y))\), we can instead write the concrete syntax \((\ast y y)\). The resulting derivation is easier to digest, but it is harder to correlate with the rules of the operational semantics:

\[
\begin{align*}
\text{FORMALVAR} & \quad x \in \text{dom} \rho \quad \rho(x) = 3 \\
\text{APPLY_MUL} & \quad \langle x, \xi, \phi, \rho \rangle \Downarrow \langle 3, \xi, \phi, \rho \rangle \\
\text{FORMALVAR} & \quad x \in \text{dom} \rho \quad \rho(x) = 3 \\
\text{APPLY_ADD} & \quad \langle (\ast x x), \xi, \phi, \rho \rangle \Downarrow \langle 9, \xi, \phi, \rho \rangle \\
& \quad \langle (\ast y y), \xi, \phi, \rho \rangle \Downarrow \langle 25, \xi, \phi, \rho \rangle
\end{align*}
\]

If we don’t label the nodes, we can squeeze in the full derivation tree:

\[
\begin{align*}
\langle x, \xi, \phi, \rho \rangle & \Downarrow \langle 3, \xi, \phi, \rho \rangle \\
\langle x, \xi, \phi, \rho \rangle & \Downarrow \langle 3, \xi, \phi, \rho \rangle \\
\langle y, \xi, \phi, \rho \rangle & \Downarrow \langle 4, \xi, \phi, \rho \rangle \\
\langle y, \xi, \phi, \rho \rangle & \Downarrow \langle 4, \xi, \phi, \rho \rangle \\
\langle (\ast x x), \xi, \phi, \rho \rangle & \Downarrow \langle 9, \xi, \phi, \rho \rangle \\
\langle (\ast y y), \xi, \phi, \rho \rangle & \Downarrow \langle 25, \xi, \phi, \rho \rangle
\end{align*}
\]

Every derivation tree is intimately related to a recursive execution of \texttt{eval}. The relationship is shown in visual form in Section 3.3 starting on page 240.

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 13 on page 77. 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 14 on page 77 asks you to show that in Impcore, the expression \((\text{if } x \ x \ 0)\) is always equivalent to \(x\). Reasoning about derivations is called metatheory.

### 1.6.2 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.
CHAPTER 1. AN Imperative Core

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
\[
⟨(+ x 0), ξ, ϕ, ρ⟩ ⇓ ⟨v, ξ′, ϕ, ρ′⟩.
\]

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 43a. In both cases, we look for inference rules with \(\text{APPLY}(+, \ldots)\) in their conclusions.

If we consult Section 1.4.6, we see there are two such rules: \(\text{APPLYUser}\) and \(\text{APPLYAdd}\).

And now we see that the claim is false! If \(ϕ(+\) refers to a user-defined function, the \(\text{APPLYUser}\) rule kicks in, and it is not safe to replace \((+ x 0)\) with \(x\). Here’s a demonstration:

\[
(\text{terrifying transcript 58}) \equiv
\]
\[
→ (\text{define } + (x y) y) ; \text{no sane person would do this}
→ (\text{define addzero } (x) (+ x 0))
→ (addzero 99)
0
→ (\text{define addzero2 } (x) x)
→ (addzero2 99)
99
\]

If a compiler writer wants to be able to replace an instance of \((+ x 0)\) with \(x\), he or she will first have to prove that the environment \(ϕ\) in which \((+ x 0)\) is evaluated never binds \(\ast\) 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.)

Section 1.9 on page 66 includes several exercises that you can solve using metatheory:

- In Exercises 14 and 15 you can investigate under what circumstances \((\text{if } x x 0)\) is equivalent to \(x\).
- In Exercise 16 you can prove that if evaluation of a \(\text{while}\) loop terminates, its value is zero.
- In Exercise 17 you can prove that evaluating an expression can’t create a new variable.
- In Exercise 19 you can prove that in Impcore, evaluating an expression always produces the same result. (In languages that support parallel execution, this property usually doesn’t hold.)
- In Exercise 24 you can prove that Impcore programs can be evaluated using a stack of mutable environments, as the implementation in Section 1.5 does.

A little metatheory goes a long way.
1.6. OPERATIONAL SEMANTICS REVISITED: PROOFS

1.6.3 Why bother with semantics, proofs, theory, and metatheory?

If you’ve never seen an operational semantics before, you might wonder what’s up with all these Greek letters and horizontal lines. What’s the point? Isn’t it easier just to look at the code?

No, it’s not. The great thing about an operational semantics is that it allows us to gloss over all sorts of “implementation details” that actually 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 what we want to understand is what programs do, we don’t care. Once you get used to the Greek letters and horizontal lines, you’ll find that they are easier to read than code—much easier. That’s why when you find a new idea in a professional paper, the idea will usually be nailed down using operational semantics. A good reason to know operational semantics is to 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 to be very much help to a programmer. A programmer is much more likely to use something like the algebraic laws in the next chapter (Section 2.4 on page 99). The programmer—or even a specialist—uses the 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, good metatheory discovers universal truths about derivations. Such truths correspond to facts about all programs in a given language. Depending on the nature of the truth in question, it 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 24 on page 78). 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.

- 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 the difference between theory and metatheory, get an idea of how a metatheoretic proof can go, and get a feeling of what metatheory can do for you.
1.7 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. You’ll see new, more interesting values in Chapter 2, coming up next. And in Chapter 6 we add 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 80 asks you to add local variables to Impcore. With this extension you’ll be able to write code like this, which adds up the odd numbers from 1 to n:

```
(> (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)
```

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. There are other kinds of extensions, including new looping constructs, that 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 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)))
```
Sidebar: 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 62, 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 29. 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 24: 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 2.17.4 on page 187.
The \texttt{while*} loop can be defined as syntactic sugar:

\[(\texttt{while* condition } e_1 \cdots e_n) \triangleq (\texttt{while condition } (\texttt{begin } e_1 \cdots e_n))\]

As another example, C’s \texttt{do...while} loop executes the body first, then the condition. In Impcore, we might define a \texttt{do-while} as syntactic sugar:

\[(\texttt{do-while body condition}) \triangleq (\texttt{begin body } (\texttt{while condition body}))\]

Finally, C’s complicated four-part \texttt{for} loop can also be defined as syntactic sugar:

\[(\texttt{for pre test post body}) \triangleq (\texttt{begin pre } (\texttt{while test } (\texttt{begin body post})))\]

You can implement these alternatives in Exercise 29 on page 80. For some sample code that adds syntactic sugar to Impcore, see Section B.7 on page 1120.

1.8 \textbf{Summary}

Impcore is a toy, but its simple, imperative control constructs—procedure definitions, conditionals, and loops—model the structure of all imperative 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 in programming languages: 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.

1.8.1 \textbf{Key words and phrases}

\section*{Programming-language theory}

\textbf{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, or perhaps about its type (Chapter 6).

\textbf{Judgment form} A template for creating \textit{judgments}. For example, \((\texttt{(+ 2 2)}, \xi, \phi, \rho) \Downarrow 4\) is a judgment, but \((e, \xi, \phi, \rho) \Downarrow v\) is a judgment form. You get from a \textit{judgment form} to a \textit{judgment} by substituting real things (e.g., \textit{abstract syntax}, values, \textit{environments}) for \textit{metavariables}.

\textbf{Metavariable} A variable used in a \textit{judgment form} or elsewhere in a \textit{theory}, which stands for something described by the judgment form or the theory. Example metavariables include

- \(e\) for an \textit{expression}
- \(d\) for a \textit{definition}
- \(x\) for a \textit{program variable}
- \(\rho\) for an \textit{environment}
- \(\sigma\) for a \textit{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.
1.8. SUMMARY

**Syntactic form** A template for creating a phrase in a programming language, like `(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 1.7 on page 60 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 judgement form expresses, in one step, the evaluation of syntax to produce a result. For example, a single judgment `⟨e, ξ, φ, ρ⟩ \Downarrow v` shows how an expression is evaluated to produce a value, or a judgment `⟨d, ξ, ϕ⟩ → ⟨ξ', φ'⟩` 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 judgement 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.
CHAPTER 1. AN IMPERATIVE CORE

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).

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\(^9\)), 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, 7, 9, and 10, its type system. A language’s theory may be used to create a definitional interpreter.

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.

\(^9\)More accurately, if you have read Chapter 10, it is like applying a value constructor to arguments.
1.8. SUMMARY

Elements and structure of programming languages

**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 ⟨ϕ, ξ⟩ 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.

**Imperative programming** A style of programming in which the primary mechanism of composition is sequential composition of side-effecting computations, like set. Imperative 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 an imperative programming language. A better example is μCLU (Chapter 8).

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 parser 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.

1.8.2 Further reading


The term “initial basis” is taken from ML, where it refers to a collection of environments binding not only values but types, “signatures,” and other entities.

Plotkin (1981) started the modern movement toward operational semantics. His paper 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.

Ken Thompson was the first I know of to create an extensible printer for C programs; his implementation appeared in Ninth Edition Research Unix. Hanson (1996, Chapter 14) describes another implementation.

Hunt and Larus (2007) describe the Singularity project and say a little bit about Sing#.

1.9 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 a set of exercises that’s good for you, I’ve organized them into groups both big and small. In each chapter, you’ll find several large groups of exercises organized around broad themes or activities. Those activities often include programming in a bridge language, working with a semantics, and modifying an interpreter—but there are others as well. Within an activity you’ll find smaller groups of closely related exercises. And each chapter’s exercises are preceded by two summaries:

- 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.

- The guide to all the exercises lists every exercise by number, and it says in a sentence or two what the exercises are about. Each small group of closely related exercises is described together, in one paragraph of the guide.

If you choose your exercises well, doing them will be the best part of your experience.
1.9. EXERCISES

Highlights

Here are some of the highlights of the exercises below:

- When combined, Exercises 1 and 2 on page 69 offer a challenging thought experiment about the syntactic structure of languages.

- Of all the exercises on programming with numbers, Exercise 10 on page 73 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 9 on page 71.

- Exercise 13 on page 77 asks you to write a derivation; to start reasoning about derivations, follow up with Exercise 14, 15, or 16 on page 77.

- This is the only chapter in which you get to do much metatheory. The very best of the metatheoretic exercises are Exercises 18 and 24 on pages 77 and 78. 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 17 and 19 on page 77 are significantly easier but also worthwhile.

- Exercises 20 and 21 on page 78 invite you to do a little language design: what if variables didn’t have to be declared before use?

- 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. To get more practice adding new syntax to an interpreter, try Exercise 29.

Guide to all the exercises

Exercises 1 and 2 asks you to think about a new idea—syntactic categories—in two different ways. Exercise 1 asks you to apply the informal idea of syntactic categories to a language you already know. Exercise 2 asks you to develop a precise, mathematical definition of syntactic category. Exercise 1 will help you solidify your understanding of the idea; Exercise 2 will challenge your mathematical skills.

In any language, programmers need to know in what parts of the program each identifier is bound in the environment. The rules determining the binding or visibility of identifiers are often called scoping rules, and Exercise 3 asks you to understand scoping rules for a short C program and to give an analogous program in Impcore.

In Impcore, the only data type is “number,” and Exercises 4 to 8 offer some classic computations with numbers. Exercises 9 and 10 look at the relationship between numbers and numerals, where a numeral is a sequence of digits. In Exercise 9, you code several functions that ask about properties of decimal numerals. Exercise 10 asks you to relate decimal numerals to binary numerals.
Exercises 11 and 12 turn to proof theory and metatheory. Exercise 11 asks you to apply proof-theoretic ideas from Section 1.6 to a familiar, simple linguistic structure: numerals again. Exercise 12 is a sequence of “finger exercises” in metatheory; you look at what’s provable about pieces of syntax, and how the appearance of code in a program doesn’t mean the code is necessarily evaluated. You also look at the limitations of the constructive style of proofs used in programming languages: sometimes you need one proof system to show a property and a second proof system to show the complement of that property!

Exercises 13 to 27 explore not only Impcore’s operational semantics but also connections between operational semantics and language design. Exercises 13 to 15 ask you to write ironclad proofs about the evaluation of a few, very simple, particular expressions in Impcore. Exercises 16 to 19 ask you to prove properties that are true of many or all expressions in Impcore: In Exercise 16 you show that if a WHILE loop terminates, its value is always zero. In Exercise 17 you show that the evaluation of an Impcore expression, no matter what the expression is, doesn’t change the set of variables bound in the environment. In Exercise 18 you figure out if eliminating SET is enough to guarantee that the bindings to the variables are unchanged. In Exercise 19 you show that Impcore is deterministic: if evaluation of an expression succeeds, it always produces the same answer.

Exercises 20 to 23 explore the semantics of various language extensions: ways to use a variable without first declaring it (Exercises 20 and 21), operational semantics of a C-style FOR loop (Exercise 22), and a probabilistic coin flip (Exercise 23).

Exercise 24 shifts your semantic attention from language design to language implementation. You prove the central fact about the implementation of Impcore, C, and other languages in the C and Algol families: environments can be implemented on a stack (the call stack).

Exercises 25 to 27 look at limitations of operational semantics. Exercise 25 asks you to develop a semantics for a language in which, as in C, the order in which arguments are evaluated is not specified. Exercises 26 and 27 ask you to develop semantics for the extended definitions check-expect and check-error.

Exercises 28 to 35 invite you to solidify your knowledge of the programming infrastructure by modifying the interpreter, usually to extend the language. Exercise 28 asks for a new primitive read, which doesn’t require any new syntax. Exercise 29 asks you to implement the loop forms described in Section 1.7, which require only syntactic sugar. Exercise 30 asks you to add local variables to Impcore, which requires you to change both the syntax and the evaluator. Exercise 31 asks you to implement the semantics for unbound variables that you developed in Exercise 20. It requires no new syntax; you change only the evaluator. Exercise 32 asks you to pass parameters by reference, not by value. Reference parameters are less popular today than in the late 20th century, but they are still offered by C++ and are still worth exploring. The last of the extensions is Exercise 33, which asks you to change not the syntax but the values used by Impcore programs. To use the new floating-point numbers, you’ll want to change the existing arithmetic primitives and also add some new ones.

Exercises 34 and 35 stick with vanilla Impcore, instead exploring other properties of the interpreter. Exercise 34 asks if the interpreter is ever slow, and if so, how you can speed it up. Exercise 35 explores a matter of software craftsmanship: how would you modify the interpreter to be sure it always closes open file descriptors?
1.9.1 Language design: learning to think about syntactic structure

1. Pick your favorite programming language and identify the syntactic categories. (If you’re not sure exactly what a syntactic category is, you’re not alone: see Exercise 2.)

• 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.

1.9.2 The difficulties of theory

2. As far as I know, the term syntactic category does not have a precise, technical definition. One interpretation is to say that a syntactic category is a nonterminal in a grammar for which more than one kind of phrase can be produced. If we look at the grammar for Impcore on page 6, we see that this interpretation allows def and exp, but it rules out function and primitive on the grounds that even though function and primitives are nonterminals that have multiple productions, each production of function or primitive always produces exactly the same kind of phrase: a single name.

But if you have studied formal language theory, you will know that any language as complex as Impcore can be described by a plethora of different grammars. I would prefer, if possible, that syntactic categories be properties of a language, not of the grammar used to describe the language.

In this Exercise, please develop a precise, formal definition of “syntactic category.”

• One starting point might be a substitution principle: if two phrases are in the same syntactic category, then in any grammatical program, if you interchange one phrase for the other, the new program is still grammatical. For example, in a C program, replacing x - 1 with *p++ cannot introduce a syntax error. If there are maximal sets of interchangeable phrases, these might be related to syntactic categories.

• Another possible approach is to define syntactic categories not in terms of concrete syntax but instead to use abstract syntax, as in Section 1.2. A study of well-formed trees, or of interchangeable trees, might lead to a satisfying definition.
In one sentence, here is the problem: develop a definition of the vague term “syntactic category” that you find precise, satisfying, and meaningful independent of how a language’s concrete syntax is expressed.

1.9.3 Programming in Impcore

3. The scope rules of Impcore are identical to that of C. Consider this C program:

```c
#include <stdio.h>

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) What does the C program print?
(b) Write, in Impcore, a sequence of four definitions that correspond to the C program.

Exercises 4–10 specify some number-theoretic functions for you to program. These exercises are adapted from Kamin (1990, Chapter 1). If you can, use recursion; it is good practice for Chapter 2:

4. Define a function $\sigma$ satisfying $\sigma(m, n) = m + (m+1) + \cdots + n$. When $m > n$, the behavior of $\sigma$ is unspecified; your implementation may do anything you like.

5. Define functions $\exp$ and $\log$. When base $b$ and exponent $n$ are nonnegative, $\exp(b, n) = b^n$, and when $b > 1$ and $n > 0$, $\log(b, n)$ 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.

6. Define a function $\choose$ such that $\choose(n, k)$ 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 $\choose(n, k)$ 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:

\[
\binom{n}{0} = 1 \quad \text{when } n \geq 0
\]

\[
\binom{n}{n} = 1 \quad \text{when } n \geq 0
\]

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad \text{when } n > 0 \text{ and } k > 0
\]
1.9. EXERCISES

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.

7. Define a function \( \text{fib} \) such that \( \text{fib}(n) \) is the \( n \)th Fibonacci number. The Fibonacci numbers are a sequence of numbers defined by these laws:

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \quad \text{when } n > 1
\end{align*}
\]

8. Define functions \( \text{prime?} \), \( \text{nthprime} \), \( \text{sumprimes} \), and \( \text{relprime?} \) 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.

9. In this exercise, you write functions that take numbers apart and look at properties of their digits. You’ll need to understand how to define decimal representations inductively, as described in Exercise 11 below.

(a) Write a function \( \text{given-positive-all-fours?} \), which when given a positive number, returns 1 if its decimal representation is all fours and 0 otherwise.

\[
\begin{align*}
\text{given-positive-all-fours?} \ 4 &\equiv 1 \\
\text{given-positive-all-fours?} \ 4444 &\equiv 1 \\
\text{given-positive-all-fours?} \ 4443 &\equiv 0 \\
\text{given-positive-all-fours?} \ 34 &\equiv 0
\end{align*}
\]

For COMP 105, Tufts University, Fall 2016 only --- do not redistribute
(b) Write a function \texttt{all-fours?}, which when given any number, returns 1 if its decimal representation is all fours and 0 otherwise. See if you can come with something a bit more satisfying than this:

\begin{verbatim}
72a \textlangle unsatisfying answer 72a\textrangle \equiv 
    (define all-fours? (n)
        (if (> n 0) (given-positive-all-fours? n) 0))
\end{verbatim}

\begin{verbatim}
72b \textlangle exercise transcripts 71\textrangle \equiv
\rightarrow (all-fours? 0) 
    0
\rightarrow (all-fours? -4) ; HORROR SHOW --- mod should always be positive!
    0
\rightarrow (all-fours? 4) 
    1
\rightarrow (all-fours? 44444) 
    1
\rightarrow (all-fours? 44443) 
    0
\rightarrow (all-fours? 34) 
    0
\end{verbatim}

(c) Define a function \texttt{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.

\begin{verbatim}
72c \textlangle exercise transcripts 71\textrangle \equiv
\rightarrow (all-one-digit? 0) 
    1
\rightarrow (all-one-digit? -4) ; HORROR SHOW --- mod should always be positive!
    1
\rightarrow (all-one-digit? 4) 
    1
\rightarrow (all-one-digit? 44444) 
    1
\rightarrow (all-one-digit? 44443) 
    0
\rightarrow (all-one-digit? 33) 
    1
\end{verbatim}
1.9. EXERCISES

(d) Define a function \texttt{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.

\[
\begin{align*}
\langle \text{exercise transcripts} \rangle &+\equiv \\
&\rightarrow (\text{increasing-digits? } 1) \\
&\quad \rightarrow (\text{increasing-digits? } 1123) \\
&\quad \rightarrow (\text{increasing-digits? } 12345) \\
&\quad \rightarrow (\text{increasing-digits? } 54321) \\
&\quad \rightarrow (\text{increasing-digits? } 12435) \\
&\quad \rightarrow (\text{increasing-digits? } -123) \\
&\quad \rightarrow (\text{increasing-digits? } 489) \\
&\quad \rightarrow (\text{increasing-digits? } 1066) \\
&\rightarrow 0
\end{align*}
\]

If you define these functions and are feeling bold, try Exercise 3 on page 197.

10. Define a function \texttt{binary} such that \texttt{(binary m)} is the number whose decimal representation looks like the binary representation of \(m\). For example, \texttt{(binary 12)} = 1100, since \(1100_2 = 12_{10}\). An ideal implementation of \texttt{binary} will work on any integer input, including negative ones. For example, \texttt{(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 11 below.

1.9.4 Digging into proof theory and metatheory

11. A \textit{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 \(d\):

\[
d \quad ::= \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
N_{10} \quad ::= \quad d \mid N_{10}d
\]

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 \textit{meaning} of a numeral is a \textit{number}. The numerals above are written in typewriter font; we’ll write numbers in ordinary math font. To write the meaning of a decimal numeral \(N_{10}\), we’ll write \(\mathcal{D}[N_{10}]\). For example, \(\mathcal{D}[297] = 297\).
Here is a proof system for computing meanings of decimal numerals. The judgment form is $D[N_{10}] = e$, where $N_{10}$ is a decimal numeral, and $e$ is a mathematical formula. (The $[· · ·]$ symbols are called “Oxford brackets,” and they are used to wrap syntax.) Here’s one rule for each decimal digit, and one rule for the case where a numeral is a numeral followed by a digit:

\[
\begin{align*}
D[0] &= 0 \\
D[1] &= 1 \\
D[2] &= 2 \\
D[3] &= 3 \\
D[4] &= 4 \\
D[5] &= 5 \\
D[6] &= 6 \\
D[7] &= 7 \\
D[8] &= 8 \\
D[9] &= 9 \\
\end{align*}
\]

\[
D[N_{10}] = e, \quad D[d] = k
\]

\[
D[N_{10}d] = 10 \times e + k
\]

The purpose of this exercise is for you to create a similar proof system for binary numerals.

(a) Define precisely what is a binary digit.

(b) Give an inductive definition of binary numerals.

(c) Give a proof system for showing the meaning of a binary numeral.

12. Impcore is an imperative core because it uses side effects. Of these side effects, the most important is mutation, also known as assignment.\(^\text{10}\) 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 18, 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 $e$ has $set$, which is intended to mean “expression $e$ has a $set$ in it.”

\[
\begin{align*}
\text{Set} & \quad \text{has set} \\
\text{set}(x,e) & \quad \text{has set} \\
\text{If1} & \quad e_1 \text{ has set} \quad \text{If2} & \quad e_2 \text{ has set} \quad \text{If3} & \quad e_3 \text{ has set} \\
& \quad \text{if}(e_1, e_2, e_3) \text{ has set} \quad & \quad \text{if}(e_1, e_2, e_3) \text{ has set} \quad & \quad \text{if}(e_1, e_2, e_3) \text{ has set} \\
\text{While1} & \quad e_1 \text{ has set} \quad \text{While2} & \quad e_2 \text{ has set} \quad \text{While3} & \quad e_3 \text{ has set} \\
& \quad \text{while}(e_1, e_2) \text{ has set} \quad & \quad \text{while}(e_1, e_2) \text{ has set} \quad & \quad \text{while}(e_1, e_2) \text{ has set} \\
\text{Begin} & \quad e_i \text{ has set} \quad \text{Apply} & \quad e_i \text{ has set} \quad \text{Begin} & \quad e_i \text{ has set} \quad \text{Apply} & \quad e_i \text{ has set} \\
& \quad \text{begin}(e_1, \ldots, e_n) \text{ has set} \quad & \quad \text{apply}(f, e_1, \ldots, e_n) \text{ has set} \quad & \quad i \in \{1, \ldots, n\} \\
& \quad i \in \{1, \ldots, n\} \\
& \quad i \in \{1, \ldots, n\} \\
\end{align*}
\]

Here are some things to notice about the proof system:

\(^{10}\)The other side effects are printing and $use$.  

For COMP 105, Tufts University, Fall 2016 only --- do not redistribute.
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 \texttt{set}.

There’s no premise on the rule for \texttt{set}. A \texttt{set} expression definitely has \texttt{set}, no matter what’s true about its subexpressions.

Any of the other expressions has \texttt{set} if and only if one of its proper subexpressions has \texttt{set}. For \texttt{if} and \texttt{while}, I can write out all the cases explicitly, but for \texttt{begin} and \texttt{apply}, I have to write rule schemas. The notation $i \in \{1, \ldots, n\}$ on the right-hand side of a rule means that the rule is repeated $n$ times: once for each value of $i$.

Here are some things to try:

(a) Show that \textit{having \texttt{set}} isn’t the same as \textit{evaluating \texttt{set}}. Give an example of an expression $e$ such that $e$ \texttt{has set}, but you can guarantee that evaluating $e$ never evaluates a \texttt{set}.

(b) Prove that the expression

\begin{verbatim}
(while (> x 1)
  (if (= 0 (mod x 2))
    (set x (/ x 2))
    (set x (+ (* 3 x) 1))))
\end{verbatim}

has \texttt{set}.

The proof system above does a perfect job telling us that an expression has \texttt{set}. But if an expression \textit{doesn’t} have \texttt{set}, the proof system can’t tell us anything! Knowing that expression doesn’t have set requires its own proof system. And knowing that they are consistent requires metatheory.

(c) Develop a proof system for yet another judgment form: \texttt{e hasn't set}. There should be a derivation of $e$ \texttt{hasn't set} exactly when expression $e$ doesn’t have a \texttt{set} in it.

Your proof system should have a structure that is closely related to the structure of the proof system for $e$ \texttt{has set}. The relationship is what a mathematician would call a \textit{dual} relationship:

- Where $e$ \texttt{has set} lacks proof rules, such as for literals and variables, $e$ \texttt{hasn't set} will have trivial proof rules with no premises.
- Where $e$ \texttt{has set} has a trivial proof rule with no premises, such as for \texttt{set}, $e$ \texttt{hasn't set} will lack proof rules. (There’s no way you can proof that a \texttt{set} expression doesn’t have \texttt{set}.)
- Where an expression has subexpressions, for $e$ \texttt{has set} it is sufficient to prove that any of the subexpressions has a \texttt{set}. But for $e$ \texttt{hasn't set} it is necessary to prove that all of the subexpressions haven’t got \texttt{set}. (This duality is an instance of DeMorgan’s Law.)

(d) This last part is hard, but it’s the payoff: Use metatheory to show that the two judgments are mutually exclusive and cover all cases. That is, for any expression $e$, it is the case that there is a valid derivation of exactly one of the two judgments $e$ \texttt{has set} and $e$ \texttt{hasn't set}. I recommend using proof by induction on the syntactic structure of $e$. 
Figure 1.4: Summary of operational semantics (expressions)
1.9. EXERCISES

\[
\frac{\langle e, \xi, \phi, \{ \} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{VAL}(x, e), \xi, \phi \rangle \rightarrow \langle \xi'[x \mapsto v], \phi \rangle}
\]
\begin{equation}
\text{(DEFINEGLOBAL)}
\end{equation}

\[
\frac{\langle x_1, \ldots, x_n \rangle \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \ldots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi[f \mapsto \text{USER}(\langle x_1, \ldots, x_n \rangle, e)] \rangle}
\]
\begin{equation}
\text{(DEFINEFUNCTION)}
\end{equation}

\[
\frac{\langle e, \xi, \phi, \{ \} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{EXP}(e), \xi, \phi \rangle \rightarrow \langle \xi'[\text{IT} \mapsto v], \phi \rangle}
\]
\begin{equation}
\text{(EVALEXP)}
\end{equation}

Figure 1.5: Summary of operational semantics (definitions)

1.9.5 Working with Impcore's semantics

13. Use the operational semantics to prove that if you evaluate \((\text{begin} (\text{set} x 3) x)\) 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 57.

14. Use the operational semantics to show that if there exist environments \(\xi, \phi, \rho\) 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 \(v_1 = v_2\).

15. Use the operational semantics to show that there exist environments \(\xi, \phi, \rho, \xi', \phi', \rho'\) 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 \(\xi, \phi, \rho, \xi'', \phi'', \rho''\) 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 \(\xi, \phi, \rho\) such that both expressions evaluate successfully.

16. Prove that the value of a \(\text{WHILE}\) expression is always zero. That is, given any \(\xi, \phi, \rho, e_1,\) and \(e_2\), prove that 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.

17. 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 \(\text{dom} \xi = \text{dom} \xi'\).

18. Is it true or false that evaluating an expression without a \(\text{SET}\) node does not change any environment? Use metatheory to justify your answer. To be sure you understand what it means to have a \(\text{SET}\) node, see Exercise 12.

19. Prove that Impcore is deterministic. That is, for any \(e\) and any environments, there is at most one \(v\) such that \(\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle\).
20. 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.\(^\text{11}\)

(c) Which of the two changes do you prefer, and why?

21. Write a program in Impcore that causes an error when run by the Impcore interpreter, prints 1 when run with the Awk-like extension from problem 20, and prints 0 when run with the Icon-like extension from problem 20. In Impcore, a “program” is simply a sequence of definitions.

22. Give operational semantics for a C-like \texttt{for}(e_1, e_2, e_3, e_4). Like a \texttt{while} expression, a \texttt{for} expression is evaluated for its side effects, so the value it returns is unimportant. Choose whatever result value you like.

23. Suppose you want extend Impcore by adding a \texttt{flip} expression, which models the flip of a coin by returning either 0 or 1 with equal probability. What rules would you add to the operational semantics for \texttt{flip}? Would the resulting language still be deterministic?

24. The operational semantics never mutates an environment; when a change in environment is needed, the semantics always creates a new environment that is equal to an existing environment with one or more new bindings. But our code uses mutable environments—and for formal parameters \( \rho \), our code uses a stack of mutable environments. (The stack is implicit in the C call stack.) Prove that using a stack of mutable environments is safe. (In this question, we don’t consider the environment of global variables \( \xi \).)

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 we’re asking you to imagine an implementation that uses an explicit, global stack of environments.

We suggest that you reimagine procedure \texttt{eval}—the implementation of the evaluation judgment \( \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle \)—as one that 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 \texttt{eval}.

\(^{11}\)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.
1.9. EXERCISES

Prove that the following properties hold.

(a) In eval, the implementation of every proof rule that ends in the judgment form $(e, \xi, \phi, \rho) \Downarrow (v, \xi', \phi, \rho')$ 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 $(e, \xi, \phi, \rho) \Downarrow (v, \xi', \phi, \rho')$. 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 is the justification for our implementation of bindval, as referred to in Section 1.5.5.

25. In some programming languages, like C, function parameters may be evaluated in any order, and the choice of order is up to the implementation. How would you write this specification in a formal semantics? What if the order is unspecified, but must be the same each time? What if the order can change each time?

26. 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.
27. 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 12.

   (a) 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.

   (b) Write a proof system for this judgment form.

   (c) 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.

   (d) 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.”

1.9.6 Modifying the Impcore interpreter

28. 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.

29. Using syntactic sugar, extend Impcore with the looping constructs discussed in Section 1.7:

   (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 22.

For some sample code that adds syntactic sugar to Impcore, see Section B.7 on page 1120.

30. 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 1106b in Section B.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. Implement your solutions to Exercise 20. Use your implementation to test your solution to Exercise 21.

32. 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 \texttt{eval} and \texttt{fetchval} to be \texttt{Value*}, and make \texttt{Valuelists} hold \texttt{Value*}s rather than \texttt{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 \texttt{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 10): “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.

33. Add floating-point numbers to Impcore. Let \texttt{Value} be a sum type containing either an integer or a floating-point number:

\[
\text{Value} = \begin{cases} 
\text{INT} & (\text{int}) \\
\text{FLOAT} & (\text{float}) 
\end{cases}
\]

The arithmetic primitives should apply integer or floating-point operations, depending on the types of their arguments; given arguments of mixed types, these functions should promote integers to floats. Add primitives \texttt{trunc} and \texttt{round} to convert floating-point values to integers; \texttt{trunc} should round toward minus infinity and \texttt{round} should round to nearest (to even if halfway between two integers). Support floating-point literals using \texttt{strtod}.

34. 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\footnote{See URL \url{http://www.cs.princeton.edu/software/cii}.} 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?

As in other programs an Impcore “program” is simply a sequence of definitions.
35. The implementation of \texttt{use} in chunk \texttt{⟨evaluate d->u.use, possibly mutating globals and functions 50c⟩} leaks open file descriptors when files have bugs. Explain how you would fix the problem.