1. Why Build Another Programming Language?

This book will show you how to build your own programming language, but first, you should ask yourself, why would I want to do this? For a few of you, the answer will be simple: because it is so much fun. However, for the rest of us, it is a lot of work to build a programming language, and we need to be sure about it before we make a start. Do you have the patience and persistence that it takes?

This chapter points out a few good reasons for building your own programming language, as well as some situations where you don't have to build your contemplated language after all; designing a class library for your application domain might be simpler and just as effective. However, libraries *have* their downsides, and sometimes only a new language will do.

After this chapter, the rest of this book will, having considered things carefully, take for granted that you have decided to build a language. In that case, you should determine some of the requirements for your language. But first, we're going to cover the following main topics in this chapter:

- Motivations for writing your own programming language
- The difference between programming languages and libraries
- The applicability of programming language tools to other software projects
- Establishing the requirements for your language
- A case study that discusses the requirements for the Unicon language

Let's start by looking at motivations.

So, you want to write your own programming language…

Sure, some programming language inventors are rock stars of computer science, such as Dennis Ritchie or Guido van Rossum! But becoming a rock star of computer science was easier back then. I heard the following report a long time ago from an attendee of the second History of Programming Languages Conference: *The consensus was that the field of programming languages is dead. All the important languages have been invented already*. This was proven wildly wrong a year or 2 later when Java hit the scene, and perhaps a dozen times since then when languages such as Go emerged. After a mere 6 decades, it would be unwise to claim our field is mature and that there's nothing new to invent that might make you famous.

Still, celebrity is a bad reason to build a programming language. The chances of acquiring fame or fortune from your programming language invention are slim. Curiosity and desire to know how things work are valid reasons, so long as you've got the time and inclination, but perhaps the best reasons for building your own programming language are need and necessity.

Some folks need to build a new language or a new implementation of an existing programming language to target a new processor or compete with a rival company. If that's not you, then perhaps you've looked at the best languages (and compilers or interpreters) available for some domain that you are developing programs for, and they are missing some key

features for what you are doing, and those missing features are causing you pain. Every once in a blue moon, someone comes up with a whole new style of computing that a new programming paradigm requires a new language for.

While we are discussing your motivations for building a language, let's talk about the different kinds of languages, organization, and the examples this book will use to guide you. Each of these topics is worth looking at.

Types of programming language implementations

Whatever your reasons, before you build a programming language, you should pick the best tools and technologies you can find to do the job. In our case, this book will pick them for you. First, there is a question of the implementation language that you are building your language in. Programming language academics like to brag about writing their language in that language itself, but this is usually only a half-truth (or someone was being very impractical and showing off at the same time). There is also the question of just what kind of programming language implementation to build:

- A pure **interpreter** that executes source code itself
- A **native compiler** and a runtime system, such as in C
- A **transpiler** that translates your language into some other high-level language
- A **bytecode compiler** with an accompanying bytecode machine, such as Java

The first option is fun but usually too slow. The second option is the best, but usually, it's too labor-intensive; a good native compiler may take many person-years of effort.

While the third option is by far the easiest and probably the most fun, and I have used it before with success, if it isn't a prototype, then it is sort

of cheating. Sure, the first version of C++ was a transpiler, but that gave way to compilers and not just because it was buggy. Strangely, generating high-level code seems to make your language even more dependent on the underlying language than the other options, and languages are moving targets. Good languages have died because their underlying dependencies disappeared or broke irreparably on them. It can be the death of a thousand small cuts.

This book chooses the fourth option: we will build a bytecode compiler with an accompanying bytecode machine because that is a sweet spot that gives the most flexibility while still offering decent performance. A chapter on native code compilation is included for those of you who require the fastest possible execution.

The notion of a bytecode machine is very old; it was made famous by UCSD's Pascal implementation and the classic SmallTalk-80 implementation, among others. It became ubiquitous to the point of entering lay English with the promulgation of Java's JVM. Bytecode machines are abstract processors interpreted by software; they are often called **virtual machines** (as in **Java Virtual Machine**), although I will not use that terminology because it is also used to refer to software tools that use real hardware instruction sets, such as IBM's classic platforms or more modern tools such as **Virtual Box**.

A bytecode machine is typically quite a bit higher level than a piece of hardware, so a bytecode implementation affords much flexibility. Let's have a quick look at what it will take to get there…

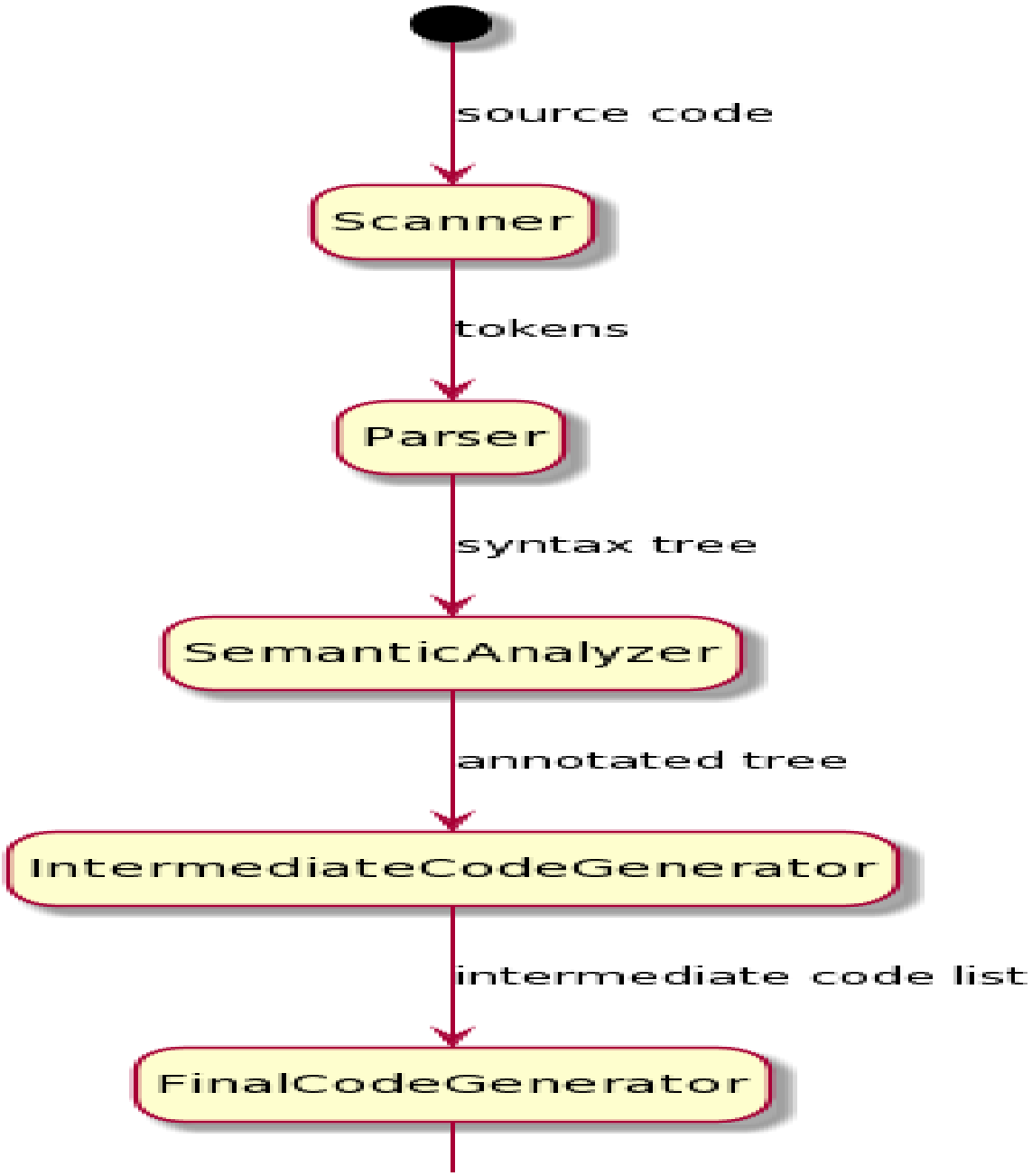
Organizing a bytecode language implementation

To a large extent, the organization of this book follows the classic organization of a bytecode compiler and its corresponding virtual machine. These components are defined here, followed by a diagram to summarize them:

- A **lexical analyzer** reads in source code characters and figures out how they are grouped into a sequence of words or tokens.
- A **syntax analyzer** reads in a sequence of tokens and determines whether that sequence is legal according to the grammar of the language. If the tokens are in a legal order, it produces a syntax tree.
- A **semantic analyzer** checks to ensure that all the names being used are legal for the operations in which they are being used. It checks their types to determine exactly what operations are being performed. All this checking makes the syntax tree heavy laden with the extra information about where variables are declared and what their types are.
- An **intermediate code generator** figures out memory locations for all the variables and all the places where a program may abruptly change execution flow, such as loops and function calls. It adds them to the syntax tree and then walks this even fatter tree before building a list of machine-independent intermediate code instructions.

- A **final code generator** turns the list of intermediate code instructions into the actual bytecode in a file format that will be efficient to load and execute.

Independent from the steps of this bytecode virtual machine compiler, a **bytecode interpreter** is written to load and execute programs. It is a gigantic loop with a switch statement in it, but for exotic programming languages, the compiler might be no big deal and all the magic will happen there. The whole organization can be summarized by the following diagram:


source code
Scanner
tokens
Parser
syntax tree
SemanticAnalyzer
annotated tree
IntermediateCodeGenerator
intermediate code list
FinalCodeGenerator

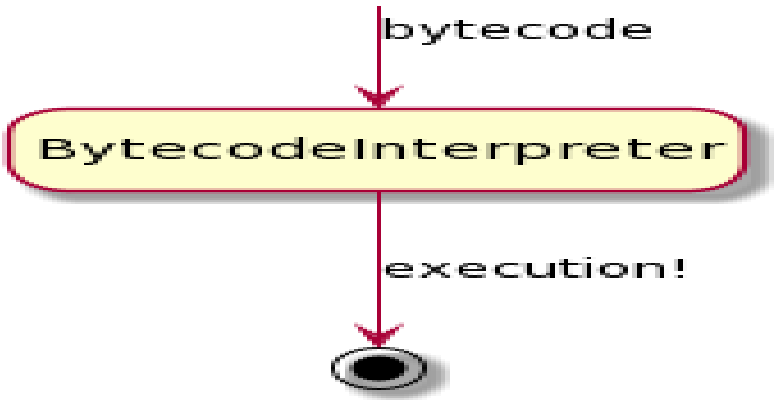


Figure 1.1 – Phases and dataflow in a simple programming language

It will take a lot of code to illustrate how to build a bytecode machine implementation of a programming language. How that code is presented is important and will tell you what you need to know going in, and much of what you may learn from going through this book.

Languages used in the examples

This book provides code examples in two languages using a **parallel translations model**. The first language is **Java**, because that language is ubiquitous. Hopefully, you know it or C++ and will be able to read the examples with intermediate proficiency. The second example language is the author's own language, **Unicon**. While reading this book, you can judge for yourself which language is better suited to building your own programming language. As many examples as possible will be provided in both languages, and the examples in the two languages will be written as similarly as possible. Sometimes, this will be to the advantage of the lesser language.

The differences between Java and Unicon will be obvious, but they are somewhat lessened in importance by the compiler construction tools we will use. We will use modern descendants of the venerable Lex and YACC tools to generate our scanner and parser, and by sticking to tools for Java and Unicon that remain as compatible as possible with the original Lex and YACC, the frontends of our compiler will be nearly identical in both languages. Lex and YACC are declarative programming languages that solve some of our hard problems at an even higher level than Java or Unicon.

While we are using Java and Unicon as our implementation languages, we will need to talk about one more language: the example language we are building. It is a stand-in for whatever language you decide to build. Somewhat arbitrarily, I will introduce a language called **Jzero** for this

purpose. Niklaus Wirth invented a toy language called **PL/0** (**programming language zero**; the name is a riff on the language name **PL/1**) that was used in compiler construction courses. Jzero will be a tiny subset of Java that serves a similar purpose. I looked pretty hard (that is, I googled Jzero and then Jzero compiler) to see if someone had already posted one we could use, and did not spot one by that name, so we will just make it up as we go along.

The Java examples in this book will be tested using OpenJDK 14; maybe other versions of Java (such as OpenJDK 12 or Oracle Java JDK) will work the same, but maybe not. You can get OpenJDK from http://openjdk.java.net, or if you are on Linux, your operating system probably has an OpenJDK package that you can install. Additional programming language construction tools (Jflex and byacc/j) that are required for the Java examples will be introduced in subsequent chapters as they are used. The Java implementations we will support might be more constrained by which versions will run these language construction tools than anything else.

The Unicon examples in this book work with Unicon version 13.2, which can be obtained from http://unicon.org. To install Unicon on Windows, you must download a .msi file and run the installer. To install on Linux, you usually do a git clone of the sources and type *make*. You will then want to add the unicon/bin directory to your PATH:

```
git clone git://git.code.sf.net/p/unicon/unicon
make
```

Having gone through our organization and the implementation that this book will use, perhaps we should take another look at when a programming language is called for, and when one can be avoided by developing a library instead.

Language versus library – what's the difference?

Don't make a programming language when a library will do the job. Libraries are by far the most common way to extend an existing programming language to perform a new task. A **library** is a set of functions or classes that can be used together to write applications for some hardware or software technology. Many languages, including C and Java, are designed almost completely to revolve around a rich set of libraries. The language itself is very simple and general, while much of what a developer must learn to develop applications consists of how to use the various libraries.

The following is what libraries can do:

- Introduce new data types (classes) and provide public functions (an API) for manipulating them
- Provide a layer of abstraction on top of a set of hardware or operating system calls

The following is what libraries cannot do:

- Introduce new control structures and syntax in support of new application domains
- Embed/support new semantics within the existing language runtime system

Libraries do some things badly, in that you might end up preferring to make a new language:

- Libraries often get larger and more complex than necessary.
- Libraries can have even steeper learning curves and poorer documentation than languages.
- Every so often, libraries have conflicts with other libraries, and version incompatibilities often break applications that use libraries.

There is a natural evolutionary path from the library to language. A reasonable approach to building a new language to support an application domain is to start by making or buying the best library available for that application domain. If the result does not meet your requirements in terms of supporting the domain and simplifying the task of writing programs for that domain, then you have a strong argument for a new language.

This book is about building your own language, not just building your own library. It turns out that learning about these tools and techniques is useful in other contexts…

Applicability to other software engineering tasks

The tools and technologies you learn about from building your own programming language can be applied to a range of other software engineering tasks. For example, you can sort almost any file or network input processing task into three categories:

- Reading XML data with an XML library
- Reading JSON data with a JSON library
- Reading anything else by writing code to parse it in its native format

The technologies in this book are useful in a wide array of software engineering tasks, which is where the third of these categories is encountered. Frequently structured data must be read in a custom file format.

For some of you, the experience of building your own programming language might be the single largest program you have written thus far. If you persist and finish it, it will teach you lots of practical software engineering skills, besides whatever you learn about compilers and interpreters and such. This will include working with large dynamic data structures, software testing, and debugging complex problems, among other skills.

That's enough of the inspirational motivation. Let's talk about what you should do first: figure out your requirements.

Establishing the requirements for your language

After you are sure you need a new programming language for what you are doing, take a few minutes to establish the requirements. This is open-ended. It is you defining what success for your project will look like. Wise language inventors do not create a whole new syntax from scratch. Instead, they define it in terms of a set of modifications to make to a popular existing language. Many great programming languages (Lisp, Forth, SmallTalk, and many others) had their success significantly limited by the degree to which their syntax was unnecessarily different from mainstream languages. Still, your language requirements include what it will look like, and that includes syntax.

More importantly, you must define a set of control structures or semantics where your programming language needs to go beyond existing language(s). This will sometimes include special support for an application domain that is not well-served by existing languages and their libraries. Such **domain-specific languages (DSLs)** are common enough that whole books are focused on that topic. Our goal for this book will be to focus on the nuts and bolts of building the compiler and runtime system for such a language, independent of whatever domain you may be working in.

In a normal software engineering process, requirements analysis would start with brainstorming lists of functional and non-functional requirements. Functional requirements for a programming language

involve the specifics of how the end user developer will interact with it. You might not anticipate all the command-line options for your language upfront, but you probably know whether interactivity is required, or whether a separate compile-step is OK. The discussion of interpreters and compilers in the previous section, and this book's presentation of a compiler, might seem to make that choice for you, but Python is an example of a language that provides a fully interactive interface, even though the source code you type in it gets crunched into bytecode rather than interpreted.

Non-functional requirements are properties that your programming language must achieve that are not directly tied to the end user developer's interactions. They include things such as what operating system(s) it must run on, how fast execution must be, or how little space the programs written in your language must run within.

The non-functional requirement regarding how fast execution must be usually determines the answer as to whether you can target a software (bytecode) machine or need to target native code. Native code is not just faster, it is also considerably more difficult to generate, and it might make your language considerably less flexible in terms of runtime system features. You might choose to target bytecode first, and then work on a native code generator afterward.

The first language I learned to program on was a BASIC interpreter in which the programs had to run within 4 KB of RAM. BASIC at the time had a low memory footprint requirement. But even in modern times, it is not uncommon to find yourself on a platform where Java won't run by

default! For example, on virtual machines with configured memory limits for user processes, you may have to learn some awkward command-line options to compile or run even simple Java programs.

Many requirements analysis processes also define a set of use cases and ask the developer to write descriptions for them. Inventing a programming language is different from your average software engineering project, but before you are finished, you may want to go there. A use case is a task that someone performs using a software application. When the software application is a programming language, if you are not careful, the use cases may be too general to be useful, such as *write my application* and *run my program*.

While those two might not be very useful, you might want to think about whether your programming language implementation must support program development, debugging, separate compilation and linking, integration with external languages and libraries, and so forth. Most of those topics are beyond the scope of this book, but we will consider some of them.

Since this book will present the implementation of a language called Jzero, here are some requirements for it. Some of these requirements may appear arbitrary. If it is not clear to you where one of them came from, it either came from our source inspiration language (plzero) or previous experience teaching compiler construction:

- **Jzero should be a strict subset of Java**. All legal Jzero programs should be legal Java programs. This requirement allows us to check

the behavior of our test programs when we are debugging our language implementation.

- **Jzero should allow enough computation to allow interesting computations**. This includes if statements, while loops, and multiple functions, along with parameters.
- **Jzero should support a few data types, including booleans, integers, arrays, and the String type**. It only needs to support a subset of their functionality, as described later. These are enough types to allow input and output of interesting values into a computation.
- **Jzero should emit decent error messages, showing the filename and line number, including messages for attempts to use Java features not in Jzero**. We will need reasonable error messages to debug the implementation.
- **Jzero should run fast enough to be practical**. This requirement is vague, but it implies that we won't be doing a pure interpreter. Pure interpreters are a very retro thing, evocative of the 1960s and 1970s.
- **Jzero should be as simple as possible so that I can explain it**. Sadly, this rules out generating native code or even JVM bytecode; we will provide our own simple bytecode machine.

Perhaps more requirements will emerge as we go along, but this is a start. Since we are constrained for time and space, perhaps this requirements list is more important for what it does not say, than for what it does say. By way of comparison, here are some of the requirements that led to the creation of the Unicon programming language.

Case study - requirements that inspired the Unicon language

This book will use the Unicon programming language, located at http://unicon.org, for a running case study. We can start with reasonable questions such as, why build Unicon, and what are its requirements? To answer the first question, we will work backward from the second one.

Unicon exists because of an earlier programming language called Icon, from the University of Arizona (http://www.cs.arizona.edu/icon/). Icon has particularly good string and list processing abilities and is used for building many scripts and utilities, as well as both programming language and natural language processing projects. Icon's fantastic built-in data types, including structure types such as lists and (hash) tables, have influenced several languages, including Python and Unicon. Icon's signature research contribution is integrating goaldirected evaluation, including backtracking and automatic resumption of generators, into a familiar mainstream syntax. Unicon requirement #1 is to preserve these best bits of Icon.

Unicon requirement #1 - preserve what people love about Icon

One of the things that people love about Icon is its expression semantics, including its generators and goal – directed evaluation. Icon also provides a rich set of built-in functions and data types so that many or most programs can be understood directly from the source code. Unicon's goal would be 100% compatibility with Icon. In the end, we achieved more like 99% compatibility.

It is a bit of a leap from preserving the best bits to the immortality goal of ensuring old source code will run forever, but for Unicon, we include that in requirement #1. We have placed a harder requirement on backward compatibility than most modern languages. While C is very backward compatible, C++, Java, Python, and Perl are examples of languages that have wandered away, in some cases far away, from being compatible with the programs written in them back in their glory days. In the case of Unicon, perhaps 99% of Icon programs run unmodified as Unicon programs.

Icon was designed for maximum programmer productivity on small-sized projects; a typical Icon program is less than 1,000 lines of code, but Icon is very high level and you can do a lot of computing in a few hundred lines of code! Still, computers keep getting more capable and users want to write much larger programs than Icon was designed to handle. Unicon requirement #2 was to support programming in large-scale projects

Unicon requirement #2 – support large-scale programs working on big data

For this reason, Unicon adds classes and packages to Icon, much like C++ adds them to C. Unicon also improved the bytecode object file format and made numerous scalability improvements to the compiler and runtime system. It also refines Icon's existing implementation to be more scalable in many specific items, such as adopting a much more sophisticated hash function.

Icon is designed for classic UNIX pipe-and-filter text processing of local files. Over time, more and more people were wanting to write with it and required more sophisticated forms of input/output, such as networking or graphics. Unicon requirement #3 is to support ubiquitous input/output capabilities at the same high level as the built – in types.

Unicon requirement #3 - high-level input/output for modern applications

Support for I/O is a moving target. At first, it included networking facilities and GDBM and ODBC database facilities to accompany Icon's 2D graphics. Then, it grew to include various popular internet protocols and 3D graphics. What input/output capabilities are ubiquitous continues to evolve and varies by platform, but touch input and gestures or shader programming capabilities are examples of things that have become rather ubiquitous by this point.

Arguably, despite billionfold improvements in CPU speed and memory size, the biggest difference between programming in 1970 and programming in 2020 is that we expect modern applications to use a myriad of sophisticated forms of I/O: graphics, networking, databases, and so forth. Libraries can provide access to such I/O, but language-level support can make it easier and more intuitive.

Icon is pretty portable, having been run on everything from Amigas to Crays to IBM mainframes with EBCDIC character sets. Although the platforms have changed almost unbelievably over the years, Unicon still retains Icon's goal of maximum source code portability: code that gets written in Unicon should continue to run unmodified on all computing platforms that matter. This leads to Unicon requirement #4.

Unicon requirement #4 - provide universally implementable system interfaces

For a very long time, portability meant running on PCs, Macs, and UNIX workstations. But again, the set of computing platforms that matter is a moving target. These days, work is underway in Unicon to support Android and iOS, in case you count them as computing platforms. Whether they count might depend on whether they are open enough and used for general computing tasks, but they are certainly capable of being used as such.

All those juicy I/O facilities that were implemented for requirement #3 must be designed in such a way that they can be multi-platform portable across all major platforms.

Having given you some of Unicon's primary requirements, here is an answer to the question, why build Unicon at all? One answer is that after studying many languages, I concluded that Icon's generators and goal-directed evaluation (requirement #1) were features that I wanted when writing programs from now on. But after allowing me to add 2D graphics to their language, Icon's inventors were no longer willing to consider further additions to meet requirements #2 and #3. Another answer is that there was a public demand for new capabilities, including volunteer partners and some financial support. Thus, Unicon was born.

Summary

In this chapter, you learned the difference between inventing a programming language and inventing a library API to support whatever kinds of computing you want to do. Several different forms of programming language implementations were considered. This first chapter allowed you to think about functional and non-functional requirements for your own language. These requirements might be different from the example requirements discussed for the Java subset Jzero and the Unicon programming language, which were both introduced.

Requirements are important because they allow you to set goals and define what success will look like. In the case of a programming language implementation, the requirements include what things will look and feel like to the programmers that use your language, as well as what hardware and software platforms it must run on. The look and feel for a programming language includes answering both external questions regarding how the language implementation and the programs written in the language are invoked, as well as internal issues such as verbosity: how much the programmer must write to accomplish a given compute task.

You may be keen to get straight to the coding part. Although the classic *build and fix* mentality of novice programmers might work on scripts and short programs, for a piece of software as large as a programming language, we need a bit more planning first. After this chapter's coverage of the requirements, *Chapter 2, Programming Language Design*, will prepare you to construct a detailed plan for the implementation that will occupy our attention for the remainder of this book!

Questions

1. What are the pros and cons of writing a language transpiler that generates C code, instead of a traditional compiler that generates assembler or native machine code?
2. What are the major components or phases in a traditional compiler?
3. From your experience, what are some pain points where programming is more difficult than it should be? What new programming language feature(s) address these pain points?
4. Write a set of functional requirements for a new programming language.